Hibernate

… 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.

> Sección 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 su proyecto
- configurar Hibernate con información sobre la base de datos, especificando propiedades de configuración.

Añadir la dependencia de Hibernate tanto en Gradle como en Maven:

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

Reemplazando `{version}` por la versión de Hibernate a utilizar.

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

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:

```
<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>6.5.2.Final</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

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 `{versión}` 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.

La lista está disponible en la documentación oficial.

Configuration using JPA XML

Siguiendo el enfoque estándar JPA, se proporcionaría un archivo llamado `persistence.xml`, que generalmente se coloca en el directorio `META-INF` de un archivo de persistencia, es decir, del archivo o directorio .jar que contiene las clases de entidad.

```
<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'`).

Se puede obtener una `EntityManagerFactory` 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 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();
```

La clase `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 `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 `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 la ruta de clase raíz:

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

# SQL statement logging
hibernate.show_sql=true
```

```
hibernate.format_sql=true
hibernate.highlight_sql=true
```

Basic configuration settings

La clase 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 solo 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, por ejemplo, *'import.sql'*, 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

Para ver el SQL generado mientras se envía a la base de datos, tiene dos opciones.

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**: si es *'true'*, imprime el registro SQL directamente en la consola
- **hibernate.format_sql**: si es *'true'*, imprime el registro SQL en un formato con sangría de varias líneas
- **hibernate.highlight_sql**: si es *'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 su implementación preferida de logging **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`. Más información

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.

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

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 está trabajando con SQL Server, es posible que 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, puede configurar SQL Server para usar la intercalación habilitada para UTF-8 (_UTF8).

Entities

Una entidad es una clase Java 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.

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. 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**.

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 {
    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** (*'field access'*)
- **acceso a propiedades** (*'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 acceden directamente a través de 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**, Hibernate accede a los atributos a través de métodos *'getter'* y *'setter'* 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 la entidad basándose en la ubicación de las anotaciones a nivel de atributo:

- Si un atributo está anotado directamente con `@Id`, Hibernate utiliza **acceso directo a campos** para ese atributo.
- Si un método *'getter'* está anotado con `@Id`, Hibernate utiliza **acceso a propiedades** para ese atributo.

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 principal 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 principal 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 principal 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.

Summary of annotations

Lista de las anotaciones vistas y si existe equivalencia con JPA.

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.

```java
@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`.

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

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

Mapping to tables

TODO

Referencias

- https://hibernate.org
- https://hibernate.org/orm/documentation/6.5
- https://hibernate.org/orm/documentation/getting-started
- https://www.baeldung.com/tag/hibernate

Licencia