Xestión de datos con Hibernate 6.4: boas prácticas e optimización

Antonio Varela

Temario

- Introducción a Hibernate
- Configuración e integración con Spring Boot
- Mapeo de Entidades e Relacións
- Persistencia y consultas
- Optimización con EhCache para mellorar o rendemento
- Conclusións e Boas Prácticas

Introducción a Hibernate

¿Qué es la persistencia?

La **persistencia** se refiere al proceso de almacenar datos de una aplicación en un medio no volátil (como una base de datos) para que puedan ser recuperados y utilizados posteriormente.

Objetivo:

- Asegurar que los datos sobrevivan más allá de la ejecución de la aplicación.

Características Principales:

- Almacenamiento de datos estructurados o no estructurados.
- Recuperación eficiente de los datos.
- Consistencia y seguridad de la información.

Opciones de persistencia

Bases de Datos Relacionales (RDBMS):

- **Ejemplos:** MySQL, PostgreSQL, Oracle, SQL Server.
- Organizan los datos en tablas, filas y columnas.
- Usan **SQL** para manipulación de datos.
- Ideal para datos estructurados y relaciones complejas.

Opciones de persistencia

Bases de Datos No Relacionales (NoSQL):

- **Ejemplos:** MongoDB, Cassandra, Redis, Elasticsearch.
- Diseñadas para manejar datos no estructurados, semiestructurados o grandes volúmenes de datos.
- Modelos flexibles como documentos, clave-valor, gráficos, etc.

Opciones de persistencia

Persistencia en Memoria:

- **Ejemplos:** Redis, H2 (embedded database).
- Almacena datos temporalmente en la RAM.
- Utilizado para almacenamiento rápido o pruebas.

ORM (Object Relational Mapping):

- **Ejemplos:** Hibernate, JPA (Java Persistence API).
- Facilita la persistencia al mapear objetos Java a tablas de bases de datos.

Formas de interactuar con bases de datos en Java

ODBC (Open Database Connectivity)

- Es una API estándar diseñada para permitir que las aplicaciones se conecten a diferentes bases de datos, independientemente del sistema de gestión de bases de datos (DBMS) utilizado. Es una solución de bajo nivel para acceso a bases de datos.

Características principales:

- Usa controladores (*drivers*) específicos para cada tipo de base de datos.
- La comunicación se realiza mediante comandos SQL enviados directamente al DBMS.
- Muy flexible, pero implica un manejo manual de conexiones, consultas y resultados.

Formas de interactuar con bases de datos en Java

ODBC (Open Database Connectivity)

Ventajas:

- Interoperabilidad: permite conectar con cualquier base de datos que tenga un controlador ODBC.
- Es una tecnología estándar y ampliamente soportada.

Desventajas:

- Es un enfoque muy “manual” y verboso.
- Los desarrolladores deben escribir y gestionar todo el código SQL, conexiones y errores de forma explícita.

Formas de interactuar con bases de datos en Java

DAO (Data Access Object)

Es un patrón de diseño que organiza el acceso a datos en una aplicación. Aunque no es una tecnología específica, define una forma estructurada de interactuar con la base de datos utilizando métodos específicos para cada operación (e.g., save(), findById()).

Características principales:

- Proporciona una capa de abstracción para separar la lógica de acceso a datos de la lógica de negocio.
- Los métodos dentro del DAO encapsulan la lógica de consultas SQL.
- Sigue siendo dependiente de tecnologías como JDBC para implementar las operaciones.

Formas de interactuar con bases de datos en Java

DAO (Data Access Object)

Ventajas:

- Organización: el código relacionado con la base de datos está centralizado en una capa específica.
- Reutilización: los métodos del DAO se pueden usar en varias partes de la aplicación.
- Reduce el acoplamiento entre la base de datos y la lógica de negocio.

Desventajas:

- Sigue siendo necesario escribir consultas SQL manualmente.
- Puede volverse engorroso para manejar relaciones complejas o grandes cantidades de datos.

Formas de interactuar con bases de datos en Java

ORM (Object-Relational Mapping)

- Es un enfoque (y a menudo un conjunto de herramientas o bibliotecas) para mapear las estructuras de una base de datos relacional (tablas, columnas, relaciones) a objetos del lenguaje de programación (en este caso, Java).

Características principales:

- Los ORM (como Hibernate) automatizan la traducción entre objetos Java y tablas de base de datos.
- Usan anotaciones o archivos de configuración para definir cómo se deben mapear las entidades y sus relaciones.
- Ofrecen un lenguaje de consulta propio, como HQL (Hibernate Query Language), además de soporte para SQL.

Formas de interactuar con bases de datos en Java

ORM (Object-Relational Mapping)

Ventajas:

- Simplifica el desarrollo al evitar el manejo manual de SQL.
- Maneja automáticamente relaciones complejas entre entidades.
- Soporte para caching, optimización y transacciones integradas.

Desventajas:

- Curva de aprendizaje más alta.
- Puede ser ineficiente si no se configura correctamente (e.g., *n+1 query problem*).

Formas de interactuar con bases de datos en Java

ODBC vs DAO vs ORM

Aspecto	ODBC	DAO	ORM
Nivel de abstracción	Bajo	Medio	Alto
Facilidad de uso	Complejo y manual	Moderado	Fácil (una vez configurado)
Automatización	Ninguna	Parcial (sólo métodos)	Alta (mapeo y consultas)
Relaciones complejas	Difícil de manejar	Posible pero tedioso	Sencillo y eficiente

Hibernate y JPA

- Hibernate es un **framework de mapeo objeto-relacional** (ORM) en Java.
- Su objetivo principal es simplificar la interacción entre una aplicación Java y una base de datos relacional, permitiendo mapear tablas y columnas a clases y atributos Java, respectivamente.

Características principales:

- Traduce automáticamente consultas en HQL (Hibernate Query Language) o métodos en el código a SQL, y viceversa.
- Proporciona herramientas avanzadas como el manejo de relaciones entre entidades, optimización de consultas y soporte para caché.
- Es uno de los ORM más populares en el ecosistema Java, gracias a su flexibilidad y capacidad para manejar casos de uso complejos.

Hibernate y JPA

Ventajas de Hibernate:

1. Reducción del código manual relacionado con SQL y JDBC.
2. Manejo automático de transacciones y relaciones entre entidades.
3. Independencia de base de datos: puedes cambiar de un sistema de base de datos a otro sin cambiar tu código.
4. Soporte para estrategias avanzadas como *lazy loading*, *eager loading* y caché de segundo nivel (EhCache, Redis, etc.).

Hibernate y JPA

Ejemplo simple:

- En lugar de escribir SQL manual para insertar datos:

```
INSERT INTO Author (id, name) VALUES (1, 'John Doe');
```

- Con Hibernate, simplemente creas una instancia de la clase Author y la guardas:

```
Author author = new Author();
author.setName("John Doe");
session.save(author);
```

Hibernate y JPA

- **JPA (Java Persistence API)** es una especificación de Java para **persistencia de datos**.
- Define un conjunto de interfaces y reglas estándar que cualquier framework ORM (como Hibernate) debe implementar para interactuar con bases de datos relacionales.
- **JPA NO es un framework**, sino una API estándar que los frameworks como Hibernate, EclipseLink o OpenJPA implementan.

Hibernate y JPA

Características principales:

- Define un modelo común para mapear objetos a tablas relacionales (ORM).
- Incluye anotaciones estándar (@Entity, @Table, @Id, etc.) que permiten definir las entidades y sus relaciones.
- Proporciona un lenguaje de consulta propio: JPQL (Java Persistence Query Language), muy similar a SQL pero orientado a objetos.

Ventajas de usar JPA:

- **Portabilidad:** Puedes cambiar entre implementaciones (e.g., de Hibernate a EclipseLink) sin modificar el código base, ya que JPA es un estándar.
- **Facilidad de uso:** Reduce la cantidad de código necesario para trabajar con bases de datos.
- **Integración con otros frameworks:** Es compatible con tecnologías como Spring y Jakarta EE.

Hibernate y JPA

Relación entre Hibernate y JPA

- Hibernate es una **implementación de JPA**. Esto significa que cumple con todas las reglas definidas por la especificación JPA, pero además incluye características adicionales que van más allá de la especificación estándar.
- **Usar Hibernate con JPA:** Si sigues los estándares de JPA (anotaciones y configuraciones), puedes cambiar a otro proveedor en el futuro (por ejemplo, EclipseLink), lo que aumenta la flexibilidad de tu aplicación.
- **Usar características exclusivas de Hibernate:** Si necesitas algo más avanzado que JPA no define (por ejemplo, un manejo específico de caché o estrategias de carga personalizada), puedes usar las APIs propias de Hibernate.

Hibernate y JPA

Comparativa de Hibernate y JPA

Aspecto	JPA	Hibernate
¿Qué es?	Una especificación (API estándar)	Un framework (implementación de JPA)
Dependencia de Java EE?	Sí (parte de Jakarta EE)	No
Anotaciones/API	Estándar (`@Entity`, `@Table`)	Estándar + Propias (`@Cache`, etc.)
Soporte de JPQL	Sí	Sí + HQL (lenguaje propio de consultas más avanzado)
Flexibilidad	Limitada a la especificación	Mayor, gracias a características avanzadas.

Cronología de Hibernate y JPA

2001 - Lanzamiento inicial de Hibernate

- Hibernate fue creado como un proyecto de código abierto por Gavin King en el año 2001.
- **Propósito inicial:** Simplificar la interacción entre aplicaciones Java y bases de datos relacionales.
- Destacó rápidamente por ser más eficiente y flexible que el acceso directo con JDBC.

2003 - Hibernate 2.0

- Introducción de características importantes:
  - Lenguaje HQL (Hibernate Query Language), un lenguaje de consultas orientado a objetos.
  - Mejoras en el mapeo de relaciones entre entidades.

Cronología de Hibernate y JPA

2006 - JPA 1.0 (Java Persistence API)

- **Primera versión de JPA, introducida como parte de Java EE 5.**
- Nace como un esfuerzo liderado por Sun Microsystems para unificar y regular el manejo de ORM en el ecosistema Java
- Hibernate se convierte en una de las implementaciones principales de JPA 1.0, junto con EclipseLink (anteriormente TopLink) y OpenJPA.
- Introducción de anotaciones estándar como @Entity, @Table, @Id.

Cronología de Hibernate y JPA

2009 - Hibernate 3.x

- Soporte completo para la especificación JPA 1.0, además de características propias avanzadas.
- Incorporación de **caché de segundo nivel**, que mejora significativamente el rendimiento.
- Extensiones para manejar bases de datos no relacionales y consultas avanzadas.

Cronología de Hibernate y JPA

2011 - JPA 2.0 (Java EE 6)

- Introducción de importantes mejoras en JPA:
  - **Criteria API:** Permite construir consultas dinámicamente en código Java sin usar cadenas de texto SQL o JPQL.
  - **Mapeo avanzado de relaciones** (e.g., herencia, claves compuestas).
  - Mayor integración con **Bean Validation** para validar entidades.
- Hibernate se adapta rápidamente para implementar estas nuevas características en su versión **Hibernate 4.x**.

Cronología de Hibernate y JPA

2014 - JPA 2.1 (Java EE 7)

- Nuevas funcionalidades, como:
  - **Consultas almacenadas (Stored Procedures):** Soporte integrado para ejecutarlas desde el ORM.
  - **Conversores personalizados:** Permiten transformar atributos entre la base de datos y Java de manera más flexible.
- Hibernate sigue siendo una de las implementaciones más populares con su versión **5.x**, integrando JPA 2.1.

Cronología de Hibernate y JPA

2019 - JPA 2.2 (Jakarta EE 8)

- Cambios menores para mejorar la compatibilidad con Java 8:
  - Soporte para las clases de fecha y hora de Java (LocalDate, LocalTime, LocalDateTime).
  - **Hibernate 5.3** introduce compatibilidad completa con JPA 2.2 y comienza a optimizar su soporte para entornos modernos como **microservicios**.

2020 - Renombramiento de JPA bajo Jakarta EE

- Con la transición de Java EE a **Jakarta EE** (liderado por Eclipse Foundation), la API de persistencia pasa a llamarse **Jakarta Persistence**.
- Hibernate continúa siendo compatible con la especificación, adaptándose al cambio de nombre.

Cronología de Hibernate y JPA

2022 - Hibernate 6.0

- **Rediseño significativo**: Se mejoró la arquitectura interna para optimizar el rendimiento y la consistencia.
- **Actualizaciones en HQL**: Se incorporaron nuevas construcciones para alinearse con las capacidades de los dialectos SQL modernos.
- **Validación mejorada**: Se implementó una validación más estricta de las consultas antes de acceder a la base de datos, mejorando la detección de errores.
- **Mejoras en la API**: Se incrementó la seguridad de tipos en las anotaciones de mapeo y se clarificó la separación entre API, SPI e implementación interna.

Cronología de Hibernate y JPA

Marzo 2023 - Hibernate 6.2

- **Compatibilidad con Jakarta Persistence 3.1**: Se aseguró la alineación con la especificación más reciente de Jakarta Persistence.
- **Soporte para registros (records)**: Se añadió soporte para los registros de Java, permitiendo una integración más fluida con las características modernas del lenguaje.
- **Manejo de estructuras (structs)**: Se mejoró el soporte para tipos de datos estructurados, facilitando la interacción con tipos de datos complejos en la base de datos.
- **Generación de valores**: Se optimizó la generación automática de valores para campos específicos, mejorando la eficiencia en la persistencia de datos.
- **Particionamiento y comando SQL** MERGE: Se incorporó soporte para estrategias de particionamiento y para el comando MERGE de SQL, ampliando las opciones de manipulación de datos.

Cronología de Hibernate y JPA

Noviembre 2023 - Hibernate 6.4

- **Eliminación lógica (Soft Delete)**: Se añadió la anotación @SoftDelete para facilitar la implementación de eliminaciones lógicas en las entidades.
- **Funciones de colecciones en HQL/Criteria**: Se incorporaron funciones para el manejo avanzado de **colecciones** en consultas HQL y Criteria, ampliando las capacidades de consulta.
- **Identificadores de inquilino (Tenant id) no basados en cadenas**: Se permitió el uso de tipos de datos distintos a String para los identificadores de inquilino en configuraciones multitenant.
- **Soporte para Java Flight Recorder (JFR)**: Se añadió integración con JFR para monitorear y analizar el rendimiento de las aplicaciones que utilizan Hibernate.

Cronología de Hibernate y JPA

Agosto 2024 - Hibernate 6.6

- **Integración con Jakarta Data**: Se completó la integración con Jakarta Data, ofreciendo nuevas capacidades para el manejo y procesamiento de datos.
- **Anotación** @ConcreteProxy: Se introdujo esta anotación para mejorar la gestión de proxies concretos en las entidades, optimizando el rendimiento y la coherencia.
- **Soporte extendido para colecciones**: Se amplió la compatibilidad con tipos de colecciones, facilitando su uso en diversas bases de datos y escenarios.
- **Herencia en embebidos**: Se añadió soporte para herencia en tipos embebidos, permitiendo modelos de datos más flexibles y reutilizables.

ORMs Alternativos a Hibernate y Comparativa

EclipseLink

- **EclipseLink** es uno de los **referentes de JPA** en el ecosistema de Java.
- **Características clave**:
  - Implementación completa de **JPA**.
  - Soporte para mapeo de **tipos avanzados** (por ejemplo, **XML** o **NoSQL**).
  - Mejor rendimiento en **consultas complejas** en algunos casos.

ORMs Alternativos a Hibernate y Comparativa

EclipseLink

- **Ventajas sobre Hibernate**:
  - **Más ligero** en aplicaciones que ya usan **JPA** como estándar.
  - Mejor integración con otras tecnologías de **Oracle** (como **Oracle Database**).

- **Limitaciones**:
  - Menos popular y documentado que Hibernate.
  - Comunidad más pequeña en comparación.

ORMs Alternativos a Hibernate y Comparativa

MyBatis

- **MyBatis** es un framework que permite la persistencia de objetos mediante SQL, en lugar de usar mapeo automático.
- **Características clave**:
  - En **MyBatis**, el desarrollador escribe las consultas SQL directamente.
  - Ofrece mayor **control** sobre el rendimiento y las consultas.

ORMs Alternativos a Hibernate y Comparativa

MyBatis

- **Ventajas sobre Hibernate**:
  - **Control total sobre SQL**: El desarrollador puede optimizar manualmente las consultas.
  - Ideal para aplicaciones donde las consultas complejas y personalizadas son necesarias.

- **Limitaciones**:
  - Menos abstracción que Hibernate. El desarrollador debe gestionar manualmente el SQL.
  - No soporta completamente las capacidades de **JPA** (como el manejo de entidades).

ORMs y NoSQL?

Desafíos de Usar ORMs con Bases de Datos NoSQL

1. **Incompatibilidad de Modelo**:
   - Los **ORMs tradicionales** como Hibernate están diseñados para bases de datos **relacionales**
2. **Escalabilidad y Flexibilidad**:
   - Las bases de datos NoSQL están diseñadas para ser **altamente escalables**
3. **Falta de Soporte de Transacciones ACID**:
   - Muchas bases de datos NoSQL no implementan transacciones **ACID**
4. **Consultas y Modelado de Datos**:
   - Los **ORMs** son excelentes para generar consultas a bases de datos SQL y crear relaciones entre tablas

ORMs y NoSQL?

Soluciones y Herramientas ORM para NoSQL

1. **MongoDB y Spring Data MongoDB**
   - **Spring Data MongoDB** es una extensión de **Spring Data** que proporciona una interfaz fácil de usar para interactuar con MongoDB desde aplicaciones Java
2. **Morphia (MongoDB ORM)**
   - **Morphia** es otro **ORM** para **MongoDB**, que proporciona una forma sencilla de mapear **objetos Java** a documentos de MongoDB.
3. **Cassandra y Spring Data Cassandra**
   - **Spring Data Cassandra** es parte de **Spring Data** y permite la interacción con **Cassandra**, una base de datos NoSQL orientada a columnas.

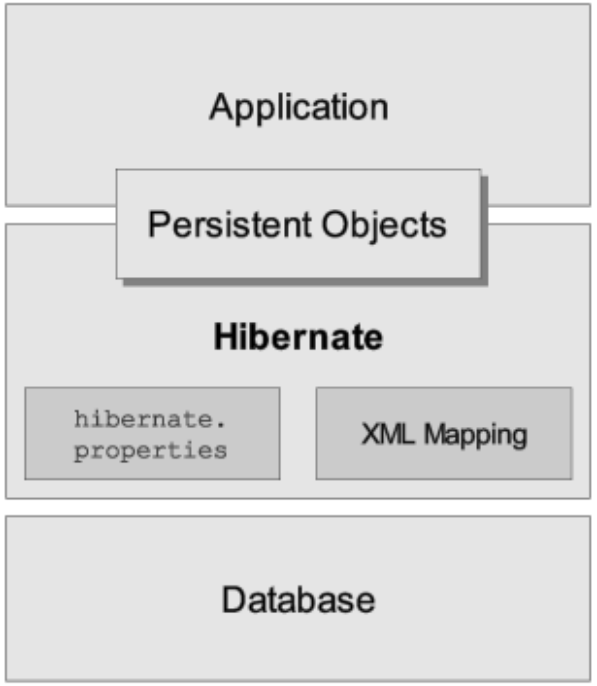
ORMs y NoSQL?

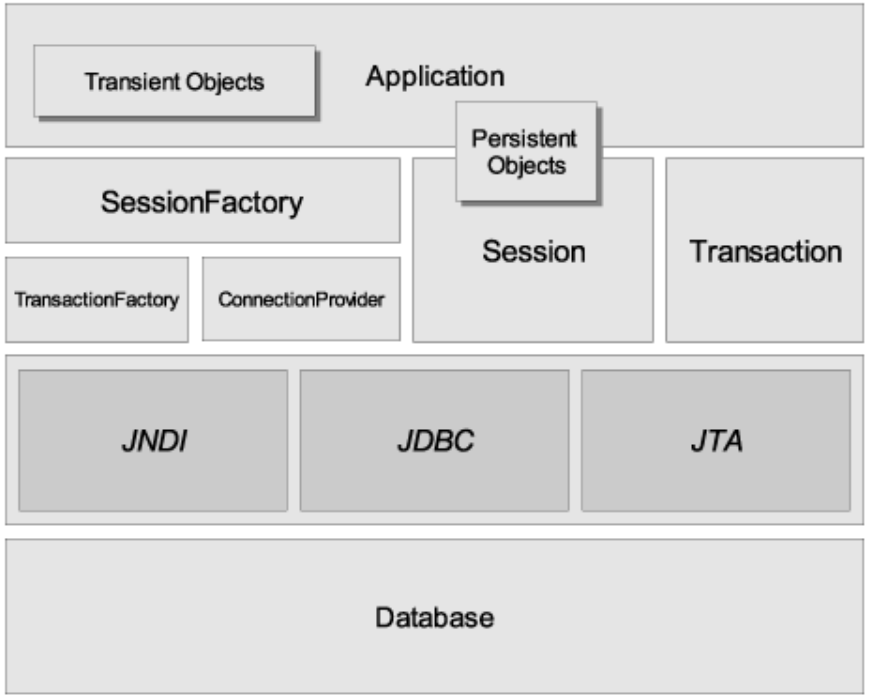
Soluciones y Herramientas ORM para NoSQL

4. **ObjectDB (Base de Datos Orientada a Objetos)**
   - **ObjectDB** es una base de datos orientada a objetos que puede actuar como un ORM propio para aplicaciones Java
5. **JPA y NoSQL (Integración con Hibernate OGM)**
   - **Hibernate OGM (Object/Grid Mapper)** extiende el uso de **JPA** para bases de datos NoSQL, permitiendo el uso de JPA con **MongoDB**, **Cassandra**, **EhCache**, **Infinispan**, entre otras.

Arquitectura de Hibernate

La arquitectura de Hibernate está diseñada para ofrecer **flexibilidad** y **rendimiento**, al mismo tiempo que abstrae las complejidades del acceso a la base de datos.





Arquitectura de Hibernate

- **SessionFactory**
  - Es el **punto de entrada principal** a la infraestructura de Hibernate.
  - **Responsabilidad**: Crear **Session** y gestionar la configuración de Hibernate.
  - Se configura al principio y es utilizado durante toda la vida de la aplicación.
  - **Ejemplo**: sessionFactory = new Configuration().configure().buildSessionFactory();

- **Session**
  - Representa una **sesión de trabajo** con la base de datos.
  - **Responsabilidad**: Interactuar con la base de datos, gestionando entidades, realizando operaciones CRUD.
  - Una **Session** está asociada con una **transacción** y proporciona el contexto en el que las entidades se cargan, se persisten y se actualizan.

Arquitectura de Hibernate

- **Transaction**
  - Una **transacción** agrupa operaciones de persistencia en una unidad atómica.
  - Hibernate gestiona las transacciones y permite un **control explícito** de las mismas.
  - **Responsabilidad**: Controlar el ciclo de vida de una transacción, desde el begin hasta el commit o rollback.

- **Query**
  - **Consulta de datos**: Hibernate proporciona **HQL** (Hibernate Query Language) y **Criteria API** para consultar las entidades.
  - **Responsabilidad**: Realizar consultas a la base de datos y mapear los resultados a objetos Java.

Ciclo de vida de una Entidad en Hibernate

Estado Transitorio:

- La entidad es **nueva** y no está asociada con ninguna sesión de Hibernate ni con la base de datos.
- No existe en la base de datos aún.

Estado Persistente:

- La entidad es **gestionada por Hibernate**.
- Está asociada a una sesión y se puede almacenar, modificar o eliminar en la base de datos.

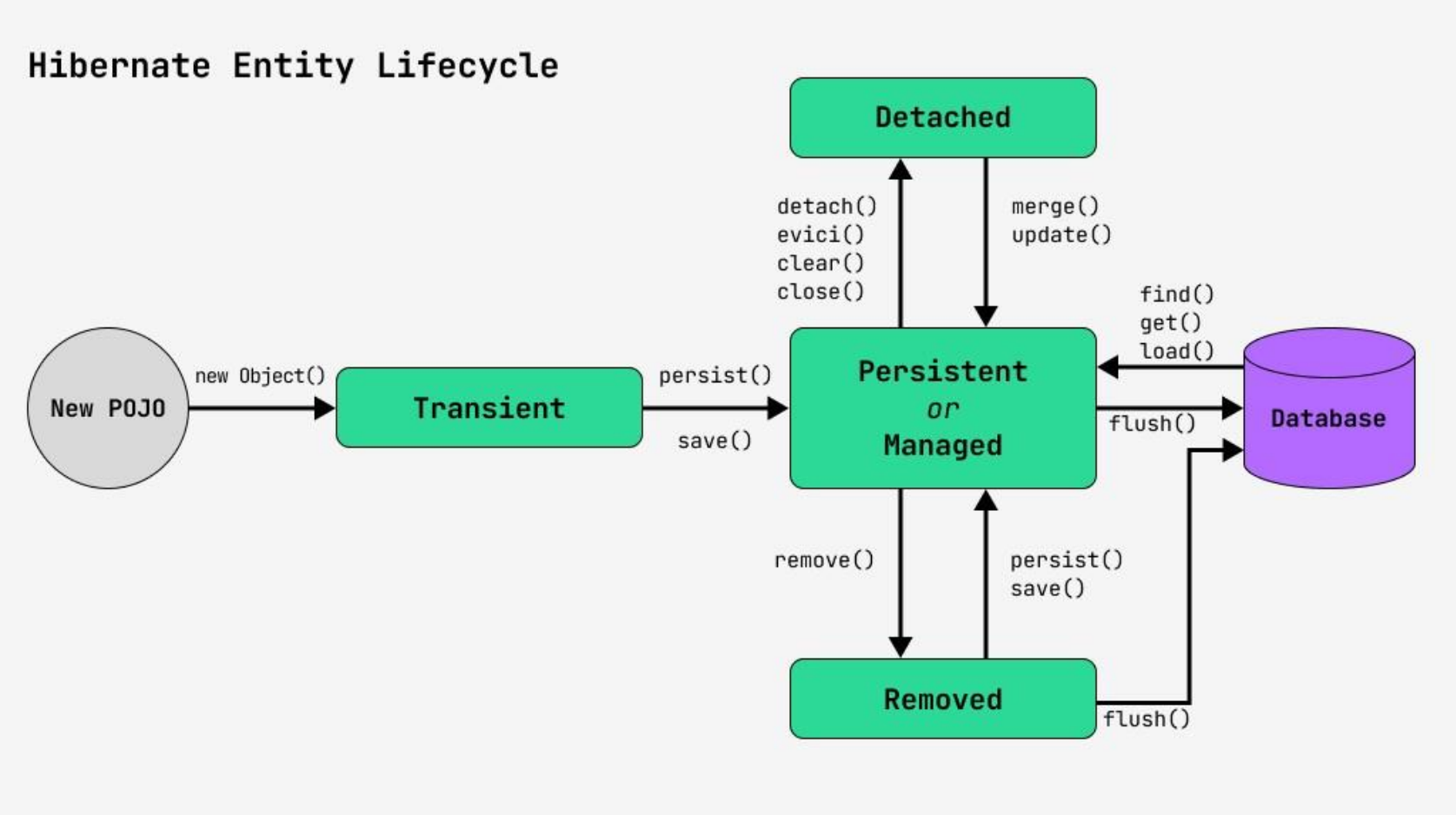
Estado Desasociado (Detached):

- La entidad ha sido **desvinculada de la sesión**, pero sigue existiendo en la base de datos.
- Puede volverse persistente de nuevo si se vuelve a asociar con una sesión.

Estado Eliminado:

- La entidad ha sido **eliminada de la base de datos** y está marcada como eliminada en Hibernate.

Ciclo de vida de una Entidad en Hibernate



Arquitectura de Hibernate

- **Transaction**
  - Una **transacción** agrupa operaciones de persistencia en una unidad atómica.
  - Hibernate gestiona las transacciones y permite un **control explícito** de las mismas.
  - **Responsabilidad**: Controlar el ciclo de vida de una transacción, desde el begin hasta el commit o rollback.

- **Query**
  - **Consulta de datos**: Hibernate proporciona **HQL** (Hibernate Query Language) y **Criteria API** para consultar las entidades.
  - **Responsabilidad**: Realizar consultas a la base de datos y mapear los resultados a objetos Java.

¿Cómo Hibernate Gestiona las Entidades?

- **Mapeo entre Objetos y Tablas**: Hibernate utiliza **anotaciones** (por ejemplo, @Entity, @Table, @Id) o archivos de mapeo XML para definir cómo se deben mapear las clases Java a las tablas en la base de datos.

```java
@Entity
@Table(name = "persona")
public class Persona {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;
    private String apellido;
}
```

- Hibernate mantiene el estado de las entidades a lo largo de las operaciones, utilizando **caché de primer nivel** (por sesión) y **caché de segundo nivel** (opcional, más allá de una sesión).

Integración con JPA

- **JPA (Java Persistence API)** es un conjunto de especificaciones para la persistencia en Java, y Hibernate es una de las implementaciones más populares.
- **Session** de Hibernate se puede mapear a **EntityManager** en JPA.
  - **EntityManager**: Es la interfaz estándar de JPA para manejar las entidades y las transacciones.

```
@PersistenceContext
private EntityManager entityManager;
```

- Hibernate actúa como la implementación detrás de JPA, gestionando las operaciones de persistencia.

Flujo de trabajo de Hibernate

- 1. **Configuración inicial**: Se configura **SessionFactory** usando el archivo hibernate.cfg.xml o anotaciones.
- 2. **Obtener una sesión**: A través de **SessionFactory**.
- 3. **Transacción**: Comienza una transacción y realiza las operaciones de persistencia.
- 4. **Operaciones CRUD**: Se crean, leen, actualizan y eliminan entidades.
- 5. **Cerrar la sesión**: Al finalizar las operaciones, la sesión se cierra.

Resumen

- Hibernate ofrece una **arquitectura modular** con componentes específicos para la gestión de sesiones, transacciones y consultas.
- **SessionFactory** y **Session** son los componentes clave que permiten la interacción con la base de datos.
- **JPA** ofrece una interfaz estándar que Hibernate implementa para mantener la portabilidad y flexibilidad.
- La **configuración** y el **ciclo de vida de las entidades** son fundamentales para entender cómo Hibernate maneja la persistencia y la interacción con la base de datos.

Configuración e integración con Spring Boot

Antonio Varela

Introducción a Spring y Spring Boot

¿Qué es Spring?

- **Spring Framework** es un framework para construir aplicaciones Java de manera modular, escalable y fácilmente configurable.
- **Principales características**:
  - Inyección de dependencias (**IoC**, Inversion of Control).
  - Soporte para programación orientada a aspectos (**AOP**).
  - Un enfoque basado en **POJOs (Plain Old Java Objects)**, lo que permite construir aplicaciones más simples y flexibles.
  - Ecosistema amplio: Spring Web, Spring Security, Spring Data, Spring Cloud, entre otros.

Introducción a Spring y Spring Boot

¿Qué es Spring Boot?

- **Spring Boot** es una extensión de Spring Framework que simplifica el desarrollo de aplicaciones al:
  - Proporcionar **configuración automática** (autoconfiguration).
  - Incluir un **servidor embebido** (como Tomcat o Jetty), eliminando la necesidad de configurarlo manualmente.
  - Ofrecer **starter dependencies**: Librerías preconfiguradas para diferentes casos de uso (Spring Boot Starter Web, Spring Boot Starter Data JPA, etc.).
  - Facilitar el **despliegue rápido** de aplicaciones mediante un único archivo JAR o WAR.

- **Spring Boot** utiliza **Spring Framework** como núcleo, pero simplifica el proceso de configuración y despliegue.

Spring Data JPA

- **Spring Data JPA** es un subproyecto de **Spring Data** que facilita el acceso a bases de datos relacionales usando **JPA (Java Persistence API)**.
- Permite escribir menos código y delegar tareas comunes, como operaciones CRUD, consultas, y más.

Principales características de Spring Data JPA:

1. **Repositorios predefinidos**:
   - Interfaz como JpaRepository que ofrece métodos como save, findById, findAll, delete, etc.
2. **Consultas personalizadas**:
   - Definición automática de consultas a partir de nombres de métodos, como findByNombre(String nombre).
3. **Soporte para paginación y ordenación**.
4. **Integración con Hibernate** como implementación por defecto de JPA.

Relación entre Spring Boot JPA y Hibernate

- **Spring Boot** actúa como el entorno que gestiona las configuraciones y facilita la integración de diferentes tecnologías.
- **JPA** es el estándar de Java para la persistencia de datos, que define una interfaz común para interactuar con bases de datos relacionales.
- **Hibernate** es una implementación de JPA, que Spring Boot utiliza por defecto para manejar la persistencia.

Flujo de trabajo:

- El desarrollador define **entidades** y usa interfaces de repositorio proporcionadas por Spring Data JPA.
- Spring Boot configura automáticamente Hibernate como la implementación de JPA.
- Hibernate traduce las operaciones de repositorio en SQL y gestiona las interacciones con la base de datos.

Ventajas de la Integración con Spring Boot

1. Configuración Automática:

- Spring Boot detecta automáticamente las dependencias de Hibernate y JPA, configurando:
  - Datasource (conexión a la base de datos) ; Dialecto de la base de datos; Transacciones y sesiones.

2. Starter Dependencies:

- Spring Boot ofrece el spring-boot-starter-data-jpa, que incluye todo lo necesario para trabajar con JPA y Hibernate.

3. Reducción del Código Boilerplate:

- Al usar Spring Data JPA, el código para las operaciones comunes de persistencia (CRUD) se reduce drásticamente.

4. Facilidad de Integración:

- Hibernate se integra perfectamente con el ecosistema de Spring (transacciones gestionadas por Spring, caché, etc.).

Configuración Básica de Spring Boot con Hibernate

- Para integrar Hibernate con Spring Boot, solo necesitas incluir el starter correspondiente y el driver del SGBD que quieras usar.
  - En este ejemplo usamos Maven como gestor de dependencias y MySql como SGBD

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
</dependency>
```

Configuración Básica de Spring Boot con Hibernate

- Configura los detalles de conexión a MySQL en el archivo src/main/resources/application.properties

```
# Configuración de la base de datos
spring.datasource.url=jdbc:mysql://localhost:3306/nombre_base_datos
spring.datasource.username=tu_usuario
spring.datasource.password=tu_contraseña

# Configuración de JPA
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

Configuración Básica de Spring Boot con Hibernate

- **spring.datasource.url:** La URL de conexión incluye el host, puerto y el nombre de la base de datos (en este caso, nombre_base_datos).
  - Ejemplo: jdbc:mysql://localhost:3306/mi_base_de_datos.
- **spring.datasource.username y spring.datasource.password:** Usuario y contraseña de la base de datos MySQL.
- **spring.jpa.hibernate.ddl-auto:** Controla cómo Hibernate maneja la estructura de la base de datos.
  - **update**: Actualiza la estructura existente.
  - **create**: Crea la base de datos desde cero en cada inicio.
  - **validate**: Solo valida la estructura sin modificarla.
- **spring.jpa.show-sql:** Muestra las consultas SQL ejecutadas en la consola.
- **spring.jpa.properties.hibernate.dialect:** Indica el dialecto específico para MySQL.

Mapeo de Entidades e Relacións

Introducción al Mapeo de Entidades y Relaciones

¿Qué es el mapeo de entidades?

- Proceso de asociar clases y sus atributos en Java con tablas y columnas en la base de datos.
- Permite interactuar con datos de manera orientada a objetos.

¿Qué es el mapeo de relaciones?

- Define cómo se asocian las entidades entre sí (uno a uno, uno a muchos, muchos a muchos).
- Se traduce en claves foráneas y tablas de relación en la base de datos.

Objetivo:

- Garantizar un acceso eficiente y correcto a los datos respetando las relaciones de negocio.

Opciones Básicas de Configuración de Entidades

@Entity

- Marca una clase como entidad gestionada por JPA/Hibernate.

```
@Entity
public class Persona { ... }
```

Opciones Básicas de Configuración de Entidades

- Una **entidad** es una clase de Java que representa una **tabla en la base de datos**.
- Cada instancia de esta clase corresponde a una **fila de la tabla**.
- Hibernate utiliza estas entidades para mapear automáticamente los datos de la base de datos a objetos en Java y viceversa.

Definición oficial (JPA):

- Una entidad es un objeto persistente cuya existencia está gestionada por un **contexto de persistencia**. Está anotada con @Entity y representa un modelo del dominio de negocio en la aplicación.

Opciones Básicas de Configuración de Entidades

Características y Propiedades de una Entidad

- **Debe estar anotada con** @Entity
- **Debe tener una clave primaria (**@Id**)**
  - Define el identificador único de la entidad (clave primaria)
  - Opciones de generación:
    - AUTO: Hibernate elige la estrategia según la base de datos.
    - IDENTITY: Utiliza una columna autoincremental.
    - SEQUENCE: Usa una secuencia específica (compatible con PostgreSQL y Oracle).
    - TABLE: Usa una tabla de claves.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

Opciones Básicas de Configuración de Entidades

Características y Propiedades de una Entidad

- **Es una clase POJO (Plain Old Java Object)**
  - Constructor por defecto (sin parámetros)
  - Métodos **getter** y **setter** para las propiedades
  - Evitar dependencias externas innecesarias

Opciones Básicas de Configuración de Entidades

Características y Propiedades de una Entidad

- **Relación directa con una tabla**
  - Por defecto, el nombre de la entidad será el mismo que el nombre de la tabla correspondiente en la base de datos.
  - Esto se puede personalizar usando la anotación `@Table`
- **Parámetros clave de `@Table`:**
  - name: Nombre de la tabla.
  - schema: Esquema de la base de datos.
  - catalog: Catálogo de la base de datos.

```
@Table(name = "personas", schema = "public")
```

Opciones Básicas de Configuración de Entidades

Características y Propiedades de una Entidad

- **Columnas mapeadas a campos**
  - Cada campo de la clase (propiedad) se mapea a una columna en la tabla.
  - Puedes personalizar este mapeo con la anotación `@Column`
- **Parámetros clave de `@Column`:**
  - name: Nombre de la columna.
  - nullable: Si permite valores null.
  - length: Tamaño máximo del campo (relevante para VARCHAR).
  - unique: Si la columna debe ser única.

```
@Column(name = "nombre_completo", length = 50, nullable = false, unique = true)
```

Opciones Básicas de Configuración de Entidades

Características y Propiedades de una Entidad

- **Ciclo de vida gestionado por Hibernate**
  - Hibernate gestiona el ciclo de vida de las entidades: **transitorio**, **persistente**, **desasociado** y **eliminado**.
- **Relaciones con otras entidades**
  - **Uno a uno** (@OneToOne).
  - **Uno a muchos** (@OneToMany).
  - **Muchos a muchos** (@ManyToMany).

Opciones Básicas de Configuración de Entidades

Características y Propiedades de una Entidad

- **Herencia soportada**
  - Hibernate permite que una entidad sea parte de una jerarquía de herencia.
  - Se pueden usar estrategias como:
    - **Tabla por clase concreta** (@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS))
    - **Una sola tabla** (@Inheritance(strategy = InheritanceType.SINGLE_TABLE))
    - **Tablas separadas para cada entidad hija** (@Inheritance(strategy = InheritanceType.JOINED))

Opciones Básicas de Configuración de Entidades

Características y Propiedades de una Entidad

- **No debe ser una clase final**
  - Hibernate necesita la posibilidad de extender las clases o de generar proxies para la gestión de las entidades.
  - Por ello, las clases de las entidades no deben ser final.
- **Propiedades transitorias (**@Transient**)**
  - Si una propiedad no debe ser persistida en la base de datos, se puede marcar como @Transient. Hibernate ignorará ese campo.

Opciones Básicas de Configuración de Entidades

```java
@Entity
@Table(name = "persona")
public class Persona {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "nombre_completo", nullable = false, length = 50)
    private String nombre;

    @Column(name = "edad")
    private int edad;

    @Transient
    private String estadoTemporal;

    // Relación uno a muchos
    @OneToMany(mappedBy = "persona", cascade = CascadeType.ALL)
    private List<Direccion> direcciones;
```

Relaciones entre entidades

Relaciones: Tipos Básicos

1. **Uno a Uno (**@OneToOne**)**
   - Relación directa entre dos entidades donde una fila en una tabla está asociada con una sola fila en otra tabla.
   - **Parámetros clave:**
     - mappedBy: Indica que el mapeo es bidireccional.
     - cascade: Define el comportamiento en cascada (por ejemplo, CascadeType.ALL).

```
@OneToOne(mappedBy = "persona", cascade = CascadeType.ALL)
private Direccion direccion;
```

Relaciones entre entidades

Relaciones: Tipos Básicos

2. **Uno a Muchos (**@OneToMany**)**
   - Una fila en la tabla principal puede estar asociada con múltiples filas en la tabla relacionada.
   - **Parámetros clave:**
     - mappedBy: Define el lado propietario de la relación.
     - cascade: Operaciones en cascada.
     - fetch: Estrategia de carga (FetchType.LAZY o FetchType.EAGER).

```
@OneToMany(mappedBy = "persona", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
private List<Orden> ordenes;
```

Relaciones entre entidades

Relaciones: Tipos Básicos

3. **Muchos a Uno (**@ManyToOne**)**
   - Muchas filas de una tabla están asociadas con una sola fila en otra tabla.

```
@ManyToOne
@JoinColumn(name = "persona_id")
private Persona persona;
```

Relaciones entre entidades

Relaciones: Tipos Básicos

4. **Muchos a Muchos (**@ManyToMany**)**
   - Asociaciones entre múltiples filas de dos tablas a través de una tabla intermedia.
   - **Parámetros clave:**
     - joinTable: Define la tabla intermedia y sus columnas.

```
@ManyToMany
@JoinTable(
    name = "persona_proyecto",
    joinColumns = @JoinColumn(name = "persona_id"),
    inverseJoinColumns = @JoinColumn(name = "proyecto_id")
)
private Set<Proyecto> proyectos;
```

Relaciones entre entidades

Opciones de Parametrización en Relaciones

- @JoinColumn
  - Especifica la columna de la clave foránea en la relación.
  - **Parámetros clave:**
    - name: Nombre de la columna en la base de datos.
    - nullable: Si puede ser nula.
    - referencedColumnName: Nombre de la columna de referencia en la tabla relacionada.

Relaciones entre entidades

Opciones de Parametrización en Relaciones

- cascade
  - Controla cómo se propagan las operaciones (persistencia, eliminación, etc.) en las relaciones.
  - Tipos comunes:
    - CascadeType.ALL: Propaga todas las operaciones.
    - CascadeType.PERSIST: Propaga persistencia.
    - CascadeType.REMOVE: Propaga eliminación.

```
@OneToOne(cascade = CascadeType.ALL)
```

Relaciones entre entidades

Opciones de Parametrización en Relaciones

- fetch
  - Define cómo se cargan los datos relacionados:
    - FetchType.LAZY: Carga diferida (recomendado para rendimiento).
    - FetchType.EAGER: Carga inmediata.

```
@ManyToOne(fetch = FetchType.LAZY)
```

Relaciones entre entidades

Opciones de Parametrización en Relaciones

- orphanRemoval
  - Borra automáticamente entidades “huérfanas” en relaciones uno a muchos.

```
@OneToMany(mappedBy = "persona", orphanRemoval = true)
```

Configuración de Herencia

Tipos de Herencia Soportados

- Una tabla por jerarquía (SINGLE_TABLE):

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
```

- Una tabla por clase concreta (TABLE_PER_CLASS):

```
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
```

- Tablas separadas con claves foráneas (JOINED):

```
@Inheritance(strategy = InheritanceType.JOINED)
```

Configuración de Herencia

@DiscriminatorColumn

- En SINGLE_TABLE, Define la columna en la tabla que almacenará el valor que distingue entre los diferentes tipos de entidades en la jerarquía.
- Sus atributos comunes son:
  - name: El nombre de la columna en la tabla (por defecto es DTYPE).
  - discriminatorType: El tipo de datos de la columna (por defecto es STRING).
  - columnDefinition: Una definición personalizada de la columna.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE) // Estrategia de herencia de Tabla Única
@DiscriminatorColumn(name = "tipo_entidad", discriminatorType = DiscriminatorType.STRING)
public abstract class Persona {
```

Configuración de Herencia

@DiscriminatorValue

- Define el valor que se almacenará en la columna definida por @DiscriminatorColumn para una subclase específica.
- Se utiliza en las subclases de la jerarquía.

```
@Entity
@DiscriminatorValue("CLIENTE")
public class Cliente extends Persona {
    private String numeroCliente;
}

@Entity
@DiscriminatorValue("EMPLEADO")
public class Empleado extends Persona {
    private String departamento;
}
```

Configuración de Embebidos

@Embedded y @Embeddable:

- Permite incluir objetos dentro de entidades, reutilizando definiciones de clases.

```
@Embeddable
public class Direccion {
    private String calle;
    private String ciudad;
}

@Entity
public class Persona {
    @Id
    private Long id;

    @Embedded
    private Direccion direccion;
}
```

Novedades en Hibernate 6.x:

- Soporte para jerarquías heredadas en objetos embebidos.

Novedades en Hibernate 6.x sobre Relaciones

1. **Consultas más expresivas en HQL y Criteria**:
   - Funciones avanzadas para manejar colecciones relacionadas.
   - Ejemplo de búsqueda dentro de una colección:

```
FROM Persona p WHERE 'Java' MEMBER OF p.habilidades
```

2. **Soporte extendido para tipos JSON**:
   - Permite mapear directamente estructuras JSON a relaciones.
3. **Mejoras en** @Any **y relaciones polimórficas**:
   - Más flexibilidad para manejar relaciones donde una clave foránea puede apuntar a diferentes tablas.

Persistencia e consultas

Creación de un Repositorio CRUD Básico

¿Qué es un repositorio?

- Patrón que abstrae el acceso a la base de datos, ofreciendo métodos predefinidos para operaciones comunes y soporte para consultas personalizadas

Extender JpaRepository **o** CrudRepository

- Hibernate y JPA permiten gestionar entidades mediante interfaces Spring Data.

Creación de un Repositorio CRUD Básico

```
@Repository
public interface PersonaRepository extends JpaRepository<Persona, Long> {
}
```

- Métodos predefinidos:
  - save(T entity): Guardar o actualizar una entidad.
  - findById(ID id): Buscar por ID.
  - findAll(): Obtener todas las entidades.
  - deleteById(ID id): Eliminar por ID.

Creación de un Repositorio CRUD Básico

Guardar una entidad:

```
Persona persona = new Persona();
persona.setNombre("Juan");
personaRepository.save(persona);
```

Buscar por ID:

```
Optional<Persona> persona = personaRepository.findById(1L);
persona.ifPresent(p -> System.out.println(p.getNombre()));
```

Creación de un Repositorio CRUD Básico

Actualizar una entidad:

```
persona.setNombre("Juan Actualizado");
personaRepository.save(persona);
```

Eliminar por ID:

```
personaRepository.deleteById(1L);
```

Creación de un Repositorio CRUD Básico

Paginación y Ordenación

- **Uso de** Pageable **y** Page **en Spring Data JPA**
  - Spring proporciona una forma sencilla de aplicar paginación en consultas.
  - Métodos útiles:
    - findAll(Pageable pageable)
    - Soporte para ordenación con Sort.

```
Pageable pageable = PageRequest.of(0, 10, Sort.by("nombre").ascending());
Page<Persona> personas = personaRepository.findAll(pageable);

for (Persona p : personas.getContent()) {
    System.out.println(p.getNombre());
}
```

Creación de un Repositorio CRUD Básico

Paginación y Ordenación

- PageRequest.of(page, size, sort):
  - page: Número de página (empezando desde 0)
  - size: Tamaño de la página
  - sort: Criterio de ordenación (opcional)

- **Resultado de la paginación:**
  - getContent(): Lista de resultados
  - getTotalPages(): Número total de páginas
  - getTotalElements(): Número total de elementos

Creación de un Repositorio CRUD Básico

Paginación y Ordenación

- Supongamos que tienes una entidad llamada Producto con los campos nombre, precio y fechaCreacion.
- Quieres realizar una paginación que ordene primero por **precio** en orden descendente y luego por **nombre** en orden ascendente.
- Vamos a crear un Servicio que abstraiga a nuestros controladores de la lógica de manejo del repositorio.

Creación de un Repositorio CRUD Básico

```
@Service
public class ProductoService {

    @Autowired
    private ProductoRepository productoRepository;

    public Page<Producto> obtenerProductosPaginados(int page, int size) {
        // Definir orden múltiple
        Sort orden = Sort.by(
            Sort.Order.desc("precio"),
            Sort.Order.asc("nombre")
        );

        Pageable pageable = PageRequest.of(page, size, orden);

        // Recuperar los datos paginados
        return productoRepository.findAll(pageable);
    }
}
```

Creación de un Repositorio CRUD Básico

Consultas Personalizadas con JPQL/HQL

- **¿Qué es JPQL?**
  - Java Persistence Query Language.

    Similar a SQL pero opera sobre entidades, no tablas.

- **Definir consultas personalizadas en un repositorio:**
  - Usar @Query con JPQL o HQL.
  - Ejemplo de una consulta personalizada:

```
@Query("SELECT p FROM Persona p WHERE p.nombre LIKE %:nombre%")
List<Persona> buscarPorNombre(@Param("nombre") String nombre);
```

Creación de un Repositorio CRUD Básico

Consultas Nativas (Native Queries)

- **¿Qué son las consultas nativas?**
  - Consultas SQL que interactúan directamente con las tablas de la base de datos.

- **Definir una consulta nativa:**
  - Usar @Query con el atributo nativeQuery = true.
  - Ejemplo:

```
@Query(value = "SELECT * FROM personas WHERE nombre LIKE %:nombre%", nativeQuery = true)
List<Persona> buscarPorNombreNativo(@Param("nombre") String nombre);
```

Creación de un Repositorio CRUD Básico

Consultas Dinámicas

- **Uso de métodos con nombres derivados (Query Methods):**
  - Spring Data genera automáticamente consultas a partir del nombre del método.
  - Ejemplo:

```
List<Persona> findByNombre(String nombre);
List<Persona> findByEdadGreaterThan(int edad);
```

- **Soporte para múltiples condiciones:**

```
List<Persona> findByNombreAndEdad(String nombre, int edad);
```

Criteria API

- **Criteria API** es una forma tipada y programática de construir consultas en JPA.
- Permite definir consultas dinámicas que se integran bien con el tipo de datos de las entidades.
- Ejemplo:
  - Supongamos que tenemos una entidad llamada Persona con los siguientes atributos

```java
@Entity
public class Persona {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    private int edad;

    // Getters y setters
}
```

  - Queremos consultar todas las personas cuyo nombre contiene “Juan” y cuya edad es mayor de 25.

Criteria API

```java
@Autowired
private EntityManager entityManager;

public List<Persona> buscarConCriteria(String nombre, int edadMinima) {
    // 1. Crear el CriteriaBuilder
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();

    // 2. Crear el CriteriaQuery para la entidad Persona
    CriteriaQuery<Persona> query = cb.createQuery(Persona.class);

    // 3. Definir la raíz de la consulta (FROM Persona)
    Root<Persona> root = query.from(Persona.class);

    // 4. Definir las condiciones (WHERE)
    Predicate nombreCondicion = cb.like(root.get("nombre"), "%" + nombre + "%");
    Predicate edadCondicion = cb.greaterThan(root.get("edad"), edadMinima);

    // 5. Combinar las condiciones con AND
    query.where(cb.and(nombreCondicion, edadCondicion));

    // 6. Ejecutar la consulta
    return entityManager.createQuery(query).getResultList();
}
```

Criteria API

Explicación del Código

1. **CriteriaBuilder**: Se usa para crear consultas. Es la entrada principal para usar la API de Criteria.
2. **CriteriaQuery<T>**: Representa la consulta que se va a construir, donde T es el tipo de resultado (en este caso, Persona).
3. **Root<T>**: Representa la raíz de la consulta, que corresponde a la entidad principal sobre la que se construye la consulta (Persona).
4. **Predicate**: Representa una condición lógica en la cláusula WHERE. Se pueden combinar múltiples predicados con AND o OR.
5. **entityManager.createQuery(query)**: Convierte la consulta programática en una consulta ejecutable y la ejecuta.

Criteria API

- Si deseas ordenar los resultados (por ejemplo, por nombre), puedes usar el método orderBy:

```
query.orderBy(cb.asc(root.get("nombre")));
```

- Si necesitas paginar los resultados, puedes usar el método setFirstResult y setMaxResults:

```
TypedQuery<Persona> typedQuery = entityManager.createQuery(query);
typedQuery.setFirstResult(0); // Página inicial
typedQuery.setMaxResults(10); // Tamaño de la página

return typedQuery.getResultList();
```

Criteria API

Ventajas de Criteria API

- **Tipo seguro**: Los campos y atributos se validan en tiempo de compilación.
- **Dinámica**: Puedes construir consultas complejas de forma programática.
- **Portabilidad**: Al ser parte de JPA, es independiente de la base de datos subyacente.

Buenas prácticas

- Usar **paginación** para evitar sobrecargar el sistema con grandes volúmenes de datos.
- Limitar el uso de **consultas nativas** para casos específicos.
- Utilizar **consultas parametrizadas** para prevenir inyección SQL.
- Aprovechar el soporte automático de **Query Methods** en Spring Data.
- Construir Consultas Dinámicas con Criteria API
- Evitar Consultas N+1
  - Usar fetch en Criteria API o HQL para cargar relaciones de manera eficiente

```
Root<Persona> root = query.from(Persona.class);
root.fetch("relacion", JoinType.LEFT); // Cargar relación con un LEFT JOIN
```

- Monitorizar y Optimizar Consultas

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Optimización con EhCache

Tipos de Caché en Hibernate

Caché de Primer Nivel (First-Level Cache):

- **Descripción:** Es una caché asociada al contexto de persistencia (EntityManager o Session).
- **Características:**
  - Está habilitada por defecto en Hibernate.
  - Funciona por entidad en el alcance de la sesión.
  - Útil para evitar consultas repetitivas dentro de la misma transacción.
- **Limitación:** Los datos en caché se eliminan al cerrar la sesión.

Tipos de Caché en Hibernate

Caché de Segundo Nivel (Second-Level Cache):

- **Descripción:** Es una caché más global que almacena entidades, colecciones y asociaciones para múltiples sesiones.

- **Características:**
  - Se habilita explícitamente y requiere un proveedor externo como EhCache.
  - Mejora el rendimiento al compartir datos entre múltiples sesiones.

- **Tipos de datos que puede almacenar:**
  - Entidades (@Cacheable o configuración XML).
  - Colecciones.
  - Consultas específicas (Query Cache).

Tipos de Caché en Hibernate

Caché de Consultas (Query Cache):

- **Descripción:** Es una capa adicional para almacenar los resultados de consultas HQL/JPQL o Criteria.
- **Requiere:** Que el caché de segundo nivel esté habilitado.
- **Configuración:**
  - Activar con @Cacheable o mediante configuración en hibernate.cfg.xml.

¿qué es EhCache?

- **EhCache** es un proveedor de caché ampliamente utilizado para implementar la caché de segundo nivel en Hibernate.
- Es un caché de alto rendimiento, ligero y fácil de configurar.
- Compatible con múltiples entornos: Hibernate, Spring y aplicaciones standalone.
- **Características principales:**
  - Soporte para caché en memoria y disco.
  - Integración con Hibernate y Spring Boot.
  - Configuración granular (por entidad o consulta).
  - Capacidad para configuraciones distribuidas (clústeres).

¿Cuándo y Por Qué Utilizar EhCache?

- **Cuando utilizarlo:**
  - Cuando las operaciones de lectura son frecuentes y el acceso a la base de datos debe minimizarse.
  - En aplicaciones con entidades que no cambian frecuentemente y son accedidas repetidamente.
  - Para optimizar el rendimiento en aplicaciones distribuidas o con múltiples instancias.
- **Razones para usarlo:**
  - Disminuir la latencia al reducir las consultas a la base de datos.
  - Mejorar el rendimiento general al evitar trabajo redundante.
  - Permitir un almacenamiento híbrido (RAM y disco).
- **Casos de uso comunes:**
  - Datos estáticos como configuraciones o catálogos.
  - Consultas complejas que producen los mismos resultados a menudo.
  - Optimización de relaciones cargadas frecuentemente.

Ventajas y Limitaciones de EhCache

- **Ventajas:**
  - **Rendimiento mejorado:** Reduce la carga en la base de datos al almacenar datos en memoria.
  - **Configuración simple:** Fácil de integrar con Hibernate y Spring Boot.
  - **Persistencia en disco:** Mantiene datos incluso después de reiniciar la aplicación.
  - **Escalabilidad:** Soporte para clústeres en implementaciones distribuidas.
  - **Flexibilidad:** Configuración por entidad, consulta o colección.

Ventajas y Limitaciones de EhCache

- **Limitaciones o consideraciones:**
  - **Consumo de memoria:** Un uso excesivo de caché puede agotar la memoria del sistema.
  - **Invalidez de caché:** Los datos desactualizados en caché pueden generar inconsistencias si no se invalidan correctamente.
  - **Coste de configuración distribuida:** En escenarios de clúster, configurar caché distribuida puede ser complejo.
  - **Datos dinámicos:** No es ideal para datos que cambian frecuentemente, ya que invalida el propósito del caché.

Ventajas y Limitaciones de EhCache

- **Buenas prácticas para mitigar limitaciones:**
  - Configurar correctamente los tiempos de expiración del caché.
  - Usar caché solo para datos que se acceden frecuentemente y no cambian a menudo.
  - Monitorizar el rendimiento de la caché con herramientas como JMX.

Novedades Relevantes en Hibernate 6.x Relacionadas con la Caché

- **Mejoras en el soporte para caché de segundo nivel:**
  - Se han optimizado las interacciones entre Hibernate y proveedores como EhCache.
  - Mejor integración para entidades que usan persistencia nativa JSON o estructuras complejas.
- **Configuración simplificada:**
  - La configuración basada en propiedades (en application.properties o hibernate.cfg.xml) ahora es más clara y precisa.
- **Soporte para claves compuestas y tipos complejos:**
  - Mejora en la serialización y almacenamiento en caché de entidades con claves compuestas o datos JSON/estructurados.
- **Compatibilidad mejorada con caching distribuido:**
  - Mejor soporte para configuraciones de clúster (por ejemplo, con herramientas como Hazelcast).

Configuración Básica de EhCache con Hibernate y Spring Boot

Dependencias necesarias:

```
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Configuración Básica de EhCache con Hibernate y Spring Boot

Configurar Hibernate para usar EhCache:

- En application.properties:

```
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.jcache.JCacheRegionFactory
spring.cache.jcache.config=classpath:ehcache.xml
```

- **Archivo de configuración** ehcache.xml

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd">
    <cache name="personaCache"
           maxEntriesLocalHeap="1000"
           timeToLiveSeconds="3600"/>
</ehcache>
```

Configuración de EhCache con Anotaciones

Habilitar Caching en tu aplicación:

- Agrega la anotación @EnableCaching en una clase de configuración de Spring Boot.
- Definir la configuración de EhCache mediante Java Config

```java
@Configuration
@EnableCaching
public class EhCacheConfig {

    @Bean
    public org.ehcache.CacheManager ehCacheManager() {
        return CacheManagerBuilder.newCacheManagerBuilder()
            .withCache("personaCache",
                CacheConfigurationBuilder.newCacheConfigurationBuilder(
                    Long.class,
                    Object.class,
                    ResourcePoolsBuilder.heap(100)) // Máximo de 100 objetos en caché
                    .withExpiry(org.ehcache.expiry.ExpiryPolicyBuilder.timeToLiveExpiration(
                        java.time.Duration.ofSeconds(3600)) // TTL: 1 hora
                    )
            ).build(true);
    }
}
```

Configuración de EhCache con Anotaciones

Anotar métodos para utilizar el caché:

- Usa las anotaciones de Spring para aplicar caché en métodos específicos.

```
@Service
public class PersonaService {

    @Cacheable(value = "personaCache", key = "#id")
    public Persona obtenerPersonaPorId(Long id) {
        // Aquí se consulta la base de datos solo si no está en caché
        return personaRepository.findById(id).orElseThrow();
    }
}
```

  - @Cacheable: Indica que el resultado de este método se almacenará en caché.
  - value: Nombre del caché definido en CacheManager.
  - key: Clave del objeto en caché (en este caso, el ID de la persona).

Ventajas y limitaciones de EhCache con Anotaciones

Ventajas de Configurar EhCache con Anotaciones:

- **Flexibilidad:** No necesitas un archivo XML separado.
- **Mantenimiento simplificado:** Toda la configuración de caché está en el código Java.
- **Fácil personalización:** Puedes configurar diferentes cachés con distintas propiedades directamente en Java.

Limitaciones:

- Puede ser menos legible en proyectos grandes, ya que las configuraciones largas en Java pueden ser más difíciles de entender que un archivo XML bien estructurado.
- Si usas una configuración distribuida (en clúster), el XML puede ser más conveniente para configurar nodos.

Estrategias de CacheConcurrencyStrategy

READ_ONLY (Solo lectura):

- Ideal para datos que no cambian después de ser cargados en la caché (datos estáticos).

- **Ventajas:**
  - No requiere sincronización, por lo que es muy rápido.
- **Limitación:**
  - No permite actualizaciones; lanzarían excepciones si se intentan modificar los datos.

Estrategias de CacheConcurrencyStrategy

NONSTRICT_READ_WRITE:

- Permite lecturas concurrentes, pero no garantiza que los datos en caché estén completamente sincronizados con la base de datos.

- **Ventajas:**
  - Buen equilibrio entre rendimiento y consistencia.
- **Limitación:**
  - Riesgo de leer datos desactualizados si otra transacción actualiza la misma entidad.
- **Casos de uso:**
  - Entidades que se actualizan infrecuentemente y no requieren consistencia estricta.

Estrategias de CacheConcurrencyStrategy

READ_WRITE:

- Asegura que los datos en caché estén siempre sincronizados con la base de datos mediante un mecanismo de bloqueo “soft locks”.

- **Ventajas:**
  - Ofrece una consistencia más estricta.
- **Limitación:**
  - Más lento que las estrategias anteriores debido al bloqueo adicional.
- **Casos de uso:**
  - Entidades que se actualizan con frecuencia y requieren consistencia estricta.

Estrategias de CacheConcurrencyStrategy

TRANSACTIONAL:

- Solo disponible con proveedores de caché que soportan transacciones distribuidas (como JTA).
- Sincroniza completamente el caché con la base de datos en todas las operaciones.

- **Ventajas:**
  - Consistencia total.
- **Limitación:**
  - Requiere más configuración y puede ser más costoso en términos de rendimiento.
- **Casos de uso:** Escenarios distribuidos o altamente críticos que requieren transacciones.

Configuración de CacheConcurrencyStrategy

- Se puede configurar CacheConcurrencyStrategy a nivel de entidad utilizando la anotación @Cache

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Producto {
    @Id
    @GeneratedValue
    private Long id;

    private String nombre;
    private Double precio;
}
```

Factores para Elegir una Estrategia

- **Frecuencia de actualización de datos:**
  - Datos estáticos: READ_ONLY.
  - Datos que cambian poco: NONSTRICT_READ_WRITE.
  - Datos dinámicos: READ_WRITE o TRANSACTIONAL.
- **Necesidad de consistencia:**
  - Baja: NONSTRICT_READ_WRITE.
  - Alta: READ_WRITE o TRANSACTIONAL.
- **Impacto en el rendimiento:**
  - Estrategias más estrictas pueden ser más lentas debido al bloqueo y sincronización.
- **Proveedor de caché:**
  - Algunos proveedores (como EhCache o Hazelcast) soportan todas las estrategias, pero TRANSACTIONAL puede requerir configuraciones adicionales (como JTA).

Errores comunes al usar caching y cómo evitarlos

Usar Caché para Datos Frecuentemente Actualizados

- **Problema:**
  - Si los datos almacenados en caché cambian frecuentemente en la base de datos, la caché puede quedarse desactualizada, causando inconsistencias.
- **Solución:**
  - No caches entidades o resultados de consultas que cambien con frecuencia.
  - Configura un tiempo de vida corto para los objetos en caché (timeToLive).
  - Usa **invalidation policies** para sincronizar los cambios en la base de datos con la caché.

Errores comunes al usar caching y cómo evitarlos

Olvidar Configurar la Invalidez de la Caché

- **Problema:**
  - Cuando los datos cambian en la base de datos, la caché no se actualiza automáticamente (a menos que se configure explícitamente).
- **Solución:**
  - Configura políticas de **invalidez de caché** para limpiar los datos desactualizados.
  - Usa mecanismos como CacheEvict en Spring:

```
@CacheEvict(value = "personaCache", key = "#id")
public void actualizarPersona(Long id, Persona persona) {
    personaRepository.save(persona);
}
```

Errores comunes al usar caching y cómo evitarlos

Caching Excesivo

- **Problema:**
  - Si almacenas demasiadas entidades o consultas en la caché, puede agotarse la memoria del sistema.
- **Solución:**
  - Limita el tamaño del caché configurando un máximo de entradas con ResourcePoolsBuilder.heap o similar.
  - Evita almacenar grandes volúmenes de datos en caché; prioriza datos críticos o frecuentemente consultados.

Errores comunes al usar caching y cómo evitarlos

No Configurar Correctamente el Caché Distribuido

- **Problema:**
  - En entornos distribuidos (clústeres), la falta de configuración adecuada puede causar inconsistencias entre nodos.
- **Solución:**
  - Usa un proveedor de caché distribuido como **Hazelcast**, **Infinispan** o **Redis**.
  - Configura las políticas de sincronización entre nodos.

Errores comunes al usar caching y cómo evitarlos

Olvidar Activar la Caché de Segundo Nivel

- **Problema:**
  - Aunque Hibernate soporta caché de segundo nivel, está **desactivada por defecto**. Esto puede llevar a pensar que el caché está funcionando cuando no lo está.
- **Solución:**
  - Habilitar explícitamente la caché de segundo nivel en la configuración

Errores comunes al usar caching y cómo evitarlos

Ignorar el Caché de Consultas (Query Cache)

- **Problema:**
  - Los desarrolladores suelen almacenar consultas personalizadas sin habilitar el caché de consultas. Esto lleva a que se realicen las mismas consultas repetidamente.
- **Solución:**
  - Habilitar el caché de consultas y configurarlo correctamente:

```
Query query = session.createQuery("FROM Persona WHERE nombre = :nombre");
query.setCacheable(true);
query.setParameter("nombre", "Juan");
List<Persona> resultados = query.list();
```

Errores comunes al usar caching y cómo evitarlos

Configurar Mal los @Cacheable en Relaciones

- **Problema:**
  - Caching en colecciones o asociaciones sin gestionar las relaciones puede causar consultas N+1 o inconsistencias.
- **Solución:**
  - Usa @Cacheable solo cuando sea necesario y combina con fetch para optimizar relaciones:

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Persona {
    @OneToMany(mappedBy = "persona", fetch = FetchType.LAZY)
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    private List<Direccion> direcciones;
}
```

Errores comunes al usar caching y cómo evitarlos

Confiar Demasiado en el Caché

- **Problema:**
  - Los desarrolladores a veces asumen que el caché solucionará todos los problemas de rendimiento, pero no optimizan las consultas y relaciones subyacentes.
- **Solución:**
  - Optimiza primero las consultas y relaciones, luego usa caché como complemento.
  - Usa herramientas de profiling (como logs de Hibernate o JProfiler) para identificar cuellos de botella antes de implementar caché.

Errores comunes al usar caching y cómo evitarlos

No Monitorizar el Caché

- **Problema:**
  - No se rastrean métricas del uso del caché, lo que puede causar problemas de rendimiento invisibles.
- **Solución:**
  - Habilitar herramientas de monitoreo como JMX para EhCache:

```
management.endpoints.jmx.exposure.include=*
spring.cache.jcache.jmx-enabled=true
```

  - Monitorizar estadísticas como:
    - Tasa de aciertos y errores (hit/miss ratio).
    - Tiempo de vida de los datos en caché.

Errores comunes al usar caching y cómo evitarlos

Caché en Datos Sensibles

- **Problema:**
  - Almacenar datos sensibles en caché (como contraseñas o información personal) puede comprometer la seguridad.
- **Solución:**
  - Evita almacenar datos sensibles en caché.
  - Si es necesario, utiliza cifrado en el nivel de caché o en las entidades.

Conclusións e Boas Prácticas

Resumen de los puntos clave

- **Hibernate y JPA:** Hemos cubierto cómo Hibernate se utiliza como una implementación del estándar JPA para gestionar la persistencia en Java, proporcionando una forma de trabajar con bases de datos de manera eficiente y flexible.
- **Configuración y Integración con Spring Boot:** Discutimos cómo configurar Hibernate en una aplicación Spring Boot, integrando JPA y Hibernate para simplificar las operaciones CRUD.
- **Mapeo de Entidades y Relaciones:** Se explicó el mapeo de entidades y las distintas estrategias de relación (uno a uno, uno a muchos, muchos a muchos), y cómo estas relaciones afectan al rendimiento y a la integridad de los datos.
- **Persistencia y Consultas:** Exploramos cómo construir repositorios en Spring Data JPA, cómo realizar consultas con JPQL/HQL y cómo optimizar el rendimiento mediante paginación y consultas personalizadas.
- **Optimización con EhCache:** Presentamos las ventajas de usar caché en Hibernate y cómo integrarlo con EhCache para mejorar el rendimiento, incluyendo la configuración y mejores prácticas.

Mejores Prácticas con Hibernate

Gestión de Transacciones:

- **Usar transacciones correctamente**: Asegúrate de que las transacciones abarcan la mínima cantidad de operaciones necesarias y que se cierren de manera adecuada para evitar problemas de concurrencia.
- **Evitar transacciones largas**: Las transacciones deben ser lo más breves posible para evitar bloqueos y mantener el rendimiento.
- **Uso de** @Transactional: En Spring, usa la anotación @Transactional para controlar el inicio y fin de las transacciones, asegurando la consistencia de los datos.

```
@Transactional
public void actualizarProducto(Long id, Producto producto) {
    productoRepository.save(producto);
}
```

Mejores Prácticas con Hibernate

Mapeo Eficiente de Entidades:

- **Seleccionar el tipo adecuado de relaciones**: Usa el tipo de relación más apropiado para tus necesidades, como @OneToMany con fetch = FetchType.LAZY para evitar cargar innecesariamente grandes colecciones.
- **Evitar N+1 Query Problem**: Asegúrate de usar estrategias de **fetching** adecuadas (e.g., fetch = FetchType.EAGER o JOIN FETCH en consultas HQL) para evitar el problema de las consultas N+1.
- **Uso de DTOs para consultas complejas**: Si necesitas resultados de consultas complejas, usa **Data Transfer Objects (DTOs)** para evitar cargar innecesarios objetos completos.

Mejores Prácticas con Hibernate

Optimización de Consultas:

- **Uso de caché de consultas**: Habilita el **caché de consultas** para consultas frecuentes y estáticas, minimizando el número de consultas a la base de datos.
- **Indexación adecuada**: Asegúrate de que las consultas más frecuentes estén indexadas correctamente en la base de datos.
- **Paginación en consultas grandes**: Usa **paginación** (Pageable) para evitar cargar grandes cantidades de datos a la vez.

```
@Query("SELECT p FROM Producto p")
Page<Producto> findAllPaginated(Pageable pageable);
```

Herramientas Adicionales Recomendadas para la Optimización

Herramientas de Monitoreo y Perfilado:

- **Hibernate Profiler**: Utiliza el **Hibernate Profiler** para analizar el rendimiento de tus consultas y detectar posibles cuellos de botella.
- **JMX y Spring Boot Actuator**: Usa JMX para monitorear el rendimiento de la caché y las transacciones en tiempo real.
- **Elasticsearch**: Si necesitas búsquedas avanzadas, considera la integración con **Elasticsearch** para consultas más rápidas y escalables.
- **JPA Criteria API**: Para consultas dinámicas y seguras, usa **Criteria API**, que es más flexible y ayuda a evitar errores en las consultas HQL/JPQL.

Herramientas Adicionales Recomendadas para la Optimización

Bibliotecas de Optimización:

- **EhCache / Hazelcast**: Ya que discutimos la integración de **EhCache**, también puedes considerar otras bibliotecas como **Hazelcast** o **Infinispan** para caché distribuido en entornos de alta disponibilidad.
- **QueryDSL**: Utiliza **QueryDSL** para generar consultas de manera más segura y optimizada, especialmente cuando se trabaja con filtros dinámicos y consultas complejas.

Herramientas Adicionales Recomendadas para la Optimización

Estrategias de Desempeño:

- **Optimización de los índices de base de datos**: Asegúrate de que la base de datos tenga los índices necesarios para mejorar el rendimiento de las consultas más frecuentes.
- **Uso de bases de datos NoSQL**: En algunos escenarios, como el manejo de grandes volúmenes de datos no estructurados, puedes considerar **bases de datos NoSQL** como **MongoDB** o **Cassandra**.

Conclusión

- **Hibernate es una herramienta poderosa** para la gestión de la persistencia en aplicaciones Java, pero requiere **buenas prácticas** y una correcta **optimización** para asegurar un rendimiento adecuado y evitar problemas como la inconsistencia de datos o el agotamiento de recursos.
- La combinación de **Hibernate**, **JPA** y herramientas de **Spring Boot** te permite construir aplicaciones escalables, eficientes y fáciles de mantener.
- **La clave del éxito** en el uso de Hibernate radica en la correcta configuración, en el mapeo eficiente de entidades y en la optimización de las consultas para asegurar una buena experiencia de usuario y rendimiento.

¿Alguna duda?

Muchas gracias ☺

Antonio Varela Nieto

antonio.varela@teslatechnologies.com