UD 3. Acceso a bases de datos vía ORM

Módulo: Acceso a datos

Germán Gascón Grau
g.gascongrau@edu.gva.es

Contenidos

- ¿Qué es ORM?
- Configuración de hibernate
- Hibernate
  - Sessions
  - Single-Table Mapping
  - Relationship-Mapping

¿Qué es ORM?

Un **ORM** (Object-Relational Mapping) es una biblioteca/API utilizada para que las interacciones con una base de datos sean más abstractas y robustas y da una serie de facilidades para acceder a ellas y exponerlas en las aplicaciones.

Traduce automáticamente las operaciones sobre objetos en el código (como crear, leer, actualizar o eliminar instancias de clases) en consultas SQL sobre las tablas de una base de datos.

POJO

Los **ORM** suelen requerir la definición de clases específicas dentro del modelo de dominio para independizar la capa de persistencia de la capa de negocio. En el mundo Java, los enfoques más utilizados se basan en **POJO** y **JavaBeans**.

POJO en Java significa Plain Old Java Object. Se trata de un objeto Java normal, que no está sujeto a restricciones especiales impuestas por un framework concreto. Una clase POJO no requiere bibliotecas específicas ni dependencias especiales en el classpath. Este enfoque mejora la legibilidad del código y favorece la reutilización en aplicaciones Java.

Los POJO están ampliamente aceptados debido a su facilidad de mantenimiento y portabilidad. Son sencillos de leer, escribir y probar.

Una clase POJO no está obligada a seguir convenciones estrictas de nomenclatura para propiedades y métodos (más allá de las buenas prácticas habituales del lenguaje). No está ligada a ningún framework Java específico, por lo que puede utilizarse en cualquier programa Java.

POJO

- Una clase POJO puede ser pública (lo habitual), aunque no es un requisito estricto del concepto.
- Debe tener **al menos un constructor** accesible. En muchos casos se utiliza un constructor público **sin argumentos**.
- Puede tener **constructores con argumentos**.
- Las propiedades pueden tener **getters y setters públicos** para permitir el acceso desde otros programas Java.
- Las **propiedades** de una clase POJO pueden tener cualquier modificador de acceso (privado, público o protegido). Se recomienda que las propiedades sean **privadas** para mejorar la encapsulación y la seguridad.
- Una clase POJO no debería extender clases predefinidas por frameworks o librerías que impongan un comportamiento específico.
- No debería implementar interfaces impuestas por frameworks.
- No debería tener anotaciones previas.

JavaBeans

- Todos los **JavaBeans son POJO**, pero no todos los POJO son JavaBeans. Puede decirse que un JavaBean es un POJO **más restrictivo**.
- Debe implementar la interfaz (de marcado) **Serializable**, incluida en el núcleo de Java.
- Las **propiedades** deben ser obligatoriamente **privadas**.
- Las propiedades deben disponer de **getters** y **setters**, siguiendo las convenciones estándar de JavaBeans.
- Debe definir, al menos, un **constructor público sin argumentos**.
- Las **propiedades** se acceden exclusivamente a través de los **constructores** y de los métodos **getters** y **setters**.

Hibernate

- Hibernate es una **solución ORM para Java**. Es un framework de persistencia de **código abierto** creado por Gavin King en el año 2001. Proporciona un potente servicio de consulta y persistencia objeto–relacional de alto rendimiento para aplicaciones Java.
- Hibernate **mapea clases Java a tablas** de bases de datos y tipos de datos Java a tipos de datos SQL, liberando al desarrollador de la mayor parte de las tareas de programación habituales relacionadas con la persistencia de datos.
- Hibernate se sitúa entre los objetos Java tradicionales y el servidor de bases de datos, gestionando todo el trabajo necesario para mantener la correspondencia entre ambos mediante mecanismos y patrones de mapeo objeto–relacional (O/R) adecuados.

Ventajas de Hibernate

- Se encarga de asignar clases Java a tablas de bases de datos mediante **archivos XML** o **anotaciones**, sin necesidad de escribir (en muchos casos) código específico de persistencia.
- Proporciona **APIs sencillas** para almacenar y recuperar objetos Java directamente desde y hacia la base de datos.
- Si hay cambios en la base de datos o en alguna tabla, normalmente **basta con modificar la configuración** de mapeo (XML o anotaciones), **sin afectar al resto de la aplicación**.
- **Abstrae los tipos SQL** específicos de cada base de datos y permite trabajar con objetos y tipos Java conocidos.
- **No requiere un servidor de aplicaciones** para funcionar; puede utilizarse en aplicaciones Java SE.
- Permite manejar **asociaciones complejas** entre objetos (uno a uno, uno a muchos, muchos a muchos, etc.) definidas en la base de datos.
- **Minimiza y optimiza el acceso a la base de datos** mediante estrategias inteligentes de obtención de datos, como lazy loading, fetching strategies y cachés.
- Proporciona **mecanismos de consulta** de datos más sencillos y orientados a objetos que el SQL tradicional.

Hibernate. Bases de datos soportadas

Hibernate es compatible con una amplia parte de los principales sistemas de gestión de bases de datos relacionales (RDBMS), entre ellos:

- DB2
- PostgreSQL
- Apache Derby, H2 y HSQLDB (bases de datos embebidas en Java)
- Microsoft SQL Server
- SAP HANA
- Google Cloud Spanner
- MariaDB y MySQL
- Oracle Database

Hibernate. Tecnologías soportadas

Hibernate es compatible con una variedad de otras tecnologías, entre las que se incluyen:

- Spring (especialmente Spring Framework y Spring Boot)
- Jakarta EE
- Conectores e integraciones con IDEs como Eclipse (Hibernate Tools) y IntelliJ IDEA
- Herramientas de construcción y gestión de dependencias como Apache Maven y Gradle
- XDoclet (herramienta histórica para generación de metadatos; hoy en día en desuso frente a anotaciones)

Hibernate. Versiones

Última versión estable: 7.2 (2025-12-12)

https://hibernate.org/orm/releases/7.2/

Hibernate. Implementaciones

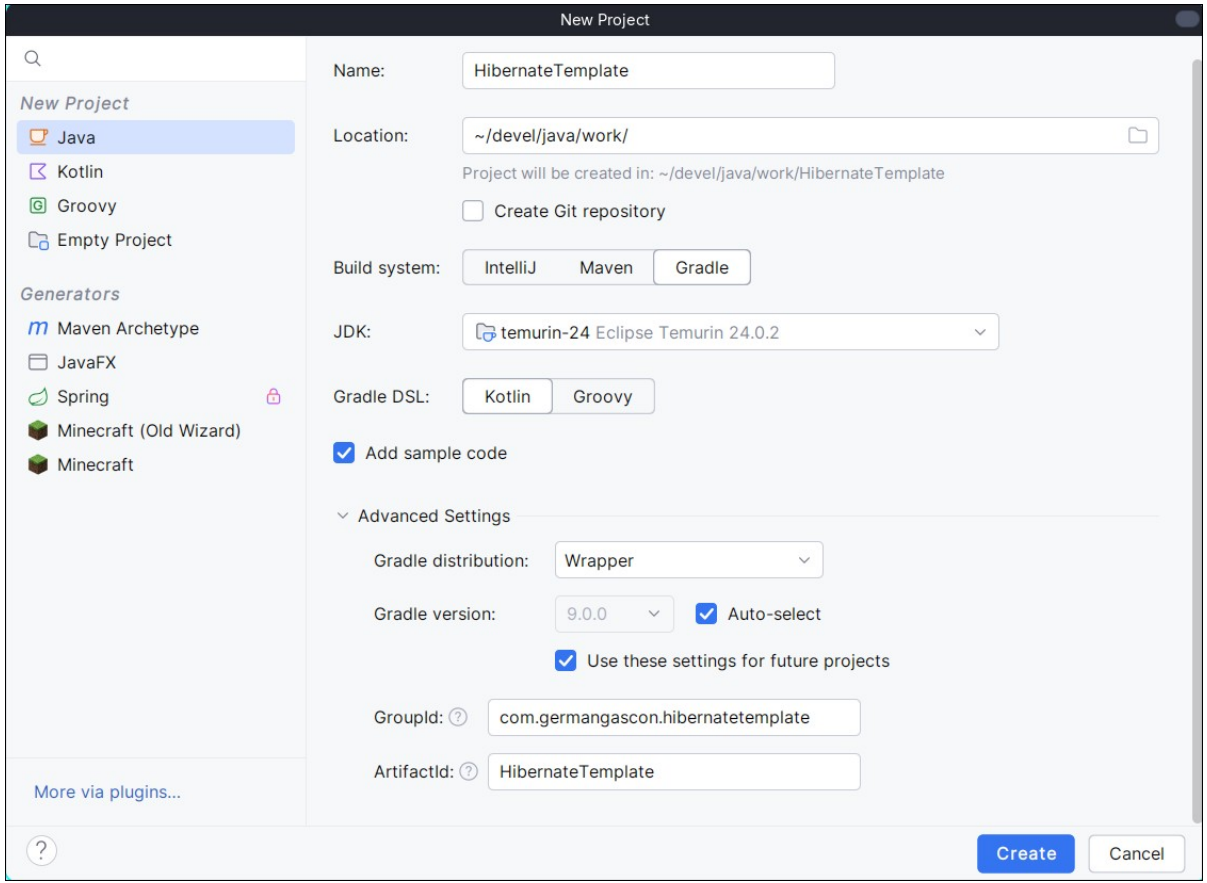
Existen dos formas de implementar Hibernate:

- La forma clásica (Hibernate clásico o legacy): Mapeo de las tablas a **archivos XML** (resources)
- La forma actual (con anotaciones): Mapeo de las tablas a clases POJO mediante **anotaciones**

Desde la versión 6 (mayo de 2022), el enfoque basado en anotaciones es el recomendado y el más utilizado en proyectos nuevos. El mapeo clásico mediante XML sigue estando soportado, aunque se considera menos habitual en desarrollos modernos.

Hibernate. Creación del proyecto

Crear un proyecto Java Gradle



Hibernate. Configurar build.gradle.kts

Es necesario añadir las siguientes dependencias:

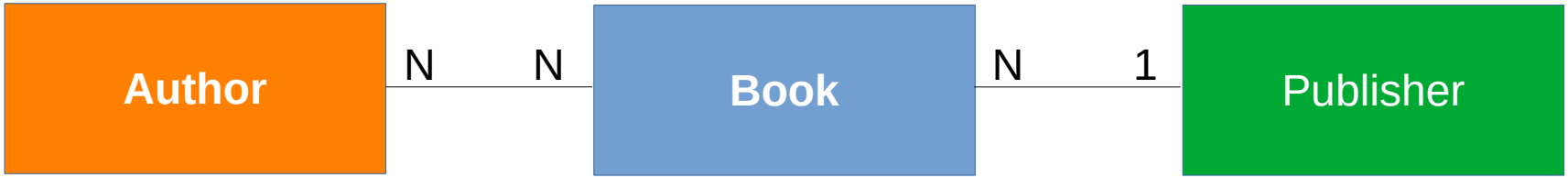
```
// Hibernate ORM
implementation("org.hibernate.orm:hibernate-core:7.2.0.Final")


// Driver JDBC de PostgreSQL
implementation("org.postgresql:postgresql:42.7.8")
```

Hibernate. Ejemplo

La mejor forma de entender el funcionamiento de Hibernate es a través de un ejemplo, así que vamos allá.

Haremos el ejemplo de una base de datos de **libros**.



Crear una base de datos PostgreSQL, en mi caso he utilizado la aplicación DBeaver y la he llamado library.

Hibernate. POJOs

Clase Author

```
public class Author {
    private Long id;
    private final String name;

    protected Author() {
        this(null);
    }

    public Author(String name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Override
    public final boolean equals(Object o) {
        if (!(o instanceof Author author)) return false;
        return id.equals(author.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }

    @Override
    public String toString() {
        return "Author{" +
                "id=" + id +
                ", name='" + name +
                '}';
    }
}
```

Hibernate. POJOs

Clase Publisher

```
public class Publisher {
    private Long id;
    private String name;

    protected Publisher() {
        this(null);
    }

    public Publisher(String name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Override
    public final boolean equals(Object o) {
        if (!(o instanceof Publisher publisher)) return false;

        return id.equals(publisher.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }

    @Override
    public String toString() {
        return "Publisher{" +
                "id=" + id +
                ", name='" + name +
                '}';
    }
}
```

Hibernate. POJOs

Clase Book

```
public class Book {
    private Long id;
    private String title;
    private Author author;
    private Publisher publisher;

    protected Book() {}

    public Book(String title, Author author, Publisher publisher) {
        this.title = title;
        this.author = author;
        this.publisher = publisher;
    }

    public Long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public Author getAuthor() {
        return author;
    }

    public Publisher getPublisher() {
        return publisher;
    }

    @Override
    public final boolean equals(Object o) {
        if (!(o instanceof Book book)) return false;

        return id.equals(book.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }

    @Override
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", title='" + title + '\'' +
                ", authorId=" + author.getId() +
                ", publisherId=" + publisher.getId() +
                '}';
    }
}
```

Hibernate. Configurar hibernate

Crea src/main/resources/hibernate.cfg.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>

        <!-- JDBC -->
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
        <property name="hibernate.connection.url">jdbc:postgresql://localhost:5432/demo_hibernate</property>
        <property name="hibernate.connection.username">demo_user</property>
        <property name="hibernate.connection.password">demo_pass</property>

        <!-- Dialecto -->
        <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>

        <!-- Esquema / DDL -->
        <!-- create-drop: crea al iniciar y borra al cerrar
             update: intenta actualizar (útil en dev)
             validate: solo valida -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Debug SQL -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>

        <!-- (Opcional) para que Hibernate sepa dónde están tus entidades si no las registras manualmente -->
    </session-factory>
</hibernate-configuration>
```

Hibernate. Configurar hibernate

Propiedades JDBC

- **hibernate.connection.driver_class**

  Indica a Hibernate qué driver JDBC usar para abrir conexiones.

- **hibernate.connection.url**

  Define la URL JDBC que identifica la conexión a la base de datos.

- **hibernate.connection.username**

- **hibernate.connection.password**

  Credenciales usadas por el driver para autenticarse en la BD

Hibernate. Configurar hibernate

Dialecto SQL

- **hibernate.dialect**

Gestión del esquema (DDL)

- **hibernate.hbm2ddl.auto**

  Controla qué hace Hibernate con las tablas al arrancar.

Valor	Comportamiento	Uso
none	No hace nada	Ocasional
validate	Comprueba que el esquema coincide	Producción
update	Intenta adaptar el esquema	Desarrollo
create	Borra y crea tablas	Ocasional / Testing
create-drop	Borra y crea tablas al iniciar, y borra al cerrar	Testing / Demos

Hibernate. Configurar hibernate

Debug SQL

- **hibernate.show_sql**

  Imprime en consola el SQL generado antes de ejecutarlo.

- **hibernate.format_sql**

  Formatea el SQL para que sea legible.

Hibernate. Entidades

¿Qué es una entidad para Hibernate?

Una **clase Java** que **representa una tabla** de la base de datos donde cada objeto representa una fila.

De momento las clases Author, Book y Publisher, son POJOs y vamos a convertirlas en Entidades.

Nuestro objetivo es que Hibernate inserte y lea filas de la tablas y las convierta en objetos Author, Book y Publisher.

Hibernate. Anotaciones

¿Qué es una anotación Java?

Las anotaciones en Java son **metadatos** que se añaden al código fuente (clases, métodos, campos, etc.) para proporcionar información adicional que puede ser utilizada por el compilador, herramientas de desarrollo o frameworks, sin cambiar la lógica directa del programa.

Son una forma de desarrollo declarativo, reemplazando a menudo archivos XML, y se identifican por el símbolo **@**

Permiten definir comportamiento, **configurar frameworks** (como Spring) y generar código de manera más limpia y eficiente, con ejemplos como @Override, @Deprecated o @SuppressWarnings, y son clave en tecnologías modernas como Hibernate, JPA, JUnit, Lombok y Spring Boot.

Hibernate. Anotaciones

Hibernate utiliza las anotaciones de la siguiente forma:

Cuando arranca (SessionFactory):

- Escanea el classpath
- Encuentra clases con @Entity
- Lee sus anotaciones
- Construye un modelo interno de metadatos. Este modelo contiene:
  - Nombre de tablas, columnas y claves primarias
  - Relaciones y cascadas
  - Fetch type

Hibernate. Entity Publisher

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

    @Column(nullable = false)
    private String name;

    protected Publisher() {
        this(null);
    }

    public Publisher(String name) {
        books = new ArrayList<>();
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Override
    public final boolean equals(Object o) {
        if (!(o instanceof Publisher publisher)) return false;

        return id.equals(publisher.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }

    @Override
    public String toString() {
        return "Publisher{" +
                "id=" + id +
                ", name='" + name +
                '}';
    }
}
```

Hibernate. Anotaciones

`@Entity`

Marca la clase como entidad Hibernate

Sin esto → Hibernate ignora la clase

Atributos más usados

- **name** (no es el nombre de la tabla)
  - Es el nombre de la entidad JPA

Hibernate. Anotaciones

```
@Table(name = "author")
```

Indica el nombre exacto de la tabla

Si no se pone, Hibernate usa el nombre de la clase (Author)

Atributos más usados

- **name**

  Nombre de la tabla

Hibernate. Anotaciones

`@Id`

Marca el atributo que actúa como identificador único (clave primaria)

Toda entidad debe tener uno.

No tiene atributos configurables.

Hibernate. Anotaciones

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

Indica que la BD genera el ID automáticamente

IDENTITY encaja con BIGSERIAL en PostgreSQL

Atributos más usados

- **strategy**
- **generator**

Hibernate. Atributos de @GeneratedValue

strategy

Define cómo se genera el identificador

- **GenerationType.*IDENTITY*** (el más usado)

  La base de datos genera el ID

- ***GenerationType.SEQUENCE***

  Hibernate pide Ids antes del INSERT

  Usa una secuencia explícita

- ***GenerationType.AUTO*** (no recomendado)

  Hibernate decide la estrategia

  Depende del dialecto

Hibernate. Atributos de @GeneratedValue

generator

Define el generador a utilizar

Debe existir uno de los siguientes:

- **@SequenceGenerator**
- **@TableGenerator**

Ejemplo

```
@SequenceGenerator(
        name = "author_seq",
        sequenceName = "author_id_seq",
        allocationSize = 1
)
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "author_seq")
private int number;
```

Hibernate. Anotaciones

@UuidGenerator

Para indicar que Hibernate debe utilizar UUID como generador.

Ventajas

- No dependencia de la BD
- Fácil en sistemas distribuidos

Ejemplo

```
@Id
@GeneratedValue
@UuidGenerator
private UUID id;
```

Hibernate. Anotaciones

```
@Column(nullable = false)
```

Mapea un atributo a una columna

nullable = false → NOT NULL en BD

Atributos más usados

- **name**
- **nullable**
- **length**
- **unique**
- **updatable**
- **insertable**

Hibernate. Atributos de @Column

name

```
@Column(name = "name")
```

- Nombre exacto de la columna en la BD
- Si no se pone, Hibernate usa el nombre del atributo

nullable

```
@Column(nullable = false)
```

- Indica si la columna acepta NULL o no

Hibernate. Atributos de @Column

unique

```
@Column(unique = false)
```

- Añade una restricción de unicidad para esa columna

updatable e insertable

```
@Column(updatable = false, updatable = false)
```

- Hibernate no escribe en esa columna
- Se usa en:
  - Vistas
  - Relaciones de solo lectura
  - Columnas gestionadas por triggers

Hibernate. Java Reflection

Hibernate crea objetos mediante **reflexión**.

La reflexión en Java permite **inspeccionar y manipular clases** en **tiempo de ejecución** (campos, métodos, constructores), incluso **aunque sean private**.

Hibernate usa reflexión para **crear instancias**, **leer/escribir campos y sincronizar** objetos con la base de datos.

El **constructor vacío es obligatorio** porque Hibernate necesita instanciar la entidad sin conocer sus datos aún.

Hibernate. Java Reflection

Errores comunes

- Olvidar `@Entity`
- No poner constructor vacío
- Pensar que Hibernate necesita setters para todo. Hibernate usa **field access**, por tanto, no necesita setters, accede directamente a los campos mediante reflexión.

Hibernate. Relaciones

Una **relación** expresa que **un registro** de una tabla está **asociado** a **uno o varios** registros de otra tabla.

Ejemplo conceptual:

“Un libro está editado por una editorial”

“Una editorial puede tener publicados varios libros”

Eso es una relación, no un atributo simple.

Hibernate. Relaciones uno a uno (1:1)

En este tipo de relaciones un registro de una tabla se relaciona exactamente con un registro de otra tabla.

Ejemplos:

- Persona y Dni
- Usuario y Perfil

En SQL

- Una tabla tiene una clave ajena única hacia la otra
- Hibernate utiliza la anotación **`@OneToOne`**

Hibernate. Relaciones uno a muchos (1:N)

En este tipo de relaciones un registro de una tabla puede relacionarse con varios registros de otra tabla.

Ejemplos:

- Libro y Editorial
- Cliente y Pedido

En SQL

La tabla del lado N tiene una clave foránea

En Hibernate

- Lado N → `@ManyToOne`
- Lado 1 → `@OneToMany`

Hibernate. Relaciones muchos a muchos (N:N)

En este tipo de relaciones varios registros de una tabla puede relacionarse con varios registros de otra tabla.

Ejemplos:

- Alumno y Asignatura
- Libro y Autor

En SQL

- Genera una tabla intermedia.

En Hibernate

- `@ManyToMany`

Hibernate. Claves ajenas (FK)

Las relaciones se almacenan con claves ajenas.

Las FK están siempre en la tabla “muchos”

Concepto SQL	Hibernate
Clave foránea	`@ManyToOne`
Columna FK	`@JoinColumn`
Relación inversa	`@OneToMany(mappedBy=...)`
Tabla intermedia	`@ManyToMany`

Hibernate. Owning Side

Uno de los conceptos más importantes de Hibernate es el Owning Side, es decir, **el lado que usa para escribir** la relación en la base de datos.

Solo el Owning Side guarda la relación.

El otro lado solo refleja, pero no manda.

Si se modifica solo el lado **no propietario**, Hibernate **ignora el cambio**.

En nuestro caso:

- Book es el owning side
- Publisher no manda sobre esa relación

Hibernate. Owning Side

En la relación 1:N entre Publisher y Libro, el Owning Side es libro.

En la clase Book tendríamos: (luego lo veremos con mayor detalle)

```
@ManyToOne
@JoinColumn(name = "publisher_id")
private Publisher publisher;
```

La anotación `@JoinColumn` indica qué columna de la tabla contiene la clave ajena. Solo tiene sentido en el Owning Side.

Sin `@JoinColumn`, Hibernate inventa los nombres y las reglas.

Hibernate. Owning Side

Examinemos con mayor detalle la relación Book - Publisher

```
@ManyToOne
@JoinColumn(name = "publisher_id")
private Publisher publisher;
```

Hibernate interpreta:

- Esta relación se guarda en la columna publisher_id
- Esa columna apunta a la clave primaria (PK) de Publisher
- Al persistir un Book, debo escribir publisher_id

Hibernate. Atributos de @JoinColum

name

```
@JoinColumn(name = "publisher_id")
```

- Nombre exacto de la columna FK
- Si no lo pones, Hibernate inventa uno. ¡Ponerlo siempre explícito!

nullable

```
@JoinColumn(nullable = false)
```

- En esto caso generaría NOT NULL e hibernate exigirá que la relación exista.

Hibernate. Atributos de @JoinColum

unique

```
@JoinColumn(unique = false)
```

- Garantiza que la FK no se repita
- Se usa típicamente en `@OneToOne`

updatable e insertable

```
@JoinColumn(updatable = false, updatable = false)
```

- Hibernate no escribe en esa columna
- Se usa en:
  - Vistas
  - Relaciones de solo lectura
  - Columnas gestionadas por triggers

Hibernate. Relaciones uno a uno (1:1)

@OneToOne

Los atributos más importantes son:

- **mappedBy**
- **fetch**
- **cascade**
- **optional**

Hibernate. Atributos de @OneToOne

mappedBy (el más usado)

- Indica que este lado NO es el owning side
- La FK está en la otra entidad
- Hibernate no escribe la relación desde aquí

Regla mental

- Si hay mappedBy, **este lado NO guarda**.

Hibernate. Atributos de @OneToOne

fetch

Parámetro JPA estándar que indica **cuándo** se cargan los datos de las relaciones.

Valores posibles:

- **EAGER**

  La relación se carga **inmediatamente** junto con la entidad principal.

- **LAZY**

  La relación se carga **bajo demanda** (cuando se accede por primera vez). Es el valor recomendado por defecto.

Hibernate. Atributos de @OneToOne

cascade

Indica a Hibernate qué operaciones se **propagan**.

Valores posibles:

- PERSIST
- MERGE
- REMOVE
- REFRESH
- DETACH
- ALL

Hibernate. Atributos de @OneToOne

optional

Indica si la relación puede ser nula o no.

Dicho de otra forma: si la entidad puede existir sin la otra.

Valores posibles:

- true
- false

Hibernate. Relaciones muchos a unos (N:1)

`@ManyToOne`

Representa el **lado propietario**, es decir, quién tiene la clave ajena.

Los atributos más importantes son:

- **fetch**
- **cascade**
- **optional**

Las propiedades de la clase anotadas con `@ManyToOne`, siempre serán un **objeto simple** (una única entidad).

El significado es el mismo que el visto para `@OneToOne`

Hibernate. Relaciones uno a muchos (1:N)

`@OneToMany`

Representa el **lado inverso** de la relación, es decir, el lado que no tiene la clave ajena.

Los atributos más importantes son:

- **mappedBy**
- **fetch**
- **cascade**
- **optional**

Las propiedades de la clase anotadas con `@OneToMany`, siempre serán una **colección de elementos** (habitualmente List o Set).

El significado es el mismo que el visto para `@OneToOne`

Hibernate. Relaciones muchos a muchos (N:N)

`@ManyToMany`

Una relación muchos-a-muchos (N:N) significa que cada lado de la relación puede tener muchos del otro.

Como en SQL no existe N:N directo, se convierte en dos 1:N mediante una **tabla intermedia**.

En Hibernate/JPA se pueden mapear de dos formas:

- `@ManyToMany` con `@JoinTable` (simple, didáctica)
- Entidad intermedia cuando la relación tiene datos propios.

¿Cómo lo mapea Hibernate?

Hibernate necesita saber:

- qué tabla intermedia usar (`@JoinTable`)
- qué columnas apuntan a cada lado (`@JoinColumn`)
- quién manda (Owning Side), el lado que tiene el `@JoinTable`, el lado inverse es el que tiene el atributo mappedBy en la anotación `@ManyToMany`

Hibernate. Relaciones muchos a muchos (N:N)

Owning Side en relaciones N:N

- El Owning Side es el lado que tiene `@JoinTable`.
- El lado con mappedBy es inverse y Hibernate no lo usa para guardar.

Consecuencia práctica

- Si solo se modifica el lado inverse → no se insertan filas en la tabla intermedia.
- Si se modifica el owning side → Hibernate inserta/borra en la tabla intermedia.

Hibernate. Atributos de @joinTable

`@JoinTable`

Los atributos más importantes son:

- **name**

  Nombre de la tabla intermedia

- **joinColumns**

  FK(s) desde la join table hacia esta entidad (owning side).

- **inverseJoinColumns**

  FK(s) desde la join table hacia la otra entidad.

Hibernate. Atributos de @JoinColumn

`@JoinTable`

Los atributos más importantes son:

- **name**

  Nombre de la tabla intermedia

- **joinColumns**

  FK(s) desde la join table hacia esta entidad (owning side).

  joinColumns = `@JoinColumn`(name = "book_id")

- **inverseJoinColumns**

  FK(s) desde la join table hacia la otra entidad.

  inverseJoinColumns = `@JoinColumn`(name = "author_id")

Hibernate. Versión final de las Entidades

Después de los conceptos vistos, veamos como quedaría n las 3 Entities (Author, Publisher y Book)

Hibernate. Entity Publisher

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

    @Column(nullable = false)
    private String name;

    @OneToMany(mappedBy = "publisher")
    private final List<Book> books;

    protected Publisher() {
        this(null);
    }

    public Publisher(String name) {
        books = new ArrayList<>();
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public List<Book> getBooks() {
        return books;
    }

    @Override
    public final boolean equals(Object o) {
        if (!(o instanceof Publisher publisher)) return false;

        return id.equals(publisher.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }

    @Override
    public String toString() {
        return "Publisher{" +
                "id=" + id +
                ", name='" + name +
                '}';
    }
}
```

Hibernate. Entity Author

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

    @Column(nullable = false)
    private final String name;

    // Inverse side
    @ManyToMany(mappedBy = "authors")
    private final Set<Book> books;

    protected Author() {
        this(null);
    }

    public Author(String name) {
        books = new HashSet<>();
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public Set<Book> getBooks() {
        return books;
    }

    @Override
    public final boolean equals(Object o) {
        if (!(o instanceof Author author)) return false;
        return id.equals(author.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }

    @Override
    public String toString() {
        return "Author{" +
                "id=" + id +
                ", name='" + name +
                '}';
    }
}
```

Hibernate. Entity Book

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

    @Column(nullable = false)
    private final String title;

    // Owning side
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(
            name = "book_author",
            joinColumns = @JoinColumn(name = "book_id", nullable = false),
            inverseJoinColumns = @JoinColumn(name = "author_id", nullable = false)
    )
    private final Set<Author> authors;

    @ManyToOne(optional = false)
    @JoinColumn(name = "publisher_id", nullable = false)
    private final Publisher publisher;

    protected Book() {
        this(null, null);
    }

    public Book(String title, Publisher publisher) {
        this.title = title;
        this.publisher = publisher;
        this.authors = new HashSet<>();
    }

    // Helpers para sincronizar ambos lados
    public void addAuthor(Author author) {
        authors.add(author);
        author.getBooks().add(this);
    }

    // Helpers para sincronizar ambos lados
    public void removeAuthor(Author author) {
        authors.remove(author);
        author.getBooks().remove(this);
    }

    public Long getId() { return id;}

    public String getTitle() { return title; }

    public Set<Author> getAuthors() { return authors; }

    public Publisher getPublisher() { return publisher; }

    @Override
    public final boolean equals(Object o) {
        if (!(o instanceof Book book)) return false;
        return id.equals(book.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }

    @Override
    public String toString() {
        return "Book{id=" + id +
                ", title='" + title + '\'' +
                ", publisherId=" + (publisher != null ? publisher.getId() : null) +
                ", authorsCount=" + authors.size() +
                '}';
    }
}
```

Hibernate. Elección de colecciones

Recordemos:

- **Set**: para elementos sin duplicados y sin relación de orden.
- **List**: relación ordenada o con índice
- **Map**: relación clave-valor.

Hibernate recomienda:

- **HashSet** para relaciones N:N ya que:
  - Evita duplicados de forma natural
  - O(1) en add, remove, contains
- **List** para cuando existe una relación de orden entre los elementos.
- **Map** solo para almacenar claves-valor.

Hibernate. SessionFactory

Hibernate separa claramente configuración de uso.

- La configuración (BD, dialecto, entidades, mappings...) se hace una vez.
- El uso (consultas, inserts, transacciones) se hace muchas veces.

SessionFactory representa Hibernate ya configurado y listo para poder crear Session.

Session representa la clase con la que interactuamos con Hibernate.

Hibernate. SessionFactory

La clase **SessionFactory** contiene:

- Configuración de Hibernate
- Metadatos de las entidades (Author, Book, Publisher)
- Dialecto SQL
- Pool de conexiones (si está configurado)
- Estrategias de generación de IDs
- Caché de segundo nivel (si existe)

Es, literalmente, la fábrica de Session

Hibernate. Clase auxiliar para la configuración

```
public class HibernateUtil {
    private static final SessionFactory sessionFactory = build();

    private static SessionFactory build() {
        var registry = new StandardServiceRegistryBuilder()
                .configure()
                .build();

        return new MetadataSources(registry)
                .addAnnotatedClass(Author.class)
                .addAnnotatedClass(Book.class)
                .addAnnotatedClass(Publisher.class)
                .buildMetadata()
                .buildSessionFactory();
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Con esta clase auxiliar garantizamos:

- Una sola instancia
- Acceso global controlado

Hibernate. Session

La clase **Session** representa la conversación de Hibernate con la base de datos. Sirve para:

- Guardar entidades
- Recuperarlas
- Modificarlas
- Borrarlas
- Ejecutar consultas
- Manejar transacciones

Hibernate. Características de Session

- NO es thread-safe
- Vive poco tiempo
- Está asociada a una transacción
- Mantiene un contexto de persistencia (1st level cache)
- Cada Session tiene su propio “mundo” de objetos

```
try (Session session =
HibernateUtil.getSessionFactory().openSession()) {
    session.beginTransaction();

    // Operaciones

    session.getTransaction().commit();
}
```

Hibernate. Métodos de Session

Transaction **beginTransaction**()

- Inicia una transacción
- Obligatorio para escritura
- Recomendado también para lectura
- Hibernate no hace auto-commit.

Transaction **getTransaction**()

- Devuelve la transacción actual
- Permite realizar commit() o rollback()

Hibernate. Métodos de Session

void **persist**(Object o)

- Pasa la entidad a estado persistent
- Programa un INSERT
- NO devuelve el id
- El INSERT ocurre en flush/commit (salvo en `GenerationType.`*`IDENTITY`*)

Object **find**(...)

- Busca por clave primaria
- Usa caché de primer nivel
- Devuelve null si no existe

List<Object> **findMultiple**(...)

- Versión en lote de find(), es decir, busca varios objetos pasándole una lista de ids.

Hibernate. Métodos de Session

Query<R> **createQuery**(...)

- Crea una consulta en HQL/JPQL, que usa:
  - **nombres de clases** (Author, Book)
  - nombres de **propiedades** (name, title, publisher)
  - y relaciones (joins por asociaciones)
- Ventajas:
  - No dependes de nombres reales de tabla/columna
  - Más portable
  - Hibernate entiende el modelo y genera SQL
- Métodos típicos del Query
  - setParameter(...)
  - getResultList()

Hibernate. Métodos de Session

NativeQuery<R> **createNativeQuery**(...)

- Crea una consulta en **SQL real**, tal cual se escribiría en PostgreSQL.
- Ventajas:
  - Puedes usar **SQL específico** (CTEs, ILIKE, JSONB, funciones, DISTINCT ON, etc.)
  - Puedes **optimizar** o hacer consultas complejas
- Desventajas:
  - Dependes de **nombres de tablas/columnas** reales
  - Menos portable
  - Hibernate no “entiende” el modelo, solo mapea el resultado
- Métodos típicos del NativeQuery
  - setParameter(...)
  - getResultList()

Hibernate. Métodos de Session

void **remove**(Object o)

- Programa un DELETE
- El DELETE se ejecuta en flush() / commit().
- La entidad debe estar **managed**, es decir, que Hibernate la esté siguiendo dentro de la Session activa.

En Hibernate las entidades pueden estar en **4 estados**:

- **Transient**
  - Objeto Java normal. Ejemplo: Author a = new Author();
  - No está en la Session.
  - No tiene porqué tener id
- **Managed**
  - El Objecto está asociado a una Session
  - Ejemplo: Author a = session.find(Author.class, 1);
- **Detached**
  - La entidad tuvo una Session, pero ya no.
  - remove lanza IllegalArgumentException
- **Removed**
  - La entidad está managed, pero marcada para borrado.

Hibernate. Métodos de Session

void **flush**()

- Fuerza la sincronización con la BD
- Ejecuta SQL pendiente
- NO hace commit

void **clear**()

- Vacía el contexto de persistencia
- CUIDADO: Todas las entidades pasan a detached
- Útil para:
  - grandes volúmenes
  - evitar consumo de memoria

Hibernate. Métodos de Session

boolean **contains**(Object o)

- Devuelve true si la entidad recibida como parámetro está gestionada
- Muy útil para depuración.

void **close**()

- Libera recursos
- Devuelve conexión al pool
- Invalida la sesión

Hibernate. Crear y utilizar Session

```
try (var session = HibernateUtil.getSessionFactory().openSession()) {
    session.beginTransaction();
    // Operaciones
    Publisher p1 = new Publisher("Prentice Hall");
    session.persist(p1);
    Publisher p2 = new Publisher("Addison Wesley");
    session.persist(p2);
    Author a = new Author("Brian Kerninghan");
    session.persist(a);

    Book b1 = new Book("The C Programming Language", p1);
    session.persist(b1);
    b1.addAuthor(a);
    Book b2 = new Book("The Practice of Programming", p2);
    session.persist(b2);
    b2.addAuthor(a);

    session.getTransaction().commit();
}
```

Hibernate. Consultas con Session

Veamos ahora cómo realizar consultas

```
try (var session = HibernateUtil.getSessionFactory().openSession()) {
    session.beginTransaction();

     // Listar todos los autores
     // createQuery utiliza nombres de clases y atributos Java no de la base de datos
     var authors = session.createQuery("from Author", Author.class)
        .getResultList();
     System.out.println("createQuery autores -> " + authors);

     // find para obtener un elemento simple
     Author author = session.find(Author.class, 1);  // null si no existe
     System.out.println("Autor con id = 1 -> " + author);

     // createQuery con parámetros
     String term = "martin";
     authors = session.createQuery(
                "select a from Author a where lower(a.name) like :p",
                Author.class
        ).setParameter("p", "%" + term.toLowerCase() + "%")
        .getResultList();
     System.out.println("Autores con nombre like %martin% -> " + authors);

     session.getTransaction().commit();
}
```

Hibernate. Consultas con Session

```java
try (var session = HibernateUtil.getSessionFactory().openSession()) {
      session.beginTransaction();

      // Paginación
      int page = 0;      // 0..n
      int pageSize = 10;
      var books = session.createQuery("from Book order by id", Book.class)
                         .setFirstResult(page * pageSize)
                         .setMaxResults(pageSize)
                         .getResultList();
      System.out.println("Books paginados -> " + books);

      // Criteria API
      var cb = session.getCriteriaBuilder();
      var cq = cb.createQuery(Book.class);
      var root = cq.from(Book.class);
      term = "clean";
      cq.select(root).where(cb.like(cb.lower(root.get("title")), "%" + term.toLowerCase() + "%"));
      books = session.createQuery(cq).getResultList();
      System.out.println("Criteria books result -> " + books);

      // SQL Nativo puro
      // createNativeQuery consultas directas a tablas/columnas.
      // Utiliza nombres de tablas y campos de tabla.
      // Potentes pero menos portables y más delicadas.
      term = "";
      authors = session.createNativeQuery(
                "select * from author where name ilike :p",
                Author.class
       ).setParameter("p", "%" + term + "%").getResultList();

      session.getTransaction().commit();
}
```

Hibernate. Generación de IDs

A lo largo del tema hemos utilizado IDs generados por la base de datos (`GenerationType.IDENTITY`).

Ventajas

- Simplicidad
- Unicidad garantizada
- Integración natural con relaciones (Fks simples, índices eficientes).

Inconvenientes

- El ID no existe hasta hacer INSERT
- Acoplamiento fuerte a la BD
- Problemas en sistemas distribuidos
- Migraciones y refactors más complejos (implica tocar Pks, Fks y tablas intermedias).

Hibernate. Generación de IDs con UUID

Una alternativa, cada vez más utilizada, es emplear **UUID**.

El UUID es generado por Hibernate antes del INSERT lo que permite la clave primaria antes de persistir en la base de datos.

```
@Id
@GeneratedValue
@UuidGenerator
private UUID id;
```

En este caso el atributo `id` tendrá valor antes de realizar la inserción del objeto en la base de datos.

Hibernate. Generación de IDs con UUID

Ventajas de los UUID

- El ID existe desde el **principio**
- **Independencia** de la base de datos
- Ideal para **sistemas distribuidos**
- Permite a Hibernate aplicar **optimizaciones** en batch y cascadas, ya que no necesita hacer INSERT inmediato, sino que puede **agrupar operaciones**.

Inconvenientes de los UUID

- Índices más **grandes** y más **lentos**
- UUID “aleatorio” rompe la **localidad**
  - Valores no secuenciales provocan **índices dispersos** en el índice B-Tree
- Menos **legible** para humanos
- **Tamaño** en red y serialización

Hibernate. Diferencias de rendimiento

Son muchos los factores que pueden influir en el rendimiento de una base de datos, pero en igualdad de condiciones, la diferencia entre ID's generados por la base de datos y UUID generados en Hibernate sería de forma aproximada la siguiente:

- Inserciones: UUID puede ser entre 1.3x y 2x más lento
- Índices: UUID consume aprox. 2x espacio y provoca más page splits
- JOINs / búsquedas por PK: UUID suele ser entre un 10 y un 30% más lento
- Lecturas puras: diferencia pequeña (a veces despreciable)

Hibernate. Implementación con UUID

La implementación completa del ejemplo de la base de datos de libros que hemos utilizado en los apuntes, la puedes encontrar en:

https://github.com/ggascon-ada/HibernatTemplate

En la rama **HibernateUUIDs** puedes ver una implementación de la base de datos de libros utilizando UUID.

https://github.com/ggascon-ada/HibernatTemplate/tree/HibernateUUIDs