Documentación
Pokemón Bytes

Documentación – Fase III

Índice

1. Introducción y Objetivos de la Fase

La Fase III se centra en la implementación de un sistema económico persistente y seguro. En un RPG, la gestión de recursos (dinero e ítems) es crítica para la progresión del jugador. El objetivo técnico principal ha sido garantizar la integridad referencial y la atomicidad de las operaciones financieras, evitando situaciones como "dinero fantasma" (gastar sin tener dinero) o "pérdida de bienes" (pagar y no recibir el objeto) ante fallos del servidor.

2. Arquitectura del Módulo Económico

El módulo sigue la arquitectura en capas establecida, utilizando DTOs para la comunicación y servicios transaccionales para la lógica.

Componentes Principales

- **TiendaController:** Endpoint (POST /api/v1/tienda/comprar). Actúa como pasarela, recibiendo la intención de compra (CompraRequest) y delegando en el servicio.
- **TiendaService:** Núcleo lógico. Verifica reglas de negocio (saldo suficiente, existencia del ítem) y ejecuta la transacción.
- **InventarioUsuario :** Resuelve la relación Muchos a Muchos entre Usuarios e Items, añadiendo el atributo de cantidad.
- **InventarioId (Clave Compuesta):** Clase embebible (@Embeddable) que garantiza la unicidad de la relación (un usuario no puede tener dos filas distintas para el mismo ítem).

3. Implementación de Transacciones Atómicas (TiendaService)

La lógica de compra reside en el método comprarItem. La característica más crítica de este servicio es el uso de la anotación @Transactional de Spring Framework.

```
@Service
public class TiendaService {

    // Inyección de dependencias (Repositorios)
    private final UserRepository userRepository;
    private final ItemRepository itemRepository;
    private final InventarioUsuarioRepository inventarioRepository;

    /**
     * Realiza la compra asegurando atomicidad .
     * Si falla el inventario, se revierte el cobro del dinero.
     */
    @Transactional // <--- Garantiza el Rollback automático
    public String comprarItem(String username, CompraRequest request) {
```

```
        // 1. Validaciones y recuperación de entidades
        Usuario usuario = userRepository.findByUsername(username)
                .orElseThrow(() -> new RuntimeException("Usuario no
                  encontrado"));

        Item item = itemRepository.findById(request.getItemId())
                .orElseThrow(() -> new RuntimeException("Ítem no existe"));

        int costoTotal = item.getPrecio() * request.getCantidad();

        // 2. Verificación de Fondos (Regla de Negocio)
        if (usuario.getDinero() < costoTotal) {
            throw new RuntimeException("Saldo insuficiente.");
        }

        // 3. Transacción Económica (Resta de Dinero)
        usuario.setDinero(usuario.getDinero() - costoTotal);
        userRepository.save(usuario);

        // 4. Actualización de Inventario (Gestión M:N)
        InventarioId inventarioId = new InventarioId(usuario.getIdUsuario(),
item.getIdItem());

        InventarioUsuario entradaInventario =
inventarioRepository.findById(inventarioId)
                .orElse(new InventarioUsuario(usuario, item, 0));

        entradaInventario.setCantidad(entradaInventario.getCantidad() +
request.getCantidad());
        inventarioRepository.save(entradaInventario);

        return "Compra exitosa. Nuevo saldo: " + usuario.getDinero();
    }
}
```

3.1. Garantía de Integridad

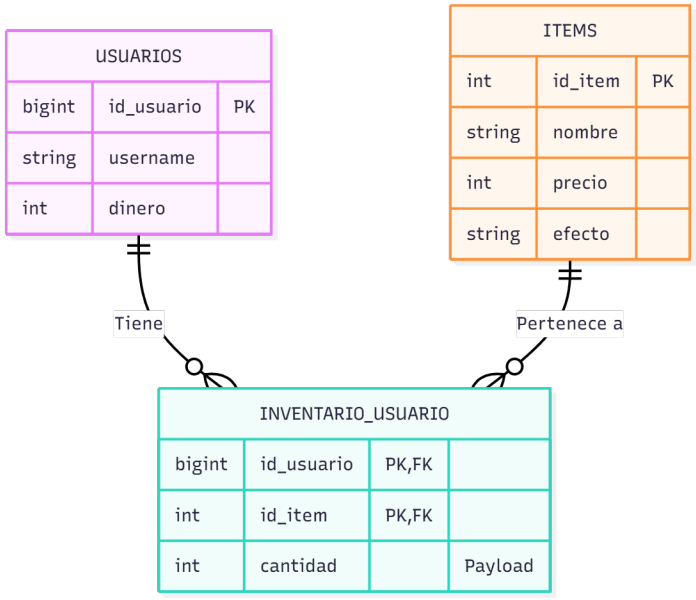
Al anotar el método con @Transactional, se establece un contexto de persistencia donde todas las operaciones de base de datos se tratan como una unidad indivisible.

Secuencia de la Transacción en Código:

1. **Validación de Existencia:** Se recuperan las entidades Usuario e Item desde la base de datos.
2. **Cálculo de Coste:** int costoTotal = item.getPrecio() * request.getCantidad();
3. **Verificación de Fondos:** Se comprueba si usuario.getDinero() >= costoTotal. Si no, se lanza una RuntimeException.
4. **Operación de Débito:** Se actualiza la entidad Usuario: usuario.setDinero(usuario.getDinero() - costoTotal).
5. **Operación de Inventario:**
    - Se busca si ya existe una relación en InventarioUsuario usando la clave compuesta.
    - Si existe, se suma la cantidad (setCantidad).
    - Si no existe, se instancia un nuevo objeto InventarioUsuario.
6. **Commit / Rollback:**
    - Si todo es correcto, Spring hace Commit y los cambios se guardan en MySQL.
    - Si ocurre cualquier error (ej. caída de base de datos al guardar el ítem después de restar el dinero), Spring ejecuta un Rollback automático, devolviendo el dinero al usuario.

4. Modelo de Datos Relacional: Relaciones M:N

Para gestionar el inventario de manera eficiente, se ha optado por un modelo relacional normalizado (3NF).



```
// 1. Definición de la Clave Primaria Compuesta
@Embeddable
@Data @NoArgsConstructor @AllArgsConstructor
public class InventarioId implements Serializable {
    private Long usuarioId;
    private Integer itemId;
}

// 2. Definición de la Entidad Intermedia
@Entity
@Table(name = "INVENTARIO_USUARIO")
@Data @NoArgsConstructor
public class InventarioUsuario {

    @EmbeddedId
    private InventarioId id; // Instancia de la clave compuesta

    @Column(nullable = false)
    private Integer cantidad; // Atributo extra de la relación

    // Vinculación con Usuario (Foreign Key 1)
    @ManyToOne(fetch = FetchType.LAZY)
    @MapsId("usuarioId")
    @JoinColumn(name = "id_usuario")
    private Usuario usuario;

    // Vinculación con Item (Foreign Key 2)
    @ManyToOne(fetch = FetchType.LAZY)
    @MapsId("itemId")
    @JoinColumn(name = "id_item")
    private Item item;
}
```

4.1. La Entidad Intermedia (InventarioUsuario)

La relación entre USUARIOS e ITEMS es de tipo Muchos a Muchos (Un usuario tiene muchos ítems, un ítem pertenece a muchos usuarios). Para añadir el atributo "Cantidad", se desarrollo esta relación en una entidad propia.

```
@Entity
@Table(name = "INVENTARIO_USUARIO")
public class InventarioUsuario {

    @EmbeddedId
    private InventarioId id; // Clave Compuesta

    @Column(nullable = false)
    private Integer cantidad; // Carga de la relación

    @ManyToOne
    @MapsId("usuarioId") // Vincula parte de la clave compuesta a la FK real
    private Usuario usuario;

    @ManyToOne
    @MapsId("itemId")
    private Item item;
}
```

4.2. Claves Compuestas (InventarioId)

Se ha implementado la clase InventarioId implementando Serializable para definir la Clave Primaria de la tabla intermedia.

```
@Embeddable
public class InventarioId implements Serializable {
    private Long usuarioId;
    private Integer itemId;
    // ... equals() y hashCode() para optimización de Hibernate
}
```

Esto asegura que la búsqueda de un objeto en la mochila sea extremadamente rápida mediante inventarioRepository.findById().

5. Diagrama de Flujo: Proceso de Compra

Este diagrama ilustra la lógica transaccional implementada en TiendaService.