Hibernate

> ⚠ **DOCUMENTO EN DESARROLLO** ⚠

Introducción

Hibernate es un poderoso **framework de mapeo objeto-relacional (ORM)** para Java que simplifica significativamente el proceso de interacción con bases de datos relacionales. Desarrollado inicialmente por Gavin King en 2001 y actualmente mantenido por Red Hat, Hibernate se ha convertido en una herramienta fundamental para desarrolladores Java que buscan gestionar la persistencia de datos de manera eficiente y transparente.

En términos simples, Hibernate permite a los desarrolladores trabajar con objetos Java en lugar de consultas SQL directas. Esto se logra mediante el mapeo de objetos Java a tablas de bases de datos y viceversa, facilitando así la manipulación y persistencia de datos en la aplicación. Hibernate se integra estrechamente con el entorno de desarrollo Java y es compatible con una amplia gama de bases de datos relacionales, lo que lo convierte en una solución flexible y escalable para aplicaciones empresariales.

Entre sus características principales se incluyen:

- **Mapeo Objeto-Relacional (ORM)**: Hibernate simplifica la representación de objetos Java como entidades persistentes en bases de datos relacionales, gestionando automáticamente la correspondencia entre las estructuras de datos y los objetos en el código.
- **Transacciones y gestión de la sesión**: proporciona un mecanismo robusto para administrar transacciones de bases de datos y sesiones de trabajo, asegurando la integridad y consistencia de los datos.
- **Consultas orientadas a objetos**: permite a los desarrolladores realizar consultas utilizando el lenguaje de consultas orientado a objetos de Hibernate (HQL), que es más intuitivo y menos propenso a errores que el SQL estándar.
- **Caché de segundo nivel**: Hibernate ofrece soporte para cache de segundo nivel, lo que mejora el rendimiento al reducir la cantidad de consultas repetitivas a la base de datos.
- **Soporte para herencia y asociaciones**: permite modelar relaciones complejas entre entidades y soporta herencia en la capa de persistencia, facilitando el diseño de bases de datos relacionales más complejas.

En resumen, Hibernate simplifica el desarrollo de aplicaciones Java al proporcionar una capa de abstracción entre la lógica de negocio y la capa de persistencia de datos, permitiendo a los desarrolladores centrarse en la funcionalidad de la aplicación sin tener que preocuparse por los detalles de la manipulación de la base de datos. Esto lo convierte en una herramienta indispensable en el arsenal de cualquier desarrollador Java que trabaje con bases de datos relacionales.

⚠ Introducción generada por ChatGPT ⚠

Configuration and bootstrap

Including Hibernate in your project build

Para utilizar Hibernate fuera del entorno de un contenedor como WildFly o Quarkus, se necesita:

- incluir el propio Hibernate ORM, junto con el controlador JDBC apropiado, como dependencias del proyecto.
- configurar Hibernate con información sobre la base de datos, especificando las propiedades de configuración.

Añadir la dependencia de Hibernate en **Maven**:

```xml
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>{version}</version>
    </dependency>
</dependencies>
```

Añadir la dependencia con **Gradle**:

```gradle
dependencies {
  implementation "org.hibernate.orm:hibernate-core:{version}"
}
```

> ⚠ **Sustituir {version}** por la última versión o la versión correspondiente.

Hibernate se divide en varios módulos/artefactos bajo el grupo `org.hibernate.orm`. El artefacto principal se llama `hibernate-core`.

Hibernate también proporciona un módulo de plataforma (**BOM** en terminología Maven) que se puede utilizar para alinear las versiones de los módulos de Hibernate junto con las versiones de sus bibliotecas. El artefacto de la plataforma se denomina `hibernate-platform`.

Para utilizar la plataforma BOM en **Maven**:

```xml
<dependencies>
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
    </dependency>
    <dependency>
        <groupId>jakarta.transaction</groupId>
        <artifactId>jakarta.transaction-api</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.hibernate.orm</groupId>
            <artifactId>hibernate-platform</artifactId>
            <version>{version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Para utilizar la plataforma BOM en **Gradle**:

```gradle
dependencies {
  implementation platform "org.hibernate.orm:hibernate-platform:{version}"

  // use the versions from the platform
  implementation "org.hibernate.orm:hibernate-core"
  implementation "jakarta.transaction:jakarta.transaction-api"
}
```

> ⚠ **Sustituir {version}** por la última versión o la versión correspondiente.

También se debe **agregar una dependencia para el controlador JDBC** de la base de datos utilizada:

- PostgreSQL or CockroachDB -> `org.postgresql:postgresql:{version}`
- MySQL or TiDB -> `com.mysql:mysql-connector-j:{version}`
- MariaDB -> `org.mariadb.jdbc:mariadb-java-client:{version}`
- DB2 -> `com.ibm.db2:jcc:{version}`
- SQL Server -> `com.microsoft.sqlserver:mssql-jdbc:${version}`
- Oracle -> `com.oracle.database.jdbc:ojdbc11:${version}`
- H2 -> `com.h2database:h2:{version}`
- HSQLDB -> `org.hsqldb:hsqldb:{version}`

Donde `{version}` es la última versión del controlador JDBC para la base de datos.

```
<dependencies>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>2.2.224</version>
    </dependency>
</dependencies>
```

Optional dependencies

Opcionalmente, también se puede agregar dependencias opcionales para trabajar con Hibernate como por ejemplo SLF4J o un Query Validator para comprobar las sentencias HQL en tiempo de compilación.

> En Maven Central se puede consultar todas las dependencias bajo el paquete `org.hibernate.orm`.

Configuration using JPA XML

Para seguir el enfoque estándar JPA, se debe proporcionar un archivo llamado `persistence.xml`, que se debe colocar en el directorio `META-INF` dentro del directorio de recursos del proyecto:

```
mi-proyecto/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── example/
│   │   │           └── MiEntidad.java
│   │   └── resources/
│   │       └── META-INF/
│   │           └── persistence.xml
│   └── test/
│       ├── java/
│       └── resources/
├── pom.xml
└── README.md
```

Un fichero `persistence.xml` tiene este aspecto:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="2.0">

    <persistence-unit name="org.hibernate.example">
        <class>org.hibernate.example.Book</class>
        <class>org.hibernate.example.Author</class>
        <properties>
            <!-- PostgreSQL -->
            <property name="jakarta.persistence.jdbc.url"
                    value="jdbc:postgresql://localhost/example"/>

            <!-- Credentials -->
            <property name="jakarta.persistence.jdbc.user" value="gavin"/>
            <property name="jakarta.persistence.jdbc.password" value="hibernate"/>

            <!-- Automatic schema export -->
            <property name="jakarta.persistence.schema-generation.database.action"
                      value="drop-and-create"/>

            <!-- SQL statement logging -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.highlight_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Cada elemento `<class>` especifica el nombre completo de una clase de entidad.

En algunos entornos de contenedores, por ejemplo, en cualquier contenedor EE, los elementos `<class>` son innecesarios, ya que el **contenedor escaneará** el archivo en busca de clases anotadas y reconocerá automáticamente cualquier clase anotada como `@Entity`.

Cada elemento `<property>` especifica una propiedad de configuración y su valor. En el ejemplo hay propiedades definidas en el estándar de JPA (`'jakarta.persistance'`) y otras propiedades son específicas de Hibernate (`'hibernate'`).

En el Javadoc de Hibernate se encuentra una lista completa de constantes asociadas a esas propiedades.

> 👀 Las propiedades de configuración con el prefijo del espacio de nombres `javax.persistence` **heredado** de Java EE aún son reconocidas, pero se debe optar por el espacio de nombres `jakarta.persistence` de Jakarta EE .

Una vez configurado el fichero, se puede obtener una `jakarta.persistence.EntityManager` llamando a `Persistence.createEntityManagerFactory()` :

```
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("org.hibernate.example");
```

Si fuera necesario, es posible anular las propiedades de configuración especificadas en el fichero `persistence.xml` :

```
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("org.hibernate.example",
            Map.of(AvailableSettings.JAKARTA_JDBC_PASSWORD, password));
```

Configuration using Hibernate API

Alternativamente, la venerable clase `org.hibernate.cfg.Configuration` permite configurar una instancia de Hibernate en código Java:

```
SessionFactory sessionFactory =
    new Configuration()
        .addAnnotatedClass(Book.class)
        .addAnnotatedClass(Author.class)
        // PostgreSQL
        .setProperty(AvailableSettings.JAKARTA_JDBC_URL, "jdbc:postgresql://localhost/example")
        // Credentials
        .setProperty(AvailableSettings.JAKARTA_JDBC_USER, user)
        .setProperty(AvailableSettings.JAKARTA_JDBC_PASSWORD, password)
        // Automatic schema export
        .setProperty(AvailableSettings.JAKARTA_HBM2DDL_DATABASE_ACTION,
                     Action.SPEC_ACTION_DROP_AND_CREATE)
        // SQL statement logging
        .setProperty(AvailableSettings.SHOW_SQL, true)
        .setProperty(AvailableSettings.FORMAT_SQL, true)
        .setProperty(AvailableSettings.HIGHLIGHT_SQL, true)
        // Create a new SessionFactory
        .buildSessionFactory();
```

Esta clase `org.hibernate.cfg.Configuration` ha sobrevivido casi sin cambios desde las primeras versiones (anteriores a la 1.0) de Hibernate, por lo que no parece particularmente moderna. Por otro lado, es muy fácil de usar y expone algunas opciones que `persistence.xml` no admite.

En realidad, la clase `org.hibernate.cfg.Configuration` es solo una fachada muy simple para la API más moderna y más poderosa (pero más compleja) definida en el paquete `org.hibernate.boot`. Esta API es útil si se tienen requisitos avanzados, por ejemplo, en el desarrollo de un framework o un contenedor.

Configuration using Hibernate properties file

Si se utiliza la API `org.hibernate.cfg.Configuration` de Hibernate, pero se quiere evitar el uso de ciertas propiedades de configuración directamente en el código Java, se pueden especificar en un archivo llamado `hibernate.properties` y colocar el archivo en el classpath del proyecto (por ejemplo en `src/main/resources`):

```
# PostgreSQL
jakarta.persistence.jdbc.url=jdbc:postgresql://localhost/example
# Credentials
jakarta.persistence.jdbc.user=hibernate
jakarta.persistence.jdbc.password=zAh7mY42MNshzAQ5

# Echo all executed SQL to console
hibernate.show_sql=true
hibernate.format_sql=true
hibernate.highlight_sql=true
```

Si se está utilizando Hibernate dentro una **aplicación Spring Boot**, es posible que se esté utilizando un fichero `application.properties` o `application.yml`, que es común en este tipo de aplicaciones, tanto para la propia configuración de Spring Boot como para la configuración del acceso a datos. En Spring Boot, muchas propiedades de Hibernate se configuran dentro de `application.properties` bajo el prefijo `spring.jpa` o `spring.datasource`:

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase
spring.datasource.username=myuser
spring.datasource.password=mypassword
spring.jpa.hibernate.ddl-auto=update
```

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

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydatabase
    username: myuser
    password: mypassword
  jpa:
    hibernate:
      ddl-auto: update
      show-sql: true
      properties:
        hibernate.generate_statistics: true
```

Basic configuration settings

La interfaz `org.hibernate.cfg.AvailableSettings` enumera todas las propiedades de configuración que entiende Hibernate.

La lista es extensa, sin embargo la mayoría de estas propiedades rara vez se necesitan y muchas **sólo existen para brindar compatibilidad con versiones anteriores de Hibernate**. Con raras excepciones, el comportamiento predeterminado de cada una de estas propiedades es el recomendable.

Las propiedades que deben configurarse para empezar son:

- **jakarta.persistence.jdbc.url**: la URL JDBC de la base de datos
- **jakarta.persistence.jdbc.user**: credenciales de acceso a la base de datos
- **jakarta.persistence.jdbc.password**: credenciales de acceso a la base de datos

En términos de optimización de rendimiento, también es recomendable configurar la propiedad `hibernate.connection.pool_size`.

Automatic schema export

Esta característica de Hibernate es una característica que permite a Hibernate **generar y exportar automáticamente el esquema de la base de datos** basado en las entidades mapeadas en la aplicación. Esto puede incluir la creación, actualización o eliminación de tablas, índices y llaves foráneas, entre otros elementos del esquema de la base de datos.

Puede hacer que Hibernate infiera el esquema de su base de datos a partir de las anotaciones de mapeo que ha especificado en su código Java, y exportar el esquema al momento de la inicialización especificando una o más de las siguientes propiedades de configuración:

- **jakarta.persistence.schema-generation.database.action**
- **jakarta.persistence.create-database-schemas** (opcional)
- **jakarta.persistence.schema-generation.create-source** (opcional)
- **jakarta.persistence.schema-generation.create-script-source** (opcional)
- **jakarta.persistence.sql-load-script-source** (opcional)

Esta característica es extremadamente útil para realizar pruebas. Durante las fases iniciales de desarrollo o cuando se está creando un prototipo, es útil tener Hibernate manejando automáticamente el esquema.

La forma más sencilla de preinicializar una base de datos con datos de prueba o "de referencia" es colocar una lista de declaraciones SQL de inserción en un archivo llamado *'import.sql'* por ejemplo, y especificar la ruta a este archivo utilizando la propiedad `jakarta.persistence.sql-load-script-source` .

Este enfoque es más limpio que escribir código Java para instanciar entidades y llamar a `persist()` en cada una de ellas.

Logging the generated SQL

Existen dos opciones para **visualizar el SQL generado** mientras se envía a la base de datos.

Una forma es establecer la propiedad `hibernate.show_sql=true` , y Hibernate registrará el SQL directamente en la consola. Puede hacer que la salida sea mucho más legible habilitando el formateo o el resaltado. Estas configuraciones son muy útiles para solucionar problemas con las declaraciones SQL generadas:

- **hibernate.show_sql=true**: imprime el registro SQL directamente en la consola
- **hibernate.format_sql=true**: imprime el registro SQL en un formato con sangría de varias líneas
- **hibernate.highlight_sql=true**: imprime el registro SQL con resaltado de sintaxis mediante códigos de escape ANSI

Como alternativa, puede habilitar el registro en nivel de depuración (debug) para la categoría `org.hibernate.SQL` utilizando una implementación de logging con SLF4J.

Minimizing repetitive mapping information

Las propiedades `hibernate.default_schema` o `hibernate.default_catalog` son muy útiles para minimizar la cantidad de información que necesitará especificar explícitamente en las anotaciones `@Table` y `@Column` .

- `hibernate.default_schema` : nombre de esquema predeterminado para entidades que no declaran explícitamente uno
- `hibernate.default_catalog` : nombre de catálogo predeterminado para entidades que no declaran explícitamente uno

Escribir su propia `PhysicalNamingStrategy` y/o `ImplicitNamingStrategy` es una forma especialmente buena de reducir el desorden de las anotaciones en las clases de entidad, e implementar sus convenciones de nomenclatura de base de datos, teniendo en cuenta las posibles estrategias de nomenclatura.

Por lo tanto, se debería hacer para cualquier modelo de datos que no sea trivial.

Nationalized character data in SQL Server

Por defecto, los tipos *'char'* y *'varchar'* de **SQL Server** no admiten datos Unicode. Sin embargo, una cadena de Java puede contener cualquier carácter Unicode. Por lo tanto, si se está trabajando con SQL Server, es posible que se necesite forzar a Hibernate a usar los tipos de columna *'nchar'* y *'nvarchar'*:

- **hibernate.use_nationalized_character_data**: *'nchar'* y *'nvarchar'* en lugar de *'char'* y *'varchar'*

Por otro lado, si solo algunos campos almacenan datos nacionalizados, utilice la anotación `@Nationalized` para indicar los campos de sus entidades que mapean estos columnas.

Como alternativa, se puede configurar SQL Server para usar la intercalación habilitada para UTF-8 (_UTF8).

Entities

Una entidad es una clase Java anotada con `@Enitity` que representa **datos en una tabla** de una base de datos relacional. Decimos que la entidad hace un mapeo o se mapea a la tabla.

Una entidad tiene **atributos**, que son propiedades o campos que se mapean a columnas de la tabla. En particular, cada entidad debe tener un identificador o id, que se mapea a la clave primaria de la tabla. El id nos permite asociar de manera única una fila de la tabla con una instancia de la clase Java, al menos dentro de un contexto de persistencia dado.

Una instancia de una clase Java no puede sobrevivir fuera de la máquina virtual a la que pertenece. Sin embargo, podemos pensar que una instancia de entidad tiene un ciclo de vida que trasciende una instancia particular en la memoria. Al proporcionar su id a Hibernate, podemos volver a materializar la instancia en un nuevo contexto de persistencia, siempre que la fila asociada esté presente en la base de datos. Por lo tanto, las operaciones `persist()` y `remove()` pueden considerarse como marcadores del inicio y el fin del ciclo de vida de una entidad, al menos en cuanto a persistencia se refiere.

Las operaciones `persist()`, `merge()`, `remove()` y `detach()` son parte del ciclo de vida de una entidad en JPA.

Así, un id representa la identidad persistente de una entidad, una identidad que sobrevive a una instancia particular en la memoria. Y esta es una diferencia importante entre la clase de entidad en sí misma y los valores de sus atributos: la entidad tiene una identidad persistente y un ciclo de vida bien definido en relación con la persistencia, mientras que un `String` o `List` que representa uno de sus valores de atributo no lo tiene.

Una entidad generalmente tiene asociaciones con otras entidades. Típicamente, una asociación entre dos entidades se mapea a una clave foránea en una de las tablas de la base de datos. Estas asociaciones pueden ser de varios tipos: **uno a uno** (`@OneToOne`), **uno a muchos** (`@OneToMany`), **muchos a uno** (`@ManyToOne`), y **muchos a muchos** (`@ManyToMany`). Un grupo de entidades mutuamente asociadas a menudo se denomina modelo de dominio, aunque también es perfectamente válido el término modelo de datos.

Entity classes

Un entidad debe ser **una clase no final** y **tener un constructor no privado y sin parámetros**.

```
@Entity
public class Book {
    @Id
    private Long id;
    private String nombre;

    // Constructor por defecto (necesario para Hibernate)
    public Book() { }

    // Constructor con parámetros
    public Book(Long id, String nombre) {
        this.id = id;
        this.nombre = nombre;
    }

    // Getters y setters
    // ...
}
```

Si una clase de entidad en Hibernate no tiene constructores con parámetros, no es necesario definir explícitamente un constructor por defecto, ya que Java lo proporciona automáticamente.

```
@Entity
public class Book {
    @Id
    private Long id;
    private String nombre;

    // Getters y setters
    // ...
}
```

Por otro lado, la clase de entidad puede ser tanto **concreta** como **abstracta**, y puede tener cualquier cantidad de constructores adicionales. La clase entidad también puede ser una **clase interna estática**.

Cada clase de entidad debe tener la anotación `@Entity`:

```
@Entity
class Book {
    // ...
}
```

Alternativamente, la clase puede identificarse como un tipo de entidad proporcionando una asignación basada en XML para la clase:

```
<entity-mappings>
    <package>org.hibernate.example</package>

    <entity class="Book">
        <attributes> ... </attributes>
    </entity>

    ...
</entity-mappings>
```

Access types

En Hibernate, cada clase entidad puede definirse con un **tipo de acceso predeterminado**, que puede ser:

- **acceso directo a campos o *'field access'***
- **acceso a propiedades o *'property access'***

Esta configuración determina cómo Hibernate accede y maneja los atributos de la clase entidad. Cuando se utiliza **acceso directo a campos**, los atributos de la clase entidad se anotan directamente, permitiendo que Hibernate acceda sin intermediarios a los campos de la clase. En este caso, Hibernate mapea directamente los atributos a los campos correspondientes en la tabla de la base de datos:

```
@Entity
public class Product {

    @Id
    private Long id;

    private String name;

    // Getters y setters
}
```

Cuando se utiliza **acceso a propiedades**, se anota el método *'getter'*, permitiendo que Hibernate acceda a los campos a través de este método, en lugar de acceder directamente a los campos:

```
@Entity
public class Product {

    private Long id;
    private String name;
```

```
    @Id
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Hibernate **determina automáticamente** el tipo de acceso de toda la entidad basándose en la **ubicación** de la primera anotación que encuentre en la clase de entidad:

- Si la primera anotación, como por ejemplo `@Id`, está en un campo, Hibernate utilizará **acceso directo a campos** en toda la clase.
- Si la primera anotación, como por ejemplo `@Id`, está en un método *getter*, Hibernate utilizará **acceso a propiedades** en toda la clase.

Por lo tanto, si no se especifica explícitamente el tipo de acceso utilizando la anotación `@Access`, Hibernate determinará automáticamente el tipo de acceso en función de dónde se encuentran las anotaciones en la clase de entidad.

> La anotación `@Column` es **opcional** en Hibernate. Si no se especifica, Hibernate mapeará automáticamente los campos de la clase a las columnas de la tabla de la base de datos con el mismo nombre. Sin embargo, `@Column` puede ser útil para personalizar el mapeo, como cambiar el nombre de la columna, definir la longitud, establecer si es nullable, entre otros.

Entity class inheritance

Una clase de entidad que no extiende ninguna otra clase de entidad, se llama entidad raíz. Cada entidad raíz debe declarar un atributo de identificador.

Una clase de entidad puede extender otra clase de entidad:

```
// Entidad raíz
@Entity
class Book {
    Book() {}

    @Id
    private Long id;

    private String name;

    // Getters y setters
}

// Entidad subclase
@Entity
class AudioBook extends Book {
    AudioBook() {}
```

```
    // Esta clase hereda los atributos de 'Book'
}
```

Una entidad de subclase hereda **todos** los atributos persistentes de cada entidad que extiende. Además, en este caso Hibernate tratará a las dos clases como entidades con derecho propio, por lo que creará tablas para cada entidad.

Sin embargo, si una clase se anota como `@MappedSuperclass`, Hibernate no la considera una entidad con derecho propio, por lo que no creará la tabla. La subclase que hereda de esta clase sigue heredando los atributos de la clase anotada como `@MappedSuperclass`:

```
// Hibernate no creará una tabla para esta entidad
@MappedSuperclass
class Book {
    Book() {}

    @Id
    private Long id;

    private String name;

    // Getters y setters
}

// Entidad subclase con tabla propia
@Entity
class AudioBook extends Book {
    AudioBook() {}

    // Esta clase hereda los atributos de 'Book'
}
```

Una clase de entidad raíz debe declarar un atributo anotado `@Id` o heredar uno de `@MappedSuperclass`.

Una entidad de subclase siempre hereda el atributo de identificador de la entidad raíz y **no puede declarar su propio atributo `@Id`**.

Identifier attributes

Un atributo de identificador suele ser un campo:

```
@Entity
class Book {
    Book() {}

    @Id
    Long id;

    ...
}
```

Pero puede ser una propiedad:

```
@Entity
class Book {
    Book() {}

    private Long id;

    @Id
```

```
    Long getId() { return id; }
    void setId(Long id) { this.id = id; }

    // ...
}
```

Un atributo de identificador debe estar anotado como `@Id` o `@EmbeddedId`.

Los valores del identificador pueden ser:

- asignado por la aplicación, es decir, por el código Java
- generado y asignado por Hibernate.

Generated identifiers

Un identificador suele ser generado por el sistema, en cuyo caso debe anotarse con `@GeneratedValue`:

```
@Id @GeneratedValue
Long id;
```

JPA define las siguientes estrategias para generar identificadores:

- **GenerationType.UUID**: se utiliza para generar identificadores únicos universales (UUID), que son valores de 128 bits generalmente representados como cadenas en formato estándar. Es útil cuando se necesitan identificadores únicos que sean globalmente únicos, no solo dentro de una tabla o una base de datos específica. El tipo en Java es UUID o String.
- **GenerationType.IDENTITY**: la base de datos genera automáticamente un valor único cada vez que se inserta una nueva fila. El tipo en Java es Long o Integer.
- **GenerationType.SEQUENCE**: utiliza una secuencia de la base de datos para generar valores únicos. El tipo en Java es Long o Integer.
- **GenerationType.TABLE**: utiliza una tabla especial en la base de datos para generar valores únicos. El tipo en Java es Long o Integer.
- **GenerationType.AUTO**: deja que el proveedor de persistencia elija la estrategia de generación más adecuada según la base de datos utilizada. El tipo en Java es Long o Integer.

Las anotaciones `@SequenceGenerator` y `@TableGenerator` permiten un mayor control sobre la generación de SEQUENCE y TABLE respectivamente:

```
@SequenceGenerator(name="bookSeq", sequenceName="seq_book", initialValue = 5, allocationSize=10)
```

Los valores se generan utilizando una secuencia de base de datos definida de la siguiente manera:

```
create sequence seq_book start with 5 increment by 10
```

De hecho, es muy común colocar la anotación `@SequenceGenerator` en el atributo `@Id` que hace uso de ella:

```
@Id
@GeneratedValue(strategy=SEQUENCE, generator="bookSeq")  // reference to generator defined below
```

```
@SequenceGenerator(name="bookSeq", sequenceName="seq_book", initialValue = 5, allocationSize=10)
Long id;
```

JPA proporciona un soporte bastante adecuado para las estrategias más comunes de generación de identificadores por el sistema. Sin embargo, para aquellas ocasiones en que no se adapta a los requisitios, Hibernate proporciona un marco muy bien diseñado para generadores definidos por el usuario.

Natural keys as identifiers

No todos los atributos identificadores se mapean a una clave sustituta (generada por el sistema). Las claves primarias que son significativas para el usuario del sistema se llaman **claves naturales**.

Cuando la clave primaria de una tabla es una clave natural, no anotamos el atributo identificador con `@GeneratedValue`, y es responsabilidad del código de la aplicación asignar un valor al atributo identificador. Únicamente se anota con `@Id`:

```
@Entity
class Book {
    @Id
    String isbn;

    ...
}
```

En resumen:

- **Clave Sustituta (*"Surrogate Key"*)**: es una clave primaria generada por el sistema, como un número de identificación único o un UUID. Este tipo de clave no tiene significado intrínseco para los usuarios del sistema y se utiliza principalmente para identificar de manera única las filas en la tabla de la base de datos. Hibernate u otras tecnologías ORM pueden encargarse de generar automáticamente estos valores.
- **Clave Natural (*"Natural Key"*)**: es una clave primaria que tiene un significado directo y relevante para los usuarios del sistema. Por ejemplo, el número de ISBN de un libro, el número de pasaporte de una persona o el código de producto. Estas claves son definidas o mantenidas por los usuarios o por el dominio del negocio, y no por la aplicación o la base de datos. En este caso, Hibernate no generará automáticamente los valores de estas claves; será responsabilidad del código de la aplicación asignar o gestionar estos valores.

Composite identifiers

Si la base de datos utiliza claves compuestas, necesitará más de un atributo de identificador. Hay dos formas de asignar claves compuestas en JPA:

- **usando un** `@IdClass`
- **usando un** `@EmbeddedId`

La forma recomendada es usar el tipo `@Embeddable` con la clave compuesta:

```
@Embeddable
record BookId(String isbn, int printing) {}
```

Ahora ya se puede utilizar la clave compuesta con `@EmbeddedId`:

```
@Entity
class Book {
    Book() {}

    @EmbeddedId
    BookId bookId;

    ...
}
```

De igual manera, ahora podemos se puede usar la clave compuesta para obtener instancias de la entidad:

```
Book book = session.find(Book.class, new BookId(isbn, printing));
```

Version attributes

Una entidad puede tener un atributo que Hibernate utiliza para la verificación de bloqueo optimista. Un atributo de versión suele ser de tipo `Integer`, `Short`, `Long`, `LocalDateTime`, `OffsetDateTime`, `ZonedDateTime` o `Instant`. Este atributo se conoce comúnmente como **atributo de versión**.

El atributo de versión se utiliza en Hibernate para implementar el bloqueo optimista. Esta técnica asegura que las actualizaciones concurrentes a la misma entidad por múltiples usuarios se gestionen correctamente para evitar la pérdida o inconsistencia de datos.

```
@Version
LocalDateTime lastUpdated;
```

Hibernate asigna automáticamente los atributos de versión cuando una entidad se vuelve persistente y los incrementa o actualiza automáticamente cada vez que se actualiza la entidad.

En otras palabras, cuando se actualiza una entidad con un atributo de versión, Hibernate compara el valor actual del atributo de versión con el valor que se leyó inicialmente al recuperar la entidad de la base de datos.

Si estos valores difieren, indica que otra transacción ha modificado la entidad desde que se leyó, y Hibernate puede decidir cómo manejar la situación (por ejemplo, lanzando una excepción o reintentando la operación).

Natural id attributes

Incluso cuando una entidad tiene una clave sustituta (clave generada por el sistema), siempre debería ser posible identificar una combinación de campos que identifiquen de manera única una instancia de la entidad, desde el punto de vista del usuario del sistema. Esta combinación de campos es su **clave natural**.

Anteriormente, consideramos el caso en que la clave natural coincide con la clave primaria. Aquí, la clave natural es una segunda clave única de la entidad, distinta de su clave primaria sustituta.

Si no se puede identificar una clave natural, podría ser una señal de que el modelo de datos no es correcto.

Dado que es extremadamente común recuperar una entidad basada en su clave natural, Hibernate tiene una manera de marcar los atributos de la entidad que componen su clave natural. Cada atributo debe ser anotado con `@NaturalId`.

```
@Entity
class Book {
    Book() {}
```

```
    @Id @GeneratedValue
    Long id; // the system-generated surrogate key

    @NaturalId
    String isbn; // belongs to the natural key

    @NaturalId
    int printing; // also belongs to the natural key

    ...
}
```

Las claves naturales (`@NaturalId`) permiten búsquedas eficientes y validaciones de unicidad basadas en atributos significativos del modelo de negocio, mientras que las claves primarias o *'surrogate keys'* (`@Id`) se utilizan principalmente para la gestión interna y referencial en la base de datos.

Si se conoce el ID (clave subrogada) y/o los valores de los atributos que forman la clave natural, se puede realizar búsquedas utilizando cualquiera de ellos.

Basic attributes

Un atributo básico de una entidad es un campo o propiedad que se asigna a una sola columna de la tabla de la base de datos asociada. La especificación JPA define un conjunto bastante limitado de tipos básicos:

- **Primitive types**: boolean, int, double, etc
- **Primitive wrappers**: Boolean, Integer, Double, etc
- **Strings**: String
- **Arbitrary-precision numeric types**: BigInteger, BigDecimal
- **Date/time types**: LocalDate, LocalTime, LocalDateTime, OffsetDateTime, Instant
- **Deprecated date/time types**: Date, Calendar
- **Deprecated JDBC date/time types**: Date, Time, Timestamp
- **Binary and character arrays**: byte[], char[]
- **UUIDs**: UUID
- **Enumerated types**: Any enum
- **Serializable types**: Any type which implements java.io.Serializable

Hibernate amplía ligeramente esta lista con los siguientes tipos:

- **Additional date/time types**: Duration, ZoneId, ZoneOffset, Year, and even ZonedDateTime
- **JDBC LOB types**: Blob, Clob, NClob
- **Java class object**: Class
- **Miscellaneous types**: Currency, URL, TimeZone

Estos tipos básicos se mapean **automáticamente** a tipos de columna SQL correspondientes y pueden ser utilizados directamente en las entidades JPA sin necesidad de definiciones adicionales.

La anotación `@Basic` especifica explícitamente que un atributo es básico, pero a menudo no es necesaria, ya que se asume que los atributos son básicos por defecto. Por otro lado, si un atributo de tipo no primitivo no puede ser nulo, se recomienda encarecidamente el uso de `@Basic(optional=false)` :

```
@Basic(optional=false) String firstName;
@Basic(optional=false) String lastName;
String middleName; // may be null
```

Los atributos de tipo primitivo se infieren como 'NOT NULL' de forma predeterminada.

Hay dos formas estándar de agregar una restricción 'NOT NULL' a una columna asignada en JPA:

- **usando** `@Basic(optional=false)`
- **usando** `@Column(nullable=false)`

La diferencia radica que mientras anotaciones como `@Entity` , `@Id` , and `@Basic` pertenecen al dominio del modelo de Java, anotaciones como `@Table` y `@Column` pertenecen a la capa del mapeo y al dominio de la base de datos relacional.

Sin embargo, hay una solución mejor y es utilizar la anotación `@NotNull` de **Bean Validation**. Simplemente hay que agregar **Hibernate Validator** a la compilación del proyecto, como se describe en dependencias opcionales.

Enumerated types

Un tipo enumerado se considera un tipo básico, pero dado que la mayoría de las bases de datos no tienen un tipo *"ENUM"* nativo, JPA proporciona una anotación especial `@Enumerated` para especificar cómo deben representarse los valores enumerados en la base de datos:

- De forma predeterminada, un enumerado se almacena como un entero, el valor de su miembro `ordinal()` , pero
- Pero si el atributo está anotado con `@Enumerated(STRING)` , se almacenará como una cadena, utilizando el valor de su miembro `name()` .

```
// here, an ORDINAL encoding makes sense
@Enumerated
@Basic(optional=false)
DayOfWeek dayOfWeek;

// but usually, a STRING encoding is better
@Enumerated(EnumType.STRING)
@Basic(optional=false)
Status status;
```

En Hibernate 6, un tipo enumerado anotado como `@Enumerated(EnumType.STRING)` se mapea como un *"VARCHAR"* en la mayoría de bases de datos, mientras que se mapea como *"ENUM"* en MySQL.

Converters

En Hibernate, los conversores son utilizados para **transformar datos de un tipo a otro** antes de ser almacenados en la base de datos y después de ser recuperados de la misma.

Estas conversiones son útiles cuando se necesita adaptar tipos de datos que no son nativamente soportados por la base de datos o cuando se desea personalizar la forma en que los datos son representados o interpretados en el modelo de la base de datos.

Un convertidor de atributos en JPA es responsable de:

- Convertir un tipo de dato Java dado a uno de los tipos mencionados anteriormente.
- Realizar cualquier otro tipo de preprocesamiento y postprocesamiento necesario en los valores de atributos básicos antes de escribirlos o leerlos desde la base de datos.

Hay dos formas de aplicar un convertidor:

- la anotación `@Convert` aplica un *"AttributeConverter"* a un atributo de entidad particular, o
- la anotación `@Converter` (o, alternativamente, la anotación `@ConverterRegistration`) registra un "*AttributeConverter*" para su aplicación automática a todos los atributos de un tipo determinado.

Compositional basic types

En Hibernate, un "tipo básico" se forma mediante la unión de dos objetos:

- **JavaType**: representa la semántica de una clase Java específica. Puede comparar instancias de la clase para determinar si un atributo de ese tipo está modificado, generar un código hash útil para la instancia y convertir valores a otros tipos.
- **JdbcType**: representa un tipo SQL entendido por JDBC. Es capaz de leer y escribir un tipo Java único desde y hacia JDBC, utilizando métodos como `setString()` y `getString()` para operaciones de escritura y lectura respectivamente.

Cuando mapeamos un atributo básico, podemos especificar explícitamente un *"JavaType"*, un *"JdbcType"*, o ambos. Sin embargo, para los tipos de Java integrados esto generalmente no es necesario:

```
@JavaType(LongJavaType.class)  // not needed, this is the default JavaType for long
long currentTimeMillis;
```

Si un *"JavaType"* dado no sabe cómo convertir sus instancias al tipo requerido por su *"JdbcType"* asociado, se puede proporcionar un _"

En resumen, Hibernate utiliza JavaType y JdbcType para manejar la conversión entre tipos de datos Java y tipos SQL, permitiendo configuraciones personalizadas mediante anotaciones y convertidores de atributos cuando es necesario.

Embeddable objects

Un objeto embebido es una clase Java cuyo estado se mapea a múltiples columnas de una tabla, pero que no tiene su propia identidad persistente. Es decir, es una clase con atributos mapeados, pero sin un atributo `@Id`.

Un objeto embebido solo puede hacerse persistente asignándolo al atributo de una entidad. Dado que el objeto embebido no tiene su propia identidad persistente, su ciclo de vida con respecto a la persistencia está completamente determinado por el ciclo de vida de la entidad a la que pertenece.

Una clase embebida debe tener la anotación `@Embeddable` en lugar de `@Entity`:

```
@Embeddable
class Name {

    @Basic(optional=false)
```

```
    String firstName;

    @Basic(optional=false)
    String lastName;

    String middleName;

    Name() {}

    Name(String firstName, String middleName, String lastName) {
        this.firstName = firstName;
        this.middleName = middleName;
        this.lastName = lastName;
    }

    ...
}
```

Una clase embebida debe cumplir los mismos requisitos que las clases de entidad, con la excepción de que una clase embebida no tiene el atributo `@Id`. En particular, **debe tener un constructor sin parámetros**.

Alternativamente, se puede definir un tipo embebido como un tipo *"record"* de Java 14:

```
record Name(String firstName, String middleName, String lastName) {}
```

Siguiendo con el ejemplo , ahora se puede usar la clase *"Name"* (o *"record"*) como el tipo de atributo de otra entidad:

```
@Entity
class Author {
    @Id @GeneratedValue
    Long id;

    @Embedded
    Name name;

    ...
}
```

JPA proporciona una anotación `@Embedded` para identificar un atributo de una entidad que hace referencia a un tipo integrable. Esta anotación es completamente **opcional**.

Associations

Una **asociación** es una relación entre entidades. Normalmente clasificamos las asociaciones en función de su multiplicidad. Si E y F son ambas clases de entidad, entonces:

- una **asociación uno a uno** relaciona como máximo una instancia única E con como máximo una instancia única de F,
- una **asociación de muchos a uno** relaciona cero o más instancias de E con una instancia única de F, y
- una **asociación de muchos a muchos** relaciona cero o más instancias de E con cero o más instancias de F.

Una asociación entre clases de entidades puede ser:

- **unidireccional**, navegable de E a F pero no de F a E, o
- **bidireccional** y navegable en cualquier dirección.

Hay tres anotaciones para mapear asociaciones: `@ManyToOne`, `@OneToMany` y `@ManyToMany`.

Many-to-one

Una asociación de muchos a uno o *"many-to-one"* es el tipo de asociación más básica que podemos imaginar. Se asigna de forma completamente natural a una clave externa en la base de datos. Casi todas las asociaciones en su modelo de dominio serán de esta forma.

La anotación `@ManyToOne` marca el lado "a uno" de la asociación, por lo que una asociación unidireccional de muchos a uno se ve así:

```
class Book {
    @Id @GeneratedValue
    Long id;

    @ManyToOne(fetch=LAZY)
    Publisher publisher;

    // ...
}
```

Aquí, la tabla 'BOOK' tiene una columna de clave externa que contiene el identificador del editor asociado.

Un defecto muy desafortunado de JPA es que las asociaciones `@ManyToOne` son de tipo `(fetch=EAGER)` de forma predeterminada y salvo casos excepcionales no es lo recomendable.

La **carga diferida** o *"Lazy Loading"* es una estrategia de Hibernate para **retrasar la carga** de datos hasta que se necesiten. En el contexto de asociaciones *"many-to-one"*, esto significa que las entidades relacionadas no se cargan de inmediato cuando se carga la entidad principal, sino que se cargan solo **cuando se accede a ellas por primera vez** mejorando el rendimiento de la aplicación al reducir la cantidad de datos cargados en la memoria.

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "customer_id")
    private Customer customer;

    // otros campos, getters y setters
}

@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<Order> orders = new HashSet<>();


    // otros campos, getters y setters
}
```

Cuando se recupera una instancia de "ORDER" desde la base de datos, Hibernate no carga inmediatamente la entidad "CUSTOMER". En lugar de eso, crea un **proxy** para "CUSTOMER". El proxy es un objeto que actúa como un sustituto y solo carga la entidad real cuando se accede a uno de sus métodos o propiedades.

```
public void example() {
    // Supongamos que tenemos una instancia de EntityManager llamada em
    Order order = em.find(Order.class, 1L);

    // En este punto, el Customer aún no ha sido cargado desde la base de datos
    System.out.println(order.getId()); // Esto no desencadena la carga del Customer

    // Acceder a una propiedad del Customer desencadena la carga del Customer
    System.out.println(order.getCustomer().getName()); // Aquí se carga el Customer desde la BDD
}
```

La mayoría de las veces, nos gustaría poder navegar fácilmente por nuestras asociaciones en ambas direcciones. Necesitamos una forma de obtener el "CUSTOMER" de un "ORDER" determinado, pero también nos gustaría poder obtener todos los "ORDER" realizados por un "CUSTOMER" determinado.

Para que esta asociación sea bidireccional, debemos agregar un atributo con valor de colección a la clase "CUSTOMER" y anotarlo `@OneToMany`.

Hibernate necesita representar asociaciones no recuperadas en tiempo de ejecución. Por lo tanto, el lado de muchos valores debe declararse usando un tipo de interfaz como `Set` o `List`, y nunca usando un tipo concreto como `HashSet` o `ArrayList`.

Una asociación de *"one-to-many"* asignada a una clave externa nunca puede contener elementos duplicados, por lo que `Set` parece ser el tipo de colección Java semánticamente más correcto para usar y esa es la práctica convencional en la comunidad de Hibernate.

One-to-one (first way)

En una relación `@OneToOne`, cada instancia de una entidad está asociada con exactamente una instancia de otra entidad. Es el tipo más simple de asociación.

La principal diferencia con una asociación `@ManyToOne` es que la columna de clave externa en la tabla también tiene una restricción *UNIQUE*, garantizando que cada valor en la columna sea único y, por lo tanto, manteniendo la relación uno a uno.

Una asociación uno a uno debe anotarse `@OneToOne`:

```
@Entity
class Author {
    @Id @GeneratedValue
    Long id;

    @OneToOne(optional=false, fetch=LAZY)
    Person author;

    // ...
}
```

Aquí, la tabla *"Author"* tiene una columna de clave externa que contiene el identificador de la *"Person"* asociada.

Podemos hacer que esta asociación sea bidireccional agregando una referencia al *"Author"* en la entidad *"Person"*:

```
@Entity
class Person {
    @Id @GeneratedValue
```

```
    Long id;

    @OneToOne(mappedBy = "Author_.PERSON")
    Author author;

    // ...
}
```

La entidad que no tiene el atributo `mappedBy` es la **propietaria de la relación**. El atributo `mappedBy` se usa para indicar que la relación está gestionada por el campo de la otra entidad.

One-to-one (second way)

Podría decirse que una forma más elegante de representar dicha relación es **compartir una clave primaria entre las dos tablas**.

Para utilizar este enfoque, la clase *"Autor"* debe anotarse así:

```
@Entity
class Author {
    @Id
    Long id;

    @OneToOne(optional=false, fetch=LAZY)
    @MapsId
    Person author;

    // ...
}
```

En comparación con el mapeo anterior:

- el atributo `@Id` ya no es `@GeneratedValue` y,
- en cambio, la asociación con `Person` se anota como `@MapsId`.

Esto le permite a Hibernate saber que la asociación con `Person` es la fuente de los valores de clave primaria para `Author`.

Aquí, no hay una columna de clave externa adicional en la tabla `Author`, ya que la columna *"id"* contiene el identificador de `Person`. Es decir, la clave primaria de la tabla `Author` cumple una doble función como clave externa que se refiere a la tabla `Person`.

Many-to-many

Una **asociación unidireccional de muchos a muchos** se representa como un atributo con valor de colección. Siempre se asigna a una tabla de asociación separada en la base de datos (una *"join table"* que contiene las claves foráneas de ambas entidades).

Suele suceder que una asociación de muchos a muchos eventualmente resulte ser una entidad disfrazada.

Se recomienda evitar el uso de `@ManyToMany` desde el principio y, en su lugar, usar una entidad intermedia para representar asociaciones muchos-a-muchos. Esto es porque, a medida que evolucionan los requisitos, puede ser necesario agregar información adicional a la asociación (por ejemplo, el porcentaje de contribución de un autor a un libro). Crear una entidad intermedia desde el inicio (como *"BookAuthorship"* que tendría asociaciones `@OneToMany` con *"Author"* y *"Book"*, y el atributo de contribución) ofrece mayor flexibilidad, facilita el mantenimiento y la escalabilidad del sistema.

Una asociación de muchos a muchos debe anotarse `@ManyToMany`:

```
@Entity
class Book {
    @Id @GeneratedValue
    Long id;

    @ManyToMany
    Set<Author> authors;

    // ...
}
```

Si la asociación es **bidireccional**, agregamos un atributo de apariencia muy similar a *"Book"*, pero esta vez debemos especificar `mappedBy` para indicar que este es el lado sin propietario de la asociación:

```
@Entity
class Book {
    @Id @GeneratedValue
    Long id;

    @ManyToMany(mappedBy=Author_.BOOKS)
    Set<Author> authors;

    // ...
}
```

Nuevamente se ha utilizado el tipo `Set` para representar la asociación. Como antes, se puede utilizar la opción de usar `Collection` o `List`. Pero en este caso sí hay una diferencia en la semántica de la asociación.

Una asociación de muchos a muchos representada como `Collection` o `List` puede contener **elementos duplicados**. Sin embargo, como antes, el orden de los elementos no es persistente. Es decir, la colección es un bolso, no un conjunto.

Collections of basic values and embeddable objects

Las entidades en Java pueden tener colecciones de tipos básicos, como listas de cadenas (`List<String>`) o arrays de enteros (`int[]`). Hibernate y JPA proporcionan anotaciones como `@Array` y `@ElementCollection` para mapear estas colecciones a la base de datos.

Sin embargo, aunque `@Array` y `@ElementCollection` pueden ser útiles en casos específicos, su uso se desaconseja generalmente por las siguientes razones:

- **Complejidad**: las colecciones de elementos básicos pueden complicar el diseño y mantenimiento del esquema de la base de datos.
- **Limitaciones**: algunas bases de datos no soportan el tipo *"ARRAY"*.
- **Buenas prácticas**: las relaciones muchos-a-muchos o uno-a-muchos suelen gestionarse mejor mediante asociaciones de clave foránea entre entidades.

Por tanto, el enfoque recomendado es modelar la colección usando **entidades separadas** y definir una **relación** entre la entidad principal y la nueva entidad.

Por ejemplo, supongamos que se requiere guardar una lista de palabras clave para cada libro. Para ello, primeramente podemos crear una entidad *"Keyword"* para el elemento básico:

```
@Entity
public class Keyword {
```

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

    private String value;

    @ManyToOne
    @JoinColumn(name = "book_id")
    private Book book;

    // getters y setters

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Keyword)) return false;
        Keyword keyword = (Keyword) o;
        return Objects.equals(getValue(), keyword.getValue()) &&
               Objects.equals(getBook(), keyword.getBook());
    }

    @Override
    public int hashCode() {
        return Objects.hash(getValue(), getBook());
    }
}
```

Luego podemos definir la relación con la entidad principal, que es *"Book"*:

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @OneToMany(mappedBy = "book", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Keyword> keywords = new ArrayList<>();

    // getters y setters

    public void addKeyword(String keywordValue) {
        Keyword keyword = new Keyword();
        keyword.setValue(keywordValue);
        keyword.setBook(this);
        keywords.add(keyword);
    }

    public void removeKeyword(Keyword keyword) {
        keywords.remove(keyword);
        keyword.setBook(null);
    }
}
```

Este enfoque tiene ciertas ventajas:

- **Flexibilidad**: permite agregar metadatos adicionales a cada elemento de la colección, es decir, podemos ampliar *"Keyword"* sin que el modelo se vea comprometido.
- **Mantenibilidad**: facilita el mantenimiento y la evolución del esquema de la base de datos.
- **Consistencia**: alineado con las prácticas estándar de diseño de bases de datos relacionales.

Collections mapped to SQL arrays

La anotación `@Array` permite mapear una colección de elementos básicos a una columna de tipo *"ARRAY"* en SQL (si la base de datos lo soporta).

```
@Entity
class Event {
    @Id @GeneratedValue
    Long id;
    ...
    @Array(length=7)
    DayOfWeek[] daysOfWeek;  // stored as a SQL ARRAY type

    // ...
}
```

Sin embargo, como se ha comentado, no es el enfoque recomendado. En la documentación oficial se amplían los motivos.

Collections mapped to a separate table

JPA define una forma estándar de asignar una colección a una tabla auxiliar y es la anotación `@ElementCollection`. Esta anotación permite mapear una colección de elementos básicos o embebidos a una tabla separada:

```
@Entity
class Event {
    @Id @GeneratedValue
    Long id;
    ...
    @ElementCollection
    List<DayOfWeek> daysOfWeek;  // stored in a dedicated table

    // ...
}
```

Sin embargo, como se ha comentado, este tampoco es el enfoque recomendado. En la documentación oficial se amplían los motivos.

equals() and hashCode()

Las clases de entidad deberían sobrescribir `equals()` y `hashCode()`, especialmente cuando las asociaciones se representan como conjuntos (`Set`) y/o variaciones como `HashSet` o `HashMap` ya que estos métodos son esenciales para determinar si dos objetos son iguales y para calcular sus valores hash tanto en Java como Hibernate.

Hay que tener en cuenta que no se debe incluir un campo mutable en `hashCode()`, ya que modificarlo cambiaría su valor hash y podría causar problemas de inconsistencia en las colecciones basadas en `Set`.

Además, aunque técnicamente no está mal incluir un identificador generado por la base de datos en `hashCode()` es arriesgado hacerlo antes de que el identificador sea generado (es decir, antes de que la entidad sea persistida en la base de datos). Esto se debe a que el valor del identificador podría cambiar y afectar la integridad de la colección.

```
@Entity
public class Keyword {

    // ...

    @Override
    public boolean equals(Object o) {
```

```
        if (this == o) return true;
        if (!(o instanceof Keyword)) return false;
        Keyword keyword = (Keyword) o;
        return Objects.equals(getValue(), keyword.getValue()) &&
               Objects.equals(getBook(), keyword.getBook());
    }

    @Override
    public int hashCode() {
        return Objects.hash(getValue(), getBook());
    }
}
```

Object/relational mapping

Dado un modelo de dominio, es decir, una colección de clases de entidad decoradas con todas las anotaciones vistas hasta ahora, Hibernate inferirá un esquema relacional completo, e incluso lo exportará a la base de datos si así se indica.

El esquema resultante será completamente será correcto aunque no perfecto, ya que, por ejemplo cada columna tipo `VARCHAR` tiene la misma longitud, es decir `VARCHAR(255)`.

Sin embargo, este tipo de flujo, denominado **mapeo descendente**, no se ajusta al escenario más común para el uso del mapeo O/R. Es raro que las clases de Java precedan al esquema relacional.

Por lo general, ya se dispone de un esquema relacional y se construye el modelo de dominio en torno a ese esquema. Esto se llama **mapeo ascendente**.

Mapping entity inheritance hierarchies

Existen tres estrategias básicas para asignar una jerarquía de entidades a tablas relacionales:

- **SINGLE_TABLE (por defecto)**: se asigna todas las clases de la jerarquía a la misma tabla. Los atributos de todas las clases de la jerarquía se combinan en esta única tabla, y se usa una columna discriminadora para identificar a qué subclase pertenece cada fila.
- **JOINED**: se asigna cada clase de la jerarquía a una tabla separada, pero cada tabla solo asigna los atributos declarados por la propia clase. Las tablas de las subclases tienen una clave foránea que referencia la tabla de la superclase. Para recuperar un objeto de una subclase, se necesita hacer un *"join"* entre las tablas. En esta estrategia hay normalización de datos, evitando valores nulos innecesarios.
- **TABLE_PER_CLASS**: cada clase concreta de la jerarquía tiene su propia tabla que incluye todos los atributos heredados. No hay tablas para las superclases abstractas, y no se utilizan claves foráneas para relacionar las tablas.

Las tres estrategias de mapeo se enumeran por `InheritanceType`. Especificamos una estrategia de mapeo de herencia usando la anotación `@Inheritance`.

```
@Entity
@Inheritance(strategy=JOINED)
class Person { ... }

@Entity
class Author { ... }
```

Mapping to tables

Las siguientes anotaciones especifican exactamente cómo se asignan los elementos del modelo de dominio a las tablas del modelo relacional:

- `@Table` : asignar una clase de entidad a su tabla principal.
- `@SecondaryTable` : definir una tabla secundaria para una clase de entidad.
- `@Table` : asignar una asociación de muchos a muchos o de muchos a uno a su tabla de asociaciones.
- `@Table` : asignar un `@ElementCollection` a su tabla.

Mapping entities to tables

De forma predeterminada, una entidad se asigna a una sola tabla, que se puede especificar usando `@Table` :

```
@Entity
@Table(name="People")
class Person {
    // ...
}
```

Sin embargo, la anotación `@SecondaryTable` nos permite distribuir sus atributos en varias tablas secundarias, es decir, la anotación `@SecondaryTable` se utiliza para mapear una entidad a múltiples tablas en una base de datos. Esto es útil cuando los datos de una entidad están distribuidos en varias tablas, pero se desea manejar la entidad como una sola unidad en el código Java.

Por ejemplo, supongamos que tenemos una entidad *"Employee"* cuyos datos están distribuidos en dos tablas: *"employee(id as PK, first_name, last_name)"* y *"employee_details(employee_id as FK, address, phone_number)"*:

```
import jakarta.persistence.*;

@Entity
@Table(name = "employee")
@SecondaryTable(name = "employee_details", pkJoinColumns = @PrimaryKeyJoinColumn(name = "employee_id"))
public class Employee {

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

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(table = "employee_details", name = "address")
    private String address;

    @Column(table = "employee_details", name = "phone_number")
    private String phoneNumber;

    // Getters and setters
}
```

La anotación `@Table` tiene argumentos como *"name"*, etcétera…

Sin embargo, **se recomienda no utilizar los argumentos *"schema"* y *"catalog"***. Especificar un *"schema"* o "*catalog*" en el código de la aplicación puede hacer que la aplicación dependa de una estructura específica de la base de datos. Es preferible

manejar la configuración del esquema o catálogo a nivel de la configuración de Hibernate (por ejemplo, en el archivo de propiedades) o en la propia URL JDBC de conexión.

La anotación `@SecondaryTable` también tiene argumentos como *"name"*, etcétera…

En este caso también **se recomienda NO utilizar los argumentos *"schema"* y *"catalog"***.

Mapping associations to tables

La anotación `@JoinTable` especifica una tabla de asociación, es decir, una tabla que contiene las claves externas de ambas entidades asociadas. Esta anotación se utiliza normalmente con asociaciones `@ManyToMany`:

```
@Entity
class Book {
    ...

    @ManyToMany
    @JoinTable(name="BooksAuthors")
    Set<Author> authors;

    // ...
}
```

Pero incluso es posible usarlo para asignar una asociación `@ManyToOne` o `@OneToOne` a una tabla de asociaciones:

```
@Entity
class Book {
    ...

    @ManyToOne(fetch=LAZY)
    @JoinTable(name="BookPublisher")
    Publisher publisher;

    // ...
}
```

Mappings to columns

Estas anotaciones especifican cómo los elementos del modelo de dominio se asignan a columnas de tablas en el modelo relacional:

- `@Column`: asigna un atributo a una columna
- `@JoinColumn`: asigna una asociación a una columna de clave externa
- `@PrimaryKeyJoinColumn`: asigna la clave primaria utilizada para unir una tabla secundaria con su tabla principal o una tabla de subclase en herencia JOINED con su tabla de clase raíz.
- `@OrderColumn`: especifica una columna que debe usarse para mantener el orden de una *"List"*.
- `@MapKeyColumn`: se especificó una columna que se debe usar para conservar las claves de un *"Map"*.

Mapping basic attributes to columns

La anotación `@Column` no solo es útil para especificar el nombre de la columna.

Esta anotación acepta una serie de atributos que puede consultarse aquí.

```
@Entity
@Table(name="Books")
@SecondaryTable(name="Editions")
class Book {
    @Id @GeneratedValue
    @Column(name="bookId") // customize column name
    Long id;

    @Column(length=100, nullable=false) // declare column as VARCHAR(100) NOT NULL
    String title;

    @Column(length=17, unique=true, nullable=false) // declare column as VARCHAR(17) NOT NULL UNIQUE
    String isbn;

    @Column(table="Editions", updatable=false) // column belongs to the secondary table, and is never updated
    int edition;
}
```

Mapping associations to foreign key columns

La anotación `@JoinColumn` se utiliza para personalizar una columna de clave externa.

Esta anotación acepta una serie de atributos que puede consultarse aquí.

Una columna de clave externa no necesariamente tiene que hacer referencia a la clave primaria de la tabla a la que se hace referencia. Es bastante aceptable que la clave externa haga referencia a cualquier otra clave única de la entidad a la que se hace referencia, incluso a una clave única de una tabla secundaria.

```
@Entity
@Table(name="Items")
class Item {
    // ...

    @ManyToOne(optional=false)  // implies nullable=false
    @JoinColumn(name = "bookIsbn", referencedColumnName = "isbn",  // a reference to a non-PK column
                foreignKey = @ForeignKey(name="ItemsToBooksBySsn")) // supply a name for the FK constraint
    Book book;

    // ...
}
```

Si no proporciona un nombre explícito usando `@ForeignKey`, Hibernate generará un nombre bastante feo.

Mapping primary key joins between tables

La anotación `@PrimaryKeyJoinColumn` es una anotación de propósito especial para mapeo:

- la columna de clave primaria de `@SecondaryTable`, que también es una clave externa que hace referencia a la tabla principal,
- la columna de clave primaria de la tabla principal asignada por una subclase en una jerarquía de herencia `JOINED`, que también es una clave externa que hace referencia a la tabla principal asignada por la entidad raíz.

Al mapear la clave primaria de una tabla de subclase, colocamos la anotación @PrimaryKeyJoinColumn en la clase de entidad:

```
@Entity
@Table(name="People")
@Inheritance(strategy=JOINED)
class Person { ... }

@Entity
@Table(name="Authors")
@PrimaryKeyJoinColumn(name="personId") // the primary key of the Authors table
class Author { ... }
```

Pero para asignar una clave primaria de una tabla secundaria, la anotación `@PrimaryKeyJoinColumn` debe ocurrir dentro de la anotación `@SecondaryTable`:

```
@Entity
@Table(name="Books")
@SecondaryTable(name="Editions",
                pkJoinColumns = @PrimaryKeyJoinColumn(name="bookId")) // the primary key of the Editions table
class Book {
    @Id @GeneratedValue
    @Column(name="bookId") // the name of the primary key of the Books table
    Long id;

    ...
}
```

Column lengths and adaptive column types

Hibernate ajusta automáticamente el tipo de columna utilizado en el DDL generado en función de la longitud de la columna especificada por la anotación `@Column`.

Por lo tanto, normalmente no necesitaremos especificar explícitamente que una columna debe ser de un tipo determinado porque Hibernate seleccionará automáticamente el tipos si es necesario para acomodar una cadena de la longitud que especifiquemos.

- **DEFAULT** → 255
- **LONG** → 32600
- **LONG16** → 32767
- **LONG32** → 2147483647

```
@Column(length=LONG)
String text;

@Column(length=LONG32)
byte[] binaryData;
```

LOBs

JPA proporciona una anotación `@Lob` que especifica que un campo debe persistirse como un BLOB o CLOB.

Los controladores JDBC son perfectamente capaces de convertir entre `String y CLOB` o entre `byte[] y BLOB`. Por lo tanto, a menos que se necesite usar específicamente estas API de LOB de JDBC, no se necesita la anotación `@Lob`.

Todo lo que necesita es especificar una longitud de columna lo suficientemente grande para acomodar los datos que planea escribir en esa columna.

```
@Column(length=LONG32) // good, correct column type inferred
String text;

@Lob // almost always unnecessary
String text;
```

Esto es particularmente cierto para **PostgreSQL** ya que no tiene un tipo de dato específico para BLOBs y CLOBs por lo que se pueden producir errores cuando se usa la anotación `@Lob` con **PostgreSQL**.

Finalmente, como alternativa, Hibernate permite declarar un atributo de tipo `java.sql.Blob` o `java.sql.Clob`:

```
@Entity
class Book {
    ...
    Clob text;
    Blob coverArt;
    ....
}
```

Mapping embeddable types to UDTs or to JSON

Hay un par de formas alternativas de representar un tipo embebido en el lado de la base de datos.

Embeddables as UDTs

Primero, una opción realmente interesante, al menos en el caso de los tipos `record` de Java y para las bases de datos que soportan **tipos definidos por el usuario (UDTs)**, es definir un UDT que represente el tipo `record`.

Solo se necesita anotar el tipo `record`, o el atributo que contiene una referencia a él, con la nueva anotación `@Struct`:

```
@Embeddable
@Struct(name="PersonName")
record Name(String firstName, String middleName, String lastName) {}
```

```
@Entity
class Person {
    // ...
    Name name;
    // ...
}
```

Lo que resulta en el siguiente UDT:

```
create type PersonName as (firstName varchar(255), middleName varchar(255), lastName varchar(255))
```

Embeddables to JSON

Una segunda opción disponible es mapear el tipo embebible a una columna JSON. Ahora bien, esto no es algo que recomendable si se está definiendo un modelo de datos desde cero, pero es al menos útil para mapear tablas preexistentes con

columnas de tipo JSON. Dado que los tipos embebibles son anidables, podemos mapear algunos formatos JSON de esta manera, e incluso consultar propiedades JSON usando HQL.

Para mapear un atributo de tipo embebible a JSON, debemos anotar el atributo con `@JdbcTypeCode(SqlTypes.JSON)`, en lugar de anotar el tipo embebible.

```
@Embeddable
record Name(String firstName, String middleName, String lastName) {}
```

```
@Entity
class Person {
    // ...
    @JdbcTypeCode(SqlTypes.JSON)
    Name name;
    // ...
}
```

También se necesita agregar **Jackson** o una implementación de JSONB al classpath en tiempo de ejecución.

Adding constraints

Hibernate añade ciertas *constraints* a la DDL generada automáticamente: restricciones de clave primaria, restricciones de clave foránea y algunas restricciones únicas. Pero es común necesitar:

- **Agregar restricciones únicas adicionales,**
- **Agregar restricciones de verificación (check constraints), o**
- **Personalizar el nombre de una restricción de clave foránea.**

Hay dos formas de agregar una restricción única a una tabla:

- usando `@Column(unique=true)` para indicar una clave única de una sola columna, o
- usando la anotación `@UniqueConstraint` para definir una restricción de unicidad en una combinación de columnas.

```
@Entity
@Table(uniqueConstraints=@UniqueConstraint(columnNames={"title", "year", "publisher_id"}))
class Book { ... }
```

La anotación `@Check` agrega una restricción de verificación a la tabla.

```
@Entity
@Check(name="ValidISBN", constraints="length(isbn)=13")
class Book { ... }
```

La anotación `@Check` se usa comúnmente a nivel de campo:

```
@Id @Check(constraints="length(isbn)=13")
String isbn;
```

Interacting with the database

Para interactuar con la base de datos, es decir, para ejecutar consultas o para insertar, actualizar o eliminar datos, se necesita una instancia de uno de los siguientes objetos:

- **EntityManager de JPA**
- **Session de Hibernate**
- **StatelessSession de Hibernate**

La interfaz `Session` extiende `EntityManager`, por lo que la única diferencia entre las dos interfaces es que `Session` ofrece algunas operaciones adicionales.

En Hibernate, cada `EntityManager` es una `Session`, y puede convertirse utilizando el método `unwrap()` de JPA para obtener la implementación subyacente de JPA:

```
Session session = entityManager.unwrap(Session.class);
```

Una instancia de `Session` (o de `EntityManager`) es una **sesión con estado**. Esta sesión media la interacción entre la aplicación y la base de datos a través de operaciones en un contexto de persistencia.

La sesión con estado realiza un seguimiento de las entidades y sus estados (nuevo, persistente, separado, eliminado) a lo largo de la transacción.

El **contexto de persistencia** es una memoria intermedia que almacena temporalmente las entidades mientras se realizan operaciones de base de datos, lo que permite optimizar el rendimiento mediante el almacenamiento en caché de primer nivel y la agrupación de operaciones.

La interfaz `StatelessSession` no tiene un contexto de persistencia.

Persistence Contexts

El **contexto de persistencia** es un concepto fundamental en Hibernate y JPA que gestiona las entidades que están en un ciclo de vida de persistencia dentro de una sesión.

Es un área de memoria gestionada por el `EntityManager` (o la `Session` en Hibernate) donde se almacenan y se controlan las entidades que se están gestionando durante la ejecución de una transacción.

Actúa como una especie de **"caché de primer nivel"**, para distinguirla de la caché de segundo nivel. Para cada instancia de entidad leída desde la base de datos dentro del ámbito de un contexto de persistencia, y para cada nueva entidad hecha persistente dentro del ámbito del contexto de persistencia, el contexto mantiene una asignación única del identificador de la instancia de entidad a la instancia en sí.

Una instancia de entidad puede estar en uno de tres estados con respecto a un contexto de persistencia dado:

- **Transitorio (*Transient*)**: nunca ha sido persistente y no está asociada con el contexto de persistencia. Estas entidades son nuevas y no han sido guardadas en la base de datos.
- **Persistente (*Persistent*)**: actualmente asociada con el contexto de persistencia. Cualquier cambio realizado en una entidad persistente se sincronizará automáticamente con la base de datos al final de la transacción o cuando se invoque `flush()`.
- **Separado (*Detached*)**: anteriormente persistente en otra sesión, pero no actualmente asociada con este contexto de persistencia. Pueden ser reconectadas a un nuevo contexto de persistencia si es necesario.

En cualquier momento dado, **una instancia puede estar asociada como máximo a un contexto de persistencia**.

La duración de un contexto de persistencia generalmente corresponde a **la duración de una transacción**. Una vez que la transacción se completa, el contexto de persistencia se cierra y todas las entidades gestionadas por él se vuelven separadas (*detached*).

> Un contexto de persistencia **no debe ser compartido entre múltiples hilos o transacciones concurrentes** debido a problemas de concurrencia y seguridad de subprocesos.

Hay diversas razones para valorar los contextos de persistencia:

- Los contextos de persistencia ayudan a **evitar la duplicidad de datos**. Si modificamos una entidad en una sección del código, cualquier otra parte del código que se ejecute dentro del mismo contexto de persistencia verá nuestra modificación.
- Después de modificar una entidad, no es necesario realizar ninguna operación explícita para pedir a Hibernate que propague ese cambio a la base de datos. En su lugar, el cambio se sincronizará automáticamente con la base de datos cuando **la sesión se vacíe** *('flush')*.
- Los contextos de persistencia pueden mejorar el rendimiento evitando viajes a la base de datos cuando se solicita repetidamente una instancia de entidad en una unidad de trabajo dada.
- Hacen posible agrupar de manera transparente múltiples operaciones de base de datos, lo cual puede optimizar el rendimiento.

Sin embargo, un contexto de persistencia también tiene ciertas restricciones:

- Los contextos de persistencia no son seguros para subprocesos y no pueden ser compartidos entre subprocesos. Compartir un contexto de persistencia entre múltiples subprocesos puede llevar a corrupción de datos y problemas de concurrencia.
- Un contexto de persistencia no puede ser reutilizado en transacciones no relacionadas, ya que eso rompería la aislación y atomicidad de las transacciones. Cada transacción debe tener su propio contexto de persistencia para garantizar la coherencia de los datos.

Creating a session

Utilizando solo las API definidas por JPA, tal y como se describe en la configuración usando XML de JPA, se puede obtener un `EntityManagerFactory`:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("my-persistence-unit");
EntityManager entityManager = emf.createEntityManager();
```

Cuando se finaliza con el `EntityManager` se debe cerrar explícitamente para liberar los recursos:

```
entityManager.close();
```

Por otro lado, si se parte de una `SessionFactory`, vista en la configuración usando la API de Hibernate, se puede hacer algo así:

```
Session session = sessionFactory.openSession();
```

Al finalizar, también se debe cerrar la sesión para liberar los recursos:

```
session.close();
```

Injecting the EntityManager

Si se está escribiendo código para algún tipo de entorno de contenedor como Quarkus, probablemente se obtendrá `EntityManager` mediante algún tipo de inyección de dependencia.

Por ejemplo, en el **contexto de Spring**, especialmente con Spring Boot, el framework se encarga de gestionar el ciclo de vida del `EntityManager` y puede ser inyectado directamente en los componentes de servicio usando `@Autowired`.

Spring se encarga de crear y gestionar esta instancia. La anotación `@Transactional` asegura que las operaciones de base de datos se realicen dentro de una transacción. Finalmente, Spring se encarga de abrir y cerrar la transacción y el `EntityManager` automáticamente.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import jakarta.persistence.EntityManager;

@Service
public class MyService {

    @Autowired
    private EntityManager entityManager;

    @Transactional
    public void performDatabaseOperations() {
        MyEntity entity = new MyEntity();
        entity.setId(1L);
        entity.setName("Test Name");

        entityManager.persist(entity);

        MyEntity retrievedEntity = entityManager.find(MyEntity.class, 1L);
        retrievedEntity.setName("Updated Name");

        entityManager.merge(retrievedEntity);
    }
}
```

Managing transactions

Para controlar transacciones de base de datos utilizando las APIs estándar de JPA, la interfaz `EntityTransaction` proporciona los métodos necesarios. El patrón recomendado para manejar transacciones con JPA es el siguiente:

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
EntityTransaction tx = entityManager.getTransaction();
try {
    tx.begin();
    // do some work
    // ...
    tx.commit();
}
catch (Exception e) {
    if (tx.isActive()) tx.rollback();
    throw e;
}
finally {
```

```
    entityManager.close();
}
```

Usando las APIs nativas de Hibernate, podríamos escribir algo muy similar a lo que haríamos con JPA estándar, pero dado que este tipo de código puede ser extremadamente tedioso, Hibernate nos ofrece una opción mucho más conveniente:

```
sessionFactory.inTransaction(session -> {
    // do the work
    // ...
});
```

JPA no tiene una forma estándar de establecer el tiempo de espera de la transacción, pero Hibernate sí la tiene:

```
session.getTransaction().setTimeout(30); // 30 seconds
```

Operations on the persistence context

Las siguientes operaciones de `EntityManager` permiten interactuar con el contexto de persistencia y programar modificaciones en los datos:

- **`persist(Object)`** : hace que un objeto transitorio (nuevo) sea persistente, programando una instrucción SQL INSERT para su ejecución futura.
- **`remove(Object)`** : hace que un objeto persistente sea transitorio, programando una instrucción SQL DELETE para su ejecución futura.
- **`merge(Object)`** : copia el estado de un objeto desasociado a una instancia persistente gestionada correspondiente y devuelve el objeto persistente. Se utiliza para actualizar el estado de un objeto desasociado y sincronizarlo con el contexto de persistencia.
- **`detach(Object)`** : desasocia un objeto persistente de la sesión sin afectar la base de datos de modo que los cambios realizados en el objeto no se sincronicen con la base de datos.
- **`clear()`** : vacía el contexto de persistencia y desasocia todas sus entidades. Se utiliza para limpiar el contexto de persistencia, eliminando todas las entidades gestionadas.
- **`flush()`** : detecta los cambios realizados en los objetos persistentes asociados con la sesión y sincroniza el estado de la base de datos con el estado de la sesión ejecutando instrucciones SQL INSERT, UPDATE y DELETE. Se utiliza para sincronizar el estado del contexto de persistencia con la base de datos de inmediato.

Tenga en cuenta que `persist()` y `remove()` no tienen un efecto inmediato en la base de datos y, en cambio, simplemente programan un comando para su ejecución posterior.

Por otro lado, excepto `getReference()` , todas las operaciones siguientes dan como resultado un acceso inmediato a la base de datos:

- **`find(Class<T> entityClass, Object primaryKey)`** : obtiene una instancia persistente de una entidad dada su clase y su identificador. Si la entidad no está en la base de datos, devuelve null.
- **`find(Class<T> entityClass, Object primaryKey, LockModeType lockMode)`** : obtiene una instancia persistente de una entidad dada su clase y su identificador pero permite especificar un modo de bloqueo optimista o pesimista al recuperar la entidad.

- `get(Class<T> entityType, Object id)` : devuelve la instancia persistente de la clase de entidad dada con el identificador proporcionado, o `null` si no existe tal instancia persistente. Si la instancia ya está asociada con la sesión, se devuelve esa instancia. Este método nunca devuelve una instancia no inicializada.

- `getReference(Class<T> entityClass, Object primaryKey)` : obtiene una referencia a una entidad persistente dada su clase y su identificador, sin cargar realmente su estado desde la base de datos. Devuelve un proxy de la entidad. La entidad no se carga hasta que se acceda a ella. Esto es útil para manejar entidades que se espera que se carguen en el futuro.

- `getReference(Object entity)` : obtiene una referencia a una entidad persistente con la misma identidad que la instancia desasociada dada, sin cargar su estado desde la base de datos.

- `refresh(Object entity)` : refresca el estado persistente de una entidad mediante una nueva consulta SQL para recuperar su estado actual desde la base de datos. Sincroniza el estado de la entidad con la base de datos, sobreescribiendo cualquier cambio no sincronizado.

- `refresh(Object entity, LockModeType lockMode)` : igual que el anterior pero permite especificar un modo de bloqueo optimista o pesimista al refrescar la entidad.

- `lock(Object entity, LockModeType lockMode)` : obtiene un bloqueo optimista o pesimista en una entidad persistente para manejar la concurrencia.

En el contexto de JPA y Hibernate, una vez que **ocurre una excepción** durante la interacción con la base de datos, el estado de la sesión (o el contexto de persistencia) puede quedar inconsistente.

Después de una excepción, es recomendable **cerrar la sesión y abrir una nueva**.

Cascading persistence operations

En el contexto de JPA y Hibernate, cuando hablamos de asociaciones entre entidades, es común que el ciclo de vida de una entidad hija esté completamente dependiente del ciclo de vida de una entidad padre. Esto es especialmente relevante en las asociaciones de muchos a uno (many-to-one) y uno a uno (one-to-one).

El uso de ***cascading*** en JPA y Hibernate permite propagar operaciones desde una entidad principal a las entidades asociadas automáticamente, simplificando la gestión del ciclo de vida de estas entidades.

Para configurar el ***cascading***, se especifica el atributo `cascade` en las anotaciones de asociación como `@OneToMany` , `@OneToOne` , `@ManyToMany` , o `@ManyToOne` . En relaciones uno a uno o muchos a uno, se puede usar `orphanRemoval=true` en las anotaciones. Esto asegura que cualquier entidad hija que ya no esté asociada con la entidad padre sea eliminada automáticamente.

```
@Entity
class Order {
    ...
    @OneToMany(mappedby=Item_.ORDER,
               // cascade persist(), remove(), and refresh() from Order to Item
               cascade={PERSIST,REMOVE,REFRESH},
               // also remove() orphaned Items
               orphanRemoval=true)
    private Set<Item> items;
    ...
}
```

Hay varios tipos de ***cascading***:

- **CascadeType.PERSIST**: Propaga la operación de persistencia. Cuando se persiste una entidad principal, las entidades asociadas también se persisten automáticamente.

- **CascadeType.MERGE**: Propaga la operación de fusión. Cuando se fusiona una entidad principal, las entidades asociadas también se fusionan.
- **CascadeType.REMOVE**: Propaga la operación de eliminación. Cuando se elimina una entidad principal, las entidades asociadas también se eliminan automáticamente.
- **CascadeType.REFRESH**: Propaga la operación de actualización. Cuando se actualiza una entidad principal, las entidades asociadas también se actualizan.
- **CascadeType.DETACH**: Propaga la operación de desasociación. Cuando una entidad principal se desasocia del contexto de persistencia, las entidades asociadas también se desasocian.
- **CascadeType.ALL**: Incluye todas las operaciones de cascada mencionadas anteriormente.

Flushing the session

Flush es el proceso mediante el cual Hibernate sincroniza el estado de las entidades que han sido modificadas en la memoria con el estado persistente en la base de datos. Esto implica la ejecución de sentencias SQL INSERT, UPDATE y DELETE para asegurar que las modificaciones hechas en el contexto de persistencia se reflejen en la base de datos.

Por defecto, *flush* se activa en las siguientes situaciones:

- Cuando se hace un COMMIT de la transacción.
- Antes de la ejecución de una consulta.
- Cuando se llama directamente a `flush()`.

Este comportamiento se puede controlar **configurando explícitamente el modo de descarga**. En JPA se modifica de esta forma:

```
entityManager.setFlushMode(FlushModeType.COMMIT);
```

- **FlushModeType.COMMIT**: *flush* antes de confirmar la transacción
- **FlushModeType.AUTO**: *flush* antes de confirmar la transacción y antes de la ejecución de una consulta cuyos resultados podrían verse afectados por modificaciones mantenidas en la memoria.

Por otro lado, en Hibernate se hace de esta forma:

```
session.setHibernateFlushMode(FlushMode.MANUAL);
```

- **FlushMode.MANUAL**: nunca se hace *flush* de forma automática
- **FlushMode.COMMIT**: *flush* antes de confirmar la transacción
- **FlushMode.AUTO**: *flush* antes de confirmar la transacción y antes de la ejecución de una consulta cuyos resultados podrían verse afectados por modificaciones mantenidas en la memoria.
- **FlushMode.ALWAYS**: *flush* antes de confirmar la transacción y antes de la ejecución de cada consulta.

Dado que el *flush* es una operación algo costosa, configurar el modo de vaciado en COMMIT ocasionalmente puede ser una optimización útil.

Reducir el costo de las operaciones de *flush* en Hibernate puede ser crucial para mejorar el rendimiento, especialmente en aplicaciones con grandes volúmenes de datos o en entornos con alta concurrencia. Otra forma efectiva de lograr esto es cargando **entidades en modo de solo lectura**:

- `Session.setDefaultReadOnly(boolean)` : establece si todas las entidades cargadas por una sesión deben ser tratadas como solo lectura por defecto.
- `SelectionQuery.setReadOnly(boolean)` : configura si las entidades devueltas por una consulta específica deben ser tratadas como solo lectura.
- `Session.setReadOnly(Object, boolean)` : cambia el estado de una entidad específica que ya ha sido cargada por la sesión para que sea tratada como de solo lectura o no.

Queries

Hibernate ofrece tres formas complementarias de escribir consultas:

- **Hibernate Query Language (HQL)**
  - HQL es un superconjunto extremadamente poderoso de JPQL (Java Persistence Query Language).
  - Abstrae la mayoría de las características de los dialectos modernos de SQL.
  - Permite escribir consultas en un lenguaje orientado a objetos que trabaja directamente con las entidades de Hibernate, en lugar de trabajar con tablas y columnas de la base de datos.
- **JPA Criteria Query API**
  - Esta API, junto con las extensiones de Hibernate, permite construir casi cualquier consulta HQL de forma programática.
  - Debido a que utiliza una API basada en clases Java, permite la verificación de tipos en tiempo de compilación, evitando errores comunes de consulta.
- **Consultas SQL nativas**
  - Permite utilizar todas las características del dialecto SQL específico de la base de datos utilizada.
  - En algunos casos, las consultas SQL nativas pueden ser más eficientes que las consultas generadas por Hibernate.

HQL queries

Las consultas se realizan mediante la API `Session` de Hibernate o `EntityManager` según el estándar JPA.

> 👀 La interfaz `Session` de Hibernate hereda de la interfaz `EntityManager` de JPA

- **Consultas de SELECCIÓN**

  Las consultas de selección son aquellas que retornan **una lista de resultados y sin modificar los datos** en la base de datos. Estas consultas generalmente comienzan con las palabras clave `select` o `from`.

  - Ejemplo con método `createSelectionQuery(String, Class)` de la interfaz `QueryProducer` de la cual hereda `Session` de la API de Hibernate:

```
// Consulta HQL de selección
List<Book> matchingBooks =
  session.createSelectionQuery("from Book where title like :titleSearchPattern", Book.class)
```

```
        .setParameter("titleSearchPattern", titleSearchPattern)
        .getResultList();
```

- Ejemplo con método `createQuery(String, Class)` de la interfaz `EntityManager` de la API estándar JPA:

```
List<Book> matchingBooks =
  entityManager.createQuery("select b from Book b where b.title like :titleSearchPattern", Book.class)
      .setParameter("titleSearchPattern", titleSearchPattern)
      .getResultList();
```

- Para ejecutar la consulta se utilizan métodos de la interfaz `SelectionQuery<R>` de Hibernate o `Query` del estándar JPA como son `getResultList()` o `getSingleResult()`:

```
// Crear un objeto de la interfaz 'TypedQuery', que es un subtipo de 'Query'
TypedQuery<Book> query =
              entityManager.createQuery("select b from Book b where b.title like :titleSearchPattern", Book.class)
                      .setParameter("titleSearchPattern", titleSearchPattern);

// Ejecutar la consulta sobre el objeto 'query'
Book book = query.getSingleResult();

// Ejecutar la consulta sobre el objeto 'query'
List<Book> books = query.getResultList();
```

- **Consultas de MUTACIÓN**

  Las consultas de mutación son aquellas que **modifican datos y retornan el número de filas afectadas**. Estas consultas generalmente comienzan con las palabras clave `insert`, `update` o `delete`.

  - Ejemplo con método `createMutationQuery(String)` de la interfaz `QueryProducer` de la cual hereda `Session` de la API de Hibernate:

```
// Consulta HQL de mutación
int updatedRows = session.createMutationQuery("update Book set price = :newPrice where title = :title")
      .setParameter("newPrice", newPrice)
      .setParameter("title", title)
      .executeUpdate();
```

- Ejemplo con método `createQuery(String)` de la interfaz `EntityManager`) de la API estándar JPA:

```
// Consulta JPA de mutación
int updatedRows = entityManager.createQuery("update Book b set b.price = :newPrice where b.title = :title")
      .setParameter("newPrice", newPrice)
      .setParameter("title", title)
      .executeUpdate();
```

- Para ejecutar la consulta se utilizan métodos de la interfaz `MutationQuery` de Hibernate o `Query` del estándar JPA:

```
// Crear un objeto de la interfaz 'Query'
Query updateQuery =
  entityManager.createQuery("update Book b set b.price = :newPrice where b.title = :title")
    .setParameter("newPrice", newPrice)
    .setParameter("title", title);
```

```
    // Ejecutar el update
    int updatedRows = updateQuery.executeUpdate();
```

Named parameters

En las consultas anteriores, tanto `:titleSearchPattern` como `:newPrice` o `:title` son **parámetros con nombre**.

También es posible identificar parámetros mediante un número. Estos parámetros se llaman **parámetros ordinales**.

```
List<Book> matchingBooks =
        session.createSelectionQuery("from Book where title like ?1", Book.class)
            .setParameter(1, titleSearchPattern)
            .getResultList();
```

> ⚠ **Nunca se debe concatenar la entrada del usuario con HQL** y pasar la cadena concatenada a `createSelectionQuery()`. Esto abriría la posibilidad de **ataques por Inyección SQL** y la ejecución de código arbitrario en el servidor de base de datos.

```
// Inyección SQL
String unsafeQuery = "from Book where title like '" + userInput + "'";
List<Book> books = session.createQuery(unsafeQuery, Book.class).getResultList();
```

Para prevenir estos ataques, **hay que usar parámetros con nombre o parámetros ordinales** ya que Hibernate maneja de forma segura el valor proporcionado por el usuario escapando cualquier carácter no seguro.

Criteria queries

En una interfaz de búsqueda, donde los usuarios pueden filtrar resultados basándose en varios criterios, como el título del libro o el nombre del autor, se puede construir una consulta HQL para obtener estos resultados. Sin embargo, construir consultas HQL mediante concatenación de cadenas puede ser frágil y propenso a errores. En su lugar, utilizar consultas basadas en criterios ofrece una alternativa más robusta y flexible.

En Hibernate 6, cada consulta HQL se compila en una consulta de criterios antes de ser traducida a SQL. Esto garantiza que la semántica de HQL y las consultas de criterios sean idénticas.

Primero se necesita un objeto para construir consultas de criterios. Utilizando las API estándar de JPA, este objeto es un `CriteriaBuilder`, y se obtiene a partir del `EntityManagerFactory`:

```
CriteriaBuilder builder = entityManagerFactory.getCriteriaBuilder();
```

Pero si se está trabajando con un `SessionFactory`, se obtiene un `HibernateCriteriaBuilder`, que extiende de `CriteriaBuilder` y añade más operaciones que JPQL no tiene:

```
HibernateCriteriaBuilder builder = sessionFactory.getCriteriaBuilder();
```

Sin embargo, si se está trabajando con `EntityManagerFactory`, es posible obtener un `HibernateCriteriaBuilder` de dos formas:

```
// Forma 1
HibernateCriteriaBuilder builder =
        entityManagerFactory.unwrap(SessionFactory.class).getCriteriaBuilder();
```

```
// Forma 2
HibernateCriteriaBuilder builder =
        (HibernateCriteriaBuilder) entityManagerFactory.getCriteriaBuilder();
```

A partir del objeto `HibernateCriteriaBuilder` o `CriteriaBuilder` se crea un objeto que corresponde con el tipo de consulta:

- **`CriteriaQuery`** : para consultas de selección

```
// Consulta de selección
CriteriaQuery<Book> criteriaQuery = criteriaBuilder.createQuery(Book.class);
Root<Book> root = criteriaQuery.from(Book.class);

// Definir las condiciones
criteriaQuery.select(root).where(criteriaBuilder.like(root.get("title"), "%Java%"));
```

- **`CriteriaUpdate`** : para modificaciones de registros

```
// Consulta de actualización
CriteriaUpdate<Book> criteriaUpdate = criteriaBuilder.createCriteriaUpdate(Book.class);
Root<Book> root = criteriaUpdate.from(Book.class);

// Definir la actualización
criteriaUpdate.set("price", 29.99)
           .where(criteriaBuilder.equal(root.get("title"), "Hibernate in Action"));
```

- **`CriteriaDelete`** : para el borrado de registros

```
// Consulta de eliminación
CriteriaDelete<Book> criteriaDelete = criteriaBuilder.createCriteriaDelete(Book.class);
Root<Book> root = criteriaDelete.from(Book.class);

// Definir la eliminación
criteriaDelete.where(criteriaBuilder.equal(root.get("title"), "Old Book Title"));
```

Una vez se ha obtenido la consulta y se ha aplicado el predicado, se ejecutan como las consultas HQL:

```
List<Book> matchingBooks =
        session.createSelectionQuery(criteriaQuery)
            .getResultList();
```

Las consultas de mutación funcionan de forma similar:

```
int rowsAffected = session.createMutationQuery(criteriaDelete).executeUpdate();
```

Native SQL queries

Las consultas se realizan mediante la API `Session` de Hibernate o `EntityManager` según el estándar JPA:

- **Consultas de SELECCIÓN de datos**

- Se utiliza el método `createNativeQuery(String, Class)` de la interfaz `QueryProducer` de la cual hereda `Session` de la API de Hibernate.
- Se utiliza el método `createNativeQuery(String, Class)` de la interfaz `EntityManager` de la API estándar JPA.
- Para ejecutar la consulta se utilizan métodos de la interfaz `NativeQuery<T>` de Hibernate o `Query` del estándar JPA como son `getResultList()` o `getSingleResult()`.

- **Consultas de MUTACIÓN de datos**
  - Se utiliza el método `createNativeMutationQuery(String)` de la interfaz `QueryProducer` de la cual hereda `Session` de la API de Hibernate.
  - Se utiliza el método `createNativeQuery(String)` de la interfaz `EntityManager` de la API estándar JPA.
  - Para ejecutar la consulta se utiliza el método `executeUpdate()` de la interfaz `MutationQuery` de Hibernate o `Query` del estándar JPA.

Para los casos más simples, Hibernate puede inferir la forma del conjunto de resultados:

```
Book book =
        session.createNativeQuery("select * from Books where isbn = ?1", Book.class)
            .getSingleResult();

String title =
        session.createNativeQuery("select title from Books where isbn = ?1", String.class)
            .getSingleResult();
```

Sin embargo, para casos más complicados, se necesita usar la anotación `@SqlResultSetMapping` para definir un mapeo con nombre, y pasar ese nombre a `createNativeQuery()`.

Por defecto, **Hibernate no hace un *flush* de la sesión** antes de la ejecución de una consulta nativa. Esto se debe a que la sesión no está al tanto de qué modificaciones mantenidas en memoria afectarían los resultados de la consulta.

Hay dos maneras de asegurarse de que el contexto de persistencia se haya sincronizado antes de ejecutar esta consulta.

Se puede simplemente forzar un *flush* llamando a `flush()`, o configurando el modo de flush a ALWAYS:

```
// Realizar el flush() antes de la consulta
session.flush();
List<Book> books = session.createNativeQuery("SELECT * FROM Book", Book.class).getResultList();

// Configurar el Modo de Flush a 'ALWAYS'
session.setHibernateFlushMode(FlushMode.ALWAYS);
List<Book> books = session.createNativeQuery("SELECT * FROM Book", Book.class).getResultList();
```

Alternativamente, se puede indicar a Hibernate qué estado modificado afecta los resultados de la consulta utilizando el método `addSynchronizedEntityClass(Class entityClass)` o `addSynchronizedEntityName(String entityName)`. Esto asegura que Hibernate realice un *flush* del contexto de persistencia para las entidades especificadas antes de ejecutar la consulta nativa.

```
List<Book> books =
        session.createNativeQuery("select * from Books", Book.class)
            .addSynchronizedEntityClass(Book.class)
            .getResultList()
```

En el caso de utilizar **HQL (Hibernate Query Language)**, Hibernate automáticamente sincroniza el contexto de persistencia (hace un *flush*) antes de ejecutar la consulta si detecta cambios pendientes que afectan los resultados.

En contraste, cuando se ejecuta una consulta **SQL nativa**, Hibernate no tiene conocimiento sobre cómo esta consulta específica puede verse afectada por el estado actual del contexto de persistencia. Es responsabilidad del programador asegurarse de que el contexto de persistencia esté sincronizado (utilizando alguna de las alternativas vistas) antes de ejecutar la consulta.

Limits, pagination, and ordering

Si una consulta puede devolver más resultados de los que se pueden manejar a la vez, se puede especificar:

- un límite en el número máximo de filas devueltas, y,
- opcionalmente, un desplazamiento (*offset*), que indica la primera fila a devolver de un conjunto de resultados ordenado. Este desplazamiento se utiliza para paginar los resultados de una consulta.

Hay dos formas de añadir un límite o desplazamiento a una consulta HQL o SQL nativa:

- utilizando la sintaxis del propio lenguaje de consulta
- utilizando métodos como `setFirstResult(...)` y `setMaxResults(...)` de la interfaz `SelectionQuery` de la API de Hibernate o `Query` de la API estándar JPA.

```
List<Book> books =
        session.createSelectionQuery("from Book where title like ?1 order by title", Book.class)
            .setParameter(1, titlePattern)
            .setMaxResults(MAX_RESULTS)
            .getResultList();
```

La clase `SelectionQuery` de Hibernate tiene una forma de **paginar los resultados** de la consulta mediante `setPage(...)`:

```
List<Book> books =
        session.createSelectionQuery("from Book where title like ?1 order by title", Book.class)
            .setParameter(1, titlePattern)
            .setPage(Page.first(MAX_RESULTS))
            .getResultList();
```

Para obtener el número de páginas de resultados se puede utilizar el métod `getResultCount()`:

```
SelectionQuery<Book> query =
        session.createSelectionQuery("from Book where title like ?1 order by title", Book.class)
            .setParameter(1, titlePattern);
long pages = query.getResultCount() / MAX_RESULTS;
List<Book> books = query.setMaxResults(MAX_RESULTS).getResultList();
```

Es bastante común que la paginación se combine con la necesidad de **ordenar los resultados** de la consulta por un campo que se determina en tiempo de ejecución. Así que, como alternativa a la cláusula `order by` de HQL, `SelectionQuery` ofrece la capacidad de especificar que los resultados de la consulta deben ordenarse por uno o más campos del tipo de entidad devuelto por la consulta mediante el uso del método `setOrder(...)`:

```
List<Book> books =
        session.createSelectionQuery("from Book where title like ?1", Book.class)
            .setParameter(1, titlePattern)
            .setOrder(List.of(Order.asc(Book_.title), Order.asc(Book_.isbn)))
```

```
            .setMaxResults(MAX_RESULTS)
            .getResultList();
```

Named queries

La anotación `@NamedQuery` permite definir una consulta HQL que se **compila y verifica** como parte del proceso de arranque, lo que permite detectar errores antes de que la consulta se ejecute realmente.

La anotación `@NamedQuery` se puede colocar en cualquier clase, incluso en una clase de entidad.

```
@NamedQuery(name="10BooksByTitle",
            query="from Book where title like :titlePattern order by title fetch first 10 rows only")
class BookQueries {}
```

Debemos asegurarnos de que la clase con la anotación `@NamedQuery` sea escaneada por Hibernate, ya sea:

- agregando `<class>org.hibernate.example.BookQueries</class>` a `persistence.xml`, o
- llamando a `configuration.addClass(BookQueries.class)`.

La anotación `@NamedNativeQuery` permite hacer lo mismo para consultas SQL nativas. Sin embargo, en este tipo de consultas Hibernate no puede validar ni verificar la corrección de una consulta escrita en el dialecto SQL nativo de la base de datos.

Según la API, las consultas con nombre se invocan mediante un método u otro:

- **Consultas de SELECCIÓN de datos**
  - Se utiliza el método `createNamedSelectionQuery(String, Class)` de la interfaz `QueryProducer` de la cual hereda `Session` de la API de Hibernate.
  - Se utiliza el método `createNamedQuery(String, Class)` de la interfaz `EntityManager` de la API estándar JPA.
- **Consultas de MUTACIÓN de datos**
  - Se utiliza el método `createNamedMutationQuery(String)` de la interfaz `QueryProducer` de la cual hereda `Session` de la API de Hibernate.
  - Se utiliza el método `createNamedQuery(String)` de la interfaz `EntityManager` de la API estándar JPA.

Controlling lookup by id

Cuando el identificador de la entidad es conocido, una consulta HQL o una consulta SQL nativa puede parecer excesiva. Además, las consultas no hacen un uso eficiente del segundo nivel de caché.

Anteriormente se vio el método `find(...)` de la interfaz `EntityManager` de la API estándar JPA. Es la forma más básica de realizar una búsqueda por ID. Sin embargo, tiene sus limitaciones y no puede hacer todo.

Por lo tanto, Hibernate, a través de `Session`, tiene algunas API que simplifican ciertas búsquedas más complicadas:

- `byId()`
- `byMultipleIds()`
- `bySimpleNaturalId()`

- `byNaturalId()`
- `byMultipleNaturalId()`

Tuning and performance

Una vez que tu programa esté en funcionamiento con Hibernate, es probable que encuentres problemas de rendimiento. Afortunadamente, la mayoría son fáciles de resolver con las herramientas que Hibernate proporciona, pero recuerda que si Hibernate complica más de lo que ayuda en algún caso, no dudes en usar otra herramienta.

Existen dos fuentes principales de posibles cuellos de botella en el rendimiento de un programa que utiliza Hibernate:

- demasiados accesos a la base de datos, y
- consumo de memoria asociado con la caché de primer nivel o caché de sesión.

Por lo tanto, la optimización del rendimiento implica principalmente reducir el número de accesos a la base de datos y/o controlar el tamaño de la caché de sesión.

Tuning the connection pool

El pool de conexiones integrado en Hibernate es adecuado para pruebas, pero no está diseñado para su uso en producción.

En su lugar, Hibernate admite una variedad de diferentes pools de conexiones, incluido **Agroal**, recomendado por Hibernate.

Para seleccionar y configurar **Agroal**, se necesita establecer **algunas propiedades de configuración adicionales**. Estas propiedades tienen el prefijo `hibernate.agroal`.

```
# The maximum number of connections present on the pool
hibernate.agroal.maxSize=10
# The minimum number of connections present on the pool
hibernate.agroal.minSize=2
# The number of connections added to the pool when it is started
hibernate.agroal.initialSize=5
# The interval between background validation checks
hibernate.agroal.validationTimeout=30000
# The duration of time a connection can be held without causing a leak to be reported
hibernate.agroal.leakTimeout=60000
# The duration for eviction of idle connections
hibernate.agroal.reapTimeout=0
# The maximum amount of time a thread can wait for a connection, after which an exception is thrown instead
hibernate.agroal.acquisitionTimeout=30000
# The maximum amount of time a connection can live, after which it is removed from the pool
hibernate.agroal.maxLifetime=1800000
```

Siempre que se establezca al menos una propiedad con el prefijo `hibernate.agroal`, el `AgroalConnectionProvider` se seleccionará automáticamente.

> Sin embargo, en un entorno de contenedor como JBoss, Tomcat o Spring Boot, generalmente **no se necesita configurar un pool de conexiones** a través de Hibernate ya que estos contenedores ofrecen la capacidad de manejar *datasources* y *pools* de conexiones.

Enabling statement batching

Una mejora en el rendimiento es **activar el agrupamiento automático de declaraciones DML**. El agrupamiento solo ayuda en casos donde un programa ejecuta muchas inserciones, actualizaciones o eliminaciones contra la misma tabla en una sola transacción.

Para activar esta mejora se utiliza la propiedad de configuración `hibernate.jdbc.batch_size`.

Association fetching

Lograr un alto rendimiento en ORM significa **minimizar la cantidad de accesos** a la base de datos. La regla más fundamental en ORM es:

- especificar explícitamente todos los datos que se van a necesitar al inicio de una sesión/transacción, y recuperarlos de inmediato en una o dos consultas,
- y solo entonces comenzar a navegar entre asociaciones de entidades persistentes.

Sin duda, la causa más común de un código de acceso a datos con bajo rendimiento en programas Java es el **problema de las consultas N + 1**. Esto es, se recupera una lista de N filas de la base de datos en una consulta inicial, y luego se obtienen las instancias asociadas de la entidad relacionada usando N consultas posteriores, una por cada fila recuperada en la consulta inicial. De ahí el concepto de las "N + 1" consultas.

Por ejemplo, dado este código:

```
// Recuperar todos los libros
List<Book> books = session.createQuery("from Book", Book.class).getResultList();

// Para cada libro, acceder a sus autores
for (Book book : books) {
    for (Author author : book.getAuthors()) {
        System.out.println(book.getTitle() + " by " + author.getName());
    }
}
```

Primero se ejecuta una consulta (`getResultList()`) para recuperar todos los libros y luego, por cada libro, se ejecuta una consulta adicional (`book.getAuthors()`) para recuperar los autores asociados. Si hay N libros, se harán N consultas adicionales para los autores, lo que resulta en un total de **N + 1 consultas**.

Si la relación entre *"Book"* y *"Author"* está configurada con carga diferida (`@ManyToMany(fetch = FetchType.LAZY)` o similar), Hibernate realizará una consulta adicional para obtener los autores de cada libro.

Sin embargo, si la relación entre *"Book"* y *"Author"* está configurada con carga inmediata (`FetchType.EAGER`), entonces Hibernate podría realizar una única consulta con un JOIN para recuperar tanto los libros como sus autores, evitando las consultas adicionales.

Hibernate recomienda el ***lazy loading*** (carga diferida) en muchas situaciones para mejorar el rendimiento ya que carga las entidades relacionadas cuando se necesitan y permite que la inicialización de la entidad principal sea más rápida si sólo se requiere esta entidad.

Por tanto, es **responsabilidad del desarrollador** saber en que situaciones o contextos se va a necesitar únicamente la entidad principal o también se van a necesitar todas las entidades asociadas.

Para solventar este problema de las **N + 1 consultas** se puede usar el `JOIN FETCH` en la consulta:

```
List<Book> books =
    session.createSelectionQuery("from Book b join fetch b.authors order by b.isbn", Book.class)
        .getResultList();
```

Esto asegurará que tanto los libros como los autores se carguen en una única consulta.

The second-level cache

Para mejorar el rendimiento de las aplicaciones que usan bases de datos, se puede usar una **caché de segundo nivel**. Esto significa que en lugar de hacer consultas a la base de datos cada vez que se necesita la información, se pueden guardar datos en memoria y compartirlos entre diferentes sesiones de la aplicación. Esto reduce la cantidad de veces que se accede a la base de datos mejorando el rendimiento.

Sin embargo, utilizar una caché de segundo nivel tiene algunos riesgos. Dado que la caché no siempre está sincronizada con la base de datos, pueden surgir problemas **si la información en la caché no coincide con la base de datos**. Además, una caché de segundo nivel puede complicar la gestión de la concurrencia, es decir, cuando múltiples usuarios o procesos intentan acceder y modificar los datos al mismo tiempo, convirtiéndose en una fuente potencial de errores difíciles de aislar y reproducir.

Por defecto, **Hibernate no guarda entidades en la caché de segundo nivel**. Para usar esta caché, se debe marcar explícitamente las entidades que se van a almacenar en ella con una anotación especial (`@Cache` de `org.hibernate.annotations`). Además, Hibernate no incluye su propia implementación de la caché, por lo que se debe configurar un proveedor de caché externo para que funcione.

Stateless sessions

Una característica de Hibernate es la interfaz `StatelessSession`, que ofrece un enfoque más directo y orientado a comandos para interactuar con la base de datos.

Se puede obtener una sesión sin estado desde el `SessionFactory`:

```
StatelessSession ss = getSessionFactory().openStatelessSession();
```

Una sesión sin estado:

- No tiene una **caché de primer nivel** (contexto de persistencia), ni interactúa con ninguna caché de segundo nivel, y
- No implementa escritura transaccional diferida ni comprobación automática de modificaciones, por lo que todas las operaciones se **ejecutan inmediatamente** cuando se llaman explícitamente.

En una sesión sin estado, siempre trabajamos con objetos *detached* (desvinculados del contexto de persistencia).

Algunos métodos de la interfaz `StatelessSession`:

- `get(Class, Object)` ➔ Obtiene un objeto separado (detached), dado su tipo y su id, ejecutando un `select`.
- `fetch(Object)` ➔ Recupera una asociación de un objeto separado.
- `refresh(Object)` ➔ Refresca el estado de un objeto separado ejecutando un `select`.
- `insert(Object)` ➔ Inserta inmediatamente el estado del objeto transitorio dado en la base de datos.
- `update(Object)` ➔ Actualiza inmediatamente el estado del objeto separado dado en la base de datos.
- `delete(Object)` ➔ Elimina inmediatamente el estado del objeto separado dado de la base de datos.
- `upsert(Object)` ➔ Inserta o actualiza inmediatamente el estado del objeto separado dado usando una declaración SQL `merge into`.

> No hay una operación `flush()`, por lo que `update()` siempre es explícito.

En ciertas circunstancias, las sesiones sin estado son más fáciles de manejar y más sencillas de entender. Sin embargo una sesión sin estado es mucho más vulnerable a los efectos de *aliasing* de datos, ya que es fácil obtener dos objetos Java no idénticos que representen la misma fila de una tabla de la base de datos.

Optimistic and pessimistic locking

Finalmente, un aspecto del comportamiento bajo carga es la contención de datos a nivel de fila. Cuando muchas transacciones intentan leer y actualizar los mismos datos, el programa puede volverse no responsivo debido a la escalación de bloqueos, bloqueos mutuos y errores de tiempo de espera en la adquisición de bloqueos.

Hay dos enfoques básicos para la **concurrencia de datos** en Hibernate:

- **Bloqueo optimista** utilizando columnas `@Version`.
- **Bloqueo pesimista a nivel de base de datos** utilizando la sintaxis SQL `for update` (o equivalente).

En la comunidad de Hibernate, es mucho **más común utilizar el bloqueo optimista**.

Donde sea posible, en un sistema multiusuario, se debe evitar mantener un bloqueo pesimista durante una interacción del usuario. De hecho, la práctica habitual es evitar tener transacciones que abarquen interacciones de usuario.

> **Para sistemas multiusuario, como podría ser aplicaciones web, el bloqueo optimista es la mejor opción**.

Dicho esto, también hay lugar para los bloqueos pesimistas, que a veces pueden reducir la probabilidad de retrocesos o *rollbacks* de transacciones.

Por lo tanto, los métodos `find()`, `lock()` y `refresh()` de la sesión reactiva aceptan un `LockMode` opcional. También podemos especificar un `LockMode` para una consulta. El modo de bloqueo puede utilizarse para solicitar un bloqueo pesimista o para personalizar el comportamiento del bloqueo optimista.

Tipos de `LockMode`:

- **READ** ➔ Un bloqueo optimista obtenido implícitamente siempre que una entidad se lea de la base de datos usando `select`.
- **OPTIMISTIC** ➔ Un bloqueo optimista obtenido cuando una entidad se lee de la base de datos, y verificado usando un `select` para comprobar la versión cuando la transacción se completa.
- **OPTIMISTIC_FORCE_INCREMENT** ➔ Un bloqueo optimista obtenido cuando una entidad se lee de la base de datos, y forzado usando una actualización para incrementar la versión cuando la transacción se completa.
- **WRITE** ➔ Un bloqueo pesimista obtenido implícitamente siempre que una entidad se escriba en la base de datos usando `update` o `insert`.
- **PESSIMISTIC_READ** ➔ Un bloqueo pesimista para compartir.
- **PESSIMISTIC_WRITE** ➔ Un bloqueo pesimista para actualización.
- **PESSIMISTIC_FORCE_INCREMENT** ➔ Un bloqueo pesimista forzado usando una actualización inmediata para incrementar la versión.

En Hibernate, si no especificas un `LockMode` explícitamente, se utiliza el comportamiento por defecto. Por defecto, Hibernate utiliza ***optimistic locking*** cuando se realiza una operación de lectura.

Esto significa que la entidad se leerá con una versión que se verifica al finalizar la transacción para detectar posibles conflictos de concurrencia. Si no se indica un `LockMode`, **Hibernate asume que debe usar el bloqueo optimista**.

Collecting statistics

La recolección de estadísticas en Hibernate se refiere a la capacidad de obtener información sobre **el rendimiento y el comportamiento** del sistema de persistencia.

Para **activar la recolección de estadísticas** en Hibernate, se debe configurar la propiedad `hibernate.generate_statistics` en el archivo de configuración `hibernate.cfg.xml` o en el archivo de configuración `hibernate.properties`.

Una vez que la recolección de estadísticas está activada, se puede acceder a estas estadísticas a través de la interfaz `SessionFactory`:

```
import org.hibernate.SessionFactory;
import org.hibernate.stat.Statistics;

public class StatisticsExample {

    public static void main(String[] args) {
        SessionFactory sessionFactory = // Obtener tu SessionFactory

        // Activar estadísticas (si no se han activado en 'hibernate.properties')
        sessionFactory.getStatistics().setStatisticsEnabled(true);

        // Obtener estadísticas
        Statistics stats = sessionFactory.getStatistics();

        // Mostrar estadísticas
        System.out.println("Query Execution Count: " + stats.getQueryExecutionCount());
        System.out.println("Entity Load Count: " + stats.getEntityLoadCount());
        System.out.println("Entity Update Count: " + stats.getEntityUpdateCount());

        // Desactivar estadísticas si ya no son necesarias
        sessionFactory.getStatistics().setStatisticsEnabled(false);
    }
}
```

Las estadísticas recolectadas pueden incluir:

- **Conteo de Consultas Ejecutadas**: Número total de consultas ejecutadas.
- **Tiempo de Consulta**: Tiempo total gastado en la ejecución de consultas.
- **Conteo de Entidades Cargadas**: Número de veces que se cargaron entidades desde la base de datos.
- **Conteo de Entidades Actualizadas**: Número de veces que se actualizaron entidades en la base de datos.
- **Conteo de Entidades Eliminadas**: Número de veces que se eliminaron entidades.

Tracking down slow queries

Cuando se descubre una consulta SQL de bajo rendimiento en producción, a veces puede ser difícil rastrear exactamente de dónde proviene la consulta en el código Java.

Hibernate ofrece dos propiedades de configuración que pueden **facilitar la identificación de una consulta lenta** y encontrar su origen:

- `hibernate.log_slow_query` ➔ que indica el tiempo mínimo de ejecución en milisegundos que caracteriza una consulta "lenta".
- `hibernate.use_sql_comments` ➔ que indica si se debe o no anteponer comentarios al SQL ejecutado.

El texto que se antepone como comentario SQL es el texto de la consulta HQL, lo que facilita su localización dentro del código Java.

El texto del comentario puede personalizarse mediante `Query.setComment(comment)` o `Query.setHint(AvailableHints.HINT_COMMENT, comment)`, o directamente mediante la anotación `@NamedQuery`.

Adding indexes

La anotación `@Index` se puede utilizar para agregar un índice a una tabla:

```
@Entity
@Table(indexes=@Index(columnList="title, year, publisher_id"))
class Book { /*...*/ }
```

Es posible especificar un orden para una columna indexada, o que el índice sea insensible a mayúsculas y minúsculas:

```
@Entity
@Table(indexes=@Index(columnList="(lower(title)), year desc, publisher_id"))
class Book { /*...*/ }
```

Esto permite crear un índice personalizado para una consulta particular.

> Se pueden crear índices personalizados para consultas específicas, pero es más adecuado definir y gestionar estos índices en **scripts SQL DDL** que en el código Java, ya que los administradores de bases de datos suelen ser los responsables de su optimización. Hibernate permite ejecutar estos scripts mediante la propiedad `javax.persistence.schema-generation.create-script-source`.

Summary of annotations

Resumen de algunas de las anotaciones disponibles en Hibernate y JPA. El ***"Javadoc"*** de las anotaciones que forman parte del estándar de JPA estan en el paquete `jakarta.persistance` de la documentación de **Jakarta 10** (última versión estable a *07/2024*).

En cambio, las anotaciones que no son del estándar y han sido añadidas por Hibernate, se encuentran en el paquete `org.hibernate.annotations` de la documentación de **Hibernate**.

Entities and embeddable types

- `@Entity`: declarar una clase de entidad - Estándar JPA
- `@MappedSuperclass`: declarar una clase que no sea una entidad con atributos mapeados heredados por una entidad - Estándar JPA
- `@Embeddable`: declarar un tipo integrable - Estándar JPA
- `@IdClass`: declarar la clase de identificador para una entidad con múltiples atributos `@Id` - Estándar JPA

Basic and embedded attributes

- `@Id`: declarar un atributo de identificador de tipo básico - Estándar JPA

- `@Version` : declarar un atributo de versión - Estándar JPA
- `@Basic` : declarar un atributo básico - Estándar JPA
- `@EmbeddedId` : declarar un atributo de identificador de tipo embebido - Estándar JPA
- `@Embedded` : declarar un atributo de tipo embebido - Estándar JPA
- `@Enumerated` : declarar un atributo de tipo enumeración y especificar cómo se codifica - Estándar JPA
- `@Array` : declarar que un atributo se asigna a un ARRAY de SQL y especificar la longitud - Hibernate
- `@ElementCollection` : declarar que una colección está asignada a una tabla dedicada - Estándar JPA

Converters and compositional basic types

- `@Converter` : registra un `AttributeConverter` - Estándar JPA
- `@Convert` : aplica un convertidor a un atributo - Estándar JPA
- `@JavaType` : especifica explícitamente una implementación de `JavaType` para un atributo básico - Hibernate
- `@JdbcType` : especifica explícitamente una implementación de `JdbcType` para un atributo básico - Hibernate
- `@JdbcTypeCode` : especifica explícitamente un código de tipo JDBC utilizado para determinar el `JdbcType` para un atributo básico. - Hibernate
- `@JavaTypeRegistration` : registra un `JavaType` para un tipo de Java determinado - Hibernate
- `@JdbcTypeRegistration` : registra un `JdbcType` para un código de tipo JDBC determinado - Hibernate

System-generated identifiers

- `@GeneratedValue` : especificar que un identificador es generado por el sistema - Estándar JPA
- `@SequenceGenerator` : definir una identificación generada respaldada por una secuencia de base de datos - Estándar JPA
- `@TableGenerator` : definir una identificación generada respaldada por una tabla de base de datos - Estándar JPA
- `@IdGeneratorType` : declarar una anotación que asocie un `Generator` personalizado con cada atributo `@Id` que anota - Hibernate
- `@ValueGenerationType` : declarar una anotación que asocie un `Generator` personalizado con cada atributo `@Basic` que anota - Hibernate

Entity associations

- `@ManyToOne` : declarar el lado de un solo valor de una asociación de muchos a uno (el lado propietario) - Estándar JPA
- `@OneToMany` : declarar el lado multivalor de una asociación de muchos a uno (el lado sin propietario) - Estándar JPA
- `@ManyToMany` : especifica una asociación de muchos valores con multiplicidad de muchos a muchos. - Estándar JPA
- `@OneToOne` : declarar cualquiera de los lados de una asociación uno a uno - Estándar JPA

- `@MapsId` : declarar que el lado propietario de una asociación @OneToOne asigna la columna de clave principal - Estándar JPA

Annotations for mapping tables

- `@Table` : asigna una clase de entidad a su tabla principal - Estándar JPA
- `@SecondaryTable` : define una tabla secundaria para una clase de entidad - Estándar JPA
- `@JoinTable` : asigna una asociación de *many-to-many* o de *many-to-one* a su tabla de asociaciones - Estándar JPA
- `@CollectionTable` : asigna un @ElementCollection a su tabla - Estándar JPA

Annotations for mapping columns

- `@Column` : asigna un atributo a una columna - Estándar JPA
- `@JoinColumn` : asigna una asociación a una columna de clave externa - Estándar JPA
- `@PrimaryKeyJoinColumn` : asigna la clave principal utilizada para unir una tabla secundaria con su tabla principal o una tabla de subclase en herencia JOINED con su tabla de clase raíz. - Estándar JPA
- `@OrderColumn` : especifica una columna que debe usarse para mantener el orden de una List. - Estándar JPA
- `@MapKeyColumn` : especifica una columna que se debe usar para conservar las claves de un Map. - Estándar JPA

Enlaces

Hibernate

- 🔶 Hibernate
- Learn JPA & Hibernate Series - Baeldung

Jakarta JPA

- 🔶 Jakarta Persistence
- Jakarta Persistence Specification Project
- Introduction to Jakarta Persistence
- JPA - Baeldung

Licencia