UNIVERSIDAD DEL BÍO-BÍO

FACULTAD DE CIENCIAS EMPRESARIALES

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN Y TECNOLOGÍA DE LA INFORMACIÓN

INGENIERÍA CIVIL EN INFORMÁTICA

Tarea 2
Inv. de Software

Docente(s): Roberto Anabalón - Gastón Márquez - César Aguilera

Nombre alumno:

- Miguel Bustamante

Sección: 1

Fecha: 13 de Noviembre, 2025

1. Descripción del problema y Requisitos

"Mueblería los muebles hermanos S.A" es una empresa que comercializa diversos tipos de muebles, los cuales poseen múltiples variaciones (material, tamaño, estilo). El gerente Gustavo necesita un sistema que permita registrar, cotizar y vender estos muebles de forma consistente, asegurando la integridad de los datos y el control de inventario.

Requisitos Funcionales

- **Gestión de Catálogo (CRUD):** El sistema debe permitir crear, listar, actualizar y desactivar los muebles del catálogo. Los atributos requeridos son: ID, nombre, tipo, precio base, stock, estado (activo/inactivo), tamaño y material.
- **Gestión de Variantes:** Se debe poder registrar variaciones (ej. "barniz premium", "cojines de seda") que modifiquen y aumenten el precio base del producto. Si un producto no tiene variaciones seleccionadas, se considera "normal" y mantiene su precio base.
- **Cotizaciones y Ventas:** El sistema debe permitir crear una cotización que contenga uno o más muebles, especificando su variante y la cantidad.
- **Control de Stock:** Si se intenta confirmar una venta y no hay stock suficiente para algún producto, el sistema debe detener la operación y devolver un mensaje de error.
- **Confirmación de Venta:** Se debe poder confirmar una cotización, lo que la convierte formalmente en una "Venta". Esta acción debe decrementar el stock de los productos vendidos.

Requisitos Técnicos

- **Stack Tecnológico:** El backend debe ser codificado usando Spring Boot, conectarse a una base de datos MySQL y utilizar JUnit para las pruebas.
- **Arquitectura:** Se debe crear una API REST para exponer la lógica de negocio.
- **Patrones de Diseño:** Se deben implementar e identificar un mínimo de 2 patrones de diseño.

2. Descripción del Proyecto y Decisiones de Diseño

Link del repositorio: https://github.com/lotussjuice/CatalogoMueblesSpring.git

Arquitectura del Sistema El proyecto se desarrolló siguiendo una **arquitectura de 4 capas** para asegurar una clara separación de responsabilidades, alta cohesión y bajo acoplamiento:

1. **Capa de Entidad:** Clases que mapean la base de datos usando JPA.
    - Mueble: Contiene la información base del producto.
    - Variante: Almacena el nombre y el aumento de precio de una variación.
    - Cotización: Representa la "orden", con un estado y un total.
    - ItemCotizacion: Tabla intermedia que representa los items de una boleta, unión de entidades para concretar.

2. **Capa de Repositorio:** Interfaces que extiende JPARepository. Spring Data JPA provee automáticamente la implementación CRUD para el acceso a datos.
3. **Capa de Servicio:** Contiene toda la lógica interna del sistema.
    - **Decisión de Diseño:** Se utilizaron **implementaciones**. Esta decisión fue clave para permitir "mockear" (simular) fácilmente.
4. **Capa de Controlador:** API REST la cual redirecciona solicitudes al sistema.

3. Patrones de Diseño Implementados

Se implementaron dos patrones de diseño para resolver problemas específicos del proyecto:

1. Patrón Builder (Creacional)

- **Planteamiento:** La entidad Mueble posee 8 atributos. Crear un objeto con un constructor tradicional sería propenso a errores dada la complejidad, o permitiría estados inválidos si se usan setters vacíos.
- **Uso y Funcionalidad:** Se implementó una clase interna estática Mueble.MuebleBuilder dentro de la entidad. Esto permite una creación de objetos fluida, legible y segura. El constructor exige los campos obligatorios (nombre, precio, stock) y permite añadir los opcionales (tipo, material) de forma encadenada.

2. Patrón Strategy (Comportamiento)

- **Planteamiento:** El sistema debe validar el stock. Sin embargo, la lógica de validación podría cambiar. Por ejemplo, "Sillas" (alta rotación) requieren stock físico, pero "Sillones" (a pedido) podrían permitir ventas sin stock. Esto más que un requerimiento comprende un cambio en la lógica de negocio de la empresa, por lo cual se optó a añadir ante casos hipotéticos. No comprende una implementación directa, pero tras pensarlo se determinó como uno de los patrones más probables de uso a futuro.

4. Testing con JUnit

A. Testing de lógica

- Se validó que la lógica de negocio interna funcione correctamente. Se usó SpringBootTest para cargar únicamente la clase de implementación del servicio. Se usó MockBean para simular todos los repositorios y servicios dependientes (extensión la cual se empleó en el sprint 2 del proyecto semestral para no comprometer la base de datos).
- **Ejemplo de tests implementados:**
    - **testConfirmarVenta_StockInsuficiente:** Verificó que, si el stock es 10 y se piden 11, se lanza una RuntimeException con el mensaje "Stock insuficiente" y que el método actualizarStock nunca es llamado.
    - **testAgregarItemConVariante_RecalculaTotal:** Verificó que el precio de la variante se suma correctamente al total de la cotización.
    - **testDesactivarMueble:** Verificó que el método save del repositorio fue llamado con un objeto Mueble cuyo estado era INACTIVO.

B. Testing de API REST

- Se validó que los endpoints de la API estuvieran conectados correctamente, que manejan JSON y que respondiera con los códigos de estado HTTP adecuados. Se usó WebMvcTest para cargar solo la capa web. Se usó MockMvc para simular peticiones HTTP. El servicio fue simulado con MockBean de igual forma.
- **Ejemplos de tests implementados:**
    - **testGetListarMuebles:** Verificó que un GET a /api/muebles devolviera un status 200 (OK) y un JSON con una lista.
    - **testPostCrearMueble:** Verificó que un POST a /api/muebles con un JSON de entrada devolviera un status 201 (Created).
    - **testPostConfirmarVenta_StockInsuficiente:** Verificó que si el servicio (mockeado) lanzaba la excepción de stock, el controlador la capturaba y respondía con un status 400 (Bad Request) y el mensaje de error.

5. Requisitos Técnicos

Software Requerido

- **Java:** JDK 21 o superior.
- **Maven:** Versión 3.9 o superior (para gestionar las dependencias y el build).
- **Base de Datos:** Un servidor MySQL. Se recomienda **XAMPP** .

6. Manual de Usuario

Paso 1: Configuración de la Base de Datos

1. Asegúrese de tener un servidor MySQL corriendo (ej. iniciado desde el panel de control de XAMPP en localhost:3306).
2. Cree una base de datos vacía. Ejemplo: *"CREATE DATABASE muebleria_db;"*
3. Abra el archivo src/main/resources/application.properties.
4. Modifique las siguientes líneas con su usuario y contraseña de MySQL (el usuario por defecto de XAMPP es "root" sin contraseña):
    - spring.datasource.username=root
    - spring.datasource.password= (déjelo vacío si no tiene contraseña)
5. La línea spring.jpa.hibernate.ddl-auto=update creará las tablas automáticamente al arrancar.

Paso 2: Ejecutar las Pruebas Unitarias (JUnit)

1. Abra una terminal en la carpeta raíz del proyecto.
2. Ejecute el comando de Maven para limpiar, compilar y probar todo:
    - ./mvnw clean test
3. **Output:** Observe la terminal. Maven compilará y ejecutará todas las clases de prueba.

4. El resultado final debe mostrar [INFO] BUILD SUCCESS, indicando que todos los tests pasaron.
5. Para un reporte detallado, puede observar la consola de comandos viendo los resultados y el checklist de los test.
6. De forma alternativa runear directamente los archivos java de la carpeta *test,* su entorno de desarrollo mostrará los resultados y retornos de cada método test.

Paso 3 (Opcional): Ejecutar el Proyecto (API REST)

1. Abra una terminal (CMD, PowerShell, etc.) en la carpeta raíz del proyecto.
2. Ejecute el siguiente comando:
    - ./mvnw spring-boot:run
3. La API REST se iniciará y estará disponible en http://localhost:1010.

Paso 4 (Opcional): Probar la API manualmente con POSTMAN

- GET /api/muebles (Listar Muebles): Devuelve una lista JSON.
- POST /api/muebles (Crear Mueble): Envíe un JSON con la estructura del DTO CrearMuebleRequest.
- POST /api/cotizaciones (Crear Cotización): Envíe la petición sin cuerpo. Devolverá una nueva cotización vacía (anote el id).
- POST /api/cotizaciones/{id}/items (Agregar Item): Envíe a (ej.) .../1/items un JSON especificando muebleId, varianteId (puede ser null) y cantidad.
- POST /api/cotizaciones/{id}/confirmar (Confirmar Venta): Envíe la petición a (ej.) .../1/confirmar.
    - Output (Éxito): Devuelve un 200 OK con la cotización en estado CONFIRMADA.
    - Output (Fallo): Devuelve un 400 Bad Request con el mensaje "Stock insuficiente...".