# 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.

> 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

TODO

---

## Enlaces de interés

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

## Licencia