Universidad del Bío-Bío

Ingeniería Civil en Informática

Tarea 2: "Mueblería Los Muebles Hermanos"

Curso: Ingeniería de Software – S1 2025

Sección: 1

Integrantes:

Alejandro Ortiz Ortega

Profesor:

Roberto Anabalón

Chillán, Noviembre de 2025

Índice

1. Enunciado

Se solicitó construir un backend completo para la mueblería ficticia **"Los Muebles Hermanos"**, aplicando buenas prácticas de **Ingeniería de Software**. El sistema debía:

- Exponer una API REST para gestionar catálogo de muebles, variaciones de precio, cotizaciones y ventas.
- Persistir la información en MySQL (H2 en modo pruebas) usando **Spring Data JPA**.
- Incorporar patrones de diseño (Strategy + Factory) para calcular precios con variaciones.
- Entregar datos de ejemplo y pruebas automatizadas que validaran los flujos críticos.

Alcance

La aplicación debía permitir:

- CRUD de muebles con atributos como tipo, tamaño, material, precio base, stock y estado.
- CRUD de variaciones asociadas a cada mueble, habilitando estrategias aditivas y porcentuales.
- Creación, confirmación y cancelación de cotizaciones, validando stock y estados antes de confirmar ventas.
- Entrega de datos iniciales para probar el flujo completo junto a un pequeño frontend estático.

2. Solución

Se implementó una API REST con **Spring Boot 3.5** y **Java 21**, organizada en capas (Controller → Service → Repository). Se usaron DTOs y mapeadores para desacoplar la API de las entidades JPA. El repositorio completo se encuentra en:

`github.com/OrvarOddr/Tarea2_IngSoft`

Dependencias y stack

- Spring Boot Starters: `spring-boot-starter-web`, `spring-boot-starter-data-jpa`, `spring-boot-starter-validation`.
- Base de datos: `mysql-connector-j` (runtime) y `h2` para pruebas.
- Testing: `spring-boot-starter-test` (JUnit 5, AssertJ, Mockito). Mock-maker inline habilitado para Java 21.
- Build: Maven Wrapper (`./mvnw`) y configuración de Surefire con `-Djdk.attach.allowAttachSelf=true`.

Modelo de dominio

El dominio principal se basa en cuatro entidades: `Mueble`, `Variacion`, `Cotizacion` y `CotizacionItem`. El mueble contiene los atributos de catálogo y se relaciona con sus variaciones:

Listing 1: Entidad Mueble con atributos de catálogo

```
@Entity
@Table(name = "mueble")
public class Mueble {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "nombre_mueble", nullable = false)
    private String nombre;
    @Enumerated(EnumType.STRING) private TipoMueble tipo;
    @Column(name = "precio_base", nullable = false, precision = 12,
        scale = 2)
    private BigDecimal precioBase;
    @Column(nullable = false) private Integer stock;
    @Enumerated(EnumType.STRING) private EstadoMueble estado;
    @Enumerated(EnumType.STRING) private Tamano tamano;
    @Column(nullable = false) private String material;
    @OneToMany(mappedBy = "mueble", cascade = CascadeType.ALL,
       orphanRemoval = true)
    private List<Variacion> variaciones = new ArrayList<>();
}
```

Estrategias de precio

Para soportar variaciones aditivas, porcentuales o nulas se implementó el patrón **Strategy** más una **Factory** que selecciona la estrategia adecuada. Esto evita usar condicionales en los servicios y permite extender fácilmente nuevas estrategias:

Listing 2: Factory que resuelve la estrategia de precio

```
@Component
public class PriceCalculatorFactory {
    private final Map<PriceStrategyType, PriceCalculator>
       calculatorMap;
    public PriceCalculatorFactory(List<PriceCalculator> calculators
       ) {
        this.calculatorMap = calculators.stream()
            .collect(Collectors.toMap(
                PriceCalculator::getType,
                Function.identity(),
                (first, second) -> first,
                () -> new EnumMap<>(PriceStrategyType.class)));
    }
    public PriceCalculator getCalculator(PriceStrategyType type) {
        return calculatorMap.getOrDefault(type, calculatorMap.get(
           PriceStrategyType.NONE));
    }
}
```

Reglas de negocio

Las reglas principales viven en los servicios. La creación y confirmación de cotizaciones validan stock, estado del mueble y pertenencia de variaciones antes de calcular precios:

Listing 3: Validaciones al crear y confirmar cotizaciones

```
public Cotizacion crearCotizacion(CotizacionRequest request) {
    if (request.items().isEmpty()) {
        throw new BusinessException("La␣cotizacion␣debe␣contener␣al
           ␣menos␣un␣mueble");
    }
    Cotizacion cotizacion = new Cotizacion();
    for (CotizacionItemRequest itemRequest : request.items()) {
        CotizacionItem item = construirItem(itemRequest);
        cotizacion.addItem(item);
    }
    return cotizacionRepository.save(cotizacion);
}

public Cotizacion confirmarCotizacion(Long id) {
    Cotizacion cotizacion = obtener(id);
    if (cotizacion.getEstado() != EstadoCotizacion.CREADA) {
        throw new BusinessException("Solo␣se␣pueden␣confirmar␣
           cotizaciones␣en␣estado␣CREADA");
    }
    cotizacion.getItems().forEach(item -> {
        Mueble mueble = muebleRepository.findById(item.getMueble().
           getId())
            .orElseThrow(() -> new ResourceNotFoundException("No␣
               existe␣mueble␣con␣id␣" + item.getMueble().getId()));
        if (mueble.getEstado() != EstadoMueble.ACTIVO) {
            throw new BusinessException("El␣mueble␣" + mueble.
               getNombre() + "␣no␣est ␣disponible");
        }
        if (mueble.getStock() < item.getCantidad()) {
            throw new BusinessException("Stock␣insuficiente␣para␣el
               ␣mueble␣" + mueble.getNombre());
        }
        mueble.setStock(mueble.getStock() - item.getCantidad());
        muebleRepository.save(mueble);
        item.setMueble(mueble);
    });
```

```
    cotizacion.setEstado(EstadoCotizacion.CONFIRMADA);
    return cotizacion;
}
```

API REST y DTOs

Los controladores exponen endpoints REST y delegan lógica a los servicios, mapeando entidades a DTOs para evitar exponer el modelo JPA:

Listing 4: Controlador del catálogo de muebles

```
@RestController
@RequestMapping("/api/muebles")
public class MuebleController {
    private final MuebleService muebleService;
    private final CatalogMapper catalogMapper;

    @GetMapping
    public List<MuebleResponse> listar(@RequestParam(name = "estado
       ", required = false) EstadoMueble estado) {
        return muebleService.listar(estado)
            .stream()
            .map(catalogMapper::toResponse)
            .collect(Collectors.toList());
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public MuebleResponse crear(@Valid @RequestBody MuebleRequest
       request) {
        Mueble creado = muebleService.crear(request);
        return catalogMapper.toResponse(creado);
    }
}
```

Persistencia, datos iniciales y despliegue

La configuración de base de datos reside en `application.properties`, con credenciales sobreescribibles por variables de entorno (`SPRING_DATASOURCE_URL`, `SPRING_DATASOURCE_USERNAME`, `SPRING_DATASOURCE_PASSWORD`, `SPRING_JPA_HIBERNATE_DDL_AUTO`, `SERVER_PORT`). El archivo `data.sql` limpia y carga datos de inicio: 8 muebles de distintos tipos/tamaños/estados y 12 variaciones (aditivas, porcentuales y ninguna) para probar rápido el flujo.

Para reproducir el ambiente se usa `docker-compose.yml` con tres servicios: `mysql` (puerto 3306, credenciales `muebles_user`/`muebles_pass`), `app` (Spring Boot en puerto 8081) y `frontend` (Nginx sirviendo los HTML en puerto 5173). Si se desea, se puede agregar un `.env` en la raíz para credenciales de MySQL y sobreescribir las variables del compose.

Patrones de diseño aplicados

- **Strategy + Factory**: cada variación de precio implementa `PriceCalculator`; la `PriceCalculatorFactory` selecciona la estrategia sin `switch` en servicios, facilitando extensiones.
- **Service Layer**: los servicios encapsulan reglas de negocio (validación de stock, estados y pertenencia de variaciones) y mantienen los controladores delgados. Ejemplo: `CotizacionService.confirmarCotizacion` aplica transacción, valida estado y descuenta stock en un solo punto.
- **DTO + Mapper**: se usan DTOs de request/response y mapeadores para no exponer entidades JPA y garantizar contratos estables en la API.
- **Repository (Spring Data JPA)**: acceso a datos desacoplado de la lógica; los repositorios definen solo interfaces, delegando la implementación a Spring.
- **Exception Handling centralizado**: `GlobalExceptionHandler` traduce excepciones de negocio a respuestas HTTP coherentes, manteniendo controladores limpios.

API resumida

- **Muebles**:
  - `GET /api/muebles?estado=` lista catálogo (opcional filtro por estado).
  - `POST /api/muebles` crea un mueble.
  - `PUT /api/muebles/{id}` actualiza datos.
  - `PATCH /api/muebles/{id}/estado` cambia estado (activo/inactivo).

- **Variaciones**:
  - `GET /api/muebles/{id}/variaciones` lista variaciones del mueble.
  - `POST /api/muebles/{id}/variaciones` crea una variación.
  - `PUT /api/muebles/{id}/variaciones/{varId}` actualiza.
  - `DELETE /api/muebles/{id}/variaciones/{varId}` elimina.
- **Cotizaciones**:
  - `POST /api/cotizaciones` crea una cotización.
  - `GET /api/cotizaciones` lista cotizaciones.
  - `POST /api/cotizaciones/{id}/confirmar` confirma y descuenta stock.
  - `POST /api/cotizaciones/{id}/cancelar` cancela.

Frontend

El frontend estático (carpeta `frontend/`) incluye vistas de catálogo, variaciones, estadísticas y cotizaciones. Se sirve con Nginx (puerto 5173 en Docker) y consume la API en `/api`; permite crear/editar muebles y variaciones, armar cotizaciones y confirmar ventas mostrando validaciones de stock.

3. Pruebas

Se creó una batería de **JUnit 5** que verifica tanto la lógica de negocio como la fábrica de estrategias:

- `PriceCalculatorFactoryTest` comprueba que cada estrategia calcule el precio correcto y que exista un fallback seguro al tipo `NONE`.
- `CotizacionServiceTest` cubre creación de cotizaciones con variaciones, confirmación que descuenta stock y errores por inventario insuficiente.
- `MuebleServiceTest` valida creación/actualización de catálogo y cambios de estado.
- `MueblesshermanosApplicationTests` asegura que el contexto Spring arranca correctamente.

Listing 5: Prueba de cálculo porcentual y fallback

```
@Test
void devuelveEstrategiaPorcentaje() {
    PriceCalculator calculator = factory.getCalculator(
       PriceStrategyType.PERCENTAGE);
    BigDecimal resultado = calculator.calculate(new BigDecimal("
       1000"), new BigDecimal("10"));
    assertThat(resultado).isEqualByComparingTo("1100.00");
}

@Test
void usaEstrategiaPorDefectoCuandoNoExiste() {
    PriceCalculator calculator = factory.getCalculator(
       PriceStrategyType.NONE);
    BigDecimal resultado = calculator.calculate(new BigDecimal("
       1000"), new BigDecimal("999"));
    assertThat(resultado).isEqualByComparingTo("1000");
}
```

Todas las pruebas se ejecutan con una base H2 aislada y el perfil de pruebas definido en `src/test/resources/application.properties`. Se disparan mediante `./mvnw test`, generando reportes en `target/surefire-reports/`.

4. Resultados y uso

Instrucciones para levantar el proyecto

- **Requisitos**: Docker + Docker Compose; o JDK 21 + `./mvnw` + MySQL en `localhost:3306` con el esquema `muebles_db`.
- **Docker Compose (recomendado)**: levanta app y MySQL en segundos.
  1. Construir y levantar:
     `docker compose build`
     `docker compose up -d`
  2. Ver logs de arranque: `docker compose logs app -f`
  3. Servicios: backend `http://localhost:8081`; frontend `http://localhost:5173`; MySQL `mysql://localhost:3306`
     (usuario/clave `muebles_user`/`muebles_pass`)
  4. Para detener y limpiar: `docker compose down -v`
- **Ejecución local (sin Docker)**:
  1. MySQL operativo; credenciales opcionales vía ambiente `SPRING_DATASOURCE_{URL,USERNAME,PASSWORD}`.
  2. API con datos precargados: `./mvnw spring-boot:run`
     Puerto **8081** (cambiable con `SERVER_PORT=8080 ./mvnw spring-boot:run`).
- **Frontend estático**: `frontend/` consume `/api`. Servir con `python3 -m http.server -directory frontend 5173`
  o `npx serve frontend`. Por defecto llama a `http://localhost:8081/api`.
- **Pruebas**: `./mvnw test` ejecuta JUnit sobre H2; reportes en `target/surefire-reports/`.

5. Conclusión

El proyecto consolida el diseño en capas con Spring, aplicando patrones de estrategia y fábrica para extender reglas de precios sin modificar los servicios. Las pruebas automatizadas garantizan los casos críticos de negocio (cálculo de precios, validación de stock y transiciones de estado), mientras que Docker Compose facilita la reproducibilidad del ambiente. El resultado es un backend mantenible, probado y listo para ser consumido por el frontend o integraciones externas.