**Repositorio Genérico en Memoria (InMemoryRepository<T>)**

La clase InMemoryRepository<T> es un repositorio genérico en memoria que permite gestionar entidades de cualquier tipo T. Esta clase proporciona métodos para guardar, buscar, actualizar y eliminar entidades, así como para encontrar entidades por campos específicos.

Para que este repositorio funcione de manera robusta, necesitamos que las entidades que gestiona (T) tengan una forma de ser identificadas. Para ello, introduciremos una interfaz Identifiable.

**1. Interfaz Identifiable**

Dado que el repositorio necesita asignar y acceder a un ID para cada entidad, es fundamental que el tipo genérico T que se utilice con InMemoryRepository implemente una interfaz que garantice la existencia de métodos getId() y setId(). Esto mejora la seguridad de tipos y la claridad del contrato.

Java

// Archivo: Identifiable.java

public interface Identifiable {

Long getId();

void setId(Long id);

}

**2. Clase InMemoryRepository<T>**

Ahora, la clase del repositorio. Hemos modificado ligeramente los métodos save y genericUpdate para usar la interfaz Identifiable, haciendo el código más seguro.

Java

// Archivo: InMemoryRepository.java

import java.lang.reflect.Method; // Necesario para la reflexión

import java.util.ArrayList; // Para la lista de resultados

import java.util.HashMap; // Para el almacenamiento de datos

import java.util.List; // Interfaz de lista

import java.util.Map; // Interfaz de mapa

import java.util.Optional; // Para manejar la posible ausencia de un resultado

import java.util.concurrent.atomic.AtomicLong; // Para generar IDs únicos

/\*\*

\* Repositorio genérico en memoria que simula operaciones CRUD.

\* La entidad T debe implementar la interfaz Identifiable.

\*

\* @param <T> El tipo de entidad que este repositorio gestionará, debe ser Identifiable.

\*/

public class InMemoryRepository<T extends Identifiable> { // T ahora debe ser Identifiable

// Almacena las entidades: clave (ID) -> valor (entidad T)

protected Map<Long, T> data = new HashMap<>();

// Generador atómico para IDs únicos y seguros en entornos concurrentes

protected AtomicLong idGenerator = new AtomicLong();

/\*\*

\* Guarda una nueva entidad en el repositorio.

\* Genera un nuevo ID, lo asigna a la entidad y la almacena.

\*

\* @param entity La entidad a guardar.

\* @return La entidad guardada con su ID asignado.

\*/

public T save(T entity) {

long id = idGenerator.incrementAndGet(); // Genera un nuevo ID único

entity.setId(id); // Asigna el ID a la entidad usando la interfaz Identifiable

// Muestra en consola la clase y el ID asignado (opcional, para depuración)

System.out.println("Guardado: " + entity.getClass().getSimpleName() + " con ID: " + id);

data.put(id, entity); // Almacena la entidad en el mapa

return entity;

}

/\*\*

\* Busca una entidad por su ID.

\*

\* @param id El ID de la entidad a buscar.

\* @return Un Optional que contiene la entidad si se encuentra, o Optional.empty() si no.

\*/

public Optional<T> findById(Long id) {

// Optional.ofNullable maneja si data.get(id) devuelve null

return Optional.ofNullable(data.get(id));

}

/\*\*

\* Retorna una lista con todas las entidades almacenadas en el repositorio.

\*

\* @return Una List con todas las entidades.

\*/

public List<T> findAll() {

// Crea una nueva ArrayList a partir de los valores del mapa para evitar mutaciones directas

return new ArrayList<>(data.values());

}

/\*\*

\* Actualiza una entidad existente en el repositorio.

\* El ID de la entidad a actualizar se pasa explícitamente.

\* La entidad actualizada debe tener el mismo ID o se le asignará el ID de la búsqueda.

\*

\* @param id El ID de la entidad a actualizar.

\* @param updatedEntity La entidad con los datos actualizados.

\* @return Un Optional que contiene la entidad actualizada si el ID existe, o Optional.empty() si no.

\*/

public Optional<T> genericUpdate(Long id, T updatedEntity) {

// Verifica si el ID existe en el repositorio

if (!data.containsKey(id)) {

System.out.println("Actualización fallida: ID " + id + " no encontrado.");

return Optional.empty();

}

// Asigna el ID de la entidad existente a la entidad actualizada para coherencia

updatedEntity.setId(id);

data.put(id, updatedEntity); // Sobrescribe la entidad existente con el mismo ID

System.out.println("Actualizado: " + updatedEntity.getClass().getSimpleName() + " con ID: " + id);

return Optional.of(updatedEntity);

}

/\*\*

\* Elimina una entidad del repositorio por su ID.

\*

\* @param id El ID de la entidad a eliminar.

\* @return Un Optional que contiene la entidad eliminada si se encontró, o Optional.empty() si no.

\*/

public Optional<T> genericDelete(Long id) {

// Verifica si el ID existe antes de intentar eliminar

if (!data.containsKey(id)) {

System.out.println("Eliminación fallida: ID " + id + " no encontrado.");

return Optional.empty();

}

// data.remove(id) devuelve el valor asociado a la clave eliminada

System.out.println("Eliminado el elemento con ID: " + id);

return Optional.ofNullable(data.remove(id));

}

/\*\*

\* Busca entidades en el repositorio por el valor de un campo específico.

\* Utiliza reflexión para acceder al método getter del campo.

\*

\* @param fieldName El nombre del campo por el que buscar (ej. "nombre", "edad").

\* @param value El valor a comparar en el campo especificado.

\* @return Una List de entidades que coinciden con el criterio de búsqueda.

\*/

public List<T> genericFindByField(String fieldName, Object value) {

List<T> results = new ArrayList<>();

try {

// Itera sobre todas las entidades almacenadas

for (T entity : data.values()) {

// Obtiene el método getter para el campo (ej. "getNombre")

Method getFieldMethod = entity.getClass().getMethod("get" + capitalize(fieldName));

// Invoca el getter en la entidad para obtener el valor del campo

Object fieldValue = getFieldMethod.invoke(entity);

// Compara el valor del campo con el valor de búsqueda

if (fieldValue != null && fieldValue.equals(value)) {

results.add(entity); // Si coinciden, añade la entidad a los resultados

}

}

} catch (Exception e) {

// Imprime la traza de la pila en caso de error de reflexión (ej. getter no encontrado)

System.err.println("Error al buscar por campo '" + fieldName + "': " + e.getMessage());

e.printStackTrace();

}

return results;

}

/\*\*

\* Método auxiliar para capitalizar la primera letra de una cadena.

\* Necesario para construir los nombres de los métodos getter (ej. "name" -> "getName").

\*

\* @param str La cadena a capitalizar.

\* @return La cadena capitalizada.

\*/

private String capitalize(String str) {

if (str == null || str.isEmpty()) {

return str;

}

return str.substring(0, 1).toUpperCase() + str.substring(1);

}

}

**3. Clases de Entidad de Ejemplo**

Para demostrar el InMemoryRepository, crearemos un par de clases de entidad que implementan Identifiable.

**Clase User**

Java

// Archivo: User.java

public class User implements Identifiable {

private Long id;

private String username;

private String email;

public User(String username, String email) {

this.username = username;

this.email = email;

}

// Getters y Setters (implementando Identifiable)

@Override

public Long getId() {

return id;

}

@Override

public void setId(Long id) {

this.id = id;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

@Override

public String toString() {

return "User{id=" + id + ", username='" + username + "', email='" + email + "'}";

}

}

**Clase Product**

Java

// Archivo: Product.java

public class Product implements Identifiable {

private Long id;

private String name;

private double price;

private int stock;

public Product(String name, double price, int stock) {

this.name = name;

this.price = price;

this.stock = stock;

}

// Getters y Setters (implementando Identifiable)

@Override

public Long getId() {

return id;

}

@Override

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

public int getStock() {

return stock;

}

public void setStock(int stock) {

this.stock = stock;

}

@Override

public String toString() {

return "Product{id=" + id + ", name='" + name + "', price=" + price + ", stock=" + stock + "}";

}

}

**4. Clase Main para Demostrar el Uso**

Finalmente, creamos una clase Main para ver cómo interactúa el InMemoryRepository con las entidades User y Product.

Java

// Archivo: Main.java

import java.util.List;

import java.util.Optional;

public class Main {

public static void main(String[] args) {

// --- Demostración con UserRepository ---

System.out.println("--- Repositorio de Usuarios ---");

InMemoryRepository<User> userRepository = new InMemoryRepository<>();

// 1. Save (Crear)

User user1 = userRepository.save(new User("alice", "alice@example.com"));

User user2 = userRepository.save(new User("bob", "bob@example.com"));

User user3 = userRepository.save(new User("charlie", "charlie@example.com"));

// 2. FindAll (Leer todos)

List<User> allUsers = userRepository.findAll();

System.out.println("\nTodos los usuarios:");

allUsers.forEach(System.out::println);

// 3. FindById (Leer por ID)

System.out.println("\nBuscar usuario con ID " + user1.getId() + ":");

Optional<User> foundUser = userRepository.findById(user1.getId());

foundUser.ifPresent(System.out::println);

System.out.println("\nBuscar usuario con ID 99 (no existente):");

Optional<User> notFoundUser = userRepository.findById(99L);

System.out.println("Usuario encontrado: " + notFoundUser.isPresent());

// 4. Update (Actualizar)

System.out.println("\nActualizar usuario " + user2.getId() + ":");

User updatedUser = new User("robert", "robert@newemail.com"); // No asignamos ID, el repositorio lo hará

Optional<User> resultUpdate = userRepository.genericUpdate(user2.getId(), updatedUser);

resultUpdate.ifPresent(u -> System.out.println("Usuario actualizado: " + u));

System.out.println("\nTodos los usuarios después de la actualización:");

userRepository.findAll().forEach(System.out::println);

// 5. Delete (Borrar)

System.out.println("\nEliminar usuario con ID " + user3.getId() + ":");

Optional<User> deletedUser = userRepository.genericDelete(user3.getId());

deletedUser.ifPresent(u -> System.out.println("Usuario eliminado: " + u));

System.out.println("\nTodos los usuarios después de la eliminación:");

userRepository.findAll().forEach(System.out::println);

// 6. FindByField (Buscar por campo)

System.out.println("\nBuscar usuario por username 'alice':");

List<User> usersByUsername = userRepository.genericFindByField("username", "alice");

usersByUsername.forEach(System.out::println);

System.out.println("\nBuscar usuario por email 'robert@newemail.com':");

List<User> usersByEmail = userRepository.genericFindByField("email", "robert@newemail.com");

usersByEmail.forEach(System.out::println);

System.out.println("\n--- Repositorio de Productos ---");

InMemoryRepository<Product> productRepository = new InMemoryRepository<>();

// Guardar productos

Product prod1 = productRepository.save(new Product("Laptop", 1200.00, 50));

Product prod2 = productRepository.save(new Product("Mouse", 25.00, 200));

System.out.println("\nTodos los productos:");

productRepository.findAll().forEach(System.out::println);

// Actualizar un producto

Product updatedProd1 = new Product("Gaming Laptop", 1500.00, 45);

productRepository.genericUpdate(prod1.getId(), updatedProd1);

System.out.println("\nProductos después de la actualización:");

productRepository.findAll().forEach(System.out::println);

// Buscar productos por stock

System.out.println("\nBuscar productos con stock 200:");

List<Product> productsByStock = productRepository.genericFindByField("stock", 200);

productsByStock.forEach(System.out::println);

}

}

**Explicación Detallada de InMemoryRepository<T extends Identifiable>**

**Componentes de la Clase**

1. **protected Map<Long, T> data = new HashMap<>();**
   * **Propósito**: Este Map es el corazón del repositorio. Actúa como una base de datos en memoria.
   * **Mecanismo**: Almacena las entidades utilizando su Long ID como clave y la entidad genérica T como valor. HashMap se elige por su eficiencia O(1) promedio para inserciones, búsquedas y eliminaciones.
   * **Visibilidad (protected)**: Permite que subclases de InMemoryRepository (si las hubiera) puedan acceder directamente a los datos.
2. **protected AtomicLong idGenerator = new AtomicLong();**
   * **Propósito**: Generar identificadores únicos para cada nueva entidad que se guarda.
   * **Mecanismo**: AtomicLong es una clase de la librería java.util.concurrent.atomic que proporciona un long que puede ser modificado de forma atómica. Esto significa que las operaciones como incrementAndGet() son seguras para hilos, garantizando que cada ID generado sea único, incluso si múltiples hilos intentan guardar entidades simultáneamente.

**Métodos de la Clase**

1. **public T save(T entity)**
   * **Función**: Persiste una nueva entidad en el repositorio. Simula la operación "Crear" (Create) en CRUD.
   * **Detalles**:
     + long id = idGenerator.incrementAndGet();: Obtiene un nuevo ID único y lo incrementa para la próxima vez.
     + entity.setId(id);: Asigna el ID generado a la entidad. Esto es posible y seguro gracias a que T ahora extends Identifiable.
     + System.out.println(...): Mensaje de depuración.
     + data.put(id, entity);: Almacena el par ID-entidad en el mapa.
     + return entity;: Devuelve la entidad con su ID ya asignado.
2. **public Optional<T> findById(Long id)**
   * **Función**: Recupera una entidad por su identificador único. Simula la operación "Leer" (Read) por ID en CRUD.
   * **Detalles**:
     + data.get(id): Intenta obtener la entidad del mapa usando el ID. Si el ID no existe, devuelve null.
     + Optional.ofNullable(...): Envuelve el resultado. Optional es una clase que representa un valor que puede estar o no presente. Esto ayuda a evitar NullPointerExceptions y hace el código más expresivo sobre la posible ausencia de un resultado.
3. **public List<T> findAll()**
   * **Función**: Retorna todas las entidades actualmente almacenadas en el repositorio. Simula la operación "Leer todas" (Read All) en CRUD.
   * **Detalles**:
     + data.values(): Obtiene una Collection de todos los valores (entidades) del mapa.
     + new ArrayList<>(...): Crea una nueva ArrayList a partir de esa colección. Esto es importante para devolver una copia, evitando que el código cliente modifique la colección subyacente del repositorio directamente.
4. **public Optional<T> genericUpdate(Long id, T updatedEntity)**
   * **Función**: Actualiza una entidad existente. Simula la operación "Actualizar" (Update) en CRUD.
   * **Detalles**:
     + if (!data.containsKey(id)): Verifica si la entidad con el id dado existe. Si no, devuelve Optional.empty() para indicar que no hay nada que actualizar.
     + updatedEntity.setId(id);: Asegura que la entidad updatedEntity tenga el mismo ID que la entidad que se está actualizando. Esto es crucial para mantener la coherencia y sobrescribir la entrada correcta en el mapa.
     + data.put(id, updatedEntity);: Sobrescribe la entidad antigua con la updatedEntity en la misma clave ID.
     + return Optional.of(updatedEntity);: Devuelve la entidad actualizada envuelta en un Optional.
5. **public Optional<T> genericDelete(Long id)**
   * **Función**: Elimina una entidad del repositorio por su ID. Simula la operación "Borrar" (Delete) en CRUD.
   * **Detalles**:
     + if (!data.containsKey(id)): Verifica la existencia.
     + data.remove(id): Elimina la entrada del mapa. Este método de Map devuelve el valor que fue eliminado (la entidad), o null si la clave no existía.
     + Optional.ofNullable(...): Envuelve el resultado de remove en un Optional.
6. **public List<T> genericFindByField(String fieldName, Object value)**
   * **Función**: Busca entidades basándose en el valor de un campo específico (ej., buscar usuarios por username).
   * **Detalles**:
     + **Reflexión (java.lang.reflect.Method)**: Este método utiliza la **reflexión de Java**, que es la capacidad de un programa de inspeccionar y modificar su propia estructura y comportamiento en tiempo de ejecución.
     + entity.getClass().getMethod("get" + capitalize(fieldName)): Construye el nombre del método getter esperado (ej., si fieldName es "username", busca "getUsername") y obtiene una referencia a ese método.
     + getFieldMethod.invoke(entity): Llama dinámicamente al método getter en la entidad para obtener el valor del campo.
     + if (fieldValue != null && fieldValue.equals(value)): Compara el valor obtenido con el valor de búsqueda proporcionado.
     + try-catch (Exception e): La reflexión puede lanzar varias excepciones (ej., NoSuchMethodException si el getter no existe, IllegalAccessException si no se puede acceder al método, InvocationTargetException si el método invocado lanza su propia excepción). Por eso es crucial un bloque try-catch.
     + **Ventaja**: Permite búsquedas flexibles sin tener que escribir un método findBy por cada campo.
     + **Desventaja**: Es más lento que el acceso directo a métodos y es propenso a errores en tiempo de ejecución si el fieldName no corresponde a un getter existente.
7. **private String capitalize(String str)**
   * **Función**: Método auxiliar para convertir la primera letra de una cadena a mayúscula.
   * **Utilidad**: Se usa en genericFindByField para construir el nombre correcto del método getter (ej., username -> Username para getUsername).

Este InMemoryRepository es un patrón fundamental para entender cómo se construyen sistemas de persistencia y cómo los genéricos y la reflexión pueden usarse para crear código reutilizable y potente en Java.