## HIBERNATE - Persistencia relacional para Java idiomático

## 1

# Documentación de referencia de Hibernate

### 3.6.0.Final

por Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, Steve Ebersole, y Hardy Ferentschik

and thanks to James Cobb (Graphic Design), Cheyenne Weaver (Graphic Design), y Bernardo Antonio Buffa Colom&#x00e9

## Prefacio

Working with both Object-Oriented software and Relational Databases can be cumbersome and time consuming. Development costs are significantly higher due to a paradigm mismatch between how data is represented in objects versus relational databases. Hibernate is an Object/Relational Mapping solution for Java environments. The term Object/Relational Mapping refers to the technique of mapping data from an object model representation to a relational data model representation (and visa versa). See *http://en.wikipedia.org/wiki/Object-relational_mapping* for a good high-level discussion.

### Nota

While having a strong background in SQL is not required to use Hibernate, having a basic understanding of the concepts can greatly help you understand Hibernate more fully and quickly. Probably the single best background is an understanding of data modeling principles. You might want to consider these resources as a good starting point:

- *http://www.agiledata.org/essays/dataModeling101.html*
- *http://en.wikipedia.org/wiki/Data_modeling*

Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities. It can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC. Hibernate's design goal is to relieve the developer from 95% of common data persistence-related programming tasks by eliminating the need for manual, hand-crafted data processing using SQL and JDBC. However, unlike many other persistence solutions, Hibernate does not hide the power of SQL from you and guarantees that your investment in relational technology and knowledge is as valid as always.

Hibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the Java-based middle-tier. However, Hibernate can certainly help you to remove or encapsulate vendor-specific SQL code and will help with the common task of result set translation from a tabular representation to a graph of objects.

Si usted es nuevo en el tema de Hibernate y del Mapeo Objeto/Relacional o inclusive en Java por favor siga los siguientes pasos:

1. Read *Capítulo 1, Tutorial* for a tutorial with step-by-step instructions. The source code for the tutorial is included in the distribution in the `doc/reference/tutorial/` directory.
2. Read *Capítulo 2, Arquitectura* to understand the environments where Hibernate can be used.

3. Déle un vistazo al directorio `eg/` en la distribución de Hibernate. Este comprende una aplicación autónoma simple. Copie su compilador JDBC al directorio `lib/` y edite `etc/hibernate.properties`, especificando los valores correctos para su base de datos. Desde un intérprete de comandos en el directorio de la distribución, escriba `ant eg` (utilizando Ant), o bajo Windows, escriba `build eg`.

4. Use this reference documentation as your primary source of information. Consider reading [*JPwH*] if you need more help with application design, or if you prefer a step-by-step tutorial. Also visit *http://caveatemptor.hibernate.org* and download the example application from [*JPwH*].

5. En el sitio web de Hibernate encontrará las respuestas a las preguntas más frecuentes.

6. En el sitio web de Hibernate encontrará los enlaces a las demostraciones de terceros, ejemplos y tutoriales.

7. El área de la comunidad en el sitio web de Hibernate es un buen recurso para encontrar patrones de diseño y varias soluciones de integración (Tomcat, JBoss AS, Struts, EJB, etc).

There are a number of ways to become involved in the Hibernate community, including

- Trying stuff out and reporting bugs. See *http://hibernate.org/issuetracker.html* details.
- Trying your hand at fixing some bugs or implementing enhancements. Again, see *http://hibernate.org/issuetracker.html* details.
- *http://hibernate.org/community.html* list a few ways to engage in the community.
  - There are forums for users to ask questions and receive help from the community.
  - There are also *IRC* [http://en.wikipedia.org/wiki/Internet_Relay_Chat] channels for both user and developer discussions.
- Helping improve or translate this documentation. Contact us on the developer mailing list if you have interest.
- Evangelizing Hibernate within your organization.

# Tutorial

Dirigido a los nuevos usuarios, este capítulo brinda una introducción a Hibernate paso por paso, empezando con una aplicación simple usando una base de datos en memoria. Este tutorial se basa en un tutorial anterior que Michael Gloegl desarrolló. Todo el código se encuentra en el directorio `tutorials/web` de la fuente del proyecto.

**Importante**

Este tutorial se basa en que el usuario tenga conocimiento de Java y SQL. Si tiene un conocimiento muy limitado de JAVA o SQL, le aconsejamos que empiece con una buena introducción a esta tecnología antes de tratar de aprender sobre Hibernate.

**Nota**

La distribución contiene otra aplicación de ejemplo bajo el directorio fuente del proyecto `tutorial/eg`.

## 1.1. Parte 1 - La primera aplicación Hibernate

Para este ejemplo, vamos a configurar una aplicación base de datos pequeña que pueda almacenar eventos a los que queremos asistir e información sobre los anfitriones de estos eventos.

**Nota**

Aunque puede utilizar cualquier base de datos con la que se sienta bien, vamos a usar *HSQLDB* [http://hsqldb.org/] (una base de datos Java en-memoria) para evitar describir la instalación/configuración de cualquier servidor de base de datos en particular.

### 1.1.1. Configuración

Lo primero que tenemos que hacer es configurar el entorno de desarrollo. Vamos a utilizar el "diseño estándar" apoyado por muchas herramientas de construcción tal como *Maven* [http://maven.org]. Maven, en particular, tiene un buen recurso que describe este *diseño* [http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html]. Como este tutorial va a ser una aplicación web, vamos a crear y a utilizar los directorios `src/main/java`, `src/main/resources` y `src/main/webapp`.

Vamos a usar Maven en este tutorial, sacando ventaja de sus funcionalidades de administración de dependencias transitivas así como la habilidad de muchos IDEs para configurar automáticamente un proyecto para nosotros con base en el descriptor maven.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.hibernate.tutorials</groupId>
    <artifactId>hibernate-tutorial</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <name>First Hibernate Tutorial</name>

    <build>
         <!-- we dont want the version to be part of the generated war file name -->
         <finalName>${artifactId}</finalName>
    </build>

    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
        </dependency>

        <!-- Because this is a web app, we also have a dependency on the servlet api. -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
        </dependency>

        <!-- Hibernate uses slf4j for logging, for our purposes here use the simple backend -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
        </dependency>

        <!-- Hibernate gives you a choice of bytecode providers between cglib and javassist -->
        <dependency>
            <groupId>javassist</groupId>
            <artifactId>javassist</artifactId>
        </dependency>
    </dependencies>

</project>
```

### Sugerencia

It is not a requirement to use Maven. If you wish to use something else to build this tutorial (such as Ant), the layout will remain the same. The only change is that you will need to manually account for all the needed dependencies. If you

use something like *Ivy* [http://ant.apache.org/ivy/] providing transitive dependency management you would still use the dependencies mentioned below. Otherwise, you'd need to grab *all* dependencies, both explicit and transitive, and add them to the project's classpath. If working from the Hibernate distribution bundle, this would mean `hibernate3.jar`, all artifacts in the `lib/required` directory and all files from either the `lib/bytecode/cglib` or `lib/bytecode/javassist` directory; additionally you will need both the servlet-api jar and one of the slf4j logging backends.

Guarde este archivo como `pom.xml` en el directorio raíz del proyecto.

### 1.1.2. La primera clase

Luego creamos una clase que representa el evento que queremos almacenar en la base de datos, es una clase JavaBean simple con algunas propiedades:

```
package org.hibernate.tutorial.domain;

import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

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

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

Esta clase utiliza convenciones de nombrado estándares de JavaBean para los métodos de propiedades getter y setter así como también visibilidad privada para los campos. Se recomienda este diseño, pero no se exige. Hibernate también puede acceder a los campos directamente, los métodos de acceso benefician la robustez de la refactorización.

La propiedad `id` tiene un valor identificador único para un evento en particular. Todas las clases de entidad persistentes necesitarán tal propiedad identificadora si queremos utilizar el grupo completo de funcionalidades de Hibernate (también algunas clases dependientes menos importantes). De hecho, la mayoría de las aplicaciones (en especial las aplicaciones web) necesitan distinguir los objetos por identificador, así que usted debe tomar esto como una funcionalidad más que una limitación. Sin embargo, usualmente no manipulamos la identidad de un objeto, por lo tanto, el método setter debe ser privado. Sólamente Hibernate asignará identificadores cuando se guarde un objeto. Como se puede ver, Hibernate puede acceder a métodos de acceso públicos, privados y protegidos, así como también a campos directamente públicos, privados y protegidos. Puede escoger y hacer que se ajuste a su diseño de su aplicación.

El constructor sin argumentos es un requerimiento para todas las clases persistentes, Hibernate tiene que crear objetos por usted utilizando Java Reflection. El constructor puede ser privado; sin embargo, se necesita la visibilidad del paquete para generar proxies en tiempo de ejecución y para la recuperación de datos de manera efectiva sin la instrumentación del código byte.

Duarde este archivo en el directorio `src/main/java/org/hibernate/tutorial/domain`.

### 1.1.3. El archivo de mapeo

Hibernate necesita saber cómo cargar y almacenar objetos de la clase persistente. En este punto es donde entra en juego el archivo de mapeo de Hibernate. Este archivo le dice a Hibernate a que tabla tiene que acceder en la base de datos, y que columnas debe utilizar en esta tabla.

La estructura básica de un archivo de mapeo se ve así:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.hibernate.tutorial.domain">
[...]
</hibernate-mapping
>
```

El DTD de Hibernate es sofisticado. Puede utilizarlo para autocompletar los elementos y atributos XML de mapeo en su editor o IDE. Abrir el archivo DTD en su editor de texto es la manera más fácil para obtener una sinopsis de todos los elementos y atributos y para ver los valores por defecto, así como algunos de los comentarios. Note que Hibernate no cargará el fichero DTD de la web, sino que primero lo buscará en la ruta de clase de la aplicación. El archivo DTD se encuentra incluido en `hibernate-core.jar` (también en `hibernate3.jar` si está usando el paquete de la distribución).

**Importante**

Omitiremos la declaración de DTD en los ejemplos posteriores para hacer más corto el código. Esto no es opcional.

Entre las dos etiquetas `hibernate-mapping`, incluya un elemento `class`. Todas las clases de entidad persistentes (de nuevo, podrían haber clases dependientes más adelante, las cuales no son entidades de primera clase) necesitan de dicho mapeo en una tabla en la base de datos SQL:

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">

    </class>

</hibernate-mapping>
```

Hasta ahora le hemos dicho a Hibernate cómo persistir y cargar el objeto de clase `Event` a la tabla `EVENTS`. Cada instancia se encuentra representada por una fila en esa tabla. Ahora podemos continuar mapeando la propiedad identificadora única a la clave primaria de la tabla. Ya que no queremos preocuparnos por el manejo de este identificador, configuramos la estrategia de generación del identificador de Hibernate para una columna clave primaria sustituta:

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
    </class>

</hibernate-mapping>
```

El elemento `id` es la declaración de la propiedad identificadora. El atributo de mapeo `name="id"` declara el nombre de la propiedad JavaBean y le dice a Hibernate que utilice los métodos `getId()` y `setId()` para acceder a la propiedad. El atributo columna le dice a Hibernate qué columna de la tabla `EVENTS` tiene el valor de la llave principal.

El elemento anidado `generator` especifica la estrategia de generación del identificador (también conocidos como ¿cómo se generan los valores del identificador?). En este caso escogimos `native`, el cual ofrece un nivel de qué tan portátil es dependiendo del dialecto configurado de la base de datos. Hibernate soporta identificadores generados por la base de datos, globalmente únicos así como asignados por la aplicación. La generación del valor del identificador también es uno de los muchos puntos de extensión de Hibernate y puede conectar su propia estrategia.

### Sugerencia

`native` is no longer consider the best strategy in terms of portability. for further discussion, see *Sección 28.4, "Generación del identificador"*

Por último es necesario decirle a Hibernate sobre las porpiedades de clase de entidad que quedan. Por defecto, ninguna propiedad de la clase se considera persistente:

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```

Al igual que con el elemento `id`, el atributo `name` del elemento `property` le dice a Hibernate que métodos getter y setter utilizar. Así que en este caso, Hibernate buscará los métodos `getDate()`, `setDate()`, `getTitle()` y `setTitle()`.

### Nota

¿Por qué el mapeo de la propiedad `date` incluye el atributo `column`, pero el de `title` no? Sin el atributo `column` Hibernate utiliza, por defecto, el nombre de propiedad como nombre de la columna. Esto funciona bien para `title`. Sin embargo, `date` es una palabra clave reservada en la mayoría de las bases de datos, así que es mejor que la mapeamos a un nombre diferente.

El mapeo de `title` carece de un atributo `type`. Los tipos que declaramos y utilizamos en los archivos de mapeo no son tipos de datos Java. Tampoco son tipos de base de datos SQL. Estos tipos se llaman *tipos de mapeo Hibernate* , convertidores que pueden traducir de tipos de datos de Java a SQL y viceversa. De nuevo, Hibernate tratará de determinar el tipo correcto de conversión y de mapeo por sí mismo si el atributo `type` no se encuentra presente en el mapeo. En algunos casos esta detección automática (utilizando Reflection en la clase Java) puede que no tenga lo que usted espera o necesita. Este es el caso de la propiedad `date`. Hibernate no puede saber is la propiedad, la cual es de `java.util.Date`, debe mapear a una columna `date`, `timestamp` o `time` de SQL. Por medio de un convertidor `timestamp`, mapeamos la propiedad y mantenemos la información completa sobre la hora y fecha.

### Sugerencia

Hibernate realiza esta determinación de tipo de mapeo usando reflection cuando se procesan los archivos de mapeo. Esto puede tomar tiempo y recursos así que el rendimiento al arrancar es importante entonces debe considerar el definir explícitamente el tipo a usar.

Guarde este archivo de mapeo como `src/main/resources/org/hibernate/tutorial/domain/Event.hbm.xml`.

## 1.1.4. Configuración de Hibernate

En este momento debe tener la clase persistente y su archivo de mapeo. Ahora debe configurar Hibernate. Primero vamos a configurar HSQLDB para que ejecute en "modo de servidor"

### Nota

Hacemos esto o lo otro y los datos permanecen entre ejecuciones.

Vamos a utilizar el plugin de ejecución Maven para lanzar el servidor HSQLDB ejecutando: `mvn exec:java -Dexec.mainClass="org.hsqldb.Server" -Dexec.args="-database.0 file:target/data/tutorial"`.Lo verá iniciando y vinculandose a un enchufe TCP/IP, allí es donde nuestra aplicación se conectará más adelante. Si quiere dar inicio con una base de datos fresca durante este tutorial, apague HSQLDB, borre todos los archivos en el directorio `target/data` e inicie HSQLDB de nuevo.

Hibernate se conectará a la base de datos de parte de su aplicación así que necesita saber cómo obtener conexiones. Para este tutorial vamos a utilizar un pool de conexiones autónomo (opuesto a `javax.sql.DataSource`). Hibernate viene con soporte para dos pools de conexiones JDBC de código abierto de terceros: *c3p0* [https://sourceforge.net/projects/c3p0] y *proxool* [http://proxool.sourceforge.net/]. Sin embargo, vamos a utilizar el pool de conexiones incluido de Hibernate para este tutorial.

### Atención

El pool de conexiones de Hibernate no está diseñado para utilizarse en producción. Le faltan varias funcionalidades que se encuentran en cualquier pool de conexiones decente.

Para la configuración de Hibernate, podemos utilizar un archivo `hibernate.properties` simple, un archivo `hibernate.cfg.xml` un poco más sofisticado, o incluso una configuración completamente programática. La mayoría de los usuarios prefieren el archivo de configuración XML:

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

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class"
>org.hsqldb.jdbcDriver</property>
        <property name="connection.url"
>jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username"
>sa</property>
        <property name="connection.password"
></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size"
>1</property>

        <!-- SQL dialect -->
        <property name="dialect"
>org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class"
>thread</property>

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class"
>org.hibernate.cache.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql"
>true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto"
>update</property>

        <mapping resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration
>
```

**Nota**

Observe que este archivo de configuración especifica un DTD diferente

Configure la `SessionFactory` de Hibernate. SessionFactory es una fábrica global responsable de una base de datos en particular. Si usted tiene varias bases de datos, para un inicio más fácil utilice varias configuraciones `<session-factory>` en varios archivos de configuración.

Los primeros cuatro elementos `property` contienen la configuración necesaria para la conexión JDBC. El elemento `property` dialecto especifica la variante SQL en particular que Hibernate genera.

**Sugerencia**

In most cases, Hibernate is able to properly determine which dialect to use. See *Sección 28.3, "Resolución del dialecto"* for more information.

La administración de la sesión automática de Hibernate para contextos de persistencia es particularmente útil en este contexto. La opción `hbm2ddl.auto` activa la generación automática de los esquemas de la base de datos directamente en la base de datos. Esto se puede desactivar, eliminando la opción de configuración o redirigiéndolo a un archivo con la ayuda de la tarea de Ant `SchemaExport`. Finalmente, agregue a la configuración el/los fichero(s) de mapeo para clases persistentes.

Guarde este archivo como `hibernate.cfg.xml` en el directorio `src/main/resources`.

### 1.1.5. Construcción con Maven

Ahora vamos a construir el tutorial con Maven. Es necesario que tenga instalado Maven; se encuentra disponible en la *página de descargas Maven* [http://maven.apache.org/download.html]. Maven leerá el archivo `/pom.xml` que creamos anteriormente y sabrá cómo realizar algunas tareas de proyectos básicos. Primero, vamos a ejecutar la meta `compile` para asegurarnos de que podemos compilar todo hasta el momento:

```
[hibernateTutorial]$ mvn compile
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building First Hibernate Tutorial
[INFO]    task-segment: [compile]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to /home/steve/projects/sandbox/hibernateTutorial/target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Tue Jun 09 12:25:25 CDT 2009
[INFO] Final Memory: 5M/547M
[INFO] ------------------------------------------------------------------------
```

### 1.1.6. Inicio y ayudantes

Es el momento de cargar y almacenar algunos objetos `Event`, pero primero tiene que completar la configuración con algo de código de infraestructura. Tiene que iniciar Hibernate construyendo un objeto `org.hibernate.SessionFactory` global y almacenarlo en algún lugar de fácil acceso en el código de la aplicación. Una `org.hibernate.SessionFactory` se utiliza para obtener instancias `org.hibernate.Session`. Una `org.hibernate.Session` representa una unidad de trabajo mono-hilo. La `org.hibernate.SessionFactory` es un objeto global seguro entre hilos que se instancia una sóla vez.

Vamos a crear una clase de ayuda `HibernateUtil` que se encargue del inicio y haga más práctico el acceso a `org.hibernate.SessionFactory`.

```
package org.hibernate.tutorial.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return new Configuration().configure().buildSessionFactory();
        }
        catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

Guarde este código como `src/main/java/org/hibernate/tutorial/util/HibernateUtil.java`

Esta clase no sólamente produce la referencia `org.hibernate.SessionFactory` global en su inicializador estático, sino que también esconde el hecho de que utiliza un singleton estático. También puede que busque la referencia `org.hibernate.SessionFactory` desde JNDI en un servidor de aplicaciones en cualquier otro lugar.

Si usted le da un nombre a `org.hibernate.SessionFactory` en su archivo de configuración, de hecho, Hibernate tratará de vincularlo a JNDI bajo ese nombre después de que ha sido construido. Otra mejor opción es utilizar el despliegue JMX y dejar que el contenedor con capacidad JMX

instancie y vincule un `HibernateService` a JNDI. Más adelante discutiremos estas opciones avanzadas.

Ahora necesita configurar un sistema de registro. Hibernate utiliza registros comunes le da dos opciones: Log4J y registros de JDK 1.4. La mayoría de los desarrolladores prefieren Log4J: copie `log4j.properties` de la distribución de Hibernate, se encuentra en el directorio `etc/`) a su directorio `src`, junto a `hibernate.cfg.xml`. Si desea tener una salida más verbosa que la que se proporcionó en la configuración del ejemplo entonces puede cambiar su configuración. Por defecto, sólo se muestra el mensaje de inicio de Hibernate en la salida estándar.

La infraestructura del tutorial está completa y estamos listos para hacer un poco de trabajo real con Hibernate.

### 1.1.7. Carga y almacenamiento de objetos

We are now ready to start doing some real work with Hibernate. Let's start by writing an `EventManager` class with a `main()` method:

```
package org.hibernate.tutorial;

import org.hibernate.Session;

import java.util.*;

import org.hibernate.tutorial.domain.Event;
import org.hibernate.tutorial.util.HibernateUtil;

public class EventManager {

    public static void main(String[] args) {
        EventManager mgr = new EventManager();

        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();

        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);
        session.save(theEvent);

        session.getTransaction().commit();
    }

}
```

En `createAndStoreEvent()` creamos un nuevo objeto `Event` y se lo entregamos a Hibernate. En ese momento, Hibernate se encarga de SQL y ejecuta un `INSERT` en la base de datos.

A org.hibernate.Session is designed to represent a single unit of work (a single atomic piece of work to be performed). For now we will keep things simple and assume a one-to-one granularity between a Hibernate org.hibernate.Session and a database transaction. To shield our code from the actual underlying transaction system we use the Hibernate `org.hibernate.Transaction` API. In this particular case we are using JDBC-based transactional semantics, but it could also run with JTA.

¿Qué hace `sessionFactory.getCurrentSession()`? Primero, la puede llamar tantas veces como desee y en donde quiera, una vez consiga su `org.hibernate.SessionFactory`. El método `getCurrentSession()` siempre retorna la unidad de trabajo "actual". ¿Recuerda que cambiamos la opción de la configuración de este mecanismo a "thread" en `src/main/resources/hibernate.cfg.xml`? Por lo tanto, el contexto de una unidad de trabajo actual se encuentra vinculada al hilo de Java actual que ejecuta nuestra aplicación.

**Importante**

Hibernate ofrece tres métodos de rastreo de sesión actual. El método basado en "hilos" no está dirigido al uso de producción; sólo es útil para prototipos y para tutoriales como este. Más adelante discutiremos con más detalles el rastreo de sesión actual.

Una org.hibernate.Session se inicia cuando se realiza la primera llamada a `getCurrentSession()` para el hilo actual. Luego Hibernate la vincula al hilo actual. Cuando termina la transacción, ya sea por medio de guardar o deshacer los cambios, Hibernate desvincula automáticamente la org.hibernate.Session del hilo y la cierra por usted. Si llama a `getCurrentSession()` de nuevo, obtiene una org.hibernate.Session nueva y obtiene una nueva org.hibernate.Session unidad de trabajo.

En relación con la unidad del campo de trabajo, ¿Se debería utilizar org.hibernate.Session de Hibernate para ejecutar una o varias operaciones de la base de datos? El ejemplo anterior utiliza una org.hibernate.Session para una operación. Sin embargo, esto es pura coincidencia; el ejemplo simplemente no es lo suficientemente complicado para mostrar cualquier otro enfoque. El ámbito de una org.hibernate.Session de Hibernate es flexible pero nunca debe diseñar su aplicación para que utilice una nueva org.hibernate.Session de Hibernate para *cada* operación de la base de datos. Aunque lo utilizamos en los siguientes ejemplos, considere la *sesión-por-operación* como un anti-patrón. Más adelante en este tutorial, se muestra una aplicación web real, lo cual le ayudará a ilustrar esto.

See *Capítulo 13, Transacciones y concurrencia* for more information about transaction handling and demarcation. The previous example also skipped any error handling and rollback.

Para ejecutar esto, utilizaremos el plugin de ejecución Maven para llamar nuestra clase con la configuración de ruta de clase necesaria: `mvn exec:java -Dexec.mainClass="org.hibernate.tutorial.EventManager" -Dexec.args="store"`

**Nota**

Es posible que primero necesite realizar `mvn compile`.

Debe ver que Hibernate inicia y dependiendo de su configuración, también verá bastantes salidas de registro. Al final, verá la siguiente línea:

```
[java] Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)
```

Este es el `INSERT` que Hibernate ejecuta.

Para listar los eventos almacenados se agrega una opción al método principal:

```
        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }
        else if (args[0].equals("list")) {
            List events = mgr.listEvents();
            for (int i = 0; i < events.size(); i++) {
                Event theEvent = (Event) events.get(i);
                System.out.println(
                        "Event: " + theEvent.getTitle() + " Time: " + theEvent.getDate()
                );
            }
        }
```

También agregamos un método `listEvents()`:

```
    private List listEvents() {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();
        List result = session.createQuery("from Event").list();
        session.getTransaction().commit();
        return result;
    }
```

Here, we are using a Hibernate Query Language (HQL) query to load all existing `Event` objects from the database. Hibernate will generate the appropriate SQL, send it to the database and populate `Event` objects with the data. You can create more complex queries with HQL. See *Capítulo 16, HQL: El lenguaje de consulta de Hibernate* for more information.

Ahora podemos llamar nuestra nueva funcionalidad, de nuevo usando el plugin de ejecución Maven: `mvn exec:java -Dexec.mainClass="org.hibernate.tutorial.EventManager" -Dexec.args="list"`

## 1.2. Part 2 - Mapeo de asociaciones

Hasta ahora hemos mapeado una clase de entidad persistente a una tabla aislada. Vamos a construir sobre esto y agregaremos algunas asociaciones de clase. Vamos a agregar personas a la aplicación y vamos a almacenar una lista de eventos en las que participan.

### 1.2.1. Mapeo de la clase Person

El primer corte de la clase `Person` se ve así:

```
package org.hibernate.tutorial.domain;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for 'id'

}
```

Guarde esto en un archivo llamado `src/main/java/org/hibernate/tutorial/domain/Person.java`

Luego, cree el nuevo archivo de mapeo como `src/main/resources/org/hibernate/tutorial/domain/Person.hbm.xml`

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
    </class>

</hibernate-mapping>
```

Finalmente, añada el nuevo mapeo a la configuración de Hibernate:

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

Vamos a crear una asociación entre estas dos entidades. Las personas pueden participar en los eventos y los eventos cuentan con participantes. Las cuestiones de diseño con las que tenemos que tratar son: direccionalidad, multiplicidad y comportamiento de la colección.

## 1.2.2. Una asociación unidireccional basada en Set

Al agregar una colección de eventos a la clase `Person`, puede navegar fácilmente a los eventos de una persona en particular, sin ejecutar una petición explícita - llamando a `Person#getEvents`. En Hibernate, las asociaciones multi-valores se representan por medio de uno de los contratos del marco de colecciones Java; aquí escogimos un `java.util.Set` ya que la colección no contendrá elementos duplicados y el orden no es relevante para nuestros ejemplos.

```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }

    public void setEvents(Set events) {
        this.events = events;
    }
}
```

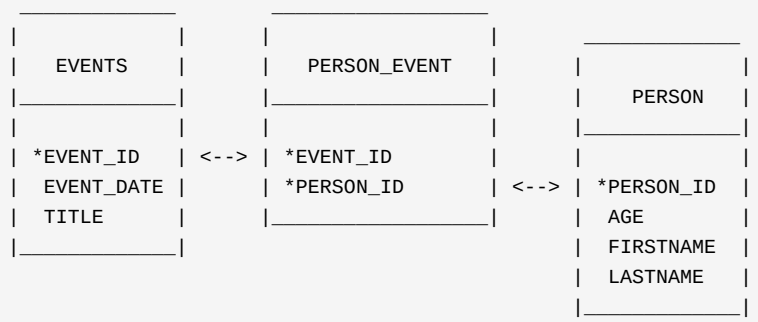
Antes de mapear esta asociación, considere el otro lado. Podriamos mantener esto unidireccional o podríamos crear otra colección en el `Event`, si queremos tener la habilidad de navegarlo desde ambas direcciones. Esto no es necesario desde un punto de vista funcional. Siempre puede ejeutar un pedido explícito para recuperar los participantes de un evento en particular. Esta es una elección de diseño que depende de usted, pero lo que queda claro de esta discusión es la multiplicidad de la asociación: "muchos" valuada en ambos lados, denominamos esto como una asociación *muchos-a-muchos*. Por lo tanto, utilizamos un mapeo muchos-a-muchos de Hibernate:

```
<class name="Person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>

    <set name="events" table="PERSON_EVENT">
        <key column="PERSON_ID"/>
        <many-to-many column="EVENT_ID" class="Event"/>
    </set>
```

```
</class>
```

Hibernate soporta un amplio rango de mapeos de colección, el más común `set`. Para una asociación muchos-a-muchos o la relación de entidad *n:m*, se necesita una tabla de asociación. Cada fila en esta tabla representa un enlace entre una persona y un evento. El nombre de esta tabla se declara con el atributo `table` del elemento `set`. El nombre de la columna identificadora en la asociación, del lado de la persona, se define con el elemento `key`, el nombre de columna para el lado del evento se define con el atributo `column` del `many-to-many`. También tiene que informarle a Hibernate la clase de los objetos en su colección (la clase del otro lado de la colección de referencias).

Por consiguiente, el esquema de base de datos para este mapeo es:




## 1.2.3. Trabajo de la asociación

Vamos a reunir a algunas personas y eventos en un nuevo método en `EventManager`:

```
private void addPersonToEvent(Long personId, Long eventId) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    Event anEvent = (Event) session.load(Event.class, eventId);
    aPerson.getEvents().add(anEvent);

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

Después de cargar una `Person` y un `Event`, simplemente modifique la colección utilizando los métodos normales de colección. No hay una llamada explícita a `update()` o `save()`; Hibernate detecta automáticamente que se ha modificado la colección y que se necesita actualizarla. Esto se denomina *chequeo automático de desactualizaciones* y también puede probarlo modificando el nombre o la propiedad de fecha de cualquiera de sus objetos. Mientras se encuentran en

estado *persistente*, es decir, enlazado a una `org.hibernate.Session` de Hibernate en particular, Hibernate monitorea cualquier cambio y ejecuta SQL de un modo escribe-detrás. El proceso de sincronización del estado de la memoria con la base de datos, usualmente sólo al final de una unidad de trabajo, se denomina *vaciado*. En nuestro código la unidad de trabajo termina con guardar o deshacer los cambios de la transacción de la base de datos.

Puede cargar una persona y un evento en diferentes unidades de trabajo. También puede modificar un objeto fuera de una `org.hibernate.Session`, cuando no se encuentra en estado persistente (si antes era persistente denominamos a este estado *separado* ). Inclusive, puede modificar una colección cuando se encuentre separada:

```
    private void addPersonToEvent(Long personId, Long eventId) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();

        Person aPerson = (Person) session
               .createQuery("select p from Person p left join fetch p.events where p.id = :pid")
                .setParameter("pid", personId)
                .uniqueResult(); // Eager fetch the collection so we can use it detached
        Event anEvent = (Event) session.load(Event.class, eventId);

        session.getTransaction().commit();

        // End of first unit of work

        aPerson.getEvents().add(anEvent); // aPerson (and its collection) is detached

        // Begin second unit of work

        Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
        session2.beginTransaction();
        session2.update(aPerson); // Reattachment of aPerson

        session2.getTransaction().commit();
    }
```

La llamada a `update` hace que un objeto separado sea persistente de nuevo enlazándolo a una nueva unidad de trabajo, así que cualquier modificación que le realizó mientras estaba separado se puede guardar en la base de datos. Esto incluye cualquier modificación (adiciones o eliminaciones) que le hizo a una colección de ese objeto entidad.

Esto no se utiliza mucho en nuestro ejemplo, pero es un concepto importante que puede incorporar en su propia aplicación. Complete este ejercicio agregando una nueva acción al método main de `EventManager` y llámela desde la línea de comandos. Si necesita los identificadores de una persona y de un evento - el método `save()` los retorna (pueda que necesite modificar algunos de los métodos anteriores para retornar ese identificador):

```
        else if (args[0].equals("addpersontoevent")) {
            Long eventId = mgr.createAndStoreEvent("My Event", new Date());
            Long personId = mgr.createAndStorePerson("Foo", "Bar");
```

```
            mgr.addPersonToEvent(personId, eventId);
            System.out.println("Added person " + personId + " to event " + eventId);
        }
```

Esto fue un ejemplo de una asociación entre dos clases igualmente importantes: dos entidades. Como se mencionó anteriormente, hay otras clases y tipos en un modelo típico, usualmente "menos importantes". Algunos de ustedes las habrán visto, como un `int` o un `java.lang.String`. Denominamos a estas clases *tipos de valor* y sus instancias *dependen* de una entidad en particular. Las instancias de estos tipos no tienen su propia identidad, ni son compartidas entre entidades. Dos personas no referencian el mismo objeto `firstname`, incluso si tienen el mismo nombre. Los tipos de valor no sólo pueden encontrarse en el JDK, sino que también puede escribir por sí mismo clases dependientes como por ejemplo, `Address` o `MonetaryAmount`. De hecho, en una aplicación Hibernate todas las clases JDK se consideran como tipos de valor.

También puede diseñar una colección de tipos de valor. Esto es conceptualmente diferente de una colección de referencias a otras entidades, pero se ve casi igual en Java.

### 1.2.4. Colección de valores

Vamos a agregar una colección de direcciones de correo electrónico a la entidad `Person`. Esto se representará como un `java.util.Set` de las instnaicas `java.lang.String`:

```
    private Set emailAddresses = new HashSet();

    public Set getEmailAddresses() {
        return emailAddresses;
    }

    public void setEmailAddresses(Set emailAddresses) {
        this.emailAddresses = emailAddresses;
    }
```

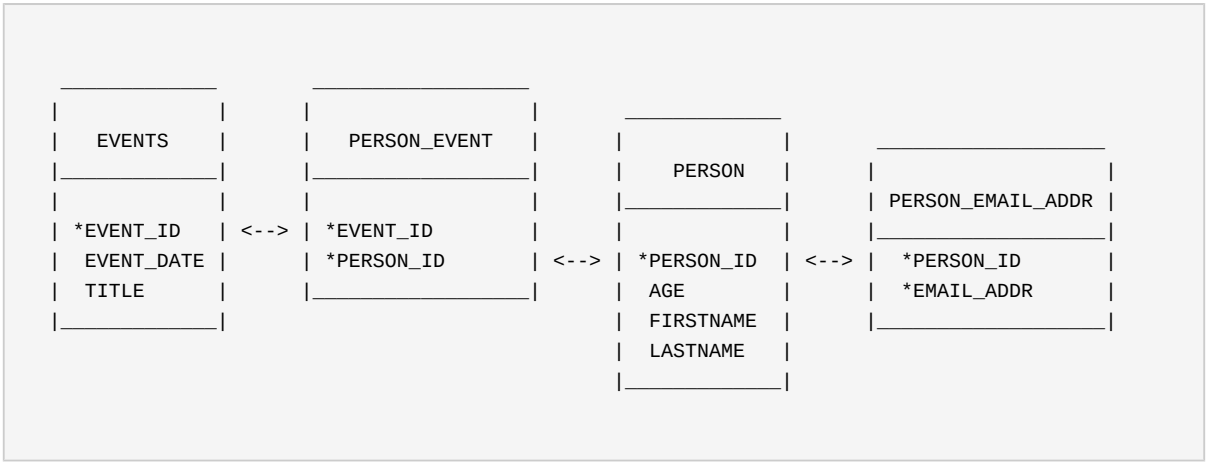
El mapeo de este `Set` es así:

```
        <set name="emailAddresses" table="PERSON_EMAIL_ADDR">
            <key column="PERSON_ID"/>
            <element type="string" column="EMAIL_ADDR"/>
        </set>
```

La diferencia comparado con el mapeo anterior es el uso de la parte `element`, que le dice a Hibernate que la colección no contiene referencias a otra entidad, sino que es una colección de elementos que son tipos de valores, aquí especificamente de tipo `String`. El nombre en minúsculas le dice que es un tipo/conversor de mapeo de Hibernate. Una vez más, el atributo `table` del elemento `set` determina el nombre de la tabla para la colección. El elemento `key` define el nombre de la columna clave foránea en la tabla de colección. El atributo `column` en el elemento

`element` define el nombre de la columna donde realmente se almacenarán los valores de la dirección de correo electrónico.

Este es el esquema actualizado:




Puede ver que la clave principal de la tabla de colección es, de hecho, una clave compuesta que utiliza ambas columnas. Esto también implica que no pueden haber direcciones de correo electrónico duplicadas por persona, la cual es exactamente la semántica que necesitamos para un conjunto en Java.

Ahora, puede tratar de agregar elementos a esta colección, al igual que lo hicimos antes vinculando personas y eventos. Es el mismo código en Java.

```
private void addEmailToPerson(Long personId, String emailAddress) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    // adding to the emailAddress collection might trigger a lazy load of the collection
    aPerson.getEmailAddresses().add(emailAddress);

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

Esta vez no utilizamos una petición de *búqueda - fetch* - para dar inicio a la colección. Monitoree su registro SQL e intente de optimizar esto con una recuperación temprana.

## 1.2.5. Asociaciones bidireccionales

A continuacion vamos a mapear una asociación bidireccional. Vamos a hacer que la asociación entre persona y evento funcione desde ambos lados en Java. El esquema de la base de datos no cambia así que todavía tendremos una multiplicidad muchos-a-muchos.

**Nota**

Una base de datos relacional es más flexible que un lenguaje de programación de red ya que no necesita una dirección de navegación; los datos se pueden ver y recuperar de cualquier forma posible.

Primero, agregue una colección de participantes a la clase `Event`:

```
private Set participants = new HashSet();

public Set getParticipants() {
    return participants;
}

public void setParticipants(Set participants) {
    this.participants = participants;
}
```

Ahora mapee este lado de la asociación en `Event.hbm.xml`.

```
<set name="participants" table="PERSON_EVENT" inverse="true">
    <key column="EVENT_ID"/>
    <many-to-many column="PERSON_ID" class="events.Person"/>
</set>
```

Estos son mapeos normales de `set` en ambos documentos de mapeo. Note que los nombres de las columnas en `key` y `many-to-many` se intercambiaron en ambos documentos de mapeo. La adición más importante aquí es el atributo `inverse="true"` en el elemento `set` del mapeo de colección de `Event`.

Esto significa que Hibernate debe tomar el otro lado, la clase `Person`, cuando necesite encontrar información sobre el enlace entre las dos. Esto será mucho más fácil de entender una vez que vea como se crea el enlace bidireccional entre nuestras dos entidades.

## 1.2.6. Trabajo con enlaces bidireccionales

Primero, recuerde que Hibernate no afecta la semántica normal de Java. ¿Cómo creamos un enlace entre `Person` y un `Event` en el ejemplo unidireccional? Agregue una instancia de `Event` a la colección de referencias de eventos de una instancia de `Person`. Si quiere que este enlace funcione bidireccionalmente, tiene que hacer lo mismo del otro lado, añadiendo una referencia `Person` a la colección en un `Event`. Este proceso de "establecer el enlace en ambos lados" es absolutamente necesario con enlaces bidireccionales.

Muchos desarrolladores programan a la defensiva y crean métodos de administración de enlaces para establecer correctamente ambos lados, (por ejemplo, en `Person`):

```
    protected Set getEvents() {
        return events;
    }

    protected void setEvents(Set events) {
        this.events = events;
    }

    public void addToEvent(Event event) {
        this.getEvents().add(event);
        event.getParticipants().add(this);
    }

    public void removeFromEvent(Event event) {
        this.getEvents().remove(event);
        event.getParticipants().remove(this);
    }
```

Los métodos get y set para la colección ahora se encuentran protegidos. Esto le permite a las clases en el mismo paquete y a las subclases acceder aún a los métodos, pero impide a cualquier otro que desordene las colecciones directamente. Repita los pasos para la colección del otro lado.

¿Y el atributo de mapeo `inverse`? Para usted y para Java, un enlace bidireccional es simplemente cuestión de establecer correctamente las referencias en ambos lados. Sin embargo, Hibernate no tiene suficiente información para organizar correctamente declaraciones `INSERT` y `UPDATE` de SQL (para evitar violaciones de restricciones). El hacer un lado de la asociación `inverse` le dice a Hibernate que lo considere un *espejo* del otro lado. Eso es todo lo necesario para que Hibernate resuelva todos los asuntos que surgen al transformar un modelo de navegación direccional a un esquema de base de datos SQL. Las reglas son muy simples: todas las asociaciones bidireccionales necesitan que uno de los lados sea `inverse`. En una asociación uno-a-muchos debe ser el lado-de-muchos; y en una asociación muchos-a-muchos, puede escoger cualquier lado.

## 1.3. Part 3 - La aplicación web EventManager

Una aplicación web de Hibernate utiliza `Session` y `Transaction` casi como una aplicación autónoma. Sin embargo, algunos patrones comunes son útiles. Ahora puede escribir un `EventManagerServlet`. Este servlet puede enumerar todos los eventos almacenados en la base de datos y proporciona una forma HTML para ingresar eventos nuevos.

### 1.3.1. Escritura de un servlet básico

Primero necesitamos crear nuestro servlet de procesamiento básico. Ya que nuestro servlet solo maneja pedidos `GET` HTTP sólamente, solo implementaremos el método `doGet()`:

```
package org.hibernate.tutorial.web;

// Imports
```

```
public class EventManagerServlet extends HttpServlet {

    protected void doGet(
            HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {

        SimpleDateFormat dateFormatter = new SimpleDateFormat( "dd.MM.yyyy" );

        try {
            // Begin unit of work
            HibernateUtil.getSessionFactory().getCurrentSession().beginTransaction();

            // Process request and render page...

            // End unit of work
            HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().commit();
        }
        catch (Exception ex) {
            HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().rollback();
            if ( ServletException.class.isInstance( ex ) ) {
                throw ( ServletException ) ex;
            }
            else {
                throw new ServletException( ex );
            }
        }
    }

}
```

Guarde este servlet como `src/main/java/org/hibernate/tutorial/web/EventManagerServlet.java`

El patrón aplicado aquí se llama *sesión-por-petición*. Cuando una petición llega al servlet, se abre una nueva `Session` de Hibernate por medio de la primera llamada a `getCurrentSession()` en el `SessionFactory`. Entonces se inicia una transacción de la base de datos. Todo acceso a los datos tiene que suceder dentro de una transacción, sin importar que los datos sean leídos o escritos . No utilice el modo auto-commit en las aplicaciones.

*No* utilice una nueva `Session` de Hibernate para cada operación de base de datos. Utilice una `Session` Hibernate que cubra el campo de todo el pedido. Utilice `getCurrentSession()` para vincularlo automáticamente al hilo de Java actual.

Después, se procesan las acciones posibles del pedido y se entrega la respuesta HTML. Llegaremos a esa parte muy pronto.

Finalmente, la unidad de trabajo termina cuando se completa el procesamiento y la entrega. Si surgió algún problema durante el procesamiento o la entrega , se presentará una excepción y la transacción de la base de datos se deshará. Esto completa el patrón `session-per-request`. En vez del código de demarcación de la transacción en todo servlet, también podría escribir un filtro de servlet. Véa el sitio web de Hibernate y el Wiki para obtener más información sobre este patrón llamado *sesión abierta en vista*. Lo necesitará tan pronto como considere representar su vista en JSP, no en un servlet.

## 1.3.2. Procesamiento y entrega

Ahora puede implementar el procesamiento del pedido y la representación de la página.

```
        // Write HTML header
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Event Manager</title></head><body>");

        // Handle actions
        if ( "store".equals(request.getParameter("action")) ) {

            String eventTitle = request.getParameter("eventTitle");
            String eventDate = request.getParameter("eventDate");

            if ( "".equals(eventTitle) || "".equals(eventDate) ) {
                out.println("<b><i>Please enter event title and date.</i></b>");
            }
            else {
                createAndStoreEvent(eventTitle, dateFormatter.parse(eventDate));
                out.println("<b><i>Added event.</i></b>");
            }
        }

        // Print page
       printEventForm(out);
       listEvents(out, dateFormatter);

       // Write HTML footer
       out.println("</body></html>");
       out.flush();
       out.close();
```

Dado que este estilo de codificación con una mezcla de Java y HTML no escalaría en una aplicación más compleja - tenga en cuenta que sólo estamos ilustrando los conceptos básicos de Hibernate en este tutorial. El código imprime una cabecera y un pie de página HTML. Dentro de esta página se imprime una forma HTML para entrada de eventos y se imprime una lista de todos los eventos en la base de datos. El primer método es trivial y su salida se realiza únicamente en HTML:

```
    private void printEventForm(PrintWriter out) {
        out.println("<h2>Add new event:</h2>");
        out.println("<form>");
        out.println("Title: <input name='eventTitle' length='50'/><br/>");
        out.println("Date (e.g. 24.12.2009): <input name='eventDate' length='10'/><br/>");
        out.println("<input type='submit' name='action' value='store'/>");
        out.println("</form>");
    }
```

El método `listEvents()` utiliza Hibernate `Session` vinculado al hilo actual para ejecutar una petición:

```
    private void listEvents(PrintWriter out, SimpleDateFormat dateFormatter) {

        List result = HibernateUtil.getSessionFactory()
                .getCurrentSession().createCriteria(Event.class).list();
        if (result.size() > 0) {
            out.println("<h2>Events in database:</h2>");
            out.println("<table border='1'>");
            out.println("<tr>");
            out.println("<th>Event title</th>");
            out.println("<th>Event date</th>");
            out.println("</tr>");
            Iterator it = result.iterator();
            while (it.hasNext()) {
                Event event = (Event) it.next();
                out.println("<tr>");
                out.println("<td>" + event.getTitle() + "</td>");
                out.println("<td>" + dateFormatter.format(event.getDate()) + "</td>");
                out.println("</tr>");
            }
            out.println("</table>");
        }
    }
```

Finalmente, la acción `store` se despacha al método `createAndStoreEvent()`, el cual también utiliza la `Session` del hilo actual:

```
    protected void createAndStoreEvent(String title, Date theDate) {
        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);

        HibernateUtil.getSessionFactory()
                .getCurrentSession().save(theEvent);
    }
```

El servlet se encuentra completo. Un pedido al servlet será procesado en una sola `Session` y `Transaction`. Como lo vimos antes en la aplicación autónoma, Hibernate puede enlazar automáticamente estos objetos al hilo actual de ejecución. Esto le da la libertad de utilizar capas en su código y acceder a la `SessionFactory` de cualquier manera que lo desee. Usualmente, usted utilizaría un diseño más sofisticado y movería el código de acceso de datos a los objetos de acceso de datos (el patrón DAO). Refiérase al Wiki de Hibernate para ver más ejemplos.

### 1.3.3. Despliegue y prueba

Para implementar esta aplicación para prueba debemos crear una Web ARchive (WAR). Primero debemos definir el descriptor WAR como `src/main/webapp/WEB-INF/web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee  http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd">

    <servlet>
        <servlet-name>Event Manager</servlet-name>
        <servlet-class>org.hibernate.tutorial.web.EventManagerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Event Manager</servlet-name>
        <url-pattern>/eventmanager</url-pattern>
    </servlet-mapping>
</web-app>
```

Para construir y desplegar llame a `mvn package` en su directorio de proyecto y copie el archivo `hibernate-tutorial.war` en su directorio `webapp` Tomcat.

**Nota**

If you do not have Tomcat installed, download it from *http://tomcat.apache.org/* and follow the installation instructions. Our application requires no changes to the standard Tomcat configuration.

Una vez que se encuentre desplegado y que Tomcat esté ejecutando, acceda la aplicación en `http://localhost:8080/hibernate-tutorial/eventmanager`. Asegúrese de ver el registro de Tomcat para ver a Hibernate iniciar cuando llegue el primer pedido a su servlet (se llama al inicializador estático en `HibernateUtil`) y para obetener la salida detallada si ocurre alguna excepción.

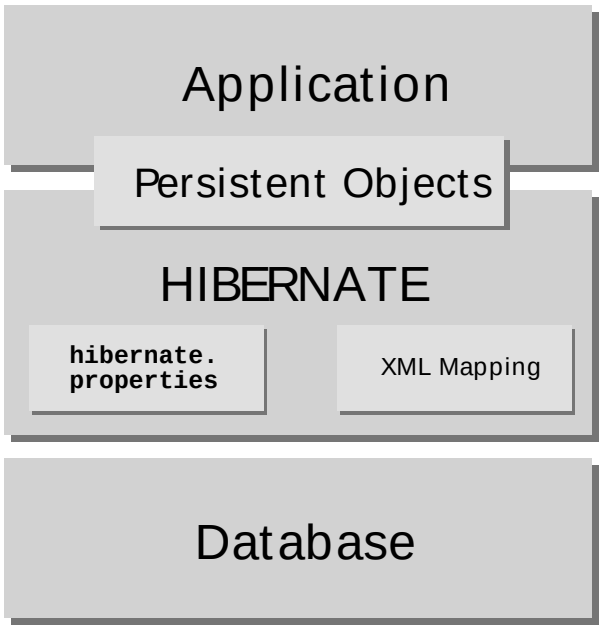
## 1.4. Resumen

Este tutorial abordó los puntos básicos de la escritura de una simple aplicación de Hibernate autónoma y una pequeña aplicación web. Encontrará más tutoriales en el website de Hibernate *http://hibernate.org*.

# Arquitectura

## 2.1. Sinopsis

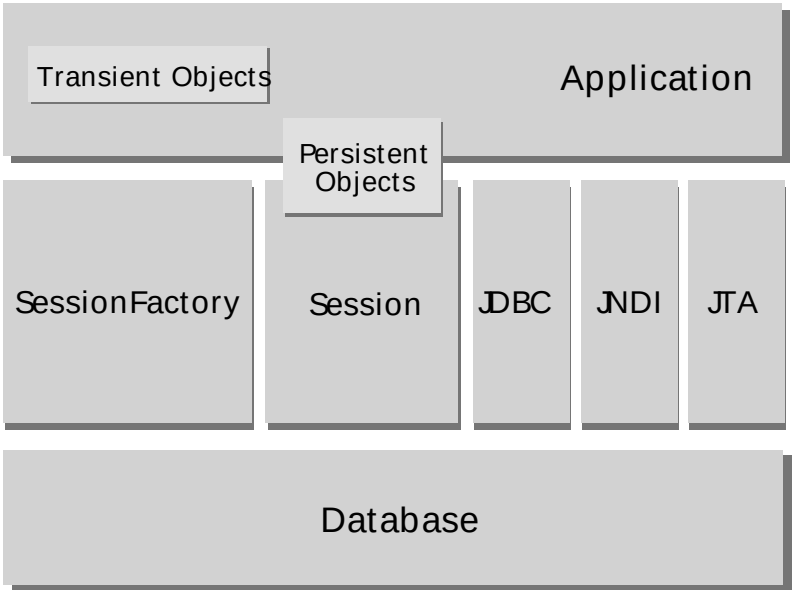
El diagrama a continuación brinda una perspectiva a alto nivel de la arquitectura de Hibernate:




Unfortunately we cannot provide a detailed view of all possible runtime architectures. Hibernate is sufficiently flexible to be used in a number of ways in many, many architectures. We will, however, illustrate 2 specifically since they are extremes.

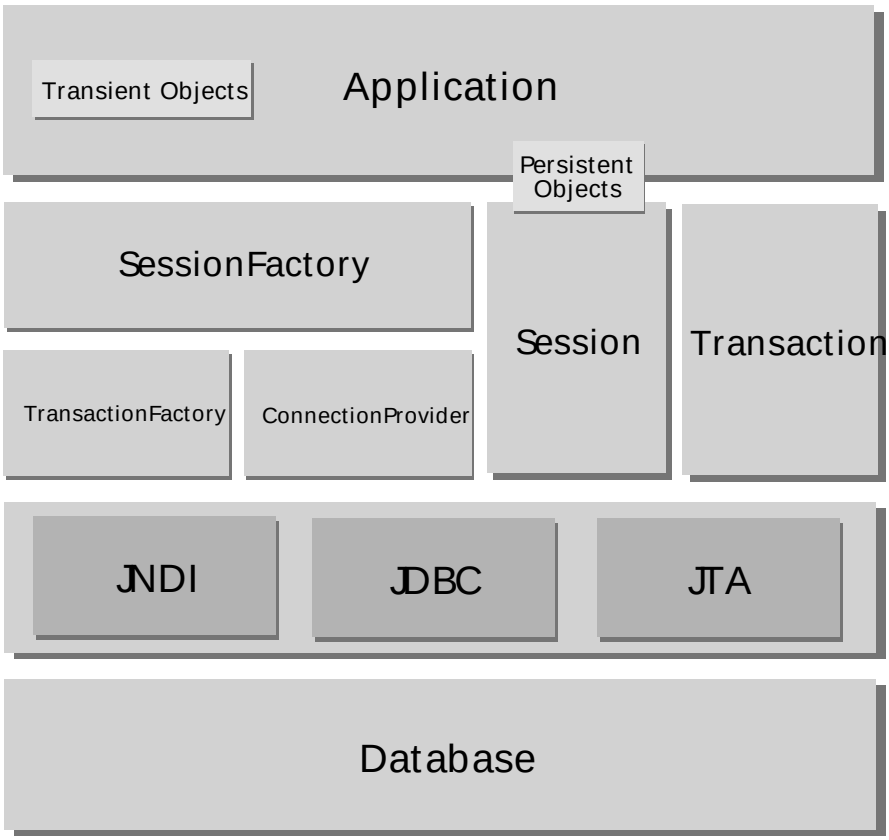
### 2.1.1. Minimal architecture

The "minimal" architecture has the application manage its own JDBC connections and provide those connections to Hibernate; additionally the application manages transactions for itself. This approach uses a minimal subset of Hibernate APIs.




## 2.1.2. Comprehensive architecture

La arquitectura "completa" abstrae la aplicación de las APIs de JDBC/JTA y permite que Hibernate se encargue de los detalles.



### 2.1.3. Basic APIs

Here are quick discussions about some of the API objects depicted in the preceding diagrams (you will see them again in more detail in later chapters).

SessionFactory (`org.hibernate.SessionFactory`)
: A thread-safe, immutable cache of compiled mappings for a single database. A factory for `org.hibernate.Session` instances. A client of `org.hibernate.connection.ConnectionProvider`. Optionally maintains a `second level cache` of data that is reusable between transactions at a process or cluster level.

Session (`org.hibernate.Session`)
: A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC `java.sql.Connection`. Factory for `org.hibernate.Transaction`. Maintains a `first level cache` of persistent the application's persistent objects and collections; this cache is used when navigating the object graph or looking up objects by identifier.

Objetos y colecciones persistentes
: Short-lived, single threaded objects containing persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one `org.hibernate.Session`. Once the `org.hibernate.Session` is closed, they will be detached and free to use in any application layer (for example, directly as data transfer objects to and from presentation). *Capítulo 11, Trabajo con objetos* discusses transient, persistent and detached object states.

Objetos y colecciones transitorios y separados
: Instances of persistent classes that are not currently associated with a `org.hibernate.Session`. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed `org.hibernate.Session`. *Capítulo 11, Trabajo con objetos* discusses transient, persistent and detached object states.

Transaction (`org.hibernate.Transaction`)
: (Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC, JTA or CORBA transaction. A `org.hibernate.Session` might span several `org.hibernate.Transactions` in some cases. However, transaction demarcation, either using the underlying API or `org.hibernate.Transaction`, is never optional.

ConnectionProvider (`org.hibernate.connection.ConnectionProvider`)
: (Optional) A factory for, and pool of, JDBC connections. It abstracts the application from underlying `javax.sql.DataSource` or `java.sql.DriverManager`. It is not exposed to application, but it can be extended and/or implemented by the developer.

TransactionFactory (`org.hibernate.TransactionFactory`)
: (Optional) A factory for `org.hibernate.Transaction` instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.

*Extension Interfaces*

Hibernate ofrece un rango de interfaces de extensión opcionales que puede implementar para personalizar el comportamiento de su capa de persistencia. Para obtener más detalles, vea la documentación de la API.

## 2.2. Integración JMX

JMX es el estándar J2EE para la gestión de componentes Java. Hibernate se puede administrar por medio de un servicio estándar JMX. Brindamos una implementación de MBean en la distribución: `org.hibernate.jmx.HibernateService`.

Another feature available as a JMX service is runtime Hibernate statistics. See *Sección 3.4.6, "Estadísticas de Hibernate"* for more information.

## 2.3. Sesiones contextuales

La mayoría de las aplicaciones que utilizan Hibernate necesitan alguna forma de sesiones "contextuales", en donde una sesión dada se encuentra en efecto en todo el campo de acción de un contexto dado. Sin embargo, a través de las aplicaciones la definición de lo que constituye un contexto es usualmente diferente y diferentes contextos definen diferentes campos de acción para la noción de actual. Las aplicaciones que utiliza Hibernate antes de la version 3.0 tienden a utilizar ya sea sesiones contextuales con base `ThreadLocal` desarrollados en casa, las clases ayudantes tales como `HibernateUtil`, o enfoques de terceros utilizados, como Spring o Pico, los cuales brindaban sesiones contextuales con base proxy/intercepción.

Comenzando con la version 3.0.1, Hibernate agregó el método `SessionFactory.getCurrentSession()`. Inicialmente, este asumió la utilización de las transacciones `JTA`, en donde la transacción `JTA` definia tanto el contexto como el campo de acción de una sesión actual. Dada la madurez de números implementaciones `JTA TransactionManager` autónomas existentes, la mayoría, si no es que todas, las aplicaciones deberían utilizar la administración de transacciones `JTA` en el caso de que se deplieguen o no en un contenedor `J2EE`. Con base en esto, las sesiones contextuales basadas en `JTA` es todo lo que usted necesita utilizar.

Sin embargo, desde la versión 3.1, el procesamiento detrás de `SessionFactory.getCurrentSession()` ahora es conectable. Para ese fin, se ha añadido una nueva interfaz de extensión, `org.hibernate.context.CurrentSessionContext`, y un nuevo parámetro de configuración, `hibernate.current_session_context_class` para permitir la conexión del campo de acción y el contexto de definición de las sesiones actuales.

Refiérase a los Javadocs para la interfaz `org.hibernate.context.CurrentSessionContext` para poder ver una discusión detallada de su contrato. Define un método único, `currentSession()`, por medio del cual la implementación es responsable de rastrear la sesión contextual actual. Tal como viene empacada, Hibernate incluye tres implementaciones de esta interfaz:

- `org.hibernate.context.JTASessionContext`: una transacción `JTA` rastrea y asume las sesiones actuales. Aquí el procesamiento es exactamente el mismo que en el enfoque más antiguo de JTA-sólamente. Refiérase a los Javadocs para obtener más información.
- `org.hibernate.context.ThreadLocalSessionContext`: las sesiones actuales son rastreadas por un hilo de ejecución. Consulte los Javadocs para obtener más detalles.
- `org.hibernate.context.ManagedSessionContext`: las sesiones actuales son rastreadas por un hilo de ejecución. Sin embargo, usted es responsable de vincular y desvincular una instancia `Session` con métodos estáticos en esta clase: no abre, vacia o cierra una `Session`.

The first two implementations provide a "one session - one database transaction" programming model. This is also known and used as *session-per-request*. The beginning and end of a Hibernate session is defined by the duration of a database transaction. If you use programmatic transaction demarcation in plain JSE without JTA, you are advised to use the Hibernate `Transaction` API to hide the underlying transaction system from your code. If you use JTA, you can utilize the JTA interfaces to demarcate transactions. If you execute in an EJB container that supports CMT, transaction boundaries are defined declaratively and you do not need any transaction or session demarcation operations in your code. Refer to *Capítulo 13, Transacciones y concurrencia* for more information and code examples.

El parámetro de configuración `hibernate.current_session_context_class` define cuales implementaciones `org.hibernate.context.CurrentSessionContext` deben utilizarse. Para compatibilidad con versiones anteriores, si este parámetro de configuración no está establecido pero si tiene configurado un `org.hibernate.transaction.TransactionManagerLookup`, Hibernate utilizará el `org.hibernate.context.JTASessionContext`. Usualmente el valor de este parámetro sólamente nombraría la clase de implementación a utilizar. Sin embargo, para las tres implementaciones incluídas existen tress nombres cortos: "jta", "thread" y "managed".

# Configuración

Hibernate está diseñado para operar en muchos entornos diferentes y por lo tanto hay un gran número de parámetros de configuración. Afortunadamente, la mayoría tiene valores predeterminados sensibles y Hibernate se distribuye con un archivo `hibernate.properties` de ejemplo en `etc/` que muestra las diversas opciones. Simplemente ponga el fichero de ejemplo en su ruta de clase y personalícelo de acuerdo a sus necesidades.

## 3.1. Configuración programática

Una instancia de `org.hibernate.cfg.Configuration` representa un conjunto entero de mapeos de los tipos Java de una aplicación a una base de datos SQL. La `org.hibernate.cfg.Configuration` se utiliza para construir una `org.hibernate.SessionFactory` inmutable. Los mapeos se compilan desde varios archivos de mapeo XML.

Puede obtener una instancia de `org.hibernate.cfg.Configuration` instanciándola directamente y especificando los documentos de mapeo XML. Si los archivos de mapeo están en la ruta de clase, utilice `addResource()`. Por ejemplo:

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

Una manera opcional es especificar la clase mapeada y dejar que Hibernate encuentre el documento de mapeo por usted:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

Luego Hibernate buscará los archivos de mapeo llamados `/org/hibernate/auction/Item.hbm.xml` y `/org/hibernate/auction/Bid.hbm.xml` en la ruta de clase. Este enfoque elimina cualquier nombre de archivo establecido manualmente.

Una `org.hibernate.cfg.Configuration` también le permite especificar las propiedades de configuración. Por ejemplo:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
```

```
    .setProperty("hibernate.order_updates", "true");
```

Esta no es la única manera de pasar propiedades de configuración a Hibernate. Algunas opciones incluyen:

1. Pasar una instancia de `java.util.Properties` a `Configuration.setProperties()`.
2. Colocar un archivo llamado `hibernate.properties` en un directorio raíz de la ruta de clase.
3. Establecer propiedades `System` utilizando `java -Dproperty=value`.
4. Incluir los elementos `<property>` en `hibernate.cfg.xml` (esto se discute más adelante).

Si quiere empezar rápidamente `hibernate.properties` es el enfoque más fácil.

La `org.hibernate.cfg.Configuration` está concebida como un objeto de tiempo de inicio que se va a descartar una vez se crea una `SessionFactory`.

## 3.2. Obtención de una SessionFactory

Cuando la `org.hibernate.cfg.Configuration` ha analizado sintácticamente todos los mapeos, la aplicación tiene que obtener una fábrica para las instancias `org.hibernate.Session`. Esta fábrica está concebida para que todos los hilos de la aplicación la compartan:

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Hibernate permite que su aplicación instancie más de una `org.hibernate.SessionFactory`. Esto es útil si está utilizando más de una base de datos.

## 3.3. Conexiones JDBC

Se aconseja que la `org.hibernate.SessionFactory` cree y almacene en pool conexiones JDBC por usted Si adopta este enfoque, el abrir una `org.hibernate.Session` es tan simple como:

```
Session session = sessions.openSession(); // open a new Session
```

En el momento en que inicie una tarea que requiera acceso a la base de datos, se obtendrá una conexión JDBC del pool.

Para que esto funcione, primero necesita pasar algunas las propiedades de conexión JDBC a Hibernate. Todos los nombres de las propiedades de Hibernate y su semántica están definidas en la clase `org.hibernate.cfg.Environment`. Ahora describiremos las configuraciones más importantes para la conexión JDBC.

Hibernate obtendrá y tendrá en pool las conexiones utilizando `java.sql.DriverManager` si configura las siguientes propiedades:

### Tabla 3.1. Propiedades JDBC de Hibernate

| Nombre de la propiedad | Propósito |
|---|---|
| hibernate.connection.driver_class | *JDBC driver class* |
| hibernate.connection.url | *JDBC URL* |
| hibernate.connection.username | *database user* |
| hibernate.connection.password | *database user password* |
| hibernate.connection.pool_size | *maximum number of pooled connections* |

Sin embargo, el algoritmo de pooling de la conexión propia de Hibernate es algo rudimentario. Está concebido para ayudarle a comenzar y *no para utilizarse en un sistema de producción* ni siquiera para pruebas de rendimiento. Para alcanzar un mejor rendimiento y estabilidad debe utilizar un pool de terceros. Sólo remplace la propiedad hibernate.connection.pool_size con configuraciones específicas del pool de conexiones. Esto desactivará el pool interno de Hibernate. Por ejemplo, es posible utilizar C3P0.

C3P0 es un pool de conexiones JDBC de código abierto distribuido junto con Hibernate en el directorio `lib`. Hibernate utilizará su `org.hibernate.connection.C3P0ConnectionProvider` para pooling de conexiones si establece propiedades `hibernate.c3p0.*`. Si quiere utilizar Proxool refiérase a `hibernate.properties` incluído en el paquete y al sitio web de Hibernate para obtener más información.

Aquí hay un archivo `hibernate.properties` de ejemplo para c3p0:

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

Para su utilización dentro de un servidor de aplicaciones, casi siempre usted debe configurar Hibernate para obtener conexiones de un `javax.sql.Datasource` del servidor de aplicaciones registrado en JNDI. Necesitará establecer al menos una de las siguientes propiedades:

### Tabla 3.2. Propiedades de la Fuente de Datos de Hibernate

| Nombre de la propiedad | Propósito |
|---|---|
| hibernate.connection.datasource | *datasource JNDI name* |
| hibernate.jndi.url | *URL del proveedor JNDI* (opcional) |
| hibernate.jndi.class | *clase del JNDI* `InitialContextFactory` (opcional) |

| Nombre de la propiedad | Propósito |
|---|---|
| hibernate.connection.username | *usuario de la base de datos* (opcional) |
| hibernate.connection.password | *contraseña del usuario de la base de datos* (opcional) |

He aquí un archivo `hibernate.properties` de ejemplo para una fuente de datos JNDI provisto por un servidor de aplicaciones:

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

Las conexiones JDBC obtenidas de una fuente de datos JNDI participarán automáticamente en las transacciones del servidor de aplicaciones administradas por el contenedor.

Pueden darse propiedades de conexión arbitrarias anteponiendo "`hibernate.connnection`" al nombre de propiedad de la conexión. Por ejemplo, puede especificar una propiedad de conexión `charSet` usando hibernate.connection.charSet.

Puede definir su propia estrategia plugin para obtener conexiones JDBC implementando la interfaz `org.hibernate.connection.ConnectionProvider` y especificando su propia implementación personalizada por medio de la propiedad hibernate.connection.provider_class.

## 3.4. Parámetros de configuración opcionales

Hay otras propiedades que controlan el comportamiento de Hibernate en tiempo de ejecución. Todas son opcionales y tienen valores razonables por defecto.

### Aviso

*Algunas de estas propiedades se encuentran a "nivel del sistema sólamente".* Las propiedades a nivel del sistema sólamente se pueden establecer por medio de `java -Dproperty=value` o `hibernate.properties`. *No* se pueden establecer por medio de las técnicas descritas anteriormente.

### Tabla 3.3. Propiedades de Configuración de Hibernate

| Nombre de la propiedad | Propósito |
|---|---|
| hibernate.dialect | El nombre de clase de un `org.hibernate.dialect.Dialect` de Hibernate, el cual le permite que genere un |

| Nombre de la propiedad | Propósito |
|---|---|
| | SQL optimizado para una base de datos relacional en particular.<br><br>**e.g.** `full.classname.of.Dialect`<br><br>En la mayoría de los casos Hibernate podrá de hecho seleccionar la implementación `org.hibernate.dialect.Dialect` correcta con base en los `JDBC metadata` que el controlador JDBC retorna. |
| hibernate.show_sql | Escribe todas las declaraciones SQL a la consola. Esta es una alternativa para establecer la categoria de registro `org.hibernate.SQL` a `debug`.<br><br>**e.g.** `true` \| `false` |
| hibernate.format_sql | Imprime el SQL en el registro y la consola.<br><br>**e.g.** `true` \| `false` |
| hibernate.default_schema | Califica los nombres de tabla sin calificar con el esquema/espacio de tabla dado en el SQL generado.<br><br>**e.g.** `SCHEMA_NAME` |
| hibernate.default_catalog | Califica los nombres de tabla sin calificar con el catálogo dado en el SQL generado.<br><br>**e.g.** `CATALOG_NAME` |
| hibernate.session_factory_name | Automáticamente se vinculará el `org.hibernate.SessionFactory` a este nombre en JNDI después de que se ha creado.<br><br>**e.g.** `jndi/composite/name` |
| hibernate.max_fetch_depth | Establece una "profundidad" máxima del árbol de recuperación por unión externa (outer join) para asociaciones de un sólo extremo (uno-a-uno, muchos-a-uno). Un `0` deshabilita la recuperación por unión externa predeterminada.<br><br>**ej.** los valores recomendados entre `0` y `3` |
| hibernate.default_batch_fetch_size | Establece un tamaño por defecto para la recuperación en lote de asociaciones de Hibernate. |

| Nombre de la propiedad | Propósito |
| --- | --- |
| | **ej.** valores recomendados `4`, `8`, `16` |
| hibernate.default_entity_mode | Establece un modo predeterminado de representación de entidades para todas las sesiones abiertas desde esta `SessionFactory`<br><br>`dynamic-map`, `dom4j`, `pojo` |
| hibernate.order_updates | Obliga a Hibernate a ordenar las actualizaciones SQL por el valor de la clave principal de los items a actualizar. Esto resultará en menos bloqueos de transacción en sistemas altamente concurrentes.<br><br>**e.g.** `true` \| `false` |
| hibernate.generate_statistics | De habilitarse, Hibernate colectará estadísticas útiles para la afinación de rendimiento.<br><br>**e.g.** `true` \| `false` |
| hibernate.use_identifier_rollback | De habilitarse, cuando se borren los objetos las propiedades identificadoras generadas se resetearán a losvalores establecidos por defecto.<br><br>**e.g.** `true` \| `false` |
| hibernate.use_sql_comments | De activarse, Hibernate generará comentarios dentro del SQL, para una depuración más fácil, por defecto es `false`.<br><br>**e.g.** `true` \| `false` |
| hibernate.id.new_generator_mappings | Setting is relevant when using `@GeneratedValue`. It indicates whether or not the new `IdentifierGenerator` implementations are used for `javax.persistence.GenerationType.AUTO`, `javax.persistence.GenerationType.TABLE` and `javax.persistence.GenerationType.SEQUENCE`. Default to `false` to keep backward compatibility.<br><br>**e.g.** `true` \| `false` |

**Nota**

We recommend all new projects which make use of to use `@GeneratedValue` to also set `hibernate.id.new_generator_mappings=true` as the new generators are more efficient and closer to the JPA 2 specification semantic. However they are not backward compatible with existing databases (if a sequence or a table is used for id generation).

### Tabla 3.4. Propiedades de JDBC y Conexiones de Hibernate

| Nombre de la propiedad | Propósito |
|---|---|
| hibernate.jdbc.fetch_size | Un valor distinto de cero que determina el tamaño de recuperación de JDBC (llama a `Statement.setFetchSize()`). |
| hibernate.jdbc.batch_size | Un valor distinto de cero habilita que Hibernate utilice las actualizaciones en lote de JDBC2.<br><br>**ej.** valores recomendados entre `5` y `30` |
| hibernate.jdbc.batch_versioned_data | Set this property to `true` if your JDBC driver returns correct row counts from `executeBatch()`. It is usually safe to turn this option on. Hibernate will then use batched DML for automatically versioned data. Defaults to `false`.<br><br>**e.g.** `true` \| `false` |
| hibernate.jdbc.factory_class | Selecciona un `org.hibernate.jdbc.Batcher` personalizado. La mayoría de las aplicaciones no necesitarán esta propiedad de configuración.<br><br>**eg.** `classname.of.BatcherFactory` |
| hibernate.jdbc.use_scrollable_resultset | Habilita a Hibernate para utilizar los grupos de resultados deslizables de JDBC2. Esta propiedad sólamente es necesaria cuando se utilizan conexiones JDBC provistas por el usuario. En el caso contrario Hibernate utiliza los metadatos de conexión.<br><br>**e.g.** `true` \| `false` |
| hibernate.jdbc.use_streams_for_binary | Utiliza flujos (streams) al escribir/leer tipos `binary` o `serializable` a/desde JDBC. *Propiedad a nivel de sistema* |

| Nombre de la propiedad | Propósito |
| --- | --- |
| | **e.g.** `true` \| `false` |
| hibernate.jdbc.use_get_generated_keys | Habilita el uso de `PreparedStatement.getGeneratedKeys()` de JDBC3 para recuperar claves generadas nativamente después de insertar. Requiere un controlador JDBC3+ y un JRE1.4+. Establézcalo como falso si su controlador tiene problemas con los generadores del identificador de Hibernate. Por defecto, se intenta determinar las capacidades del controlador utilizando los metadatos de conexión.<br><br>**e.g.** `true\|false` |
| hibernate.connection.provider_class | EL nombre de clase de un `org.hibernate.connection.ConnectionProvider` personalizado que proporcione conexiones JDBC a Hibernate.<br><br>**e.g.** `classname.of.ConnectionProvider` |
| hibernate.connection.isolation | Establece el nivel de aislamiento de la transacción JDBC. Comprueba `java.sql.Connection` para valores significativos pero observe que la mayoría de las bases de datos no soportan todos los niveles de aislamiento y algunos definen nivekes de aislamiento adicionales y no estándares.<br><br>**e.g.** `1, 2, 4, 8` |
| hibernate.connection.autocommit | Habilita un guardado automático (autocommit) para las conexiones JDBC en pool (no se recomienda).<br><br>**e.g.** `true` \| `false` |
| hibernate.connection.release_mode | Especifica el momento en que Hibernate debe liberar las conexiones JDBC. Por defecto, una conexión JDBC es retenida hasta que la sesión se cierra o se desconecta explícitamente. Para una fuente de datos JTA del servidor de aplicaciones, debe utilizar `after_statement` para liberar agresivamente las conexiones después de cada llamada JDBC. Para una |

| Nombre de la propiedad | Propósito |
|---|---|
| | conexión no JTA, frecuentemente tiene sentido el liberar la conexión al final de cada transacción, el utilizar`after_transaction`. `auto` escogerá `after_statement` para las estrategias de transacción JTA y CMT y `after_transaction` para la estrategia JDBC de transacción.<br><br>**e.g.** `auto` (default) \| `on_close` \| `after_transaction` \| `after_statement`<br><br>This setting only affects `Session`s returned from `SessionFactory.openSession`. For `Session`s obtained through `SessionFactory.getCurrentSession`, the `CurrentSessionContext` implementation configured for use controls the connection release mode for those `Session`s. See *Sección 2.3, "Sesiones contextuales"* |
| hibernate.connection.<*propertyName*> | Pasar la propiedad JDBC <*propertyName*> a `DriverManager.getConnection()`. |
| hibernate.jndi.<*propertyName*> | Pasar la propiedad <*propertyName*> al `InitialContextFactory` JNDI. |

### Tabla 3.5. Propiedades de Caché de Hibernate

| Nombre de la propiedad | Propósito |
|---|---|
| `hibernate.cache.provider_class` | El nombre de clase de un `CacheProvider` personalizado.<br><br>**e.g.** `classname.of.CacheProvider` |
| `hibernate.cache.use_minimal_puts` | Optimiza la operación del caché de segundo nivel para minimizar escrituras, con el costo de lecturas más frecuentes. Esta configuración es más útil para cachés en clúster y en Hibernate3, está habilitado por defecto para implementaciones de caché en clúster.<br><br>**e.g.** `true\|false` |
| `hibernate.cache.use_query_cache` | Habilita el caché de consultas. Las consultas individuales todavía tienen que establecerse con cachés.<br><br>**e.g.** `true\|false` |

| Nombre de la propiedad | Propósito |
| --- | --- |
| `hibernate.cache.use_second_level_cache` | Se puede utilizar para deshabilitar por completo el caché de segundo nivel, que está habilitado por defecto para clases que especifican un mapeo `<cache>`.<br><br>**e.g.** `true\|false` |
| `hibernate.cache.query_cache_factory` | El nombre de clase de una interfaz `QueryCache` personalizada, por defecto al `StandardQueryCache` incorporado.<br><br>**e.g.** `classname.of.QueryCache` |
| `hibernate.cache.region_prefix` | Un prefijo que se debe utilizar para los nombres de región del caché de segundo nivel.<br><br>**e.g.** `prefix` |
| `hibernate.cache.use_structured_entries` | Obliga a Hibernate a almacenar los datos en el caché de segundo nivel en un formato más amigable para personas.<br><br>**e.g.** `true\|false` |
| `hibernate.cache.default_cache_concurrency_strategy` | Setting used to give the name of the default `org.hibernate.annotations.CacheConcurrencyStrategy` to use when either `@Cacheable` or `@Cache` is used. `@Cache(strategy="..")` is used to override this default. |

## Tabla 3.6. Propiedades de Transacción de Hibernate

| Nombre de la propiedad | Propósito |
| --- | --- |
| `hibernate.transaction.factory_class` | El nombre de clase de un `TransactionFactory` a utilizar con la API de `Transaction` de Hibernate (por defecto es `JDBCTransactionFactory`).<br><br>**e.g.** `classname.of.TransactionFactory` |
| `jta.UserTransaction` | Un nombre JNDI utilizado por `JTATransactionFactory` para obtener la `UserTransaction` de JTA del servidor de aplicaciones.<br><br>**e.g.** `jndi/composite/name` |
| `hibernate.transaction.manager_lookup_class` | El nombre de clase de un `TransactionManagerLookup`. Se requiere |

| Nombre de la propiedad | Propósito |
| --- | --- |
| | cuando el chaché a nivel de MVJ está habilitado o cuando se utiliza un generador alto/bajo en un entorno JTA.<br><br>**e.g.** `classname.of.TransactionManagerLookup` |
| `hibernate.transaction.flush_before_completion` | If enabled, the session will be automatically flushed during the before completion phase of the transaction. Built-in and automatic session context management is preferred, see *Sección 2.3, "Sesiones contextuales"*.<br><br>**e.g.** `true` \| `false` |
| `hibernate.transaction.auto_close_session` | If enabled, the session will be automatically closed during the after completion phase of the transaction. Built-in and automatic session context management is preferred, see *Sección 2.3, "Sesiones contextuales"*.<br><br>**e.g.** `true` \| `false` |

### Tabla 3.7. Propiedades Misceláneas

| Nombre de la propiedad | Propósito |
| --- | --- |
| `hibernate.current_session_context_class` | Supply a custom strategy for the scoping of the "current" `Session`. See *Sección 2.3, "Sesiones contextuales"* for more information about the built-in strategies.<br><br>**e.g.** `jta` \| `thread` \| `managed` \| `custom.Class` |
| `hibernate.query.factory_class` | Elige la implementación de análisis sintáctico HQL.<br><br>**ej.** `org.hibernate.hql.ast.ASTQueryTranslatorFactory` o `org.hibernate.hql.classic.ClassicQueryTranslatorFactory` |
| `hibernate.query.substitutions` | Se utiliza para mapear desde tokens en consultas Hibernate a tokens SQL. (por ejemplo, los tokens pueden ser nombres de función o literales).<br><br>**e.g.** `hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC` |

| Nombre de la propiedad | Propósito |
| --- | --- |
| `hibernate.hbm2ddl.auto` | Exporta o valida automáticamente DDL de esquema a la base de datos cuando se crea la `SessionFactory`. Con `create-drop` se desechará el esquema de la base de datos cuando la `SessionFactory` se cierre explícitamente.<br><br>**e.g.** `validate` \| `update` \| `create` \| `create-drop` |
| `hibernate.hbm2ddl.import_file` | Comma-separated names of the optional files containing SQL DML statements executed during the `SessionFactory` creation. This is useful for testing or demoing: by adding INSERT statements for example you can populate your database with a minimal set of data when it is deployed.<br><br>File order matters, the statements of a give file are executed before the statements of the following files. These statements are only executed if the schema is created ie if `hibernate.hbm2ddl.auto` is set to `create` or `create-drop`.<br><br>**e.g.** `/humans.sql,/dogs.sql` |
| `hibernate.bytecode.use_reflection_optimizer` | Enables the use of bytecode manipulation instead of runtime reflection. This is a System-level property and cannot be set in `hibernate.cfg.xml`. Reflection can sometimes be useful when troubleshooting. Hibernate always requires either CGLIB or javassist even if you turn off the optimizer.<br><br>**e.g.** `true` \| `false` |
| `hibernate.bytecode.provider` | Both javassist or cglib can be used as byte manipulation engines; the default is `javassist`.<br><br>**e.g.** `javassist` \| `cglib` |

### 3.4.1. Dialectos de SQL

Siempre configure la propiedad `hibernate.dialect` a la subclase correcta `org.hibernate.dialect.Dialect` para su base de datos. Si especifica un dialecto, Hibernate

utilizará valores predeterminados de manera sensible para algunas de las otras propiedades enumeradas anteriormente, ahorrándole el esfuerzo de especificarlas manualmente.

**Tabla 3.8. Dialectos SQL de Hibernate(`hibernate.dialect`)**

| RDBMS | Dialecto |
|---|---|
| DB2 | `org.hibernate.dialect.DB2Dialect` |
| DB2 AS/400 | `org.hibernate.dialect.DB2400Dialect` |
| DB2 OS390 | `org.hibernate.dialect.DB2390Dialect` |
| PostgreSQL | `org.hibernate.dialect.PostgreSQLDialect` |
| MySQL | `org.hibernate.dialect.MySQLDialect` |
| MySQL con InnoDB | `org.hibernate.dialect.MySQLInnoDBDialect` |
| MySQL con MyISAM | `org.hibernate.dialect.MySQLMyISAMDialect` |
| Oracle (cualquier versión) | `org.hibernate.dialect.OracleDialect` |
| Oracle 9i | `org.hibernate.dialect.Oracle9iDialect` |
| Oracle 10g | `org.hibernate.dialect.Oracle10gDialect` |
| Sybase | `org.hibernate.dialect.SybaseDialect` |
| Sybase Anywhere | `org.hibernate.dialect.SybaseAnywhereDialect` |
| Microsoft SQL Server | `org.hibernate.dialect.SQLServerDialect` |
| SAP DB | `org.hibernate.dialect.SAPDBDialect` |
| Informix | `org.hibernate.dialect.InformixDialect` |
| HypersonicSQL | `org.hibernate.dialect.HSQLDialect` |
| Ingres | `org.hibernate.dialect.IngresDialect` |
| Progress | `org.hibernate.dialect.ProgressDialect` |
| Mckoi SQL | `org.hibernate.dialect.MckoiDialect` |
| Interbase | `org.hibernate.dialect.InterbaseDialect` |
| Pointbase | `org.hibernate.dialect.PointbaseDialect` |
| FrontBase | `org.hibernate.dialect.FrontbaseDialect` |
| Firebird | `org.hibernate.dialect.FirebirdDialect` |

## 3.4.2. Recuperación por Unión Externa - Outer Join Fetching

Si su base de datos soporta uniones externas del estilo ANSI, Oracle o Sybase, frecuentemente la *recuperación por unión externa* aumentará el rendimiento limitando el número de llamadas a la base de datos. La recuperación por unión externa permite que un gráfico completo de objetos conectados por asociaciones muchos-a-uno, uno-a-muchos, muchos-a-muchos y uno-a-uno sea recuperado en un sólo `SELECT` SQL.

La recuperación por unión externa puede ser deshabilitada *globalmente* estableciendo la propiedad `hibernate.max_fetch_depth` como `0`. Un valor de `1` o mayor habilita la recuperación por unión externa para asociaciones uno-a-uno y muchos-a-uno que hayan sido mapeadas con `fetch="join"`.

See *Sección 21.1, "Estrategias de recuperación"* for more information.

### 3.4.3. Flujos Binarios

Oracle limita el tamaño de arrays de `byte` que se puedan pasar a/desde su controlador JDBC. Si desea utilizar instancias grandes de tipo `binary` o `serializable`, usted debe habilitar `hibernate.jdbc.use_streams_for_binary`. *Esta es una configuración a nivel de sistema sólamente.*

### 3.4.4. Caché de segundo nivel y de lectura

The properties prefixed by `hibernate.cache` allow you to use a process or cluster scoped second-level cache system with Hibernate. See the *Sección 21.2, "El Caché de Segundo Nivel"* for more information.

### 3.4.5. Sustitución de Lenguaje de Consulta

Puede definir nuevos tokens de consulta de Hibernate utilizando `hibernate.query.substitutions`. Por ejemplo:

```
hibernate.query.substitutions true=1, false=0
```

Esto causaría que los tokens `true` y `false` sean traducidos a literales enteros en el SQL generado.

```
hibernate.query.substitutions toLowercase=LOWER
```

Esto le permitiría renombrar la función `LOWER` de SQL.

### 3.4.6. Estadísticas de Hibernate

Si habilita `hibernate.generate_statistics`, Hibernate expondrá un número de métricas que son útiles al afinar un sistema en ejecución por medio de `SessionFactory.getStatistics()`. Incluso se puede configurar Hibernate para exponer estas estadísticas por medio de JMX. Lea el Javadoc de las interfaces en `org.hibernate.stats` para obtener más información.

## 3.5. Registros de mensajes (Logging)

Hibernate utiliza *Simple Logging Facade for Java* [http://www.slf4j.org/] (SLF4J) con el fin de registrar varios eventos del sistema. SLF4J puede direccionar su salida de registro a varios

marcos de trabajo de registro (NOP, Simple, log4j versión 1.2, JDK 1.4 logging, JCL o logback) dependiendo de su enlace escogido. Con el fin de configurar el registro necesitará `slf4j-api.jar` en su ruta de clase junto con el archivo jar para su enlace preferido - `slf4j-log4j12.jar` en el caso de Log4J. Consulte la *documentación* [http://www.slf4j.org/manual.html] SLF4J para obtener mayores detalles. Para usar Log4j también necesitará poner un archivo `log4j.properties` en su ruta de clase. Un archivo de propiedades de ejemplo se distribuye junto con Hibernate en el directorio `src/`.

Le recomendamos bastante que se familiarice con los mensajes de registro de Hibernate. Se ha trabajado bastante para hacer que los registros de Hibernate sean tan detallados como sea posible, sin hacerlos ilegibles. Es un dispositivo esencial en la resolución de problemas. Las categorías de registro más interesantes son las siguientes:

### Tabla 3.9. Categorías de Registro de Hibernate

| Categoría | Función |
|---|---|
| `org.hibernate.SQL` | Registra todas las declaraciones DML de SQL a medida que se ejecutan |
| `org.hibernate.type` | Registra todos los parámetros JDBC |
| `org.hibernate.tool.hbm2ddl` | Registra todas las declaraciones DDL de SQL a medida que se ejecutan |
| `org.hibernate.pretty` | Registra el estado de todas las entidades (máximo 20 entidades) asociadas con la sesión en tiempo de limpieza (flush) |
| `org.hibernate.cache` | Registra toda la actividad del caché de segundo nivel |
| `org.hibernate.transaction` | Registra la actividad relacionada con la transacción |
| `org.hibernate.jdbc` | Registra toda adquisición de recursos JDBC |
| `org.hibernate.hql.ast.AST` | Registra los ASTs de HQL y SQL, durante análisis de consultas. |
| `org.hibernate.secure` | Registra todas las peticiones de autorización JAAS |
| `org.hibernate` | Registra todo. Hay mucha información, pero es útil para la resolución de problemas |

Al desarrollar aplicaciones con Hibernate, casi siempre debe trabajar con `debug` habilitado para la categoría `org.hibernate.SQL` o, alternativamente, la propiedad `hibernate.show_sql` habilitada.

## 3.6. Implementación de una `NamingStrategy`

La interfaz `org.hibernate.cfg.NamingStrategy` le permite especificar un "estándar de nombrado" para objetos de la base de datos y los elementos del esquema.

Puede proveer reglas para generar automáticamente identificadores de la base de datos a partir de identificadores JDBC o para procesar nombres "lógicos" de columnas y tablas dadas en el archivo de mapeo en nombres "físicos" de columnas y tablas. Esta funcionalidad ayuda a reducir

la verborragia del documento de mapeo, eliminando ruidos repetitivos (por ejemplo, prefijos `TBL_`). Hibernate utiliza una estrategia por defecto bastante mínima.

Puede especificar una estrategia diferente llamando a `Configuration.setNamingStrategy()` antes de agregar los mapeos:

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

`org.hibernate.cfg.ImprovedNamingStrategy` es una estrategia incorporada que puede ser un punto de partida útil para algunas aplicaciones.

## 3.7. Archivo de configuración XML

Un enfoque alternativo de configuración es especificar una configuración completa en un archivo llamado `hibernate.cfg.xml`. Este archivo se puede utilizar como un remplazo del archivo `hibernate.properties` o en el caso de que ambos se encuentren presentes, para sobrescribir propiedades.

El archivo de configuración XML por defecto se espera en la raíz de su `CLASSPATH`. Este es un ejemplo:

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

<hibernate-configuration>

    <!-- a SessionFactory instance listed as /jndi/name -->
    <session-factory
        name="java:hibernate/SessionFactory">

        <!-- properties -->
        <property name="connection.datasource">java:/comp/env/jdbc/MyDB</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="transaction.factory_class">
            org.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction">java:comp/UserTransaction</property>

        <!-- mapping files -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

        <!-- cache settings -->
        <class-cache class="org.hibernate.auction.Item" usage="read-write"/>
```

```
        <class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
        <collection-cache collection="org.hibernate.auction.Item.bids" usage="read-write"/>

    </session-factory>

</hibernate-configuration>
```

La ventaja de este enfoque es la externalización de los nombres de los archivos de mapeo a la configuración. El `hibernate.cfg.xml` también es más práctico una vez que haya afinado el caché de Hibernate. Puede escoger ya sea `hibernate.properties` o `hibernate.cfg.xml`. Ambos son equivalentes, excepto por los beneficios de utilizar la sintaxis XML que mencionados anteriormente.

Con la configuración XML, iniciar Hibernate es tan simple como:

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

Puede seleccionar un fichero de configuración XML diferente utilizando:

```
SessionFactory sf = new Configuration()
    .configure("catdb.cfg.xml")
    .buildSessionFactory();
```

## 3.8. Integración con Servidores de Aplicaciones J2EE

Hibernate tiene los siguientes puntos de integración con la infraestructura J2EE:

- *Fuentes de datos administrados por el contenedor*: Hibernate puede utilizar conexiones JDBC administradas por el contenedor y provistas a través de JNDI. Usualmente, un `TransactionManager` compatible con JTA y un `ResourceManager` se ocupan de la administración de transacciones (CMT), especialmente del manejo de transacciones distribuídas a través de varias fuentes de datos. También puede demarcar los límites de las transacciones programáticamente (BMT) o puede que quiera utilizar para esto la API opcional de `Transaction` de Hibernate para mantener portátil su código.

- *Vinculación Automática JNDI*: Hibernate puede vincular sus `SessionFactory` a JNDI después del inicio.

- *Vinculación de Sesión JTA:* La `Session` de Hibernate se puede vincular automáticamente al ámbito de transacciones JTA. Simplemente busque la `SessionFactory` de JNDI y obténga la `Session` actual. Deje que Hibernate se ocupe de vaciar y cerrar la `Session` cuando se

complete su transacción JTA. La demarcación de transacción puede ser declarativa (CMT) o programática (BMT/UserTransaction).

- *Despliegue JMX:* Si tiene un servidor de aplicaciones con capacidad para JMX (por ejemplo, JBoss AS), puede escoger el desplegar Hibernate como un MBean administrado. Esto le ahorra el código de una línea de inicio para construir su `SessionFactory` desde una `Configuration`. El contenedor iniciará su `HibernateService`, e idealmente también cuidará de las dependencias entre servicios (la fuente de datos debe estar disponible antes de que Hibernate inicie, etc).

Dependiendo de su entorno, podría tener que establecer la opción de configuración `hibernate.connection.aggressive_release` como true si su servidor de aplicaciones muestra excepciones "contención de conexión".

### 3.8.1. Configuración de la estrategia de transacción

La API de `Session` de Hibernate es independiente de cualquier demarcación de transacción en su arquitectura. Si deja que Hibernate utilice JDBC directamente, a través de un pool de conexiones, puede comenzar y acabar sus transacciones llamando la API de JDBC. Si ejecuta en un servidor de aplicaciones J2EE, puede que quiera utilizar transacciones administradas por bean y llamar la API de JTA y `UserTransaction` cuando sea necesario.

Para mantener su código portable entre estos dos (y otros) entornos le recomendamos la API de `Transaction` de Hibernate, que envuelve y oculta el sistema subyacente. Tiene que especificar una clase fábrica para las instancias de `Transaction` estableciendo la propiedad de configuración `hibernate.transaction.factory_class` de Hibernate.

Existen tres opciones estándares o incorporadas:

`org.hibernate.transaction.JDBCTransactionFactory`
: delega a transacciones de bases de datos (JDBC) (por defecto)

`org.hibernate.transaction.JTATransactionFactory`
: delega a transacciones administradas por el contenedor si una transacción existente se encuentra en proceso en este contexto (por ejemplo, un método de bean de sesión EJB). De otra manera, se inicia una nueva transacción y se utilizan las transacciones administradas por bean.

`org.hibernate.transaction.CMTTransactionFactory`
: delega a transacciones JTA administradas por el contenedor

También puede definir sus propias estrategias de transacción (por ejemplo, para un servicio de transacción CORBA).

Algunas funcionalidades en Hibernate (por ejemplo, el caché de segundo nivel, las sesiones contextuales, etc.) requieren acceso al `TransactionManager` de JTA en un entorno administrado. En un servidor de aplicaciones tiene que especificar cómo Hibernate debe obtener una referencia al `TransactionManager`, ya que J2EE no estandariza un sólo mecanismo:

**Tabla 3.10. TransactionManagers de JTA**

| Transaction Factory | Servidor de Aplicaciones |
|---|---|
| `org.hibernate.transaction.JBossTransactionManagerLookup` | JBoss AS |
| `org.hibernate.transaction.WeblogicTransactionManagerLookup` | Weblogic |
| `org.hibernate.transaction.WebSphereTransactionManagerLookup` | WebSphere |
| `org.hibernate.transaction.WebSphereExtendedJTATransactionLookup` | WebSphere 6 |
| `org.hibernate.transaction.OrionTransactionManagerLookup` | Orion |
| `org.hibernate.transaction.ResinTransactionManagerLookup` | Resin |
| `org.hibernate.transaction.JOTMTransactionManagerLookup` | JOTM |
| `org.hibernate.transaction.JOnASTransactionManagerLookup` | JOnAS |
| `org.hibernate.transaction.JRun4TransactionManagerLookup` | JRun4 |
| `org.hibernate.transaction.BESTransactionManagerLookup` | Borland ES |
| `org.hibernate.transaction.JBossTSStandaloneTransactionManagerLookup` | JBoss TS used standalone (ie. outside JBoss AS and a JNDI environment generally). Known to work for `org.jboss.jbossts:jbossjta:4.11.0.Final` |

## 3.8.2. `SessionFactory` enlazado a JNDI

Una `SessionFactory` de Hibernate vinculada a JNDI puede simplificar la búsqueda de la fábrica y la creación de nuevas `Session`es. Sin embargo, esto no se relaciona con un `Datasource` vinculado a JNDI; simplemente que ambos utilizan el mismo registro.

Si desea tener la `SessionFactory` vinculada a un espacio de nombres de JNDI, especifique un nombre (por ejemplo, `java:hibernate/SessionFactory`) utilizando la propiedad `hibernate.session_factory_name`. Si se omite esta propiedad, la `SessionFactory` no será vinculada a JNDI. Esto es particularmente útil en entornos con una implementación JNDI de sólo lectura por defecto (por ejemplo, en Tomcat).

Al vincular la `SessionFactory` a JNDI, Hibernate utilizará los valores de `hibernate.jndi.url`, `hibernate.jndi.class` para instanciar un contexto inicial. Si éstos no se especifican, se utilizará el `InitialContext` por defecto.

Hibernate colocará automáticamente la `SessionFactory` en JNDI después de que llame a `cfg.buildSessionFactory()`. Esto significa que tendrá al menos esta llamada en algún código de inicio o clase de utilidad en su aplicación, a menos de que utilice el despliegue JMX con el `HibernateService` (esto se discute más adelante en mayor detalle).

Si utiliza una `SessionFactory` JNDI, un EJB or cualquier otra clase puede llegar a obtener el `SessionFactory` utilizando una búsqueda JNDI.

It is recommended that you bind the `SessionFactory` to JNDI in a managed environment and use a `static` singleton otherwise. To shield your application code from these details, we also recommend to hide the actual lookup code for a `SessionFactory` in a helper class, such as `HibernateUtil.getSessionFactory()`. Note that such a class is also a convenient way to startup Hibernate—see chapter 1.

### 3.8.3. Administración de contexto de Sesión Actual con JTA

The easiest way to handle `Sessions` and transactions is Hibernate's automatic "current" `Session` management. For a discussion of contextual sessions see *Sección 2.3, "Sesiones contextuales"*. Using the `"jta"` session context, if there is no Hibernate `Session` associated with the current JTA transaction, one will be started and associated with that JTA transaction the first time you call `sessionFactory.getCurrentSession()`. The `Sessions` retrieved via `getCurrentSession()` in the `"jta"` context are set to automatically flush before the transaction completes, close after the transaction completes, and aggressively release JDBC connections after each statement. This allows the `Sessions` to be managed by the life cycle of the JTA transaction to which it is associated, keeping user code clean of such management concerns. Your code can either use JTA programmatically through `UserTransaction`, or (recommended for portable code) use the Hibernate `Transaction` API to set transaction boundaries. If you run in an EJB container, declarative transaction demarcation with CMT is preferred.

### 3.8.4. Despliegue JMX

La línea `cfg.buildSessionFactory()` todavía se tiene que ejecutar en algún sitio para obtener una `SessionFactory` en JNDI. Puede hacer esto ya sea en un bloque inicializador `static` (como aquel en `HibernateUtil`) o bien puede desplegar Hibernate como un *servicio administrado*.

Hibernate se distribuye con `org.hibernate.jmx.HibernateService` para desplegar en un servidor de aplicaciones con capacidades JMX, como JBoss AS. El despliegue y la configuracón reales son específicos del vendedor. He aquí un ejemplo de `jboss-service.xml` para JBoss 4.0.x:

```
<?xml version="1.0"?>
<server>

<mbean code="org.hibernate.jmx.HibernateService"
    name="jboss.jca:service=HibernateFactory,name=HibernateFactory">

    <!-- Required services -->
    <depends>jboss.jca:service=RARDeployer</depends>
    <depends>jboss.jca:service=LocalTxCM,name=HsqlDS</depends>

    <!-- Bind the Hibernate service to JNDI -->
    <attribute name="JndiName">java:/hibernate/SessionFactory</attribute>

    <!-- Datasource settings -->
    <attribute name="Datasource">java:HsqlDS</attribute>
    <attribute name="Dialect">org.hibernate.dialect.HSQLDialect</attribute>
```

```
    <!-- Transaction integration -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribute>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup</attribute>
    <attribute name="FlushBeforeCompletionEnabled">true</attribute>
    <attribute name="AutoCloseSessionEnabled">true</attribute>

    <!-- Fetching options -->
    <attribute name="MaximumFetchDepth">5</attribute>

    <!-- Second-level caching -->
    <attribute name="SecondLevelCacheEnabled">true</attribute>
    <attribute name="CacheProviderClass">org.hibernate.cache.EhCacheProvider</attribute>
    <attribute name="QueryCacheEnabled">true</attribute>

    <!-- Logging -->
    <attribute name="ShowSqlEnabled">true</attribute>

    <!-- Mapping files -->
    <attribute name="MapResources">auction/Item.hbm.xml,auction/Category.hbm.xml</attribute>

</mbean>

</server>
```

Este archivo se implementa en un directorio llamado `META-INF` y se encuentra empacado en un archivo JAR con la extensión `.sar` (archivo de servicio). También necesita empacar Hibernate, sus bibliotecas de terceros requeridas, sus clases persistentes compiladas, así como sus archivos de mapeo en el mismo archivo. Sus beans empresariales (usualmente beans de sesión) se pueden dejar en su propio archivo JAR, pero puede incluir este archivo EJB JAR en el archivo de servicio principal para obtener una unidad desplegable en vivo (sin apagarlo). Consulte la documentación de JBoss AS para obtener más información sobre el servicio JMX y la implementación de EJB.

# Clases persistentes

Persistent classes are classes in an application that implement the entities of the business problem (e.g. Customer and Order in an E-commerce application). The term "persistent" here means that the classes are able to be persisted, not that they are in the persistent state (see *Sección 11.1, "Estados de objeto de Hibernate"* for discussion).

Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model. However, none of these rules are hard requirements. Indeed, Hibernate assumes very little about the nature of your persistent objects. You can express a domain model in other ways (using trees of `java.util.Map` instances, for example).

## 4.1. Ejemplo simple de POJO

### Ejemplo 4.1. Simple POJO representing a cat

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
private Long id; // identifier

private Date birthdate;
private Color color;
private char sex;
private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

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

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }

    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }
```

```
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }

    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kitten.setMother(this);
    kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}
```

En las siguientes secciones vamos a explorar en mayor detalle las cuatro reglas principales de las clases persistentes.

### 4.1.1. Implemente un constructor sin argumentos

`Cat` has a no-argument constructor. All persistent classes must have a default constructor (which can be non-public) so that Hibernate can instantiate them using `java.lang.reflect.Constructor.newInstance()`. It is recommended that this constructor be defined with at least *package* visibility in order for runtime proxy generation to work properly.

### 4.1.2. Provide an identifier property

**Nota**

Historically this was considered option. While still not (yet) enforced, this should be considered a deprecated feature as it will be completely required to provide a identifier property in an upcoming release.

`Cat` has a property named `id`. This property maps to the primary key column(s) of the underlying database table. The type of the identifier property can be any "basic" type (see *???*). See *Sección 9.4, "Componentes como identificadores compuestos"* for information on mapping composite (multi-column) identifiers.

**Nota**

Identifiers do not necessarily need to identify column(s) in the database physically defined as a primary key. They should just identify columns that can be used to uniquely identify rows in the underlying table.

Le recomendamos que declare propiedades identificadoras nombradas-consistentemente en clases persistentes. y que utilice un tipo nulable (por ejemplo, no primitivo).

### 4.1.3. Prefer non-final classes (semi-optional)

A central feature of Hibernate, *proxies* (lazy loading), depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods. You can persist `final` classes that do not implement an interface with Hibernate; you will not, however, be able to use proxies for lazy association fetching which will ultimately limit your options for performance tuning. To persist a `final` class which does not implement a "full" interface you must disable proxy generation. See *Ejemplo 4.2, "Disabling proxies in hbm.xml"* and *Ejemplo 4.3, "Disabling proxies in annotations"*.

#### Ejemplo 4.2. Disabling proxies in `hbm.xml`

```
<class name="Cat" lazy="false"...>...</class>
```

#### Ejemplo 4.3. Disabling proxies in annotations

```
@Entity @Proxy(lazy=false) public class Cat { ... }
```

If the `final` class does implement a proper interface, you could alternatively tell Hibernate to use the interface instead when generating the proxies. See *Ejemplo 4.4, "Proxying an interface in hbm.xml"* and *Ejemplo 4.5, "Proxying an interface in annotations"*.

### Ejemplo 4.4. Proxying an interface in `hbm.xml`

```
<class name="Cat" proxy="ICat"...>...</class>
```

### Ejemplo 4.5. Proxying an interface in annotations

```
@Entity @Proxy(proxyClass=ICat.class) public class Cat implements ICat { ... }
```

You should also avoid declaring `public final` methods as this will again limit the ability to generate *proxies* from this class. If you want to use a class with `public final` methods, you must explicitly disable proxying. Again, see *Ejemplo 4.2, "Disabling proxies in hbm.xml"* and *Ejemplo 4.3, "Disabling proxies in annotations"*.

## 4.1.4. Declare métodos de acceso y de modificación para los campos persistentes (opcional)

`Cat` declares accessor methods for all its persistent fields. Many other ORM tools directly persist instance variables. It is better to provide an indirection between the relational schema and internal data structures of the class. By default, Hibernate persists JavaBeans style properties and recognizes method names of the form `getFoo`, `isFoo` and `setFoo`. If required, you can switch to direct field access for particular properties.

Properties need *not* be declared public. Hibernate can persist a property declared with `package`, `protected` or `private` visibility as well.

# 4.2. Implementación de herencia

Una subclase también tiene que cumplir con la primera y la segunda regla. Hereda su propiedad identificadora de la superclase `Cat`. Por ejemplo:

```
package eg;

public class DomesticCat extends Cat {
        private String name;

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

```
}
```

## 4.3. Implementando `equals()` y `hashCode()`

Tiene que sobrescribir los métodos `equals()` y `hashCode()` si:

- piensa poner instancias de clases persistentes en un `Set` (la forma recomendada de representar asociaciones multivaluadas); *y*
- piensa utilizar reasociación de instancias separadas.

Hibernate garantiza la equivalencia de identidad persistente (fila de base de datos) y de identidad Java sólamente dentro del ámbito de una sesión en particular. De modo que en el momento en que mezcla instancias recuperadas en sesiones diferentes, tiene que implementar `equals()` y `hashCode()` si desea tener una semántica significativa para `Sets`.

La forma más obvia es implementar `equals()`/`hashCode()` comparando el valor identificador de ambos objetos. Si el valor es el mismo, ambos deben ser la misma fila de la base de datos ya que son iguales. Si ambos son agregados a un `Set`, sólo tendremos un elemento en el `Set`). Desafortunadamente, no puede utilizar este enfoque con identificadores generados. Hibernate sólo asignará valores identificadores a objetos que son persistentes; una instancia recién creada no tendrá ningún valor identificador. Además, si una instancia no se encuentra guardada y está actualmente en un `Set`, al guardarla se asignará un valor identificador al objeto. Si `equals()` y `hashCode()` están basados en el valor identificador, el código hash podría cambiar, rompiendo el contrato del `Set`. Consulte el sitio web de Hibernate y allí encontrará una discusión completa sobre este problema. Este no es un problema de Hibernate, sino de la semántica normal de Java de identidad de objeto e igualdad.

Le recomendamos implementar `equals()` y `hashCode()` utilizando *igualdad de clave empresarial (Business key equality)*. Igualdad de clave empresarial significa que el método `equals()` sólamente compara las propiedades que forman la clave empresarial. Esta es una clave que podría identificar nuestra instancia en el mundo real (una clave candidata *natural*):

```
public class Cat {

    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if ( !(other instanceof Cat) ) return false;

        final Cat cat = (Cat) other;

        if ( !cat.getLitterId().equals( getLitterId() ) ) return false;
        if ( !cat.getMother().equals( getMother() ) ) return false;

        return true;
    }
```

```
    public int hashCode() {
        int result;
        result = getMother().hashCode();
        result = 29 * result + getLitterId();
        return result;
    }

}
```

A business key does not have to be as solid as a database primary key candidate (see *Sección 13.1.3, "Consideración de la identidad del objeto"*). Immutable or unique properties are usually good candidates for a business key.

## 4.4. Modelos dinámicos

**Nota**

*The following features are currently considered experimental and may change in the near future.*

Las entidades persistentes no necesariamente tienen que estar representadas como clases POJO o como objetos JavaBean en tiempo de ejecución. Hibernate también soporta modelos dinámicos (utilizando `Map`eos de `Map`eos en tiempo de ejecución) y la representación de entidades como árboles de DOM4J. No escriba clases persistentes con este enfoque, sólamente archivos de mapeo.

By default, Hibernate works in normal POJO mode. You can set a default entity representation mode for a particular `SessionFactory` using the `default_entity_mode` configuration option (see *Tabla 3.3, "Propiedades de Configuración de Hibernate"*).

Los siguientes ejemplos demuestran la representación utilizando `Map`eos. Primero, en el archivo de mapeo tiene que declararse un `entity-name` en lugar de, o además de un nombre de clase:

```
<hibernate-mapping>

    <class entity-name="Customer">

        <id name="id"
            type="long"
            column="ID">
            <generator class="sequence"/>
        </id>

        <property name="name"
            column="NAME"
            type="string"/>

        <property name="address"
            column="ADDRESS"
```

```
            type="string"/>

        <many-to-one name="organization"
            column="ORGANIZATION_ID"
            class="Organization"/>

        <bag name="orders"
            inverse="true"
            lazy="false"
            cascade="all">
            <key column="CUSTOMER_ID"/>
            <one-to-many class="Order"/>
        </bag>

    </class>

</hibernate-mapping>
```

Aunque las asociaciones se declaran utilizando nombres de clase destino, el tipo destino de una asociación puede ser además una entidad dinámica en lugar de un POJO.

Después de establecer el modo de entidad predeterminado como `dynamic-map` para la `SessionFactory`, puede trabajar en tiempo de ejecución con `Map`eos de `Map`eos:

```
Session s = openSession();
Transaction tx = s.beginTransaction();

// Create a customer
Map david = new HashMap();
david.put("name", "David");

// Create an organization
Map foobar = new HashMap();
foobar.put("name", "Foobar Inc.");

// Link both
david.put("organization", foobar);

// Save both
s.save("Customer", david);
s.save("Organization", foobar);

tx.commit();
s.close();
```

Una de las ventajas principales de un mapeo dinámico es el rápido tiempo de entrega del prototipado sin la necesidad de implementar clases de entidad. Sin embargo, pierde el chequeo de tipos en tiempo de compilación y muy probablemente tendrá que tratar con muchas excepciones en tiempo de ejecución. Gracias al mapeo de Hibernate, el esquema de base de datos se puede normalizar y volver sólido, permitiendo añadir una implementación apropiada del modelo de dominio más adelante.

Los modos de representación de entidad se pueden establecer por `Session`:

```
Session dynamicSession = pojoSession.getSession(EntityMode.MAP);

// Create a customer
Map david = new HashMap();
david.put("name", "David");
dynamicSession.save("Customer", david);
...
dynamicSession.flush();
dynamicSession.close()
...
// Continue on pojoSession
```

Tenga en cuenta que la llamada a `getSession()` utilizando un `EntityMode` está en la API de `Session`, no en la de `SessionFactory`. De esta forma, la nueva `Session` comparte la conexión JDBC, la transacción y otra información de contexto. Esto significa que no tiene que llamar a `flush()` ni a `close()` en la `Session` secundaria, y también tiene que dejar el manejo de la transacción y de la conexión a la unidad de trabajo primaria.

More information about the XML representation capabilities can be found in *Capítulo 20, Mapeo XML*.

## 4.5. Tuplizers

`org.hibernate.tuple.Tuplizer` and its sub-interfaces are responsible for managing a particular representation of a piece of data given that representation's `org.hibernate.EntityMode`. If a given piece of data is thought of as a data structure, then a tuplizer is the thing that knows how to create such a data structure, how to extract values from such a data structure and how to inject values into such a data structure. For example, for the POJO entity mode, the corresponding tuplizer knows how create the POJO through its constructor. It also knows how to access the POJO properties using the defined property accessors.

There are two (high-level) types of Tuplizers:

- `org.hibernate.tuple.entity.EntityTuplizer` which is responsible for managing the above mentioned contracts in regards to entities
- `org.hibernate.tuple.component.ComponentTuplizer` which does the same for components

Users can also plug in their own tuplizers. Perhaps you require that `java.util.Map` implementation other than `java.util.HashMap` be used while in the dynamic-map entity-mode. Or perhaps you need to define a different proxy generation strategy than the one used by default. Both would be achieved by defining a custom tuplizer implementation. Tuplizer definitions are attached to the entity or component mapping they are meant to manage. Going back to the example of our `Customer` entity, *Ejemplo 4.6, “Specify custom tuplizers in annotations”* shows how to specify a custom `org.hibernate.tuple.entity.EntityTuplizer` using annotations while *Ejemplo 4.7, “Specify custom tuplizers in hbm.xml”* shows how to do the same in `hbm.xml`

### Ejemplo 4.6. Specify custom tuplizers in annotations

```
@Entity
@Tuplizer(impl = DynamicEntityTuplizer.class)
public interface Cuisine {
    @Id
    @GeneratedValue
    public Long getId();
    public void setId(Long id);

    public String getName();
    public void setName(String name);

    @Tuplizer(impl = DynamicComponentTuplizer.class)
    public Country getCountry();
    public void setCountry(Country country);
}
```

### Ejemplo 4.7. Specify custom tuplizers in `hbm.xml`

```
<hibernate-mapping>
    <class entity-name="Customer">
        <!--
            Override the dynamic-map entity-mode
            tuplizer for the customer entity
        -->
        <tuplizer entity-mode="dynamic-map"
                class="CustomMapTuplizerImpl"/>

        <id name="id" type="long" column="ID">
            <generator class="sequence"/>
        </id>

        <!-- other properties -->
        ...
    </class>
</hibernate-mapping>
```

## 4.6. EntityNameResolvers

`org.hibernate.EntityNameResolver` is a contract for resolving the entity name of a given entity instance. The interface defines a single method `resolveEntityName` which is passed the entity instance and is expected to return the appropriate entity name (null is allowed and would indicate that the resolver does not know how to resolve the entity name of the given entity instance). Generally speaking, an `org.hibernate.EntityNameResolver` is going to be most useful in the case of dynamic models. One example might be using proxied interfaces as your domain model. The hibernate test suite has an example of this exact style of usage under the org.hibernate.test.dynamicentity.tuplizer2. Here is some of the code from that package for illustration.

```
/**
 * A very trivial JDK Proxy InvocationHandler implementation where we proxy an
 * interface as the domain model and simply store persistent state in an internal
 * Map.  This is an extremely trivial example meant only for illustration.
 */
public final class DataProxyHandler implements InvocationHandler {
        private String entityName;
        private HashMap data = new HashMap();

        public DataProxyHandler(String entityName, Serializable id) {
                this.entityName = entityName;
                data.put( "Id", id );
        }

        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                String methodName = method.getName();
                if ( methodName.startsWith( "set" ) ) {
                        String propertyName = methodName.substring( 3 );
                        data.put( propertyName, args[0] );
                }
                else if ( methodName.startsWith( "get" ) ) {
                        String propertyName = methodName.substring( 3 );
                        return data.get( propertyName );
                }
                else if ( "toString".equals( methodName ) ) {
                        return entityName + "#" + data.get( "Id" );
                }
                else if ( "hashCode".equals( methodName ) ) {
                        return new Integer( this.hashCode() );
                }
                return null;
        }

        public String getEntityName() {
                return entityName;
        }

        public HashMap getData() {
                return data;
        }
}

public class ProxyHelper {
    public static String extractEntityName(Object object) {
        // Our custom java.lang.reflect.Proxy instances actually bundle
        // their appropriate entity name, so we simply extract it from there
        // if this represents one of our proxies; otherwise, we return null
        if ( Proxy.isProxyClass( object.getClass() ) ) {
            InvocationHandler handler = Proxy.getInvocationHandler( object );
            if ( DataProxyHandler.class.isAssignableFrom( handler.getClass() ) ) {
                DataProxyHandler myHandler = ( DataProxyHandler ) handler;
                return myHandler.getEntityName();
            }
        }
        return null;
    }

    // various other utility methods ....
```

```
}

/**
 * The EntityNameResolver implementation.
 *
 * IMPL NOTE : An EntityNameResolver really defines a strategy for how entity names
 * should be resolved.  Since this particular impl can handle resolution for all of our
 * entities we want to take advantage of the fact that SessionFactoryImpl keeps these
 * in a Set so that we only ever have one instance registered.  Why?  Well, when it
 * comes time to resolve an entity name, Hibernate must iterate over all the registered
 * resolvers.  So keeping that number down helps that process be as speedy as possible.
 * Hence the equals and hashCode implementations as is
 */
public class MyEntityNameResolver implements EntityNameResolver {
    public static final MyEntityNameResolver INSTANCE = new MyEntityNameResolver();

    public String resolveEntityName(Object entity) {
        return ProxyHelper.extractEntityName( entity );
    }

    public boolean equals(Object obj) {
        return getClass().equals( obj.getClass() );
    }

    public int hashCode() {
        return getClass().hashCode();
    }
}

public class MyEntityTuplizer extends PojoEntityTuplizer {
       public MyEntityTuplizer(EntityMetamodel entityMetamodel, PersistentClass mappedEntity) {
               super( entityMetamodel, mappedEntity );
        }

        public EntityNameResolver[] getEntityNameResolvers() {
               return new EntityNameResolver[] { MyEntityNameResolver.INSTANCE };
        }

    public String determineConcreteSubclassEntityName(Object entityInstance, SessionFactoryImplementor factory) {
        String entityName = ProxyHelper.extractEntityName( entityInstance );
        if ( entityName == null ) {
            entityName = super.determineConcreteSubclassEntityName( entityInstance, factory );
        }
        return entityName;
    }

    ...
```

Con el fin de registrar un `org.hibernate.EntityNameResolver` los usuarios deben:

1. Implement a custom tuplizer (see *Sección 4.5, "Tuplizers"*), implementing the `getEntityNameResolvers` method

2. Registrarlo con el `org.hibernate.impl.SessionFactoryImpl` (el cual es la clase de implementación para `org.hibernate.SessionFactory`) usando el método `registerEntityNameResolver`.

# Mapeo O/R Básico

## 5.1. Declaración de mapeo

Object/relational mappings can be defined in three approaches:

- using Java 5 annotations (via the Java Persistence 2 annotations)
- using JPA 2 XML deployment descriptors (described in chapter XXX)
- using the Hibernate legacy XML files approach known as hbm.xml

Annotations are split in two categories, the logical mapping annotations (describing the object model, the association between two entities etc.) and the physical mapping annotations (describing the physical schema, tables, columns, indexes, etc). We will mix annotations from both categories in the following code examples.

JPA annotations are in the `javax.persistence.*` package. Hibernate specific extensions are in `org.hibernate.annotations.*`. You favorite IDE can auto-complete annotations and their attributes for you (even without a specific "JPA" plugin, since JPA annotations are plain Java 5 annotations).

Here is an example of mapping

```
package eg;

@Entity
@Table(name="cats") @Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorValue("C") @DiscriminatorColumn(name="subclass", discriminatorType=CHAR)
public class Cat {

   @Id @GeneratedValue
   public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public BigDecimal getWeight() { return weight; }
   public void setWeight(BigDecimal weight) { this.weight = weight; }
   private BigDecimal weight;

   @Temporal(DATE) @NotNull @Column(updatable=false)
   public Date getBirthdate() { return birthdate; }
   public void setBirthdate(Date birthdate) { this.birthdate = birthdate; }
   private Date birthdate;

   @org.hibernate.annotations.Type(type="eg.types.ColorUserType")
   @NotNull @Column(updatable=false)
   public ColorType getColor() { return color; }
   public void setColor(ColorType color) { this.color = color; }
   private ColorType color;

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

```
   public String getSex() { return sex; }
   public void setSex(String sex) { this.sex = sex; }
   private String sex;

   @NotNull @Column(updatable=false)
   public Integer getLitterId() { return litterId; }
   public void setLitterId(Integer litterId) { this.litterId = litterId; }
   private Integer litterId;

   @ManyToOne @JoinColumn(name="mother_id", updatable=false)
   public Cat getMother() { return mother; }
   public void setMother(Cat mother) { this.mother = mother; }
   private Cat mother;

   @OneToMany(mappedBy="mother") @OrderBy("litterId")
   public Set<Cat> getKittens() { return kittens; }
   public void setKittens(Set<Cat> kittens) { this.kittens = kittens; }
   private Set<Cat> kittens = new HashSet<Cat>();
}

@Entity @DiscriminatorValue("D")
public class DomesticCat extends Cat {

   public String getName() { return name; }
   public void setName(String name) { this.name = name }
   private String name;
}

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

The legacy hbm.xml approach uses an XML schema designed to be readable and hand-editable. The mapping language is Java-centric, meaning that mappings are constructed around persistent class declarations and not table declarations.

Observe que, incluso aunque muchos de los usuarios de Hibernate eligen escribir el XML a mano, existe un número de herramientas para generar el documento de mapeo, incluyendo XDoclet, Middlegen y AndroMDA.

Este es un ejemplo de mapeo:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat"
            table="cats"
            discriminator-value="C">

                <id name="id">
                        <generator class="native"/>
                </id>
```

```
                <discriminator column="subclass"
                     type="character"/>

                <property name="weight"/>

                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>

                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>

                <property name="sex"
                    not-null="true"
                    update="false"/>

                <property name="litterId"
                    column="litterId"
                    update="false"/>

                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>

                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>

                <subclass name="DomesticCat"
                    discriminator-value="D">

                        <property name="name"
                            type="string"/>

                </subclass>

        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

We will now discuss the concepts of the mapping documents (both annotations and XML). We will only describe, however, the document elements and attributes that are used by Hibernate at runtime. The mapping document also contains some extra optional attributes and elements that affect the database schemas exported by the schema export tool (for example, the `not-null` attribute).

### 5.1.1. Entity

An entity is a regular Java object (aka POJO) which will be persisted by Hibernate.

To mark an object as an entity in annotations, use the `@Entity` annotation.

```
@Entity
public class Flight implements Serializable {
    Long id;

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

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

That's pretty much it, the rest is optional. There are however any options to tweak your entity mapping, let's explore them.

`@Table` lets you define the table the entity will be persisted into. If undefined, the table name is the unqualified class name of the entity. You can also optionally define the catalog, the schema as well as unique constraints on the table.

```
@Entity
@Table(name="TBL_FLIGHT",
       schema="AIR_COMMAND",
       uniqueConstraints=
           @UniqueConstraint(
               name="flight_number",
               columnNames={"comp_prefix", "flight_number"} ) )
public class Flight implements Serializable {
    @Column(name="comp_prefix")
    public String getCompagnyPrefix() { return companyPrefix; }

    @Column(name="flight_number")
    public String getNumber() { return number; }
}
```

The constraint name is optional (generated if left undefined). The column names composing the constraint correspond to the column names as defined before the Hibernate `NamingStrategy` is applied.

`@Entity.name` lets you define the shortcut name of the entity you can used in JP-QL and HQL queries. It defaults to the unqualified class name of the class.

Hibernate goes beyond the JPA specification and provide additional configurations. Some of them are hosted on `@org.hibernate.annotations.Entity`:

- `dynamicInsert` / `dynamicUpdate` (defaults to false): specifies that `INSERT` / `UPDATE` SQL should be generated at runtime and contain only the columns whose values are not null. The `dynamic-`

`update` and `dynamic-insert` settings are not inherited by subclasses. Although these settings can increase performance in some cases, they can actually decrease performance in others.

- `selectBeforeUpdate` (defaults to false): specifies that Hibernate should *never* perform an SQL `UPDATE` unless it is certain that an object is actually modified. Only when a transient object has been associated with a new session using `update()`, will Hibernate perform an extra SQL `SELECT` to determine if an `UPDATE` is actually required. Use of `select-before-update` will usually decrease performance. It is useful to prevent a database update trigger being called unnecessarily if you reattach a graph of detached instances to a `Session`.

- `polymorphisms` (defaults to `IMPLICIT`): determines whether implicit or explicit query polymorphisms is used. *Implicit* polymorphisms means that instances of the class will be returned by a query that names any superclass or implemented interface or class, and that instances of any subclass of the class will be returned by a query that names the class itself. *Explicit* polymorphisms means that class instances will be returned only by queries that explicitly name that class. Queries that name the class will return only instances of subclasses mapped. For most purposes, the default `polymorphisms=IMPLICIT` is appropriate. Explicit polymorphisms is useful when two different classes are mapped to the same table This allows a "lightweight" class that contains a subset of the table columns.

- `persister`: specifies a custom `ClassPersister`. The `persister` attribute lets you customize the persistence strategy used for the class. You can, for example, specify your own subclass of `org.hibernate.persister.EntityPersister`, or you can even provide a completely new implementation of the interface `org.hibernate.persister.ClassPersister` that implements, for example, persistence via stored procedure calls, serialization to flat files or LDAP. See `org.hibernate.test.CustomPersister` for a simple example of "persistence" to a `Hashtable`.

- `optimisticLock` (defaults to `VERSION`): determines the optimistic locking strategy. If you enable `dynamicUpdate`, you will have a choice of optimistic locking strategies:

  - `version`: chequea las columnas de versión/sello de fecha

  - `all`: chequea todas las columnas

  - `dirty`: chequea las columnas modificadas permitiendo algunas actualizaciones concurrentes

  - `none`: no utilice bloqueo optimista

  Le recomendamos *mucho* que utilice columnas de versión/sello de fecha para el bloqueo optimista con Hibernate. Esta estrategia optimiza el rendimiento y maneja correctamente las modificaciones realizadas a las instancias separadas, (por ejemplo, cuando se utiliza `Session.merge()`).

### Sugerencia

Be sure to import `@javax.persistence.Entity` to mark a class as an entity. It's a common mistake to import `@org.hibernate.annotations.Entity` by accident.

Some entities are not mutable. They cannot be updated or deleted by the application. This allows Hibernate to make some minor performance optimizations.. Use the `@Immutable` annotation.

You can also alter how Hibernate deals with lazy initialization for this class. On `@Proxy`, use `lazy`=false to disable lazy fetching (not recommended). You can also specify an interface to use for lazy initializing proxies (defaults to the class itself): use `proxyClass` on `@Proxy`. Hibernate will initially return proxies (Javassist or CGLIB) that implement the named interface. The persistent object will load when a method of the proxy is invoked. See "Initializing collections and proxies" below.

`@BatchSize` specifies a "batch size" for fetching instances of this class by identifier. Not yet loaded instances are loaded batch-size at a time (default 1).

You can specific an arbitrary SQL WHERE condition to be used when retrieving objects of this class. Use `@Where` for that.

In the same vein, `@Check` lets you define an SQL expression used to generate a multi-row *check* constraint for automatic schema generation.

There is no difference between a view and a base table for a Hibernate mapping. This is transparent at the database level, although some DBMS do not support views properly, especially with updates. Sometimes you want to use a view, but you cannot create one in the database (i.e. with a legacy schema). In this case, you can map an immutable and read-only entity to a given SQL subselect expression using `@org.hibernate.annotations.Subselect`:

```
@Entity
@Subselect("select item.name, max(bid.amount), count(*) "
        + "from item "
        + "join bid on bid.item_id = item.id "
        + "group by item.name")
@Synchronize( {"item", "bid"} ) //tables impacted
public class Summary {
    @Id
    public String getId() { return id; }
    ...
}
```

Declara las tablas con las cuales se debe sincronizar esta entidad, asegurándose de que el auto-vaciado ocurra correctamente y que las consultas frente a la entidad derivada no devuelvan datos desactualizados. El `<subselect>` se encuentra disponible tanto como un atributo y como un elemento anidado de mapeo.

We will now explore the same options using the hbm.xml structure. You can declare a persistent class using the `class` element. For example:

```
<class
        name="ClassName"                                          (1)
        table="tableName"                                         (2)
        discriminator-value="discriminator_value"                 (3)
        mutable="true|false"                                      (4)
        schema="owner"                                            (5)
        catalog="catalog"                                         (6)
        proxy="ProxyInterface"                                    (7)
        dynamic-update="true|false"                               (8)
        dynamic-insert="true|false"                               (9)
        select-before-update="true|false"                         (10)
        polymorphism="implicit|explicit"                          (11)
        where="arbitrary sql where condition"                     (12)
        persister="PersisterClass"                                (13)
        batch-size="N"                                            (14)
        optimistic-lock="none|version|dirty|all"                  (15)
        lazy="true|false"                                         (16)
        entity-name="EntityName"                                  (17)
        check="arbitrary sql check condition"                     (18)
        rowid="rowid"                                             (19)
        subselect="SQL expression"                                (20)
        abstract="true|false"                                     (21)
        node="element-name"
/>
```

1. `name` (opcional): El nombre completamente calificado de la clase Java persistente (o interfaz). Si se omite este atributo, se asume que el mapeo es para una entidad que no es POJO.
2. `table` (opcional - por defecto es el nombre de la clase no calificado): El nombre de su tabla en la base de datos.
3. `discriminator-value` (opcional - predeterminado al nombre de la clase): Un valor que distingue subclases individuales, usado para el comportamiento polimórfico. Los valores aceptables incluyen `null` y `not null`.
4. `mutable` (opcional, por defecto es `true`): Especifica que las instancias de la clase (no) son mutables.
5. `schema` (opcional): Sobrescribe el nombre del esquema especificado por el elemento raíz `<hibernate-mapping>`.
6. `catalog` (opcional): Sobrescribe el nombre del catálogo especificado por el elemento raíz `<hibernate-mapping>`.
7. `proxy` (opcional): Especifica una interfaz a utilizar para los proxies de inicialización perezosa. Puede especificar el nombre mismo de la clase.

8. `dynamic-update` (opcional, por defecto es `false`): Especifica que el SQL `UPDATE` debe ser generado en tiempo de ejecución y puede contener sólamente aquellas columnas cuyos valores hayan cambiado.
9. `dynamic-insert` (opcional, por defecto es `false`): Especifica que el SQL `INSERT` debe ser generado en tiempo de ejecución y debe contener sólamente aquellas columnas cuyos valores no son nulos.
10. `select-before-update` (opcional, por defecto es `false`): Especifica que Hibernate *nunca* debe realizar un `UPDATE` SQL a menos de que se tenga certeza de que realmente se haya modificado un objeto. Sólo cuando un objeto transitorio ha sido asociado con una sesión nueva utilizando `update()`), Hibernate realizará una SQL `SELECT` extra para determinar si realmente se necesita un `UPDATE`.
11. `polymorphisms` (optional - defaults to `implicit`): determines whether implicit or explicit query polymorphisms is used.
12. `where` (opcional) especifica una condición SQL `WHERE` arbitraria para utilizarla en la recuperación de objetos de esta clase.
13. `persister` (opcional): Especifica un `ClassPersister` personalizado.
14. `batch-size` (opcional, por defecto es `1`) especifica un "tamaño de lote" para buscar instancias de esta clase por identificador.
15. `optimistic-lock` (opcional, por defecto es `version`): Determina la estrategia optimista de bloqueo.
16. `lazy` (opcional): La recuperación perezosa se puede deshabilitar por completo al establecer `lazy="false"`.
17. `entity-name` (optional - defaults to the class name): Hibernate3 allows a class to be mapped multiple times, potentially to different tables. It also allows entity mappings that are represented by Maps or XML at the Java level. In these cases, you should provide an explicit arbitrary name for the entity. See *Sección 4.4, "Modelos dinámicos"* and *Capítulo 20, Mapeo XML* for more information.
18. `check` (opcional): Una expresión SQL utilizada para generar una restricción *check* multi-filas para la generación automática de esquemas.
19. `rowid` (opcional): Hibernate puede utilizar los llamados ROWIDs en las bases de datos. Por ejemplo, en Oracle, Hibernate puede utilizar la columna extra `rowid` para actualizaciones rápidas si usted establece esta opción como `rowid`. Un ROWID es un detalle de implementación y representa la posición física de la tupla almacenada.
20. `subselect` (opcional): Mapea una entidad inmutable y de sólo lectura a una subselección de base de datos. Es útil si quiere tener una vista en vez de una tabla base. Vea a continuación para obtener más información.
21. `abstract` (opcional): Utilizado para marcar superclases abstractas en las jerarquías `<union-subclass>`.

Es perfectamente aceptable que la clase persistente mencionada sea una interfaz. Puede declarar clases que implementan esa interfaz utilizando el elemento `<subclass>`. Puede persistir cualquier clase interna *estática*. Debe especificar el nombre de la clase utilizando la forma estándar, por ejemplo, `e.g.Foo$Bar`.

Here is how to do a virtual view (subselect) in XML:

```
<class name="Summary">
    <subselect>
        select item.name, max(bid.amount), count(*)
        from item
        join bid on bid.item_id = item.id
        group by item.name
    </subselect>
    <synchronize table="item"/>
    <synchronize table="bid"/>
    <id name="name"/>
    ...
</class>
```

The `<subselect>` is available both as an attribute and a nested mapping element.

### 5.1.2. Identifiers

Mapped classes *must* declare the primary key column of the database table. Most classes will also have a JavaBeans-style property holding the unique identifier of an instance.

Mark the identifier property with `@Id`.

```
@Entity
public class Person {
   @Id Integer getId() { ... }
   ...
}
```

In hbm.xml, use the `<id>` element which defines the mapping from that property to the primary key column.

```
<id
        name="propertyName"                                          (1)
        type="typename"                                              (2)
        column="column_name"                                         (3)
        unsaved-value="null|any|none|undefined|id_value"             (4)
        access="field|property|ClassName">                           (5)
        node="element-name|@attribute-name|element/@attribute|."

        <generator class="generatorClass"/>
</id>
```

1. `name` (opcional): El nombre de la propiedad del identificador. s
2. `type` (opcional): un nombre que indica el tipo de Hibernate.
3. `column` (opcional - por defecto es el nombre de la propiedad): El nombre de la columna de la clave principal.

4. `unsaved-value` (opcional - por defecto es un valor "sensible"): Un valor de la propiedad identificadora que indica que una instancia está recién instanciada (sin guardar), distinguiéndola de las instancias separadas que fueron guardadas o cargadas en una sesión previa.
5. `access` (opcional - por defecto es `property`): La estrategia que Hibernate debe utilizar para acceder al valor de la propiedad.

Si se omite el atributo `name`, se asume que la clase no tiene propiedad identificadora.

The `unsaved-value` attribute is almost never needed in Hibernate3 and indeed has no corresponding element in annotations.

You can also declare the identifier as a composite identifier. This allows access to legacy data with composite keys. Its use is strongly discouraged for anything else.

#### 5.1.2.1. Composite identifier

You can define a composite primary key through several syntaxes:

- use a component type to represent the identifier and map it as a property in the entity: you then annotated the property as `@EmbeddedId`. The component type has to be `Serializable`.
- map multiple properties as `@Id` properties: the identifier type is then the entity class itself and needs to be `Serializable`. This approach is unfortunately not standard and only supported by Hibernate.
- map multiple properties as `@Id` properties and declare an external class to be the identifier type. This class, which needs to be `Serializable`, is declared on the entity via the `@IdClass` annotation. The identifier type must contain the same properties as the identifier properties of the entity: each property name must be the same, its type must be the same as well if the entity property is of a basic type, its type must be the type of the primary key of the associated entity if the entity property is an association (either a `@OneToOne` or a `@ManyToOne`).

As you can see the last case is far from obvious. It has been inherited from the dark ages of EJB 2 for backward compatibilities and we recommend you not to use it (for simplicity sake).

Let's explore all three cases using examples.

##### 5.1.2.1.1. id as a property using a component type

Here is a simple example of `@EmbeddedId`.

```
@Entity
class User {
   @EmbeddedId
   @AttributeOverride(name="firstName", column=@Column(name="fld_firstname")
```

```
   UserId id;

   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;
}
```

You can notice that the `UserId` class is serializable. To override the column mapping, use `@AttributeOverride`.

An embedded id can itself contains the primary key of an associated entity.

```
@Entity
class Customer {
   @EmbeddedId CustomerId id;
   boolean preferredCustomer;

   @MapsId("userId")
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   @OneToOne User user;
}

@Embeddable
class CustomerId implements Serializable {
   UserId userId;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

In the embedded id object, the association is represented as the identifier of the associated entity. But you can link its value to a regular association in the entity via the `@MapsId` annotation. The `@MapsId` value correspond to the property name of the embedded id object containing

the associated entity's identifier. In the database, it means that the `Customer.user` and the `CustomerId.userId` properties share the same underlying column (`user_fk` in this case).

**Sugerencia**

The component type used as identifier must implement `equals()` and `hashCode()`.

In practice, your code only sets the `Customer.user` property and the user id value is copied by Hibernate into the `CustomerId.userId` property.

**Aviso**

The id value can be copied as late as flush time, don't rely on it until after flush time.

While not supported in JPA, Hibernate lets you place your association directly in the embedded id component (instead of having to use the `@MapsId` annotation).

```
@Entity
class Customer {
   @EmbeddedId CustomerId id;
   boolean preferredCustomer;
}

@Embeddable
class CustomerId implements Serializable {
   @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;


   //implements equals and hashCode
}
```

Let's now rewrite these examples using the hbm.xml syntax.

```
<composite-id
        name="propertyName"
        class="ClassName"
        mapped="true|false"
        access="field|property|ClassName"
        node="element-name|.">

        <key-property name="propertyName" type="typename" column="column_name"/>
        <key-many-to-one name="propertyName" class="ClassName" column="column_name"/>
        ......
</composite-id>
```

First a simple example:

```
<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName" column="fld_firstname"/>
      <key-property name="lastName"/>
   </composite-id>
</class>
```

Then an example showing how an association can be mapped.

```
<class name="Customer">
   <composite-id name="id" class="CustomerId">
      <key-property name="firstName" column="userfirstname_fk"/>
      <key-property name="lastName" column="userfirstname_fk"/>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>

   <many-to-one name="user">
      <column name="userfirstname_fk" updatable="false" insertable="false"/>
      <column name="userlastname_fk" updatable="false" insertable="false"/>
   </many-to-one>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

Notice a few things in the previous example:

- the order of the properties (and column) matters. It must be the same between the association and the primary key of the associated entity
- the many to one uses the same columns as the primary key and thus must be marked as read only (`insertable` and `updatable` to false).
- unlike with `@MapsId`, the id value of the associated entity is not transparently copied, check the `foreign` id generator for more information.

The last example shows how to map association directly in the embedded id component.

```
<class name="Customer">
   <composite-id name="id" class="CustomerId">
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

This is the recommended approach to map composite identifier. The following options should not be considered unless some constraint are present.

#### 5.1.2.1.2. Multiple id properties without identifier type

Another, arguably more natural, approach is to place `@Id` on multiple properties of your entity. This approach is only supported by Hibernate (not JPA compliant) but does not require an extra embeddable component.

```
@Entity
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;
```

```
   boolean preferredCustomer;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

In this case `Customer` is its own identifier representation: it must implement `Serializable` and must implement `equals()` and `hashCode()`.

In hbm.xml, the same mapping is:

```
<class name="Customer">
   <composite-id>
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

#### 5.1.2.1.3. Multiple id properties with with a dedicated identifier type

`@IdClass` on an entity points to the class (component) representing the identifier of the class. The properties marked `@Id` on the entity must have their corresponding property on the `@IdClass`. The return type of search twin property must be either identical for basic properties or must correspond to the identifier class of the associated entity for an association.

**Aviso**

This approach is inherited from the EJB 2 days and we recommend against its use. But, after all it's your application and Hibernate supports it.

```
@Entity
@IdClass(CustomerId.class)
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;

   boolean preferredCustomer;
}

class CustomerId implements Serializable {
   UserId user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;

   //implements equals and hashCode
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

`Customer` and `CustomerId` do have the same properties `customerNumber` as well as `user`. `CustomerId` must be `Serializable` and implement `equals()` and `hashCode()`.

While not JPA standard, Hibernate let's you declare the vanilla associated property in the `@IdClass`.

```
@Entity
@IdClass(CustomerId.class)
```

```
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;

   boolean preferredCustomer;
}

class CustomerId implements Serializable {
   @OneToOne User user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;

   //implements equals and hashCode
}

@Embeddable
class UserId implements Serializable {
  String firstName;
  String lastName;
}
```

This feature is of limited interest though as you are likely to have chosen the `@IdClass` approach to stay JPA compliant or you have a quite twisted mind.

Here are the equivalent on hbm.xml files:

```
<class name="Customer">
   <composite-id class="CustomerId" mapped="true">
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>
```

```
   <property name="age"/>
</class>
```

### 5.1.2.2. Identifier generator

Hibernate can generate and populate identifier values for you automatically. This is the recommended approach over "business" or "natural" id (especially composite ids).

Hibernate offers various generation strategies, let's explore the most common ones first that happens to be standardized by JPA:

- IDENTITY: supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type `long`, `short` or `int`.
- SEQUENCE (called `seqhilo` in Hibernate): uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a named database sequence.
- TABLE (called `MultipleHiLoPerTableGenerator` in Hibernate) : uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a table and column as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database.
- AUTO: selects `IDENTITY`, `SEQUENCE` or `TABLE` depending upon the capabilities of the underlying database.

**Importante**

We recommend all new projects to use the new enhanced identifier generators. They are deactivated by default for entities using annotations but can be activated using `hibernate.id.new_generator_mappings=true`. These new generators are more efficient and closer to the JPA 2 specification semantic.

However they are not backward compatible with existing Hibernate based application (if a sequence or a table is used for id generation). See XXXXXXX *???* for more information on how to activate them.

To mark an id property as generated, use the `@GeneratedValue` annotation. You can specify the strategy used (default to `AUTO`) by setting `strategy`.

```
@Entity
public class Customer {
   @Id @GeneratedValue
   Integer getId() { ... };
}

@Entity
```

```
public class Invoice {
   @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
   Integer getId() { ... };
}
```

`SEQUENCE` and `TABLE` require additional configurations that you can set using `@SequenceGenerator` and `@TableGenerator`:

- `name`: name of the generator
- `table` / `sequenceName`: name of the table or the sequence (defaulting respectively to `hibernate_sequences` and `hibernate_sequence`)
- `catalog` / `schema`:
- `initialValue`: the value from which the id is to start generating
- `allocationSize`: the amount to increment by when allocating id numbers from the generator

In addition, the `TABLE` strategy also let you customize:

- `pkColumnName`: the column name containing the entity identifier
- `valueColumnName`: the column name containing the identifier value
- `pkColumnValue`: the entity identifier
- `uniqueConstraints`: any potential column constraint on the table containing the ids

To link a table or sequence generator definition with an actual generated property, use the same name in both the definition `name` and the generator value `generator` as shown below.

```
@Id
@GeneratedValue(
    strategy=GenerationType.SEQUENCE,
    generator="SEQ_GEN")
@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence",
    allocationSize=20
)
public Integer getId() { ... }
```

The scope of a generator definition can be the application or the class. Class-defined generators are not visible outside the class and can override application level generators. Application level generators are defined in JPA's XML deployment descriptors (see XXXXXX *???*):

```
<table-generator name="EMP_GEN"
            table="GENERATOR_TABLE"
```

```
            pk-column-name="key"
            value-column-name="hi"
            pk-column-value="EMP"
            allocation-size="20"/>

//and the annotation equivalent

@javax.persistence.TableGenerator(
    name="EMP_GEN",
    table="GENERATOR_TABLE",
    pkColumnName = "key",
    valueColumnName = "hi"
    pkColumnValue="EMP",
    allocationSize=20
)

<sequence-generator name="SEQ_GEN"
    sequence-name="my_sequence"
    allocation-size="20"/>

//and the annotation equivalent

@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence",
    allocationSize=20
)
```

If a JPA XML descriptor (like `META-INF/orm.xml`) is used to define the generators, `EMP_GEN` and `SEQ_GEN` are application level generators.

**Nota**

Package level definition is not supported by the JPA specification. However, you can use the `@GenericGenerator` at the package level (see ???).

These are the four standard JPA generators. Hibernate goes beyond that and provide additional generators or additional options as we will see below. You can also write your own custom identifier generator by implementing `org.hibernate.id.IdentifierGenerator`.

To define a custom generator, use the `@GenericGenerator` annotation (and its plural counter part `@GenericGenerators`) that describes the class of the identifier generator or its short cut name (as described below) and a list of key/value parameters. When using `@GenericGenerator` and assigning it via `@GeneratedValue.generator`, the `@GeneratedValue.strategy` is ignored: leave it blank.

```
@Id @GeneratedValue(generator="system-uuid")
@GenericGenerator(name="system-uuid", strategy = "uuid")
public String getId() {
```

```
@Id @GeneratedValue(generator="trigger-generated")
@GenericGenerator(
    name="trigger-generated",
    strategy = "select",
    parameters = @Parameter(name="key", value = "socialSecurityNumber")
)
public String getId() {
```

The hbm.xml approach uses the optional `<generator>` child element inside `<id>`. If any parameters are required to configure or initialize the generator instance, they are passed using the `<param>` element.

```
<id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

#### 5.1.2.2.1. Various additional generators

Todos los generadores implementan la interfaz `org.hibernate.id.IdentifierGenerator`. Esta es una interfaz muy simple. Algunas aplicaciones pueden decidir brindar sus propias implementaciones especializadas. Sin embargo, Hibernate provee un rango de implementaciones ya incorporadas. Los nombres de atajo para los generadores incorporados son los siguientes:

`increment`
: genera indentificadores de tipo `long`, `short` o `int` que sólamente son únicos cuando ningún otro proceso está insertando datos en la misma tabla. *No lo utilice en un clúster.*

`identity`
: soporta columnas de identidad en DB2, MySQL, MS SQL Server, Sybase y HypersonicSQL. El identificador devuelto es de tipo `long`, `short` o `int`.

`sequence`
: usa una secuencia en DB2, PostgreSQL, Oracle, SAP DB, McKoi o un generador en Interbase. El identificador devuelto es de tipo `long`, `short` o `int`.

`hilo`
: utiliza un algoritmo alto/bajo para generar eficientemente identificadores de tipo `long`, `short` o `int`, dada una tabla y columna como fuente de valores altos (por defecto `hibernate_unique_key` y `next_hi` respectivamente). El algoritmo alto/bajo genera identificadores que son únicos sólamente para una base de datos particular.

`seqhilo`
: utiliza un algoritmo alto/bajo para generar eficientemente identificadores de tipo `long`, `short` o `int`, dada una secuencia de base de datos.

`uuid`
: Generates a 128-bit UUID based on a custom algorithm. The value generated is represented as a string of 32 hexidecimal digits. Users can also configure it to use a separator (config parameter "separator") which separates the hexidecimal digits into 8{sep}8{sep}4{sep}8{sep}4. Note specifically that this is different than the IETF RFC 4122 representation of 8-4-4-4-12. If you need RFC 4122 compliant UUIDs, consider using "uuid2" generator discussed below.

`uuid2`
: Generates a IETF RFC 4122 compliant (variant 2) 128-bit UUID. The exact "version" (the RFC term) generated depends on the pluggable "generation strategy" used (see below). Capable of generating values as `java.util.UUID`, `java.lang.String` or as a byte array of length 16 (`byte[16]`). The "generation strategy" is defined by the interface `org.hibernate.id.UUIDGenerationStrategy`. The generator defines 2 configuration parameters for defining which generation strategy to use:

  `uuid_gen_strategy_class`
  : Names the UUIDGenerationStrategy class to use

  `uuid_gen_strategy`
  : Names the UUIDGenerationStrategy instance to use

  Out of the box, comes with the following strategies:

  - `org.hibernate.id.uuid.StandardRandomStrategy` (the default) - generates "version 3" (aka, "random") UUID values via the `randomUUID` method of `java.util.UUID`
  - `org.hibernate.id.uuid.CustomVersionOneStrategy` - generates "version 1" UUID values, using IP address since mac address not available. If you need mac address to be used, consider leveraging one of the existing third party UUID generators which sniff out mac address and integrating it via the `org.hibernate.id.UUIDGenerationStrategy` contract. Two such libraries known at time of this writing include *http://johannburkard.de/software/uuid/* and *http://commons.apache.org/sandbox/id/uuid.html*

`guid`
: utiliza una cadena GUID generada por base de datos en MS SQL Server y MySQL.

`native`
: selecciona `identity`, `sequence` o `hilo` dependiendo de las capacidades de la base de datos subyacente.

`assigned`
: deja a la aplicación asignar un identificador al objeto antes de que se llame a `save()`. Esta es la estrategia por defecto si no se especifica un elemento `<generator>`.

`select`
: recupera una clave principal asignada por un disparador de base de datos seleccionando la fila por alguna clave única y recuperando el valor de la clave principal.

`foreign`
: utiliza el identificador de otro objeto asociado. Generalmente se usa en conjunto cón a una asociación de clave principal `<one-to-one>`.

`sequence-identity`
: una estrategia de generación de secuencias especilizadas que utiliza una secuencia de base de datos para el valor real de la generación, pero combina esto junto con JDBC3 getGeneratedKeys para devolver el valor del identificador generado como parte de la ejecución de la declaración de inserción. Esta estrategia está soportada sólamente en los controladores 10g de Oracle destinados para JDK1.4. Los comentarios en estas declaraciones de inserción están desactivados debido a un error en los controladores de Oracle.

#### 5.1.2.2.2. Algoritmo alto/bajo

Los generadores `hilo` y `seqhilo` brindan dos implementaciones opcionales del algoritmo alto/bajo. La primera implementación necesita de una tabla "especial" de base de datos para tener el siguiente valor "alto" disponible. La segunda utiliza una secuencia del estilo de Oracle, donde se encuentre soportada.

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

Desafortunadamente, no puede utilizar `hilo` cuando le provea su propia `Connection` a Hibernate. Cuando Hibernate está utilizando una fuente de datos del servidor de aplicaciones para obtener conexiones alistadas con JTA, usted tiene que configurar el `hibernate.transaction.manager_lookup_class`.

#### 5.1.2.2.3. Algoritmo UUID

El UUID contiene: la dirección IP, el tiempo de iniciación de la MVJ, con una precisión de un cuarto de segundo, el tiempo de sistema y un valor de contador (único en la MVJ). No es posible obtener una dirección MAC o una dirección de memoria desde el código Java, así que esto es la mejor opción sin tener que utilizar JNI.

#### 5.1.2.2.4. Columnas de identidad y secuencias

Para las bases de datos que soportan columnas de identidad (DB2, MySQL, Sybase, MS SQL), puede utilizar generación de claves `identity`. Para las bases de datos que soportan las secuencias (DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB) puede utilizar la generación de claves del estilo `sequence`. Ambas estrategias requieren dos consultas SQL para insertar un nuevo objeto. Por ejemplo:

```
<id name="id" type="long" column="person_id">
        <generator class="sequence">
                <param name="sequence">person_id_sequence</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="person_id" unsaved-value="0">
        <generator class="identity"/>
</id>
```

Para desarrollos a través de plataformas, la estrategia `native` eligirá entre las estrategias `identity`, `sequence` e `hilo`, dependiendo de las capacidades de la base de datos subyacente.

#### 5.1.2.2.5. Identificadores asignados

If you want the application to assign identifiers, as opposed to having Hibernate generate them, you can use the `assigned` generator. This special generator uses the identifier value already assigned to the object's identifier property. The generator is used when the primary key is a natural key instead of a surrogate key. This is the default behavior if you do not specify `@GeneratedValue` nor `<generator>` elements.

El generador `assigned` hace que Hibernate utilice `unsaved-value="undefined"`. Esto fuerza a Hibernate a ir a la base de datos para determinar si una instancia es transitoria o separada, a menos de que haya una propiedad de versión o sello de fecha, o que usted defina `Interceptor.isUnsaved()`.

#### 5.1.2.2.6. Claves primarias asignadas por disparadores

Hibernate no genera DDL con disparadores. Es para los esquemas heredados sólamente.

```
<id name="id" type="long" column="person_id">
        <generator class="select">
                <param name="key">socialSecurityNumber</param>
        </generator>
</id>
```

En el ejemplo anterior, hay una propiedad única llamada `socialSecurityNumber`, Esta está definida por la clase, como una clave natural y una clave sustituta llamada `person_id`, cuyo valor es generado por un disparador.

#### 5.1.2.2.7. Identity copy (foreign generator)

Finally, you can ask Hibernate to copy the identifier from another associated entity. In the Hibernate jargon, it is known as a foreign generator but the JPA mapping reads better and is encouraged.

```
@Entity
class MedicalHistory implements Serializable {
  @Id @OneToOne
  @JoinColumn(name = "person_id")
  Person patient;
}

@Entity
public class Person implements Serializable {
  @Id @GeneratedValue Integer id;
}
```

Or alternatively

```
@Entity
class MedicalHistory implements Serializable {
  @Id Integer id;

  @MapsId @OneToOne
  @JoinColumn(name = "patient_id")
  Person patient;
}

@Entity
class Person {
  @Id @GeneratedValue Integer id;
}
```

In hbm.xml use the following approach:

```
<class name="MedicalHistory">
   <id name="id">
      <generator class="foreign">
         <param name="property">patient</param>
      </generator>
   </id>
   <one-to-one name="patient" class="Person" constrained="true"/>
</class>
```

### 5.1.2.3. Generadores mejorados del identificador

Desde el lanzamiento 3.2.3, hay 2 nuevos generadores, los cuales representan una nueva reflexión sobre dos aspectos diferentes de la generación del identificador. El primer aspecto es qúe tan portátil es la base de datos; el segudno es la optimización. La optimización significa que no tiene que preguntarle a la base de datos por toda petición de un nuevo valor identificador. Estos dos nuevos generadores tienen el propósito de tomar el lugar de algunos de los generadores nombrados que describimos anteriormente, empezando por 3.3.x. Sin embargo, están incluídos en los lanzamientos actuales y puede ser referenciados por FQN.

El primero de estos nuevos generadores es `org.hibernate.id.enhanced.SequenceStyleGenerator`, el cual tiene el propósito, primero, de ser el reemplazo para el generador `sequence` y segundo, de ser un generador de portabilidad mejor que `native`. Esto se debe a que `native` generalmente escoge entre `identity` y `sequence`, los cuales tienen una gran diferencia semántica que puede crear problemas sutiles en las aplicaciones mirando la portabilidad. Sin embargo, `org.hibernate.id.enhanced.SequenceStyleGenerator`, logra la portabilidad de una manera diferente. Escoge entre una tabla o una secuencia en la base de datos para almacenar sus valores en subida, dependiendo de las capacidades del dialecto que se está utilizando. La diferencia enter esto y `native` es que el almacenamiento basado en tablas y secuencias tienen la misma semántica. De hecho, las secuencias son exactamente lo que Hibernate trata de emular con sus generadores basados en tablas. Este generador tiene un número de parámetros de configuración:

- `sequence_name` (opcional, por defecto es `hibernate_sequence`): el nombre de la secuencia o la tabla a utilizar.
- `initial_value` (opcional, por defecto es `1`): el valor inicial a recuperarse de la secuencia/tabla. En términos de creación de secuencias, esto es análogo a la cláusula que usualmente se llama "STARTS WITH".
- `increment_size` (opcional - por defecto es `1`): el valor por el cual las llamadas subsecuentes a la secuencia/tabla deben diferir. En términos de creación de secuencias, esto es análogo a la cláusula que usualmente se llama "INCREMENT BY".
- `force_table_use` (opcional - por defecto es `false`): ¿debemos forzar el uso de una tabla como la estructura de respaldo aunque puede que el dialecto soporte la secuencia?
- `value_column` (opcional - por defecto es `next_val`): solo es relevante para estructuras de tablas, es el nombre de la columna en la tabla, la cual se usa para mantener el valor.
- `optimizer` (optional - defaults to `none`): See *Sección 5.1.2.3.1, “Optimización del generador del identificador”*

El segundo de estos nuevos generadores es `org.hibernate.id.enhanced.TableGenerator`, el cual tiene el propósito, primero, de reemplazar el generador `table`, auqnue de hecho funciona como `org.hibernate.id.MultipleHiLoPerTableGenerator`, y segundo, como una re-implementación de `org.hibernate.id.MultipleHiLoPerTableGenerator` que utiliza la noción de los optimizadores enchufables. Esencialmente, este generador define una tabla capaz de mantener un número de valores de incremento diferentes de manera simultánea usando múltiples filas tecleadas claramente. Este generador tiene un número de parámetros de configuración:

- `table_name` (opcional - por defecto es `hibernate_sequences`): el nombre de la tabla a utilizar.
- `value_column_name` (opcional - por defecto es `next_val`): el nombre de la columna en la tabla que se utiliza para mantener el valor.
- `segment_column_name` (opcional - por defecto es `sequence_name`): el nombre de la columna en la tabla que se utiliza para mantener la "llave segmento". Este es el valor que identifica que valor de incremento utilizar.
- `segment_value` (opcional - por defecto es `default`): El valor "llave segmento" para el segmento desde el cual queremos sacar los valores de incremento para este generador.
- `segment_value_length` (opcional - por defecto es `255`): Se utiliza para la generación de esquemas; el tamaño de la columna a crear esta columna de llave de segmento.
- `initial_value` (opcional - por defecto es `1`): El valor inicial a recuperar de la tabla.
- `increment_size` (opcional - por defecto es `1`): El valor por el cual deben diferir las llamadas subsecuentes a la tabla.
- `optimizer` (optional - defaults to `??`): See *Sección 5.1.2.3.1, "Optimización del generador del identificador"*.

#### 5.1.2.3.1. Optimización del generador del identificador

For identifier generators that store values in the database, it is inefficient for them to hit the database on each and every call to generate a new identifier value. Instead, you can group a bunch of them in memory and only hit the database when you have exhausted your in-memory value group. This is the role of the pluggable optimizers. Currently only the two enhanced generators (*Sección 5.1.2.3, "Generadores mejorados del identificador"* support this operation.

- `none` (generalmente este el es valor predeterminado si no se especifica un optimizador): esto no realizará ninguna optimización y accederá a la base de datos para toda petición.
- `hilo`: aplica un algoritmo hi/lo a los valores recuperados de la base de datos. Se espera que los valores de la base de datos para este optimizador sean secuenciales. Los valores recuperados de la estructura de la base de datos para este optimizador indican el "número del grupo". El `increment_size` se multiplica por ese valor en la memoria para definir un grupo "hi value".
- `pooled`: como en el caso de `hilo`, este optimizador trata de minimizar el número de hits a la base de datos. Sin embargo, aquí simplemente almacenamos el valor inicial para el "siguiente grupo" en la estructura de la base de datos en lugar de un valor secuencial en combinación con un algoritmo de agrupamiento en-memoria. Aquí, `increment_size` ser refiere a los valores que provienen de la base de datos.

### 5.1.2.4. Partial identifier generation

Hibernate supports the automatic generation of some of the identifier properties. Simply use the `@GeneratedValue` annotation on one or several id properties.

**Aviso**

The Hibernate team has always felt such a construct as fundamentally wrong. Try hard to fix your data model before using this feature.

```
@Entity
public class CustomerInventory implements Serializable {
  @Id
  @TableGenerator(name = "inventory",
    table = "U_SEQUENCES",
    pkColumnName = "S_ID",
    valueColumnName = "S_NEXTNUM",
    pkColumnValue = "inventory",
    allocationSize = 1000)
  @GeneratedValue(strategy = GenerationType.TABLE, generator = "inventory")
  Integer id;


  @Id @ManyToOne(cascade = CascadeType.MERGE)
  Customer customer;
}

@Entity
public class Customer implements Serializable {
   @Id
   private int id;
}
```

You can also generate properties inside an `@EmbeddedId` class.

## 5.1.3. Optimistic locking properties (optional)

When using long transactions or conversations that span several database transactions, it is useful to store versioning data to ensure that if the same entity is updated by two conversations, the last to commit changes will be informed and not override the other conversation's work. It guarantees some isolation while still allowing for good scalability and works particularly well in read-often write-sometimes situations.

You can use two approaches: a dedicated version number or a timestamp.

Una propiedad de versión o de sello de fecha nunca debe ser nula para una instancia separada. Hibernate detectará cualquier instancia con una versión o sello de fecha nulo como transitoria, sin importar qué otras estrategias `unsaved-value` se hayan especificado. *El declarar una propiedad de versión o sello de fecha nulable es una forma fácil de evitar cualquier problema con la re-unión transitiva en Hibernate. Es especialmente útil para la gente que utiliza identificadores asignados o claves compuestas*.

### 5.1.3.1. Version number

You can add optimistic locking capability to an entity using the `@Version` annotation:

```
@Entity
public class Flight implements Serializable {
...
    @Version
```

```
    @Column(name="OPTLOCK")
    public Integer getVersion() { ... }
}
```

The version property will be mapped to the `OPTLOCK` column, and the entity manager will use it to detect conflicting updates (preventing lost updates you might otherwise see with the last-commit-wins strategy).

The version column may be a numeric. Hibernate supports any kind of type provided that you define and implement the appropriate `UserVersionType`.

The application must not alter the version number set up by Hibernate in any way. To artificially increase the version number, check in Hibernate Entity Manager's reference documentation `LockModeType.OPTIMISTIC_FORCE_INCREMENT` or `LockModeType.PESSIMISTIC_FORCE_INCREMENT`.

If the version number is generated by the database (via a trigger for example), make sure to use `@org.hibernate.annotations.Generated(GenerationTime.ALWAYS)`.

To declare a version property in hbm.xml, use:

```
<version
        column="version_column"                                    1
        name="propertyName"                                        2
        type="typename"                                            3
        access="field|property|ClassName"                          4
        unsaved-value="null|negative|undefined"                    5
        generated="never|always"                                   6
        insert="true|false"                                        7
        node="element-name|@attribute-name|element/@attribute|."
/>
```

1. `column` (opcional - por defecto es el nombre de la propiedad): El nombre de la columna que tiene el número de la versión.
2. `name`: El nombre de una propiedad de la clase persistente.
3. `type` (opcional - por defecto es `integer`): El tipo del número de la versión.
4. `access` (opcional - por defecto es `property`): La estrategia que Hibernate utiliza para acceder al valor de la propiedad.
5. `unsaved-value` (opcional - por defecto es `undefined`): Un valor de la propiedad de versión que indica que una instancia se encuentra recién instanciada (sin guardar), distinguiéndola de las instancias separadas que se guardaron o se cargaron en una sesión previa. `undefined` especifica que se debe utilizar el valor de la propiedad identificadora.
6. `generated` (opcional - por defecto es `never`): Especifica que este valor de la propiedad de la versión es generado por la base de datos. Vea la discusión de las *propiedades generadas* para obtener mayor información.

7. `insert` (opcional - por defectos es `true`): Especifica si la columna de la versión debe incluirse en las declaraciones de inserción SQL. Se puede configurar como `false` si la columna de la base de datos se define con un valor predeterminado de `0`.

### 5.1.3.2. Timestamp

Alternatively, you can use a timestamp. Timestamps are a less safe implementation of optimistic locking. However, sometimes an application might use the timestamps in other ways as well.

Simply mark a property of type `Date` or `Calendar` as `@Version`.

```
@Entity
public class Flight implements Serializable {
...
    @Version
    public Date getLastUpdate() { ... }
}
```

When using timestamp versioning you can tell Hibernate where to retrieve the timestamp value from - database or JVM - by optionally adding the `@org.hibernate.annotations.Source` annotation to the property. Possible values for the value attribute of the annotation are `org.hibernate.annotations.SourceType.VM` and `org.hibernate.annotations.SourceType.DB`. The default is `SourceType.DB` which is also used in case there is no `@Source` annotation at all.

Like in the case of version numbers, the timestamp can also be generated by the database instead of Hibernate. To do that, use `@org.hibernate.annotations.Generated(GenerationTime.ALWAYS)`.

In hbm.xml, use the `<timestamp>` element:

```
<timestamp
        column="timestamp_column"                                    (1)
        name="propertyName"                                          (2)
        access="field|property|ClassName"                            (3)
        unsaved-value="null|undefined"                               (4)
        source="vm|db"                                               (5)
        generated="never|always"                                     (6)
        node="element-name|@attribute-name|element/@attribute|."
/>
```

1. `column` (opcional - por defecto es el nombre de la propiedad): El nombre de una columna que tiene el sello de fecha.
2. `name`: El nombre de una propiedad del estilo JavaBeans de tipo Java `Date` o `Timestamp` de la clase persistente.

3. `access` (opcional - por defecto es `property`): La estrategia que Hibernate utiliza para acceder al valor de la propiedad.
4. `unsaved-value` (opcional - por defecto es `null`): Un valor de propiedad de versión que indica que una instancia está recién instanciada (sin guardar), distinguiéndola de instancias separadas que hayan sido guardadas o cargadas en una sesión previa. `Undefined` especifica que debe utilizarse el valor de la propiedad identificadora.
5. `source` (opcional - por defecto es `vm`): ¿Desde dónde debe recuperar Hibernate el valor del sello de fecha? ¿Desde la base de datos o desde la MVJ actual? Los sellos de fecha con base en la base de datos provocan un gasto general debido a que Hibernate tiene que llegar hasta la base de datos para poder determinar el "siguiente valor". Es más seguro utilizarlo en entornos con clústers. No todos los `Dialects` soportan la recuperación del sello de fecha actual de la base de datos. Los otros pueden ser poco seguros para utilizarlos como bloqueo debido a la falta de precisión (por ejemplo, Oracle 8).
6. `generated` (opcional - por defecto es `never`): Especifica que este valor de la propiedad del sello de fecha en realidad es generado por la base de datos. Consulte la discusión de las *propiedades generadas* para obtener mayor información.

**Nota**

`<Timestamp>` es equivalente a `<version type="timestamp">`. Y `<timestamp source="db">` es equivalente a `<version type="dbtimestamp">`.

## 5.1.4. Propiedad

You need to decide which property needs to be made persistent in a given entity. This differs slightly between the annotation driven metadata and the hbm.xml files.

### 5.1.4.1. Property mapping with annotations

In the annotations world, every non static non transient property (field or method depending on the access type) of an entity is considered persistent, unless you annotate it as `@Transient`. Not having an annotation for your property is equivalent to the appropriate `@Basic` annotation.

The `@Basic` annotation allows you to declare the fetching strategy for a property. If set to `LAZY`, specifies that this property should be fetched lazily when the instance variable is first accessed. It requires build-time bytecode instrumentation, if your classes are not instrumented, property level lazy loading is silently ignored. The default is `EAGER`. You can also mark a property as not optional thanks to the `@Basic.optional` attribute. This will ensure that the underlying column are not nullable (if possible). Note that a better approach is to use the `@NotNull` annotation of the Bean Validation specification.

Let's look at a few examples:

```
public transient int counter; //transient property
```

```
private String firstname; //persistent property

@Transient
String getLengthInMeter() { ... } //transient property

String getName() {... } // persistent property

@Basic
int getLength() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(EnumType.STRING)
Starred getNote() { ... } //enum persisted as String in database
```

`counter`, a transient field, and `lengthInMeter`, a method annotated as `@Transient`, and will be ignored by the Hibernate. `name`, `length`, and `firstname` properties are mapped persistent and eagerly fetched (the default for simple properties). The `detailedComment` property value will be lazily fetched from the database once a lazy property of the entity is accessed for the first time. Usually you don't need to lazy simple properties (not to be confused with lazy association fetching). The recommended alternative is to use the projection capability of JP-QL (Java Persistence Query Language) or Criteria queries.

JPA support property mapping of all basic types supported by Hibernate (all basic Java types , their respective wrappers and serializable classes). Hibernate Annotations supports out of the box enum type mapping either into a ordinal column (saving the enum ordinal) or a string based column (saving the enum string representation): the persistence representation, defaulted to ordinal, can be overridden through the `@Enumerated` annotation as shown in the `note` property example.

In plain Java APIs, the temporal precision of time is not defined. When dealing with temporal data you might want to describe the expected precision in database. Temporal data can have `DATE`, `TIME`, or `TIMESTAMP` precision (ie the actual date, only the time, or both). Use the `@Temporal` annotation to fine tune that.

`@Lob` indicates that the property should be persisted in a Blob or a Clob depending on the property type: `java.sql.Clob`, `Character[]`, `char[]` and java.lang.`String` will be persisted in a Clob. `java.sql.Blob`, `Byte[]`, `byte[]` and `Serializable` type will be persisted in a Blob.

```
@Lob
public String getFullText() {
    return fullText;
}

@Lob
public byte[] getFullCode() {
    return fullCode;
```

```
}
```

If the property type implements `java.io.Serializable` and is not a basic type, and if the property is not annotated with `@Lob`, then the Hibernate `serializable` type is used.

#### 5.1.4.1.1. Type

You can also manually specify a type using the `@org.hibernate.annotations.Type` and some parameters if needed. `@Type.type` could be:

1. El nombre de un tipo básico de Hibernate: `integer, string, character, date, timestamp, float, binary, serializable, object, blob`, etc.
2. El nombre de una clase Java con un tipo básico predeterminado: `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob`, etc.
3. El nombre de una clase Java serializable.
4. El nombre declase de un tipo personalizado: `com.illflow.type.MyCustomType` etc.

If you do not specify a type, Hibernate will use reflection upon the named property and guess the correct Hibernate type. Hibernate will attempt to interpret the name of the return class of the property getter using, in order, rules 2, 3, and 4.

`@org.hibernate.annotations.TypeDef` and `@org.hibernate.annotations.TypeDefs` allows you to declare type definitions. These annotations can be placed at the class or package level. Note that these definitions are global for the session factory (even when defined at the class level). If the type is used on a single entity, you can place the definition on the entity itself. Otherwise, it is recommended to place the definition at the package level. In the example below, when Hibernate encounters a property of class `PhoneNumer`, it delegates the persistence strategy to the custom mapping type `PhoneNumberType`. However, properties belonging to other classes, too, can delegate their persistence strategy to `PhoneNumberType`, by explicitly using the `@Type` annotation.

**Nota**

Package level annotations are placed in a file named `package-info.java` in the appropriate package. Place your annotations before the package declaration.

```
@TypeDef(
   name = "phoneNumber",
   defaultForType = PhoneNumber.class,
   typeClass = PhoneNumberType.class
)

@Entity
public class ContactDetails {
   [...]
   private PhoneNumber localPhoneNumber;
   @Type(type="phoneNumber")
```

```
   private OverseasPhoneNumber overseasPhoneNumber;
   [...]
}
```

The following example shows the usage of the `parameters` attribute to customize the TypeDef.

```
//in org/hibernate/test/annotations/entity/package-info.java
@TypeDefs(
    {
    @TypeDef(
        name="caster",
        typeClass = CasterStringType.class,
        parameters = {
            @Parameter(name="cast", value="lower")
        }
    )
    }
)
package org.hibernate.test.annotations.entity;

//in org/hibernate/test/annotations/entity/Forest.java
public class Forest {
    @Type(type="caster")
    public String getSmallText() {
    ...
}
```

When using composite user type, you will have to express column definitions. The `@Columns` has been introduced for that purpose.

```
@Type(type="org.hibernate.test.annotations.entity.MonetaryAmountUserType")
@Columns(columns = {
    @Column(name="r_amount"),
    @Column(name="r_currency")
})
public MonetaryAmount getAmount() {
    return amount;
}


public class MonetaryAmount implements Serializable {
    private BigDecimal amount;
    private Currency currency;
    ...
}
```

#### 5.1.4.1.2. Access type

By default the access type of a class hierarchy is defined by the position of the `@Id` or `@EmbeddedId` annotations. If these annotations are on a field, then only fields are considered for persistence and the state is accessed via the field. If there annotations are on a getter, then only the getters

are considered for persistence and the state is accessed via the getter/setter. That works well in practice and is the recommended approach.

### Nota

The placement of annotations within a class hierarchy has to be consistent (either field or on property) to be able to determine the default access type. It is recommended to stick to one single annotation placement strategy throughout your whole application.

However in some situations, you need to:

- force the access type of the entity hierarchy
- override the access type of a specific entity in the class hierarchy
- override the access type of an embeddable type

The best use case is an embeddable class used by several entities that might not use the same access type. In this case it is better to force the access type at the embeddable class level.

To force the access type on a given class, use the `@Access` annotation as showed below:

```
@Entity
public class Order {
   @Id private Long id;
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   @Embedded private Address address;
   public Address getAddress() { return address; }
   public void setAddress() { this.address = address; }
}

@Entity
public class User {
   private Long id;
   @Id public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   private Address address;
   @Embedded public Address getAddress() { return address; }
   public void setAddress() { this.address = address; }
}

@Embeddable
@Access(AcessType.PROPERTY)
public class Address {
   private String street1;
   public String getStreet1() { return street1; }
   public void setStreet1() { this.street1 = street1; }
```

```
   private hashCode; //not persistent
}
```

You can also override the access type of a single property while keeping the other properties standard.

```
@Entity
public class Order {
   @Id private Long id;
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
   @Transient private String userId;
   @Transient private String orderId;

   @Access(AccessType.PROPERTY)
   public String getOrderNumber() { return userId + ":" + orderId; }
   public void setOrderNumber() { this.userId = ...; this.orderId = ...; }
}
```

In this example, the default access type is `FIELD` except for the `orderNumber` property. Note that the corresponding field, if any must be marked as `@Transient` or `transient`.

**@org.hibernate.annotations.AccessType**

The annotation `@org.hibernate.annotations.AccessType` should be considered deprecated for FIELD and PROPERTY access. It is still useful however if you need to use a custom access type.

#### 5.1.4.1.3. Optimistic lock

It is sometimes useful to avoid increasing the version number even if a given property is dirty (particularly collections). You can do that by annotating the property (or collection) with `@OptimisticLock(excluded=true)`.

More formally, specifies that updates to this property do not require acquisition of the optimistic lock.

#### 5.1.4.1.4. Declaring column attributes

The column(s) used for a property mapping can be defined using the `@Column` annotation. Use it to override default values (see the JPA specification for more information on the defaults). You can use this annotation at the property level for properties that are:

- not annotated at all
- annotated with `@Basic`

- annotated with `@Version`
- annotated with `@Lob`
- annotated with `@Temporal`

```
@Entity
public class Flight implements Serializable {
...
@Column(updatable = false, name = "flight_name", nullable = false, length=50)
public String getName() { ... }
```

The `name` property is mapped to the `flight_name` column, which is not nullable, has a length of 50 and is not updatable (making the property immutable).

This annotation can be applied to regular properties as well as `@Id` or `@Version` properties.

```
@Column(
    name="columnName";                                          (1)
    boolean unique() default false;                             (2)
    boolean nullable() default true;                            (3)
    boolean insertable() default true;                          (4)
    boolean updatable() default true;                           (5)
    String columnDefinition() default "";                       (6)
    String table() default "";                                  (7)
    int length() default 255;                                   (8)
    int precision() default 0; // decimal precision             (9)
    int scale() default 0; // decimal scale
```

1. `name` (optional): the column name (default to the property name)
2. `unique` (optional): set a unique constraint on this column or not (default false)
3. `nullable` (optional): set the column as nullable (default true).
4. `insertable` (optional): whether or not the column will be part of the insert statement (default true)
5. `updatable` (optional): whether or not the column will be part of the update statement (default true)
6. `columnDefinition` (optional): override the sql DDL fragment for this particular column (non portable)
7. `table` (optional): define the targeted table (default primary table)
8. `length` (optional): column length (default 255)
8. `precision` (optional): column decimal precision (default 0)
10. `scale` (optional): column decimal scale if useful (default 0)

#### 5.1.4.1.5. Formula

Sometimes, you want the Database to do some computation for you rather than in the JVM, you might also create some kind of virtual column. You can use a SQL fragment (aka formula) instead of mapping a property into a column. This kind of property is read only (its value is calculated by your formula fragment).

```
@Formula("obj_length * obj_height * obj_width")
public long getObjectVolume()
```

The SQL fragment can be as complex as you want and even include subselects.

#### 5.1.4.1.6. Non-annotated property defaults

If a property is not annotated, the following rules apply:

- If the property is of a single type, it is mapped as @Basic
- Otherwise, if the type of the property is annotated as @Embeddable, it is mapped as @Embedded
- Otherwise, if the type of the property is `Serializable`, it is mapped as `@Basic` in a column holding the object in its serialized version
- Otherwise, if the type of the property is `java.sql.Clob` or `java.sql.Blob`, it is mapped as `@Lob` with the appropriate `LobType`

### 5.1.4.2. Property mapping with hbm.xml

El elemento `<property>` declara una propiedad persistente estilo JavaBean de la clase.

```
<property
        name="propertyName"                                          1
        column="column_name"                                         2
        type="typename"                                              3
        update="true|false"                                          4
        insert="true|false"                                          4
        formula="arbitrary SQL expression"                           5
        access="field|property|ClassName"                            6
        lazy="true|false"                                            7
        unique="true|false"                                          8
        not-null="true|false"                                        9
        optimistic-lock="true|false"                                 10
        generated="never|insert|always"                              11
        node="element-name|@attribute-name|element/@attribute|."
        index="index_name"
```

```
        unique_key="unique_key_id"
        length="L"
        precision="P"
        scale="S"
/>
```

1. `name`: el nombre de la propiedad, con la letra inicial en minúscula.
2. `column` (opcional - por defecto es el nombre de la propiedad): El nombre de la columna de la tabla de base de datos mapeada. Esto se puede especificar también con los elemento(s) anidado(s) `<column>`.
3. `type` (opcional): un nombre que indica el tipo de Hibernate.
4. `update, insert` (opcional - por defecto es `true`): Especifica que las columnas mapeadas deben ser incluídas en las declaraciones SQL `UPDATE` y/o `INSERT` . Especificando ambas como `false` permite una propiedad "derivada", cuyo valor se inicia desde alguna otra propiedad que mapee a la misma columna (o columnas) o por un disparador u otra aplicación.
5. `formula` (opcional): una expresión SQL que define el valor para una propiedad *computada*. Las propiedades computadas no tienen una columna mapeada propia.
6. `access` (opcional - por defecto es `property`): La estrategia que Hibernate utiliza para acceder al valor de la propiedad.
7. `lazy` (opcional - por defecto es `false`): Especifica que se debe recuperar perezosamente esta propiedad cuando se acceda por primera vez la variable de instancia. Requiere instrumentación de código byte en tiempo de compilación.
8. `unique` (opcional): Activa la generación DDL de una restricción de unicidad para las columnas. Además, permite que ésta sea el objetivo de una `property-ref`.
9. `not-null` (opcional): Activa la generación DDL de una restricción de nulabilidad para las columnas.
10. `optimistic-lock` (opcional - por defecto es `true`): Especifica que las actualizaciones a esta propiedad requieren o no de la obtención de un bloqueo optimista. En otras palabras, determina si debe ocurrir un incremento de versión cuando la propiedad se encuentre desactualizada.
11. `generated` (opcional - por defecto es `never`): Especifica que este valor de la propiedad es de hecho generado por la base de datos. Consulte discusión sobre las *propiedades generadas* para obtener mayor información.

*escribanombre* puede ser:

1. El nombre de un tipo básico de Hibernate: `integer, string, character, date, timestamp, float, binary, serializable, object, blob`, etc.
2. El nombre de una clase Java con un tipo básico predeterminado: `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob`, etc.
3. El nombre de una clase Java serializable.
4. El nombre declase de un tipo personalizado: `com.illflow.type.MyCustomType` etc.

Si no especifica un tipo, Hibernate utilizará reflección sobre la propiedad mencionada para deducir el tipo Hibernate correcto. Hibernate intentará interpretar el nombre de la clase de

retorno del getter de la propiedad utilizando las reglas 2, 3 y 4 en ese mismo orden. En algunos casos necesitará el atributo `type`. Por ejemplo, para distinguir entre `Hibernate.DATE` y `Hibernate.TIMESTAMP`, o especificar un tipo personalizado.

El atributo `access` le permite controlar el cómo Hibernate accederá a la propiedad en tiempo de ejecución. Por defecto, Hibernate llamará al par de getter/setter de la propiedad. Si usted especifica `access="field"`, Hibernate se saltará el par get/set y accederá al campo directamente utilizando reflección. Puede especificar su propia estrategia de acceso a la propiedad mencionando una clase que implemente la interfaz `org.hibernate.property.PropertyAccessor`.

Una funcionalidad especialmente poderosa son las propiedades derivadas. Estas propiedades son, por definición, de sólo lectura. El valor de la propiedad se computa en tiempo de carga. Usted declara la computación como una expresión SQL y ésta se traduce como una cláusula de subconsulta `SELECT` en la consulta SQL que carga una instancia:

```
<property name="totalPrice"
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
                WHERE li.productId = p.productId
                AND li.customerId = customerId
                AND li.orderNumber = orderNumber )"/>
```

Puede referenciar la tabla de las entidades sin declarar un alias o una columna particular. En el ejemplo dado sería `customerId`. También puede utilizar el elemento anidado de mapeo `<formula>` si no quiere utilizar el atributo.

### 5.1.5. Embedded objects (aka components)

Embeddable objects (or components) are objects whose properties are mapped to the same table as the owning entity's table. Components can, in turn, declare their own properties, components or collections

It is possible to declare an embedded component inside an entity and even override its column mapping. Component classes have to be annotated at the class level with the `@Embeddable` annotation. It is possible to override the column mapping of an embedded object for a particular entity using the `@Embedded` and `@AttributeOverride` annotation in the associated property:

```
@Entity
public class Person implements Serializable {

    // Persistent component using defaults
    Address homeAddress;

    @Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="iso2", column = @Column(name="bornIso2") ),
            @AttributeOverride(name="name", column = @Column(name="bornCountryName") )
    } )
```

```
    Country bornIn;
    ...
}
```

```
@Embeddable
public class Address implements Serializable {
    String city;
    Country nationality; //no overriding here
}
```

```
@Embeddable
public class Country implements Serializable {
    private String iso2;
    @Column(name="countryName") private String name;

    public String getIso2() { return iso2; }
    public void setIso2(String iso2) { this.iso2 = iso2; }


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

An embeddable object inherits the access type of its owning entity (note that you can override that using the `@Access` annotation).

The `Person` entity has two component properties, `homeAddress` and `bornIn`. `homeAddress` property has not been annotated, but Hibernate will guess that it is a persistent component by looking for the `@Embeddable` annotation in the Address class. We also override the mapping of a column name (to `bornCountryName`) with the `@Embedded` and `@AttributeOverride` annotations for each mapped attribute of `Country`. As you can see, `Country` is also a nested component of `Address`, again using auto-detection by Hibernate and JPA defaults. Overriding columns of embedded objects of embedded objects is through dotted expressions.

```
@Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="city", column = @Column(name="fld_city") ),
            @AttributeOverride(name="nationality.iso2", column = @Column(name="nat_Iso2") ),
          @AttributeOverride(name="nationality.name", column = @Column(name="nat_CountryName") )
            //nationality columns in homeAddress are overridden
    } )
    Address homeAddress;
```

Hibernate Annotations supports something that is not explicitly supported by the JPA specification. You can annotate a embedded object with the `@MappedSuperclass` annotation to make the superclass properties persistent (see `@MappedSuperclass` for more informations).

You can also use association annotations in an embeddable object (ie `@OneToOne`, `@ManyToOne`, `@OneToMany` or `@ManyToMany`). To override the association columns you can use `@AssociationOverride`.

If you want to have the same embeddable object type twice in the same entity, the column name defaulting will not work as several embedded objects would share the same set of columns. In plain JPA, you need to override at least one set of columns. Hibernate, however, allows you to enhance the default naming mechanism through the `NamingStrategy` interface. You can write a strategy that prevent name clashing in such a situation. `DefaultComponentSafeNamingStrategy` is an example of this.

If a property of the embedded object points back to the owning entity, annotate it with the `@Parent` annotation. Hibernate will make sure this property is properly loaded with the entity reference.

In XML, use the `<component>` element.

```
<component
        name="propertyName"                                    1
        class="className"                                      2
        insert="true|false"                                    3
        update="true|false"                                    4
        access="field|property|ClassName"                      5
        lazy="true|false"                                      6
        optimistic-lock="true|false"                           7
        unique="true|false"                                    8
        node="element-name|."
>

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

1. `name`: El nombre de la propiedad.
2. `class` (opcional - por defecto es el tipo de la propiedad determinado por reflección): El nombre de la clase del componente (hijo).
3. `insert`: ¿Las columnas mapeadas aparacen en `INSERT`s SQL?
4. `update`: ¿Las columnas mapeadas aparacen en `UPDATE`s SQL?
5. `access` (opcional - por defecto es `property`): La estrategia que Hibernate utiliza para acceder al valor de la propiedad.
6. `lazy` (opcional - por defecto es `false`): Especifica que este componente debe ser recuperado perezosamente cuando se acceda a la variable de instancia por primera vez. Requiere instrumentación de código byte en tiempo de compilación.

7. `optimistic-lock` (opcional - por defecto es `true`): Especifica que las actualizaciones de este componente requieren o no la adquisición de un bloqueo optimista. Determina si debe ocurrir un incremento de versión cuando esta propiedad se encuentra desactualizada.
8. `unique` (opcional - por defecto es `false`): Especifica que existe una restricción de unicidad sobre todas las columnas mapeadas del componente.

Las etiquetas hijas `<property>` mapean propiedades de la clase hija a las columnas de la tabla.

El elemento `<component>` permite un subelemento `<parent>` que mapea una propiedad de la clase del componente como una referencia a la entidad contenedora.

The `<dynamic-component>` element allows a `Map` to be mapped as a component, where the property names refer to keys of the map. See *Sección 9.5, "Componentes dinámicos"* for more information. This feature is not supported in annotations.

## 5.1.6. Inheritance strategy

Java is a language supporting polymorphism: a class can inherit from another. Several strategies are possible to persist a class hierarchy:

- Single table per class hierarchy strategy: a single table hosts all the instances of a class hierarchy
- Joined subclass strategy: one table per class and subclass is present and each table persist the properties specific to a given subclass. The state of the entity is then stored in its corresponding class table and all its superclasses
- Table per class strategy: one table per concrete class and subclass is present and each table persist the properties of the class and its superclasses. The state of the entity is then stored entirely in the dedicated table for its class.

### 5.1.6.1. Single table per class hierarchy strategy

With this approach the properties of all the subclasses in a given mapped class hierarchy are stored in a single table.

Each subclass declares its own persistent properties and subclasses. Version and id properties are assumed to be inherited from the root class. Each subclass in a hierarchy must define a unique discriminator value. If this is not specified, the fully qualified Java class name is used.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
```

```
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

In hbm.xml, for the table-per-class-hierarchy mapping strategy, the `<subclass>` declaration is used. For example:

```
<subclass
        name="ClassName"                                    (1)
        discriminator-value="discriminator_value"           (2)
        proxy="ProxyInterface"                              (3)
        lazy="true|false"                                   (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        entity-name="EntityName"
        node="element-name"
        extends="SuperclassName">

        <property .... />
        .....
</subclass>
```

1. `name`: El nombre de clase completamente calificado de la subclase.
2. `discriminator-value` (opcional - por defecto es el nombre de la clase): Un valor que distingue subclases individuales.
3. `proxy` (opcional): Especifica una clase o interfaz que se utiliza para proxies de inicialización perezosa.
4. `lazy` (opcional, por defecto es `true`): El establecer `lazy="false"` desactiva el uso de la recuperación perezosa.

For information about inheritance mappings see *Capítulo 10, Mapeo de herencias*.

#### 5.1.6.1.1. Discriminador

Discriminators are required for polymorphic persistence using the table-per-class-hierarchy mapping strategy. It declares a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. Hibernate Core supports the follwoing restricted set of types as discriminator column: `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

Use the `@DiscriminatorColumn` to define the discriminator column as well as the discriminator type.

**Nota**

The enum `DiscriminatorType` used in `javax.persitence.DiscriminatorColumn` only contains the values `STRING`,

> `CHAR` and `INTEGER` which means that not all Hibernate supported types are available via the `@DiscriminatorColumn` annotation.

You can also use `@DiscriminatorFormula` to express in SQL a virtual discriminator column. This is particularly useful when the discriminator value can be extracted from one or more columns of the table. Both `@DiscriminatorColumn` and `@DiscriminatorFormula` are to be set on the root entity (once per persisted hierarchy).

`@org.hibernate.annotations.DiscriminatorOptions` allows to optionally specify Hibernate specific discriminator options which are not standardized in JPA. The available options are `force` and `insert`. The `force` attribute is useful if the table contains rows with "extra" discriminator values that are not mapped to a persistent class. This could for example occur when working with a legacy database. If `force` is set to `true` Hibernate will specify the allowed discriminator values in the `SELECT` query, even when retrieving all instances of the root class. The second option - `insert` - tells Hibernate whether or not to include the discriminator column in SQL `INSERT`s. Usually the column should be part of the `INSERT` statement, but if your discriminator column is also part of a mapped composite identifier you have to set this option to `false`.

### Sugerencia

There is also a `@org.hibernate.annotations.ForceDiscriminator` annotation which is deprecated since version 3.6. Use `@DiscriminatorOptions` instead.

Finally, use `@DiscriminatorValue` on each class of the hierarchy to specify the value stored in the discriminator column for a given entity. If you do not set `@DiscriminatorValue` on a class, the fully qualified class name is used.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

In hbm.xml, the `<discriminator>` element is used to define the discriminator column or formula:

```
<discriminator
        column="discriminator_column"                    1
        type="discriminator_type"                        2
```

```
        force="true|false"                          (3)
        insert="true|false"                         (4)
        formula="arbitrary sql expression"          (5)
/>
```

(1) `column` (opcional - por defecto es `class`) el nombre de la columna discriminadora.

(2) `type` (opcional - por defecto es `string`) un nombre que indica el tipo Hibernate.

(3) `force` (opcional - por defecto es `false`) "fuerza" a Hibernate para especificar los valores discriminadores permitidos incluso cuando se recuperan todas las instancias de la clase raíz.

(4) `insert` (opcional - por defecto es `true`): establecido como `false` si su columna discriminadora también es parte de un identificador mapeado compuesto. Lle dice a Hibernate que no incluya la columna en los SQLs `INSERT`.

(5) `formula` (opcional): una expresión SQL arbitraria que se ejecuta cuando se tenga que evaluar un tipo. Permite la discriminación con base en el contenido.

Los valores reales de la columna discriminadora están especificados por el atributo `discriminator-value` de los elementos `<class>` y `<subclass>`.

El atributo `formula` le permite declarar una expresión SQL arbitraria que será utilizada para evaluar el tipo de una fila. Por ejemplo:

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
    type="integer"/>
```

### 5.1.6.2. Joined subclass strategy

Each subclass can also be mapped to its own table. This is called the table-per-subclass mapping strategy. An inherited state is retrieved by joining with the table of the superclass. A discriminator column is not required for this mapping strategy. Each subclass must, however, declare a table column holding the object identifier. The primary key of this table is also a foreign key to the superclass table and described by the `@PrimaryKeyJoinColumn`s or the `<key>` element.

```
@Entity @Table(name="CATS")
@Inheritance(strategy=InheritanceType.JOINED)
public class Cat implements Serializable {
    @Id @GeneratedValue(generator="cat-uuid")
    @GenericGenerator(name="cat-uuid", strategy="uuid")
    String getId() { return id; }

    ...
}

@Entity @Table(name="DOMESTIC_CATS")
@PrimaryKeyJoinColumn(name="CAT")
public class DomesticCat extends Cat {
    public String getName() { return name; }
```

```
}
```

**Nota**

The table name still defaults to the non qualified class name. Also if `@PrimaryKeyJoinColumn` is not set, the primary key / foreign key columns are assumed to have the same names as the primary key columns of the primary table of the superclass.

In hbm.xml, use the `<joined-subclass>` element. For example:

```
<joined-subclass
        name="ClassName"                                    (1)
        table="tablename"                                   (2)
        proxy="ProxyInterface"                              (3)
        lazy="true|false"                                   (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

1. `name`: El nombre de clase completamente calificado de la subclase.
2. `table`: El nombre de tabla de la subclase.
3. `proxy` (opcional): Especifica una clase o interfaz que se debe utilizar para proxies de inicialización perezosa.
4. `lazy` (opcional, por defecto es `true`): El establecer `lazy="false"` desactiva el uso de la recuperación perezosa.

Use the `<key>` element to declare the primary key / foreign key column. The mapping at the start of the chapter would then be re-written as:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
                    <key column="CAT"/>
                    <property name="name" type="string"/>
                </joined-subclass>
        </class>

        <class name="eg.Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

For information about inheritance mappings see *Capítulo 10, Mapeo de herencias*.

### 5.1.6.3. Table per class strategy

A third option is to map only the concrete classes of an inheritance hierarchy to tables. This is called the table-per-concrete-class strategy. Each table defines all persistent states of the class, including the inherited state. In Hibernate, it is not necessary to explicitly map such inheritance hierarchies. You can map each class as a separate entity root. However, if you wish use polymorphic associations (e.g. an association to the superclass of your hierarchy), you need to use the union subclass mapping.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Flight implements Serializable { ... }
```

Or in hbm.xml:

```
<union-subclass
        name="ClassName"                                    1
        table="tablename"                                   2
        proxy="ProxyInterface"                              3
        lazy="true|false"                                   4
```

```
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        abstract="true|false"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <property .... />
        .....
</union-subclass>
```

1. `name`: El nombre de clase completamente calificado de la subclase.
2. `table`: El nombre de tabla de la subclase.
3. `proxy` (opcional): Especifica una clase o interfaz que se debe utilizar para proxies de inicialización perezosa.
4. `lazy` (opcional, por defecto es `true`): El establecer `lazy="false"` desactiva el uso de la recuperación perezosa.

No se necesita una columna o una columna clave discriminadora para esta estrategia de mapeo.

For information about inheritance mappings see *Capítulo 10, Mapeo de herencias*.

### 5.1.6.4. Inherit properties from superclasses

This is sometimes useful to share common properties through a technical or a business superclass without including it as a regular mapped entity (ie no specific table for this entity). For that purpose you can map them as `@MappedSuperclass`.

```
@MappedSuperclass
public class BaseEntity {
    @Basic
    @Temporal(TemporalType.TIMESTAMP)
    public Date getLastUpdate() { ... }
    public String getLastUpdater() { ... }
    ...
}

@Entity class Order extends BaseEntity {
    @Id public Integer getId() { ... }
    ...
}
```

In database, this hierarchy will be represented as an `Order` table having the `id`, `lastUpdate` and `lastUpdater` columns. The embedded superclass property mappings are copied into their entity subclasses. Remember that the embeddable superclass is not the root of the hierarchy though.

### Nota

Properties from superclasses not mapped as `@MappedSuperclass` are ignored.

### Nota

The default access type (field or methods) is used, unless you use the `@Access` annotation.

### Nota

The same notion can be applied to `@Embeddable` objects to persist properties from their superclasses. You also need to use `@MappedSuperclass` to do that (this should not be considered as a standard EJB3 feature though)

### Nota

It is allowed to mark a class as `@MappedSuperclass` in the middle of the mapped inheritance hierarchy.

### Nota

Any class in the hierarchy non annotated with `@MappedSuperclass` nor `@Entity` will be ignored.

You can override columns defined in entity superclasses at the root entity level using the `@AttributeOverride` annotation.

```
@MappedSuperclass
public class FlyingObject implements Serializable {

    public int getAltitude() {
        return altitude;
    }

    @Transient
    public int getMetricAltitude() {
        return metricAltitude;
    }

    @ManyToOne
    public PropulsionType getPropulsion() {
```

```
        return metricAltitude;
    }
    ...
}

@Entity
@AttributeOverride( name="altitude", column = @Column(name="fld_altitude") )
@AssociationOverride(
   name="propulsion",
   joinColumns = @JoinColumn(name="fld_propulsion_fk")
)
public class Plane extends FlyingObject {
    ...
}
```

The `altitude` property will be persisted in an `fld_altitude` column of table `Plane` and the propulsion association will be materialized in a `fld_propulsion_fk` foreign key column.

You can define `@AttributeOverride`(s) and `@AssociationOverride`(s) on `@Entity` classes, `@MappedSuperclass` classes and properties pointing to an `@Embeddable` object.

In hbm.xml, simply map the properties of the superclass in the `<class>` element of the entity that needs to inherit them.

### 5.1.6.5. Mapping one entity to several tables

While not recommended for a fresh schema, some legacy databases force your to map a single entity on several tables.

Using the `@SecondaryTable` or `@SecondaryTables` class level annotations. To express that a column is in a particular table, use the `table` parameter of `@Column` or `@JoinColumn`.

```
@Entity
@Table(name="MainCat")
@SecondaryTables({
    @SecondaryTable(name="Cat1", pkJoinColumns={
        @PrimaryKeyJoinColumn(name="cat_id", referencedColumnName="id")
    ),
    @SecondaryTable(name="Cat2", uniqueConstraints={@UniqueConstraint(columnNames={"storyPart2"})})
})
public class Cat implements Serializable {

    private Integer id;
    private String name;
    private String storyPart1;
    private String storyPart2;

    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
```

```
    }

    @Column(table="Cat1")
    public String getStoryPart1() {
        return storyPart1;
    }

    @Column(table="Cat2")
    public String getStoryPart2() {
        return storyPart2;
    }
}
```

In this example, `name` will be in `MainCat`. `storyPart1` will be in `Cat1` and `storyPart2` will be in `Cat2`. `Cat1` will be joined to `MainCat` using the `cat_id` as a foreign key, and `Cat2` using `id` (ie the same column name, the `MainCat` id column has). Plus a unique constraint on `storyPart2` has been set.

There is also additional tuning accessible via the `@org.hibernate.annotations.Table` annotation:

- `fetch`: If set to JOIN, the default, Hibernate will use an inner join to retrieve a secondary table defined by a class or its superclasses and an outer join for a secondary table defined by a subclass. If set to `SELECT` then Hibernate will use a sequential select for a secondary table defined on a subclass, which will be issued only if a row turns out to represent an instance of the subclass. Inner joins will still be used to retrieve a secondary defined by the class and its superclasses.
- `inverse`: If true, Hibernate will not try to insert or update the properties defined by this join. Default to false.
- `optional`: If enabled (the default), Hibernate will insert a row only if the properties defined by this join are non-null and will always use an outer join to retrieve the properties.
- `foreignKey`: defines the Foreign Key name of a secondary table pointing back to the primary table.

Make sure to use the secondary table name in the `appliesto` property

```
@Entity
@Table(name="MainCat")
@SecondaryTable(name="Cat1")
@org.hibernate.annotations.Table(
   appliesTo="Cat1",
   fetch=FetchMode.SELECT,
   optional=true)
public class Cat implements Serializable {

    private Integer id;
    private String name;
    private String storyPart1;
    private String storyPart2;
```

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

    public String getName() {
        return name;
    }

    @Column(table="Cat1")
    public String getStoryPart1() {
        return storyPart1;
    }

    @Column(table="Cat2")
    public String getStoryPart2() {
        return storyPart2;
    }
}
```

In hbm.xml, use the `<join>` element.

```
<join
        table="tablename"                    (1)
        schema="owner"                       (2)
        catalog="catalog"                    (3)
        fetch="join|select"                  (4)
        inverse="true|false"                 (5)
        optional="true|false">               (6)

        <key ... />

        <property ... />
        ...
</join>
```

1. `table`: El nombre de la tabla unida.
2. `schema` (opcional): Sobrescribe el nombre del esquema especificado por el elemento raíz `<hibernate-mapping>`.
3. `catalog` (opcional): Sobrescribe el nombre del catálogo especificado por el elemento raíz `<hibernate-mapping>`.
4. `fetch` (opcional - por defecto es `join`): Si se establece como `join`, por defecto, Hibernate utilizará una unión interior (inner join) para recuperar un `<join>` definido por una clase o sus superclases. Utilizará una unión externa (outer join) para un `<join>` definido por una subclase. Si se establece como `select`, entonces Hibernate utilizará una selección secuencial para un `<join>` definido en una subclase. Esto se publicará sólamente si una

fila representa una instancia de la subclase. Las uniones interiores todavía serán utilizadas para recuperar un `<join>` definido por la clase y sus superclases.

5. `inverse` (opcional - por defecto es `false`): De activarse, Hibernate no tratará de insertar o actualizar las propiedades definidas por esta unión.
6. `optional` (opcional - por defecto es `false`): De activarse, Hibernate insertará una fila sólo si las propiedades definidas por esta unión son no-nulas. Siempre utilizará una unión externa para recuperar las propiedades.

Por ejemplo, la información domiciliaria de una persona se puede mapear a una tabla separada, preservando a la vez la semántica de tipo de valor para todas las propiedades:

```
<class name="Person"
    table="PERSON">

    <id name="id" column="PERSON_ID">...</id>

    <join table="ADDRESS">
        <key column="ADDRESS_ID"/>
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </join>
    ...
```

Con frecuencia, esta funcionalidad sólamente es útil para los modelos de datos heredados. Recomendamos menos tablas que clases y un modelo de dominio más detallado. Sin embargo, es útil para cambiar entre estrategias de mapeo de herencias en una misma jerarquía, como se explica más adelante.

## 5.1.7. Mapping one to one and one to many associations

To link one entity to an other, you need to map the association property as a to one association. In the relational model, you can either use a foreign key or an association table, or (a bit less common) share the same primary key value between the two entities.

To mark an association, use either `@ManyToOne` or `@OnetoOne`.

`@ManyToOne` and `@OneToOne` have a parameter named `targetEntity` which describes the target entity name. You usually don't need this parameter since the default value (the type of the property that stores the association) is good in almost all cases. However this is useful when you want to use interfaces as the return type instead of the regular entity.

Setting a value of the `cascade` attribute to any meaningful value other than nothing will propagate certain operations to the associated object. The meaningful values are divided into three categories.

1. basic operations, which include: `persist`, `merge`, `delete`, `save-update`, `evict`, `replicate`, `lock` and `refresh`;

2. special values: `delete-orphan` or `all` ;
3. comma-separated combinations of operation names: `cascade="persist,merge,evict"` or `cascade="all,delete-orphan"`. See *Sección 11.11, "Persistencia transitiva"* for a full explanation. Note that single valued many-to-one associations do not support orphan delete.

By default, single point associations are eagerly fetched in JPA 2. You can mark it as lazily fetched by using `@ManyToOne(fetch=FetchType.LAZY)` in which case Hibernate will proxy the association and load it when the state of the associated entity is reached. You can force Hibernate not to use a proxy by using `@LazyToOne(NO_PROXY)`. In this case, the property is fetched lazily when the instance variable is first accessed. This requires build-time bytecode instrumentation. lazy="false" specifies that the association will always be eagerly fetched.

With the default JPA options, single-ended associations are loaded with a subsequent select if set to `LAZY`, or a SQL JOIN is used for `EAGER` associations. You can however adjust the fetching strategy, ie how data is fetched by using `@Fetch`. `FetchMode` can be `SELECT` (a select is triggered when the association needs to be loaded) or `JOIN` (use a SQL JOIN to load the association while loading the owner entity). `JOIN` overrides any lazy attribute (an association loaded through a `JOIN` strategy cannot be lazy).

### 5.1.7.1. Using a foreign key or an association table

An ordinary association to another persistent class is declared using a

- `@ManyToOne` if several entities can point to the the target entity
- `@OneToOne` if only a single entity can point to the the target entity

and a foreign key in one table is referencing the primary key column(s) of the target table.

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}
```

The `@JoinColumn` attribute is optional, the default value(s) is the concatenation of the name of the relationship in the owner side, _ (underscore), and the name of the primary key column in the owned side. In this example `company_id` because the property name is `company` and the column id of Company is `id`.

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE}, targetEntity=CompanyImpl.class )
    @JoinColumn(name="COMP_ID")
```

```
    public Company getCompany() {
        return company;
    }
    ...
}

public interface Company {
    ...
}
```

You can also map a to one association through an association table. This association table described by the `@JoinTable` annotation will contains a foreign key referencing back the entity table (through `@JoinTable.joinColumns`) and a a foreign key referencing the target entity table (through `@JoinTable.inverseJoinColumns`).

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinTable(name="Flight_Company",
        joinColumns = @JoinColumn(name="FLIGHT_ID"),
        inverseJoinColumns = @JoinColumn(name="COMP_ID")
    )
    public Company getCompany() {
        return company;
    }
    ...
}
```

### Nota

You can use a SQL fragment to simulate a physical join column using the `@JoinColumnOrFormula` / `@JoinColumnOrformulas` annotations (just like you can use a SQL fragment to simulate a property column via the `@Formula` annotation).

```
@Entity
public class Ticket implements Serializable {
    @ManyToOne
    @JoinColumnOrFormula(formula="(firstname + ' ' + lastname)")
    public Person getOwner() {
        return person;
    }
    ...
}
```

You can mark an association as mandatory by using the `optional=false` attribute. We recommend to use Bean Validation's `@NotNull` annotation as a better alternative however. As a consequence, the foreign key column(s) will be marked as not nullable (if possible).

When Hibernate cannot resolve the association because the expected associated element is not in database (wrong id on the association column), an exception is raised. This might be inconvenient for legacy and badly maintained schemas. You can ask Hibernate to ignore such elements instead of raising an exception using the `@NotFound` annotation.

### Ejemplo 5.1. @NotFound annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @NotFound(action=NotFoundAction.IGNORE)
    public Parent getParent() { ... }
    ...
}
```

Sometimes you want to delegate to your database the deletion of cascade when a given entity is deleted. In this case Hibernate generates a cascade delete constraint at the database level.

### Ejemplo 5.2. @OnDelete annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @OnDelete(action=OnDeleteAction.CASCADE)
    public Parent getParent() { ... }
    ...
}
```

Foreign key constraints, while generated by Hibernate, have a fairly unreadable name. You can override the constraint name using `@ForeignKey`.

### Ejemplo 5.3. @ForeignKey annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @ForeignKey(name="FK_PARENT")
    public Parent getParent() { ... }
    ...
}

alter table Child add constraint FK_PARENT foreign key (parent_id) references Parent
```

Sometimes, you want to link one entity to an other not by the target entity primary key but by a different unique key. You can achieve that by referencing the unique key column(s) in `@JoinColumn.referenceColumnName`.

```
@Entity
class Person {
   @Id Integer personNumber;
   String firstName;
   @Column(name="I")
   String initial;
   String lastName;
}

@Entity
class Home {
   @ManyToOne
   @JoinColumns({
      @JoinColumn(name="first_name", referencedColumnName="firstName"),
      @JoinColumn(name="init", referencedColumnName="I"),
      @JoinColumn(name="last_name", referencedColumnName="lastName"),
   })
   Person owner
}
```

This is not encouraged however and should be reserved to legacy mappings.

In hbm.xml, mapping an association is similar. The main difference is that a `@OneToOne` is mapped as `<many-to-one unique="true"/>`, let's dive into the subject.

```
<many-to-one
        name="propertyName"                                          1
        column="column_name"                                         2
        class="ClassName"                                            3
        cascade="cascade_style"                                      4
        fetch="join|select"                                          5
        update="true|false"                                          6
        insert="true|false"                                          6
        property-ref="propertyNameFromAssociatedClass"               7
        access="field|property|ClassName"                            8
        unique="true|false"                                          9
        not-null="true|false"                                        10
        optimistic-lock="true|false"                                 11
        lazy="proxy|no-proxy|false"                                  12
        not-found="ignore|exception"                                 13
        entity-name="EntityName"                                     14
        formula="arbitrary SQL expression"                           15
```

```
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        index="index_name"
        unique_key="unique_key_id"
        foreign-key="foreign_key_name"
/>
```

1. `name`: El nombre de la propiedad.
2. `column` (opcional): El nombre de la columna de la clave foránea. Esto también se puede especificar por medio de uno o varios elementos anidados `<column>`.
3. `class` (opcional - por defecto es el tipo de la propiedad determinado por reflección): El nombre de la clase asociada.
4. `cascade` (opcional) especifica qué operaciones deben ir en cascada desde el objeto padre hasta el objeto asociado.
5. `fetch` (opcional - por defecto es `select`): Escoge entre la recuperación de unión exterior (outer-join) o la recuperación por selección secuencial.
6. `update, insert` (opcional - por defecto es `true`) especifica que las columnas mapeadas deben ser incluídas en las declaraciones SQL `UPDATE` y/o `INSERT`. El establecer ambas como `false` permite una asociación puramente "derivada" cuyo valor es inicializado desde alguna otra propiedad que mapea a la misma columna (o columnas), por un disparador o por otra aplicación.
7. `property-ref`: (opcional): El nombre de una propiedad de la clase asociada que se encuentra unida a su llave foránea. Si no se especifica, se utiliza la llave principal de la clase asociada.
8. `access` (opcional - por defecto es `property`): La estrategia que Hibernate utiliza para acceder al valor de la propiedad.
9. `unique` (opcional): Activa la generación DDL de una restricción de unicidad para la columna de clave foránea. Además, permite que éste sea el objetivo de una `property-ref`. puede hacer que la asociación sea de multiplicidad uno-a-uno.
10. `not-null` (opcional): Activa la generación DDL de una restricción de nulabilidad para las columnas de clave foránea.
11. `optimistic-lock` (opcional - por defecto es `true`): Especifica que las actualizaciones a esta propiedad requieren o no de la obtención de un bloqueo optimista. En otras palabras, determina si debe ocurrir un incremento de versión cuando la propiedad se encuentre desactualizada.
12. `lazy` (opcional - por defecto es `proxy`): Por defecto, las asociaciones de punto único van con proxies. `lazy="no-proxy"` especifica que esta propiedad debe ser recuperada perezosamente cuando se acceda por primera vez a la variable de instancia. Requiere instrumentación del código byte en tiempo de compilación. `lazy="false"` especifica que la asociación siempre será recuperada tempranamente.
13. `not-found` (opcional - por defecto es `exception`): Especifica cómo se manejarán las claves foráneas que referencian las filas que hacen falta. `ignore` tratará una fila perdida como una asociación nula.
14. `entity-name` (opcional): El nombre de entidad de la clase asociada.

15. `formula` (opcional): una expresión SQL que define el valor para una clave foránea *computada*.

Setting a value of the `cascade` attribute to any meaningful value other than `none` will propagate certain operations to the associated object. The meaningful values are divided into three categories. First, basic operations, which include: `persist, merge, delete, save-update, evict, replicate, lock and refresh`; second, special values: `delete-orphan`; and third,`all` comma-separated combinations of operation names: `cascade="persist,merge,evict"` or `cascade="all,delete-orphan"`. See *Sección 11.11, "Persistencia transitiva"* for a full explanation. Note that single valued, many-to-one and one-to-one, associations do not support orphan delete.

Este es un ejemplo de una declaración típica `muchos-a-uno`:

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

El atributo `property-ref` se debe utilizar sólamente para el mapeo de datos heredados donde una clave foránea referencia una clave única de la tabla asociada, distinta de la clave principal. Este es un modelo relacional complicado y confuso. Por ejemplo, si la clase `Product` tuviera un número único serial que no es la clave principal, el atributo `unique` controla la generación de DDL de Hibernate con la herramienta SchemaExport.

```
<property name="serialNumber" unique="true" type="string" column="SERIAL_NUMBER"/>
```

Entonces el mapeo para `OrderItem` puede utilizar:

```
<many-to-one name="product" property-ref="serialNumber" column="PRODUCT_SERIAL_NUMBER"/>
```

Sin embargo, esto ciertamente no se aconseja.

Si la clave única referenciada abarca múltiples propiedades de la entidad asociada, debe mapear las propiedades dentro de un elemento nombrado `<properties>`.

Si la clave única referenciada es propiedad de un componente, usted puede especificar una ruta de propiedad:

```
<many-to-one name="owner" property-ref="identity.ssn" column="OWNER_SSN"/>
```

### 5.1.7.2. Sharing the primary key with the associated entity

The second approach is to ensure an entity and its associated entity share the same primary key. In this case the primary key column is also a foreign key and there is no extra column. These associations are always one to one.

#### Ejemplo 5.4. One to One association

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

    @OneToOne(cascade = CascadeType.ALL)
    @MapsId
    public Heart getHeart() {
        return heart;
    }
    ...
}

@Entity
public class Heart {
    @Id
    public Long getId() { ...}
}
```

**Nota**

Many people got confused by these primary key based one to one associations. They can only be lazily loaded if Hibernate knows that the other side of the association is always present. To indicate to Hibernate that it is the case, use `@OneToOne(optional=false)`.

In hbm.xml, use the following mapping.

```
<one-to-one
        name="propertyName"                                          1
        class="ClassName"                                            2
        cascade="cascade_style"                                      3
        constrained="true|false"                                     4
        fetch="join|select"                                          5
        property-ref="propertyNameFromAssociatedClass"               6
        access="field|property|ClassName"                            7
        formula="any SQL expression"                                 8
```

```
        lazy="proxy|no-proxy|false"                                    (9)
        entity-name="EntityName"                                       (10)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        foreign-key="foreign_key_name"
/>
```

1. `name`: El nombre de la propiedad.
2. `class` (opcional - por defecto es el tipo de la propiedad determinado por reflección): El nombre de la clase asociada.
3. `cascade` (opcional) especifica qué operaciones deben ir en cascada desde el objeto padre hasta el objeto asociado.
4. `constrained` (opcional): especifica que una restricción de clave foránea en la clave principal de la tabla mapeada referencia la tabla de la clase asociada. Esta opción afecta el orden en que van en la cascada `save()` y `delete()` y determina si la asociación puede ser virtualizada por proxies. La herramienta de exportación de esquemas también lo utiliza.
5. `fetch` (opcional - por defecto es `select`): Escoge entre la recuperación de unión exterior (outer-join) o la recuperación por selección secuencial.
6. `property-ref` (opcional): El nombre de una propiedad de la clase asociada que esté unida a la clave principal de esta clase. Si no se especifica, se utiliza la clave principal de la clase asociada.
7. `access` (opcional - por defecto es `property`): La estrategia que Hibernate utiliza para acceder al valor de la propiedad.
8. `formula` (opcional): Casi todas las asociaciones uno-a-uno mapean a la clave principal de la entidad propietaria. Si este no es el caso, puede especificar otra columna, o columnas, o una expresión para unir utilizando una fórmula SQL. Para un obtener un ejemplo consulte `org.hibernate.test.onetooneformula`.
9. `lazy` (opcional - por defecto es `proxy`): Por defecto, las asociaciones de punto único van con proxies. `lazy="no-proxy"` especifica que esta propiedad debe ser traída perezosamente cuando se acceda por primera vez la variable de instancia. Requiere instrumentación del código byte en tiempo de compilación. `lazy="false"` especifica que la asociación siempre será recuperada tempranamente. *Observe que si `constrained="false"`, la aplicación de proxies es imposible e Hibernate recuperará tempranamente la asociación*.
10. `entity-name` (opcional): El nombre de entidad de la clase asociada.

Las asociaciones de claves principales no necesitan una columna extra de la tabla. Si dos filas están relacionadas por la asociación entonces las dos filas de tablas comparten el mismo valor de clave principal. Para que dos objetos estén relacionados por una asociación de clave principal, asegúrese de que se les asigne el mismo valor de identificador.

Para una asociación de clave principal, agregue los siguientes mapeos a `Employee` y `Person` respectivamente:

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

Asegúrese de que las claves principales de las filas relacionadas en las tablas PERSON y EMPLOYEE sean iguales. Utilizamos una estrategia especial de generación de identificador de Hibernate denominada `foreign`:

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

A una instancia recién guardada de `Person` se le asigna el mismo valor de clave principal que se le asignó a la instancia `Employee` referida por la propiedad `employee` de esa `Person`.

### 5.1.8. Natural-id

Although we recommend the use of surrogate keys as primary keys, you should try to identify natural keys for all entities. A natural key is a property or combination of properties that is unique and non-null. It is also immutable. Map the properties of the natural key as `@NaturalId` or map them inside the `<natural-id>` element. Hibernate will generate the necessary unique key and nullability constraints and, as a result, your mapping will be more self-documenting.

```
@Entity
public class Citizen {
    @Id
    @GeneratedValue
    private Integer id;
    private String firstname;
    private String lastname;

    @NaturalId
    @ManyToOne
    private State state;

    @NaturalId
    private String ssn;
    ...
}



//and later on query
List results = s.createCriteria( Citizen.class )
```

```
                .add( Restrictions.naturalId().set( "ssn", "1234" ).set( "state", ste ) )
                .list();
```

Or in XML,

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
</natural-id>
```

Le recomendamos bastante que implemente `equals()` y `hashCode()` para comparar las propiedades de clave natural de la entidad.

Este mapeo no está concebido para la utilización con entidades que tienen claves principales naturales.

- `mutable` (opcional - por defecto es `false`): Por defecto, se asume que las propiedades de identificadores naturales son inmutables (constantes).

### 5.1.9. Any

There is one more type of property mapping. The `@Any` mapping defines a polymorphic association to classes from multiple tables. This type of mapping requires more than one column. The first column contains the type of the associated entity. The remaining columns contain the identifier. It is impossible to specify a foreign key constraint for this kind of association. This is not the usual way of mapping polymorphic associations and you should use this only in special cases. For example, for audit logs, user session data, etc.

The `@Any` annotation describes the column holding the metadata information. To link the value of the metadata information and an actual entity type, The `@AnyDef` and `@AnyDefs` annotations are used. The `metaType` attribute allows the application to specify a custom type that maps database column values to persistent classes that have identifier properties of the type specified by `idType`. You must specify the mapping from values of the `metaType` to class names.

```
@Any( metaColumn = @Column( name = "property_type" ), fetch=FetchType.EAGER )
@AnyMetaDef(
    idType = "integer",
    metaType = "string",
    metaValues = {
        @MetaValue( value = "S", targetEntity = StringProperty.class ),
        @MetaValue( value = "I", targetEntity = IntegerProperty.class )
    } )
@JoinColumn( name = "property_id" )
public Property getMainProperty() {
    return mainProperty;
}
```

Note that `@AnyDef` can be mutualized and reused. It is recommended to place it as a package metadata in this case.

```
//on a package
@AnyMetaDef( name="property"
    idType = "integer",
    metaType = "string",
    metaValues = {
        @MetaValue( value = "S", targetEntity = StringProperty.class ),
        @MetaValue( value = "I", targetEntity = IntegerProperty.class )
    } )
package org.hibernate.test.annotations.any;


//in a class
    @Any( metaDef="property", metaColumn = @Column( name = "property_type" ), fetch=FetchType.EAGER )
    @JoinColumn( name = "property_id" )
    public Property getMainProperty() {
        return mainProperty;
    }
```

The hbm.xml equivalent is:

```
<any name="being" id-type="long" meta-type="string">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

### Nota

You cannot mutualize the metadata in hbm.xml as you can in annotations.

```
<any
        name="propertyName"                          (1)
        id-type="idtypename"                         (2)
        meta-type="metatypename"                     (3)
        cascade="cascade_style"                      (4)
        access="field|property|ClassName"            (5)
        optimistic-lock="true|false"                 (6)
>
        <meta-value ... />
        <meta-value ... />
        .....
```

```
        <column .... />
        <column .... />
        .....
</any>
```

1. `name`: el nombre de la propiedad.
2. `id-type`: el tipo del identificador.
3. `meta-type` (opcional - por defecto es `string`): Cualquier tipo que se permita para un mapeo discriminador.
4. `cascade` (opcional- por defecto es `none`): el estilo de cascada.
5. `access` (opcional - por defecto es `property`): La estrategia que Hibernate utiliza para acceder al valor de la propiedad.
6. `optimistic-lock` (opcional - por defecto es `true`): Especifica si las actualizaciones de esta propiedad requieren o no de la adquisición del bloqueo optimista. Define si debe ocurrir un incremento de versión cuando esta propiedad está desactualizada.

## 5.1.10. Propiedades

El elemento `<properties>` permite la definición de un grupo de propiedades lógico con nombre de una clase. El uso más importante de la contrucción es que permite que una combinación de propiedades sea el objetivo de una `property-ref`. También es una forma práctica de definir una restricción de unicidad multicolumna. Por ejemplo:

```
<properties
        name="logicalName"                          (1)
        insert="true|false"                         (2)
        update="true|false"                         (3)
        optimistic-lock="true|false"                (4)
        unique="true|false"                         (5)
>

        <property ...../>
        <many-to-one .... />
        ........
</properties>
```

1. `name`: El nombre lógico del agrupamiento. *No* es un nombre de propiedad.
2. `insert`: ¿Las columnas mapeadas aparacen en `INSERT`s SQL?
3. `update`: ¿Las columnas mapeadas aparacen en `UPDATE`s SQL?
4. `optimistic-lock` (opcional - por defecto es `true`): Especifica que las actualizaciones de estas propiedades requieren o no de la adquisición de un bloqueo optimista. Determina si debe ocurrir un incremento de versión cuando estas propiedades están desactualizadas.
5. `unique` (opcional - por defecto es `false`): Especifica que existe una restricción de unicidad sobre todas las columnas mapeadas del componente.

Por ejemplo, si tenemos el siguiente mapeo de `<properties>`:

```
<class name="Person">
    <id name="personNumber"/>

    ...
    <properties name="name"
            unique="true" update="false">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </properties>
</class>
```

Puede que tenga alguna asociación de datos heredados que se refiera a esta clave única de la tabla de `Person`, en lugar de la clave principal:

```
<many-to-one name="owner"
         class="Person" property-ref="name">
    <column name="firstName"/>
    <column name="initial"/>
    <column name="lastName"/>
</many-to-one>
```

### Nota

When using annotations as a mapping strategy, such construct is not necessary as the binding between a column and its related column on the associated table is done directly

```
@Entity
class Person {
   @Id Integer personNumber;
   String firstName;
   @Column(name="I")
   String initial;
   String lastName;
}

@Entity
class Home {
   @ManyToOne
   @JoinColumns({
      @JoinColumn(name="first_name", referencedColumnName="firstName"),
      @JoinColumn(name="init", referencedColumnName="I"),
      @JoinColumn(name="last_name", referencedColumnName="lastName"),
   })
   Person owner
```

```
    }
```

No recomendamos el uso de este tipo de cosas fuera del contexto del mapeo de datos heredados.

## 5.1.11. Some hbm.xml specificities

The hbm.xml structure has some specificities naturally not present when using annotations, let's describe them briefly.

### 5.1.11.1. Doctype

Todos los mapeos XML deben declarar el tipo de documento que se muestra. El DTD en sí se puede encontrar en la URL mencionada anteriormente, en el directorio `hibernate-x.x.x/src/org/hibernate` , o en `hibernate3.jar`. Hibernate siempre buscará el DTD primero en la ruta de clase. Si el DTD realiza búsquedas utilizando una conexión de Internet, verifique que su declaración DTD frente al contenido de su ruta de clase.

#### 5.1.11.1.1. EntityResolver

Hibernate tratará primero de resolver los DTDs en su ruta de clase. La manera en que lo hace es registrando una implementación `org.xml.sax.EntityResolver` personalizada con el SAXReader que utiliza para leer los archivos xml. Este `EntityResolver` personalizado reconoce dos diferentes espacios de nombre del identificador del sistema.

- a `hibernate namespace` is recognized whenever the resolver encounters a systemId starting with `http://www.hibernate.org/dtd/`. The resolver attempts to resolve these entities via the classloader which loaded the Hibernate classes.
- un `user namespace` se reconoce cuando el resolvedor se encuentra con un identificador del sistema utilizando un protocolo URL `classpath://`. El resolvedor intentará resolver estas entidades por medio de (1) el cargador de clase del contexto del hilo actual y (2) el cargador de clase, el cual cargó las clases de Hibernate.

Este es un ejemplo de la utilización de los espacios de nombre del usuario:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" [
    <!ENTITY types SYSTEM "classpath://your/domain/types.xml">
]>

<hibernate-mapping package="your.domain">
    <class name="MyEntity">
        <id name="id" type="my-custom-id-type">
            ...
        </id>
```

```
    <class>
    &types;
</hibernate-mapping>
```

En donde `types.xml` es un recurso en el paquete `your.domain` y comprende un *typedef* personalizado.

### 5.1.11.2. Mapeo de Hibernate

Este elemento tiene varios atributos opcionales. Los atributos `schema` y `catalog` especifican que las tablas a las que se refiere en este mapeo pertenecen al esquema y/o catálogo mencionado(s). De especificarse, los nombres de tablas serán calificados por el nombre del esquema y del catálogo dados. De omitirse, los nombres de las tablas no serán calificados. El atributo `default-cascade` especifica qué estilo de cascada se debe asumir para las propiedades y colecciones que no especifican un atributo `cascade`. Por defecto, el atributo `auto-import` nos permite utilizar nombres de clase sin calificar en el lenguaje de consulta.

```
<hibernate-mapping
         schema="schemaName"                                    (1)
         catalog="catalogName"                                  (2)
         default-cascade="cascade_style"                        (3)
         default-access="field|property|ClassName"              (4)
         default-lazy="true|false"                              (5)
         auto-import="true|false"                               (6)
         package="package.name"                                 (7)
 />
```

1. `schema` (opcional): El nombre de un esquema de la base de datos.
2. `catalog` (opcional): El nombre de un catálogo de la base de datos.
3. `default-cascade` (opcional - por defecto es `none`): Un estilo de cascada por defecto.
4. `default-access` (opcional - por defecto es `property`): La estrategia que Hibernate debe utilizar para acceder a todas las propiedades. Puede ser una implementación personalizada de `PropertyAccessor`.
5. `default-lazy` (opcional - por defecto es `true`): El valor por defecto para los atributos `lazy` no especificados de mapeos de clase y de colección.
6. `auto-import` (opcional - por defecto es `true`): Especifica si podemos utilizar nombres de clases no calificados de clases en este mapeo en el lenguaje de consulta.
7. `package` (opcional): Especifica un prefijo de paquete que se debe utilizar para los nombres de clase no calificados en el documento de mapeo.

Si tiene dos clases persistentes con el mismo nombre (sin calificar), debe establecer `auto-import="false"`. Se presentará una excepción si usted intenta asignar dos clases al mismo nombre "importado".

El elemento `hibernate-mapping` le permite anidar varios mapeos `<class>` persistentes, como se mostró anteriormente. Sin embargo, es una buena práctica (y algunas herramientas esperan) que mapee sólamente una clase persistente, o a una sóla jerarquía de clases, en un archivo de mapeo y nombrarlo como la superclase persistente. Por ejemplo, `Cat.hbm.xml`, `Dog.hbm.xml`, o si utiliza herencia, `Animal.hbm.xml`.

### 5.1.11.3. Key

The `<key>` element is featured a few times within this guide. It appears anywhere the parent mapping element defines a join to a new table that references the primary key of the original table. It also defines the foreign key in the joined table:

```
<key
        column="columnname"                    (1)
        on-delete="noaction|cascade"           (2)
        property-ref="propertyName"            (3)
        not-null="true|false"                  (4)
        update="true|false"                    (5)
        unique="true|false"                    (6)
/>
```

1. `column` (opcional): El nombre de la columna de la clave foránea. Esto también se puede especificar por medio de uno o varios elementos anidados `<column>`.
2. `on-delete` (opcional - por defecto es `noaction`): Especifica si la restricción de clave foránea tiene el borrado en cascada activado a nivel de base de datos.
3. `property-ref` (opcional): Especifica que la clave foránea referencia columnas que no son la clave principal de la tabla original. Se proporciona para los datos heredados.
4. `not-null` (opcional): Especifica que las columnas de la clave foránea son no nulables. Esto se implica cuando la clave foránea también es parte de la clave principal.
5. `update` (opcional): Especifica que la clave foránea nunca se debe actualizar. Esto se implica cuando la clave foránea también es parte de la clave principal.
6. `unique` (opcional): Especifica que la clave foránea debe tener una restricción de . Esto se implica cuando la clave foránea también es la clave principal.

Para los sistemas en donde el rendimiento es importante, todas las claves deben ser definidas `on-delete="cascade"`. Hibernate utiliza una restricción `ON CASCADE DELETE` a nivel de base de datos, en vez de muchas declaraciones `DELETE` individuales. Tenga en cuenta que esta funcionalidad evita la estrategia de bloqueo optimista normal de Hibernate para datos versionados.

Los atributos `not-null` y `update` son útiles al mapear una asociación uno a muchos unidireccional. Si mapea una unidireccional uno a muchos a una clave foránea no nulable, *tiene* que declarar la columna clave utilizando `<key not-null="true">`.

### 5.1.11.4. Import

Si su aplicación tiene dos clases persistentes con el mismo nombre y no quiere especificar el nombre del paquete completamenta calificado en las consultas Hibernate, las clases pueden ser "importadas" explícitamente, en lugar de depender de `auto-import="true"`. Incluso puede importar clases e interfaces que no estén mapeadas explícitamente:

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="ClassName"                                    (1)
        rename="ShortName"                                   (2)
/>
```

(1) `class`: El nombre de clase completamente calificado de cualquier clase Java.

(2) `rename` (opcional - por defecto es el nombre de clase sin calificar): Un nombre que se puede utilizar en el lenguaje de consulta.

**Nota**

This feature is unique to hbm.xml and is not supported in annotations.

### 5.1.11.5. Los elementos columna y fórmula

Los elementos de mapeo que acepten un atributo `column` aceptarán opcionalmente un subelemento `<column>`. De manera similar, `<formula>` es una alternativa al atributo `formula`. Por ejemplo:

```
<column
        name="column_name"
        length="N"
        precision="N"
        scale="N"
        not-null="true|false"
        unique="true|false"
        unique-key="multicolumn_unique_key_name"
        index="index_name"
        sql-type="sql_type_name"
        check="SQL expression"
        default="SQL expression"
        read="SQL expression"
        write="SQL expression"/>
```

```
<formula>SQL expression</formula>
```

La mayoría de los atributos en `column` proporcionan una manera de personalizar el DDL durante la generación del esquema automático. Los atributos `read` y `write` le permiten especificar SQL personalizado que Hibernate utilizará para acceder el valor de la columna. Para obtener mayor información sobre esto, consulte la discusión sobre *expresiones de lectura y escritura de columnas*.

Los elementos `column` y `formula` incluso se pueden combinar dentro del mismo mapeo de propiedad o asociación para expresar, por ejemplo, condiciones de unión exóticas.

```
<many-to-one name="homeAddress" class="Address"
        insert="false" update="false">
    <column name="person_id" not-null="true" length="10"/>
    <formula>'MAILING'</formula>
</many-to-one>
```

# 5.2. Tipos de Hibernate

## 5.2.1. Entidades y Valores

En relación con el servicio de persistencia, los objetos a nivel de lenguaje Java se clasifican en dos grupos:

Una *entidad* existe independientemente de cualquier otro objeto que referencie a la entidad. Compare esto con el modelo habitual de Java en donde un objeto no referenciado es recolectado como basura. Las entidades deben ser guardadas y borradas explícitamente. Sin embargo, los grabados y borrados se pueden *tratar en cascada* desde una entidad padre a sus hijos. Esto es diferente al modelo de persistencia de objetos por alcance (ODMG) y corresponde más a cómo se utilizan habitualmente los objetos de aplicación en sistemas grandes. Las entidades soportan referencias circulares y compartidas, que también pueden ser versionadas.

El estado persistente de una entidad consta de las referencias a otras entidades e instancias de tipo *valor*. Los valores son primitivos: colecciones (no lo que está dentro de la colección), componentes y ciertos objetos inmutables. A diferencia de las entidades, los valores en particular las colecciones y los componentes, *son* persistidos y borrados por alcance. Como los objetos valor y primitivos son persistidos y borrados junto con sus entidades contenedoras, no se pueden versionar independientemente. Los valores no tienen identidad independiente, por lo que dos entidades o colleciones no los pueden compartir.

Hasta ahora, hemos estado utilizando el término "clase persistente" para referirnos a entidades. Continuaremos haciéndolo así. Sin embargo, no todas la clases con estado persistente definidas por el usuario son entidades. Un *componente* es una clase definida por el usuario con semántica de valor. Una propiedad Java de tipo `java.lang.String` también tiene semántica de valor.

Dada esta definición, podemos decir que todos los tipo (clases) provistos por el JDK tienen una semántica de tipo valor en Java, mientras que los tipos definidos por el usuario se pueden mapear con semántica de tipo valor o de entidad. La desición corre por cuenta del desarrollador de la aplicación. Una clase entidad en un modelo de dominio son las referencias compartidas a una sola instancia de esa clase, mientras que la composición o agregación usualmente se traducen a un tipo de valor.

Volveremos a revisar ambos conceptos a lo largo de este manual de referencia.

EL desafío es mapear el sistema de tipos de Java ( la definición de entidades y tipos de valor de los desarrolladores al sistema de tipos de SQL/la base de datos. El puente entre ambos sistemas lo brinda Hibernate. Para las entidades utilizamos `<class>`, `<subclass>`, etc. Para los tipos de valor utilizamos `<property>`, `<component>`, etc, usualmente con un atributo `type`. El valor de este atributo es el nombre de un *tipo de mapeo* de Hibernate. Hibernate proporciona un rango de mapeos para tipos de valores del JDK estándar. Puede escribir sus propios mapeos de tipo e implementar sus estrategias de conversión personalizadas.

Todos los tipos incorporados de Hibernate soportan la semántica de nulos, a excepción de las colecciones.

### 5.2.2. Tipos de valores básicos

Los *tipos de mapeo básicos* incorporados se pueden categorizar así:

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`
: Mapeos de tipos de primitivos de Java o de clases de envoltura a los tipos de columna SQL (específica del vendedor). `boolean, yes_no` y `true_false` son codificaciones alternativas a `boolean` de Java o `java.lang.Boolean`.

`string`
: Un mapeo del tipo `java.lang.String` a `VARCHAR` (u Oracle `VAARCHAR2`).

`date, time, timestamp`
: Mapeos de tipo desde `java.util.Date` y sus subclases a tipos SQL `DATE`, `TIME` y `TIMESTAMP` (o equivalente).

`calendar, calendar_date`
: Mapeos de tipo desde `java.util.Date` y tipos SQL `TIMESTAMP` y `DATE` (o equivalente).

`big_decimal, big_integer`
: Mapeos de tipo desde `java.math.BigDecimal` y `java.math.BigInteger` a `NUMERIC` (o `NUMBER` de Oracle).

`locale, timezone, currency`
: Mapeos de tipo desde `java.util.Locale`, `java.util.TimeZone` y `java.util.Currency` a `VARCHAR` (o `VARCHAR2` de Oracle). Las instancias de `Locale` y `Currency` son mapeadas a sus códigos ISO. Las instancias de `TimeZone` son mapeadas a sus `ID`.

`class`
: Un mapeo de tipo `java.lang.Class` a `VARCHAR` (o `VARCHAR2` de Oracle). Una `Class` es mapeada a su nombre completamente calificado.

`binary`
: Mapea arreglos de bytes a un tipo binario SQL apropiado.

`text`
: Mapea cadenas largas de Java al tipo SQL `CLOB` o `TEXT`.

`serializable`
: Mapea tipos serializables Java a un tipo binario SQL apropiado. También puede indicar el tipo `serializable` de Hibernate con el nombre de una clase o interfaz serializable Java que no sea por defecto un tipo básico.

`clob, blob`
: Mapeos de tipo para las clases JDBC `java.sql.Clob` y `java.sql.Blob`. Estos tipos pueden ser inconvenientes para algunas aplicaciones, pues el objeto blob o clob no pueden ser reusados fuera de una transacción. Además, el soporte del controlador suele ser malo e inconsistente.

`imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date, imm_serializable, imm_binary`
: Los mapeos de tipo para lo que usualmente se considera tipos Java mutables. Aquí es donde Hibernate realiza ciertas optimizaciones apropiadas sólamente para tipos Java inmutables y la aplicación trata el objeto como inmutable. Por ejemplo, no debe llamar `Date.setTime()` para una instancia mapeada como `imm_timestamp`. Para cambiar el valor de la propiedad y hacer que ese cambio sea persistente, la aplicación tiene que asignar un objeto nuevo, no idéntico, a la propiedad.

Los identificadores únicos de entidades y colecciones pueden ser de cualquier tipo básico excepto `binary`, `blob` y `clob`. Los identificadores compuestos también están permitidos, a continuación encontrará mayor información.

Los tipos de valor básicos tienen sus constantes `Type` correspondientes definidas en `org.hibernate.Hibernate`. Por ejemplo, `Hibernate.STRING` representa el tipo `string`.

## 5.2.3. Tipos de valor personalizados

Es relativamente fácil para los desarrolladores crear sus propios tipos de valor. Por ejemplo, puede que quiera persistir propiedades del tipo `java.lang.BigInteger` a columnas `VARCHAR`. Hibernate no provee un tipo incorporado para esto. Los tipos personalizados no están limitados a mapear una propiedad o elemento de colección a una sola columna de tabla. Así, por ejemplo, podría tener una propiedad Java `getName()`/`setName()` de tipo `java.lang.String` que es persistida a las columnas `FIRST_NAME`, `INITIAL`, `SURNAME`.

Para implementar un tipo personalizado, implemente `org.hibernate.UserType` o `org.hibernate.CompositeUserType` y declare las propiedades utilizando el nombre de clase

completamente calificado del tipo. Revise `org.hibernate.test.DoubleStringType` para ver qué clases de cosas son posibles.

```
<property name="twoStrings" type="org.hibernate.test.DoubleStringType">
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

Observe el uso de etiquetas `<column>` para mapear una propiedad a múltiples columnas.

Las interfaces `CompositeUserType`, `EnhancedUserType`, `UserCollectionType`, y `UserVersionType` brindan soporte para usos más especializados.

Incluso usted puede proporcionar parámetros a un `UserType` en el archivo de mapeo. Para hacer esto, su `UserType` tiene que implementar la interfaz `org.hibernate.usertype.ParameterizedType`. Para brindar parámetros a su tipo personalizado, puede utilizar el elemento `<type>` en sus archivos de mapeo.

```
<property name="priority">
    <type name="com.mycompany.usertypes.DefaultValueIntegerType">
        <param name="default">0</param>
    </type>
</property>
```

Ahora el `UserType` puede recuperar el valor del parámetro denominado `default` del objeto `Properties` que se le pasa.

Si utiliza cierto `UserType` muy frecuentemente, puede ser útil el definir un nombre más corto para este. Puede hacer esto utilizando el elemento `<typedef>`. Los typedefs asignan un nombre a un tipo personalizado y también pueden contener una lista de valores predeterminados de parámetros si el tipo se encuentra parametrizado.

```
<typedef class="com.mycompany.usertypes.DefaultValueIntegerType" name="default_zero">
    <param name="default">0</param>
</typedef>
```

```
<property name="priority" type="default_zero"/>
```

También es posible sobrescribir los parámetros provistos en un typedef sobre una base de caso por caso utilizando parámetros de tipo en el mapeo de la propiedad.

Aunque el amplio espectro de tipos incorporados y de soporte para los componentes de Hibernate significa que necesitará usar un tipo personalizado muy raramente, se considera como una buena práctica el utilizar tipos personalizados para clases no-entidades que aparezcan frecuentemente

en su aplicación. Por ejemplo, una clase `MonetaryAmount` es una buena candidata para un `CompositeUserType`, incluso cuando puede ser fácilmente mapeada como un componente. Un razón para esto es la abstracción. Con un tipo personalizado, sus documentos de mapeo estarán protegidos contra posibles cambios futuros en la forma de representar valores monetarios.

## 5.3. Mapeo de una clase más de una vez

Es posible proporcionar más de un mapeo para una clase persistente en particular. En este caso usted debe especificar un *nombre de entidad* para aclarar entre las instancias de las dos entidades mapeadas. Por defecto, el nombre de la entidad es el mismo que el nombre de la clase. Hibernate le deja especificar el nombre de entidad al trabajar con objetos persistentes, al escribir consultas, o al mapear asociaciones a la entidad mencionada.

```
<class name="Contract" table="Contracts"
        entity-name="CurrentContract">
    ...
    <set name="history" inverse="true"
            order-by="effectiveEndDate desc">
        <key column="currentContractId"/>
        <one-to-many entity-name="HistoricalContract"/>
    </set>
</class>

<class name="Contract" table="ContractHistory"
        entity-name="HistoricalContract">
    ...
    <many-to-one name="currentContract"
            column="currentContractId"
            entity-name="CurrentContract"/>
</class>
```

Las asociaciones ahora se especifican utilizando `entity-name` en lugar de `class`.

**Nota**

This feature is not supported in Annotations

## 5.4. Identificadores SQL en comillas

Puede forzar a Hibernate a que utilice comillas con un identificador en el SQL generado encerrando el nombre de tabla o de columna entre comillas sencillas en el documento de mapeo. Hibernate utilizará el estilo de comillas para el `Dialect` SQL. Usualmente comillas dobles, a excepción de corchetes para SQL Server y comillas sencillas para MySQL.

```
@Entity @Table(name="`Line Item`")
class LineItem {
   @id @Column(name="`Item Id`") Integer id;
```

```
   @Column(name="`Item #`") int itemNumber
}

<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

## 5.5. Propiedades generadas

Las propiedades generadas son propiedades cuyos valores son generados por la base de datos. Usualmente, las aplicaciones de Hibernate necesitaban `refrescar` los objetos que contenian cualquier propiedad para la cual la base de datos generará valores. Sin embargo, el marcar propiedades como generadas deja que la aplicación delegue esta responsabilidad a Hibernate. Cuando Hibernate emite un INSERT or UPDATE SQL para una entidad la cual ha definido propiedades generadas, inmediatamente emite un select para recuperar los valores generados.

Las propiedades marcadas como generadas tienen que ser además no insertables y no actualizables. Sólamente las *versiones*, *sellos de fecha*, y *propiedades simples* se pueden marcar como generadas.

`never` (por defecto): el valor dado de la propiedad no es generado dentro de la base de datos.

`insert`: el valor dado de la propiedad es generado en insert, pero no es regenerado en las actualizaciones posteriores. Las propiedades como fecha-creada (created-date) se encuentran dentro de esta categoría. Aunque las propiedades *versión* y *sello de fecha* se pueden marcar como generadas, esta opción no se encuentra disponible.

`always`: el valor de la propiedad es generado tanto en insert como en update.

To mark a property as generated, use `@Generated`.

## 5.6. Column transformers: read and write expressions

Hibernate allows you to customize the SQL it uses to read and write the values of columns mapped to *simple properties*. For example, if your database provides a set of data encryption functions, you can invoke them for individual columns like this:

```
@Entity
class CreditCard {
   @Column(name="credit_card_num")
   @ColumnTransformer(
      read="decrypt(credit_card_num)",
      write="encrypt(?)")
   public String getCreditCardNumber() { return creditCardNumber; }
   public void setCreditCardNumber(String number) { this.creditCardNumber = number; }
   private String creditCardNumber;
}
```

or in XML

```
<property name="creditCardNumber">
        <column
          name="credit_card_num"
          read="decrypt(credit_card_num)"
          write="encrypt(?)"/>
</property>
```

**Nota**

You can use the plural form `@ColumnTransformers` if more than one columns need to define either of these rules.

If a property uses more that one column, you must use the `forColumn` attribute to specify which column, the expressions are targeting.

```
@Entity
class User {
   @Type(type="com.acme.type.CreditCardType")
   @Columns( {
      @Column(name="credit_card_num"),
      @Column(name="exp_date") } )
   @ColumnTransformer(
      forColumn="credit_card_num",
      read="decrypt(credit_card_num)",
      write="encrypt(?)")
   public CreditCard getCreditCard() { return creditCard; }
   public void setCreditCard(CreditCard card) { this.creditCard = card; }
   private CreditCard creditCard;
}
```

Hibernate aplica las expresiones personalizadas de manera automática cuando la propiedad se referencia en una petición. Esta funcionalidad es similar a una propiedad derivada `formula` con dos diferencias:

- Esta propiedad está respaldada por una o más columnas que se exportan como parte de la generación automática del esquema.
- La propiedad es de lectura y escritura no de sólo lectura.

Si se especifica la expresión `write` debe contener exactamente un parámetro de sustitución '?' para el valor.

## 5.7. Objetos de bases de datos auxiliares

Los objetos de bases de datos auxiliares permiten la creación - CREATE - y eliminación - DROP - de objetos de bases de datos arbitrarios. Junto con las herramientas de evolución del esquema

de Hibernate, tienen la habilidad de definir de manera completa el esquema de un usuario dentro de los archivos de mapeo de Hibernate. Aunque están diseñados específicamente para crear y eliminar cosas como disparadores - triggers- o procedimientos almacenados, realmente cualquier comando SQL se puede ejecutar por medio de un método `java.sql.Statement.execute()` aquí es válido (por ejemplo, ALTERs, INSERTS, etc). Básicamente, hay dos modos para definir objetos de bases de datos auxiliares:

El primer modo es para numerar explícitamente los comandos CREATE y DROP en el archivo de mapeo:

```
<hibernate-mapping>
    ...
    <database-object>
        <create>CREATE TRIGGER my_trigger ...</create>
        <drop>DROP TRIGGER my_trigger</drop>
    </database-object>
</hibernate-mapping>
```

El segundo modo es para proporcionar una clase personalizada que construye los comandos CREATE y DROP. Esta clase personalizada tiene que implementar la interfaz `org.hibernate.mapping.AuxiliaryDatabaseObject`.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
    </database-object>
</hibernate-mapping>
```

Adicionalmente, estos objetos de la base de datos se pueden incluir de manera opcional de forma que aplique sólamente cuando se utilicen ciertos dialectos.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="org.hibernate.dialect.Oracle9iDialect"/>
        <dialect-scope name="org.hibernate.dialect.Oracle10gDialect"/>
    </database-object>
</hibernate-mapping>
```

**Nota**

This feature is not supported in Annotations

# Types

As an Object/Relational Mapping solution, Hibernate deals with both the Java and JDBC representations of application data. An online catalog application, for example, most likely has `Product` object with a number of attributes such as a `sku`, `name`, etc. For these individual attributes, Hibernate must be able to read the values out of the database and write them back. This 'marshalling' is the function of a *Hibernate type*, which is an implementation of the `org.hibernate.type.Type` interface. In addition, a *Hibernate type* describes various aspects of behavior of the Java type such as "how is equality checked?" or "how are values cloned?".

**Importante**

A Hibernate type is neither a Java type nor a SQL datatype; it provides a information about both.

When you encounter the term *type* in regards to Hibernate be aware that usage might refer to the Java type, the SQL/JDBC type or the Hibernate type.

Hibernate categorizes types into two high-level groups: value types (see *Sección 6.1, "Value types"*) and entity types (see *Sección 6.2, "Entity types"*).

## 6.1. Value types

The main distinguishing characteristic of a value type is the fact that they do not define their own lifecycle. We say that they are "owned" by something else (specifically an entity, as we will see later) which defines their lifecycle. Value types are further classified into 3 sub-categories: basic types (see *Sección 6.1.1, "Basic value types"*), composite types (see *Sección 6.1.2, "Composite types"*) amd collection types (see *Sección 6.1.3, "Collection types"*).

### 6.1.1. Basic value types

The norm for basic value types is that they map a single database value (column) to a single, non-aggregated Java type. Hibernate provides a number of built-in basic types, which we will present in the following sections by the Java type. Mainly these follow the natural mappings recommended in the JDBC specification. We will later cover how to override these mapping and how to provide and use alternative type mappings.

#### 6.1.1.1. java.lang.String

`org.hibernate.type.StringType`

Maps a string to the JDBC VARCHAR type. This is the standard mapping for a string if no Hibernate type is specified.

Registered under `string` and `java.lang.String` in the type registry (see *Sección 6.5, "Type registry"*).

`org.hibernate.type.MaterializedClob`
Maps a string to a JDBC CLOB type

Registered under `materialized_clob` in the type registry (see *Sección 6.5, "Type registry"*).

`org.hibernate.type.TextType`
Maps a string to a JDBC LONGVARCHAR type

Registered under `text` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.2. `java.lang.Character` (or char primitive)

`org.hibernate.type.CharacterType`
Maps a char or `java.lang.Character` to a JDBC CHAR

Registered under `char` and `java.lang.Character` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.3. `java.lang.Boolean` (or boolean primitive)

`org.hibernate.type.BooleanType`
Maps a boolean to a JDBC BIT type

Registered under `boolean` and `java.lang.Boolean` in the type registry (see *Sección 6.5, "Type registry"*).

`org.hibernate.type.NumericBooleanType`
Maps a boolean to a JDBC INTEGER type as 0 = false, 1 = true

Registered under `numeric_boolean` in the type registry (see *Sección 6.5, "Type registry"*).

`org.hibernate.type.YesNoType`
Maps a boolean to a JDBC CHAR type as ('N' | 'n') = false, ( 'Y' | 'y' ) = true

Registered under `yes_no` in the type registry (see *Sección 6.5, "Type registry"*).

`org.hibernate.type.TrueFalseType`
Maps a boolean to a JDBC CHAR type as ('F' | 'f') = false, ( 'T' | 't' ) = true

Registered under `true_false` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.4. `java.lang.Byte` (or byte primitive)

`org.hibernate.type.ByteType`
Maps a byte or `java.lang.Byte` to a JDBC TINYINT

Registered under `byte` and `java.lang.Byte` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.5. `java.lang.Short` (or short primitive)

`org.hibernate.type.ShortType`
Maps a short or `java.lang.Short` to a JDBC SMALLINT

Registered under `short` and `java.lang.Short` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.6. `java.lang.Integer` (or int primitive)

`org.hibernate.type.IntegerTypes`
Maps an int or `java.lang.Integer` to a JDBC INTEGER

Registered under `int` and `java.lang.Integer`in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.7. `java.lang.Long` (or long primitive)

`org.hibernate.type.LongType`
Maps a long or `java.lang.Long` to a JDBC BIGINT

Registered under `long` and `java.lang.Long` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.8. `java.lang.Float` (or float primitive)

`org.hibernate.type.FloatType`
Maps a float or `java.lang.Float` to a JDBC FLOAT

Registered under `float` and `java.lang.Float` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.9. `java.lang.Double` (or double primitive)

`org.hibernate.type.DoubleType`
Maps a double or `java.lang.Double` to a JDBC DOUBLE

Registered under `double` and `java.lang.Double` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.10. `java.math.BigInteger`

`org.hibernate.type.BigIntegerType`
Maps a `java.math.BigInteger` to a JDBC NUMERIC

Registered under `big_integer` and `java.math.BigInteger` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.11. `java.math.BigDecimal`

`org.hibernate.type.BigDecimalType`
Maps a `java.math.BigDecimal` to a JDBC NUMERIC

Registered under `big_decimal` and `java.math.BigDecimal` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.12. `java.util.Date` or `java.sql.Timestamp`

`org.hibernate.type.TimestampType`
Maps a `java.sql.Timestamp` to a JDBC TIMESTAMP

Registered under `timestamp`, `java.sql.Timestamp` and `java.util.Date` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.13. `java.sql.Time`

`org.hibernate.type.TimeType`
Maps a `java.sql.Time` to a JDBC TIME

Registered under `time` and `java.sql.Time` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.14. `java.sql.Date`

`org.hibernate.type.DateType`
Maps a `java.sql.Date` to a JDBC DATE

Registered under `date` and `java.sql.Date` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.15. `java.util.Calendar`

`org.hibernate.type.CalendarType`
Maps a `java.util.Calendar` to a JDBC TIMESTAMP

Registered under `calendar`, `java.util.Calendar` and `java.util.GregorianCalendar` in the type registry (see *Sección 6.5, "Type registry"*).

`org.hibernate.type.CalendarDateType`
Maps a `java.util.Calendar` to a JDBC DATE

Registered under `calendar_date` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.16. `java.util.Currency`

`org.hibernate.type.CurrencyType`
Maps a `java.util.Currency` to a JDBC VARCHAR (using the Currency code)

Registered under `currency` and `java.util.Currency` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.17. `java.util.Locale`

`org.hibernate.type.LocaleType`
Maps a `java.util.Locale` to a JDBC VARCHAR (using the Locale code)

Registered under `locale` and `java.util.Locale` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.18. `java.util.TimeZone`

`org.hibernate.type.TimeZoneType`
Maps a `java.util.TimeZone` to a JDBC VARCHAR (using the TimeZone ID)

Registered under `timezone` and `java.util.TimeZone` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.19. `java.net.URL`

`org.hibernate.type.UrlType`
Maps a `java.net.URL` to a JDBC VARCHAR (using the external form)

Registered under `url` and `java.net.URL` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.20. `java.lang.Class`

`org.hibernate.type.ClassType`
Maps a `java.lang.Class` to a JDBC VARCHAR (using the Class name)

Registered under `class` and `java.lang.Class` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.21. `java.sql.Blob`

`org.hibernate.type.BlobType`
Maps a `java.sql.Blob` to a JDBC BLOB

Registered under `blob` and `java.sql.Blob` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.22. `java.sql.Clob`

`org.hibernate.type.ClobType`
Maps a `java.sql.Clob` to a JDBC CLOB

Registered under `clob` and `java.sql.Clob` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.23. byte[]

`org.hibernate.type.BinaryType`
Maps a primitive byte[] to a JDBC VARBINARY

Registered under `binary` and `byte[]` in the type registry (see *Sección 6.5, "Type registry"*).

`org.hibernate.type.MaterializedBlobType`
Maps a primitive byte[] to a JDBC BLOB

Registered under `materialized_blob` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.24. byte[]

`org.hibernate.type.BinaryType`
Maps a java.lang.Byte[] to a JDBC VARBINARY

Registered under `wrapper-binary`, `Byte[]` and `java.lang.Byte[]` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.25. char[]

`org.hibernate.type.CharArrayType`
Maps a char[] to a JDBC VARCHAR

Registered under `characters` and `char[]` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.26. char[]

`org.hibernate.type.CharacterArrayType`
Maps a java.lang.Character[] to a JDBC VARCHAR

Registered under `wrapper-characters`, `Character[]` and `java.lang.Character[]` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.27. `java.util.UUID`

`org.hibernate.type.UUIDBinaryType`
Maps a java.util.UUID to a JDBC BINARY

Registered under `uuid-binary` and `java.util.UUID` in the type registry (see *Sección 6.5, "Type registry"*).

`org.hibernate.type.UUIDCharType`
Maps a java.util.UUID to a JDBC CHAR (though VARCHAR is fine too for existing schemas)

Registered under `uuid-char` in the type registry (see *Sección 6.5, "Type registry"*).

`org.hibernate.type.PostgresUUIDType`
: Maps a java.util.UUID to the PostgreSQL UUID data type (through `Types#OTHER` which is how the PostgreSQL JDBC driver defines it).

  Registered under `pg-uuid` in the type registry (see *Sección 6.5, "Type registry"*).

#### 6.1.1.28. `java.io.Serializable`

`org.hibernate.type.SerializableType`
: Maps implementors of java.lang.Serializable to a JDBC VARBINARY

  Unlike the other value types, there are multiple instances of this type. It gets registered once under `java.io.Serializable`. Additionally it gets registered under the specific `java.io.Serializable` implementation class names.

### 6.1.2. Composite types

**Nota**

The Java Persistence API calls these embedded types, while Hibernate traditionally called them components. Just be aware that both terms are used and mean the same thing in the scope of discussing Hibernate.

Components represent aggregations of values into a single Java type. For example, you might have an Address class that aggregates street, city, state, etc information or a Name class that aggregates the parts of a person's Name. In many ways a component looks exactly like an entity. They are both (generally speaking) classes written specifically for the application. They both might have references to other application-specific classes, as well as to collections and simple JDK types. As discussed before, the only distinguishing factory is the fact that a component does not own its own lifecycle nor does it define an identifier.

### 6.1.3. Collection types

**Importante**

It is critical understand that we mean the collection itself, not its contents. The contents of the collection can in turn be basic, component or entity types (though not collections), but the collection itself is owned.

Collections are covered in *Capítulo 7, Mapeos de colección* .

## 6.2. Entity types

The definition of entities is covered in detail in *Capítulo 4, Clases persistentes*. For the purpose of this discussion, it is enough to say that entities are (generally application-specific) classes which

correlate to rows in a table. Specifically they correlate to the row by means of a unique identifier. Because of this unique identifier, entities exist independently and define their own lifecycle. As an example, when we delete a `Membership`, both the `User` and `Group` entities remain.

**Nota**

This notion of entity independence can be modified by the application developer using the concept of cascades. Cascades allow certain operations to continue (or "cascade") across an association from one entity to another. Cascades are covered in detail in *Capítulo 8, Mapeos de asociación*.

## 6.3. Significance of type categories

Why do we spend so much time categorizing the various types of types? What is the significance of the distinction?

The main categorization was between entity types and value types. To review we said that entities, by nature of their unique identifier, exist independently of other objects whereas values do not. An application cannot "delete" a Product sku; instead, the sku is removed when the Product itself is deleted (obviously you can *update* the sku of that Product to null to make it "go away", but even there the access is done through the Product).

Nor can you define an association *to* that Product sku. You *can* define an association to Product *based on* its sku, assuming sku is unique, but that is totally different.

TBC...

## 6.4. Custom types

Hibernate makes it relatively easy for developers to create their own *value* types. For example, you might want to persist properties of type `java.lang.BigInteger` to `VARCHAR` columns. Custom types are not limited to mapping values to a single table column. So, for example, you might want to concatenate together `FIRST_NAME`, `INITIAL` and `SURNAME` columns into a `java.lang.String`.

There are 3 approaches to developing a custom Hibernate type. As a means of illustrating the different approaches, lets consider a use case where we need to compose a `java.math.BigDecimal` and `java.util.Currency` together into a custom `Money` class.

### 6.4.1. Custom types using `org.hibernate.type.Type`

The first approach is to directly implement the `org.hibernate.type.Type` interface (or one of its derivatives). Probably, you will be more interested in the more specific `org.hibernate.type.BasicType` contract which would allow registration of the type (see *Sección 6.5, "Type registry"*). The benefit of this registration is that whenever the metadata for a particular property does not specify the Hibernate type to use, Hibernate will consult the registry

for the exposed property type. In our example, the property type would be `Money`, which is the key we would use to register our type in the registry:

### Ejemplo 6.1. Defining and registering the custom Type

```
public class MoneyType implements BasicType {
    public String[] getRegistrationKeys() {
        return new String[] { Money.class.getName() };
    }

        public int[] sqlTypes(Mapping mapping) {
             // We will simply use delegation to the standard basic types for BigDecimal and
 Currency for many of the
            // Type methods...
            return new int[] {
                    BigDecimalType.INSTANCE.sqlType(),
                    CurrencyType.INSTANCE.sqlType(),
            };
            // we could also have honored any registry overrides via...
            //return new int[] {
                                                    //
 mappings.getTypeResolver().basic( BigDecimal.class.getName() ).sqlTypes( mappings )[0],
                                                    //
 mappings.getTypeResolver().basic( Currency.class.getName() ).sqlTypes( mappings )[0]
            //};
        }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throws SQLE
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index, boolean[] settable, SessionImplementor
            throws SQLException {
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}

Configuration cfg = new Configuration();
```

```
cfg.registerTypeOverride( new MoneyType() );
cfg...;
```

**Importante**

It is important that we registered the type *before* adding mappings.

## 6.4.2. Custom types using `org.hibernate.usertype.UserType`

**Nota**

Both `org.hibernate.usertype.UserType` and `org.hibernate.usertype.CompositeUserType` were originally added to isolate user code from internal changes to the `org.hibernate.type.Type` interfaces.

The second approach is the use the `org.hibernate.usertype.UserType` interface, which presents a somewhat simplified view of the `org.hibernate.type.Type` interface. Using a `org.hibernate.usertype.UserType`, our `Money` custom type would look as follows:

### Ejemplo 6.2. Defining the custom UserType

```
public class MoneyType implements UserType {
    public int[] sqlTypes() {
        return new int[] {
                BigDecimalType.INSTANCE.sqlType(),
                CurrencyType.INSTANCE.sqlType(),
        };
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(ResultSet rs, String[] names, Object owner) throws SQLException {
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

   public void nullSafeSet(PreparedStatement st, Object value, int index) throws SQLException {
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
```

```
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}
```

There is not much difference between the `org.hibernate.type.Type` example and the `org.hibernate.usertype.UserType` example, but that is only because of the snippets shown. If you choose the `org.hibernate.type.Type` approach there are quite a few more methods you would need to implement as compared to the `org.hibernate.usertype.UserType`.

### 6.4.3. Custom types using `org.hibernate.usertype.CompositeUserType`

The third and final approach is the use the `org.hibernate.usertype.CompositeUserType` interface, which differs from `org.hibernate.usertype.UserType` in that it gives us the ability to provide Hibernate the information to handle the composition within the `Money` class (specifically the 2 attributes). This would give us the capability, for example, to reference the `amount` attribute in an HQL query. Using a `org.hibernate.usertype.CompositeUserType`, our `Money` custom type would look as follows:

#### Ejemplo 6.3. Defining the custom CompositeUserType

```
public class MoneyType implements CompositeUserType {
    public String[] getPropertyNames() {
          // ORDER IS IMPORTANT!   it must match the order the columns are defined in the
 property mapping
        return new String[] { "amount", "currency" };
    }

    public Type[] getPropertyTypes() {
        return new Type[] { BigDecimalType.INSTANCE, CurrencyType.INSTANCE };
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object getPropertyValue(Object component, int propertyIndex) {
        if ( component == null ) {
            return null;
        }

        final Money money = (Money) component;
        switch ( propertyIndex ) {
            case 0: {
                return money.getAmount();
            }
            case 1: {
                return money.getCurrency();
            }
            default: {
```

```
                throw new HibernateException( "Invalid property index [" + propertyIndex + "]" );
            }
        }
    }

        public void setPropertyValue(Object component, int propertyIndex, Object value) throws HibernateException
        if ( component == null ) {
            return;
        }

        final Money money = (Money) component;
        switch ( propertyIndex ) {
            case 0: {
                money.setAmount( (BigDecimal) value );
                break;
            }
            case 1: {
                money.setCurrency( (Currency) value );
                break;
            }
            default: {
               throw new HibernateException( "Invalid property index [" + propertyIndex + "]" );
            }
        }
        }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throws SQLEx
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index, SessionImplementor session) throws SQLE
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}
```

## 6.5. Type registry

Internally Hibernate uses a registry of basic types (see *Sección 6.1.1, "Basic value types"*) when it needs to resolve the specific `org.hibernate.type.Type` to use in certain situations. It also provides a way for applications to add extra basic type registrations as well as override the standard basic type registrations.

To register a new type or to override an existing type registration, applications would make use of the `registerTypeOverride` method of the `org.hibernate.cfg.Configuration` class when bootstrapping Hibernate. For example, lets say you want Hibernate to use your custom `SuperDuperStringType`; during bootstrap you would call:

### Ejemplo 6.4. Overriding the standard `StringType`

```
Configuration cfg = ...;
cfg.registerTypeOverride( new SuperDuperStringType() );
```

The argument to `registerTypeOverride` is a `org.hibernate.type.BasicType` which is a specialization of the `org.hibernate.type.Type` we saw before. It adds a single method:

### Ejemplo 6.5. Snippet from BasicType.java

```
/**
         * Get the names under which this type should be registered in the type registry.
         *
         * @return The keys under which to register this type.
         */
        public String[] getRegistrationKeys();
```

One approach is to use inheritance (`SuperDuperStringType` extends `org.hibernate.type.StringType`); another is to use delegation.

# Mapeos de colección

## 7.1. Colecciones persistentes

Naturally Hibernate also allows to persist collections. These persistent collections can contain almost any other Hibernate type, including: basic types, custom types, components and references to other entities. The distinction between value and reference semantics is in this context very important. An object in a collection might be handled with "value" semantics (its life cycle fully depends on the collection owner), or it might be a reference to another entity with its own life cycle. In the latter case, only the "link" between the two objects is considered to be a state held by the collection.

As a requirement persistent collection-valued fields must be declared as an interface type (see *Ejemplo 7.2, "Collection mapping using @OneToMany and @JoinColumn"*). The actual interface might be `java.util.Set`, `java.util.Collection`, `java.util.List`, `java.util.Map`, `java.util.SortedSet`, `java.util.SortedMap` or anything you like ("anything you like" means you will have to write an implementation of `org.hibernate.usertype.UserCollectionType`).

Notice how in *Ejemplo 7.2, "Collection mapping using @OneToMany and @JoinColumn"* the instance variable `parts` was initialized with an instance of `HashSet`. This is the best way to initialize collection valued properties of newly instantiated (non-persistent) instances. When you make the instance persistent, by calling `persist()`, Hibernate will actually replace the `HashSet` with an instance of Hibernate's own implementation of `Set`. Be aware of the following error:

### Ejemplo 7.1. Hibernate uses its own collection implementations

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);

kittens = cat.getKittens(); // Okay, kittens collection is a Set
(HashSet) cat.getKittens(); // Error!
```

Las colecciones persistentes inyectadas por Hibernate se comportan como `HashMap`, `HashSet`, `TreeMap`, `TreeSet` o `ArrayList`, dependiendo del tipo de interfaz.

Las instancias de colecciones tienen el comportamiento usual de los tipos de valor. Son automáticamente persistidas al ser referenciadas por un objeto persistente y se borran automáticamente al desreferenciarse. Si una colección se pasa de un objeto persistente a otro, puede que sus elementos se muevan de una tabla a otra. Dos entidades no pueden compartir una referencia a la misma instancia de colección. Debido al modelo relacional subyacente, las

propiedades valuadas en colección no soportan la semántica de valor nulo. Hibernate no distingue entre una referencia de colección nula y una colección vacía.

**Nota**

Use persistent collections the same way you use ordinary Java collections. However, ensure you understand the semantics of bidirectional associations (see *Sección 7.3.2, "Asociaciones bidireccionales"*).

## 7.2. How to map collections

Using annotations you can map `Collections`, `Lists`, `Maps` and `Sets` of associated entities using @OneToMany and @ManyToMany. For collections of a basic or embeddable type use @ElementCollection. In the simplest case a collection mapping looks like this:

### Ejemplo 7.2. Collection mapping using @OneToMany and @JoinColumn

```
@Entity
public class Product {

    private String serialNumber;
    private Set<Part> parts = new HashSet<Part>();

    @Id
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }

    @OneToMany
    @JoinColumn(name="PART_ID")
    public Set<Part> getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
}


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

Product describes a unidirectional relationship with Part using the join column PART_ID. In this unidirectional one to many scenario you can also use a join table as seen in *Ejemplo 7.3, "Collection mapping using @OneToMany and @JoinTable"*.

### Ejemplo 7.3. Collection mapping using @OneToMany and @JoinTable

```
@Entity
public class Product {
```

```
    private String serialNumber;
    private Set<Part> parts = new HashSet<Part>();

    @Id
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }

    @OneToMany
    @JoinTable(
            name="PRODUCT_PARTS",
            joinColumns = @JoinColumn( name="PRODUCT_ID"),
            inverseJoinColumns = @JoinColumn( name="PART_ID")
    )
    public Set<Part> getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
}


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

Without describing any physical mapping (no `@JoinColumn` or `@JoinTable`), a unidirectional one to many with join table is used. The table name is the concatenation of the owner table name, _, and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table, _, and the owner primary key column(s) name. The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s) name. A unique constraint is added to the foreign key referencing the other side table to reflect the one to many.

Lets have a look now how collections are mapped using Hibernate mapping files. In this case the first step is to chose the right mapping element. It depends on the type of interface. For example, a `<set>` element is used for mapping properties of type `Set`.

### Ejemplo 7.4. Mapping a Set using <set>

```
<class name="Product">
    <id name="serialNumber" column="productSerialNumber"/>
    <set name="parts">
        <key column="productSerialNumber" not-null="true"/>
        <one-to-many class="Part"/>
    </set>
</class>
```

In *Ejemplo 7.4, “Mapping a Set using <set>”* a *one-to-many association* links the `Product` and `Part` entities. This association requires the existence of a foreign key column and possibly an index column to the `Part` table. This mapping loses certain semantics of normal Java collections:

- Una instancia de la clase entidad contenida no puede pertenecer a más de una instancia de la colección.

- Una instancia de la clase entidad contenida no puede aparecer en más de un valor del índice de colección.

Looking closer at the used `<one-to-many>` tag we see that it has the following options.

### Ejemplo 7.5. options of <one-to-many> element

```
<one-to-many
        class="ClassName"                          (1)
        not-found="ignore|exception"               (2)
        entity-name="EntityName"                   (3)
        node="element-name"
        embed-xml="true|false"
    />
```

(1) `class` (requerido): El nombre de la clase asociada.

(2) `not-found` (opcional - por defecto es `exception`): Especifica cómo serán manejados los identificadores en caché que referencien filas perdidas. `ignore` tratará una fila perdida como una asociación nula.

(3) `entity-name` (opcional): El nombre de entidad de la clase asociada como una alternativa para `class`.

El elemento `<one-to-many>` no necesita declarar ninguna columna. Ni es necesario especificar el nombre de `table` en ningún sitio.

**Aviso**

If the foreign key column of a `<one-to-many>` association is declared `NOT NULL`, you must declare the `<key>` mapping `not-null="true"` or *use a bidirectional association* with the collection mapping marked `inverse="true"`. See *Sección 7.3.2, "Asociaciones bidireccionales"*.

Apart from the `<set>` tag as shown in *Ejemplo 7.4, "Mapping a Set using <set>"*, there is also `<list>`, `<map>`, `<bag>`, `<array>` and `<primitive-array>` mapping elements. The `<map>` element is representative:

### Ejemplo 7.6. Elements of the <map> mapping

```
<map
    name="propertyName"                            (1)
    table="table_name"                             (2)
    schema="schema_name"                           (3)
    lazy="true|extra|false"                        (4)
```

```
    inverse="true|false"                                                    5
    cascade="all|none|save-update|delete|all-delete-orphan|delete-orphan"   6
    sort="unsorted|natural|comparatorClass"                                 7
    order-by="column_name asc|desc"                                         8
    where="arbitrary sql where condition"                                   9
    fetch="join|select|subselect"                                           10
    batch-size="N"                                                          11
    access="field|property|ClassName"                                       12
    optimistic-lock="true|false"                                            13
    mutable="true|false"                                                    14
    node="element-name|."
    embed-xml="true|false"
>

    <key .... />
    <map-key .... />
    <element .... />
</map>
```

1. `name`: el nombre de la propiedad de colección
2. `table` (opcional - por defecto es el nombre de la propiedad): el nombre de la tabla de colección. No se utiliza para asociaciones uno-a-muchos.
3. `schema` (opcional): el nombre de un esquema de tablas para sobrescribir el esquema declarado en el elemento raíz
4. `lazy` (opcional - por defecto es `true`): deshabilita la recuperación perezosa y especifica que la asociación siempre es recuperada tempranamente. También se puede utilizar para activar una recuperación "extra-perezoza", en donde la mayoría de las operaciones no inicializan la colección. Esto es apropiado para colecciones grandes.
5. `inverse` (opcional - por defecto es `false`): marca esta colección como el extremo "inverso" de una asociación bidireccional.
6. `cascade` (opcional - por defecto es `none`): habilita operaciones en cascada para entidades hijas.
7. `sort` (opcional): especifica una colección con ordenamiento `natural`, o una clase comparadora dada.
8. `order-by` (optional): specifies a table column or columns that define the iteration order of the `Map`, `Set` or bag, together with an optional `asc` or `desc`.
9. `where` (opcional): especifica una condición `WHERE` de SQL arbitraria que se utiliza al recuperar o quitar la colección. Esto es útil si la colección debe contener sólamente un subconjunto de los datos disponibles.
10. `fetch` (opcional, por defecto es `select`): Elige entre la recuperación por unión externa (outer-join), la recuperación por selección secuencial y la recuperación por subselección secuencial.
11. `batch-size` (opcional, por defecto es `1`): especifica un "tamaño de lote" para recuperar perezosamente instancias de esta colección.

12. `access` (opcional - por defecto es `property`): La estrategia que Hibernate utiliza para acceder al valor de la propiedad de colección.
13. `optimistic-lock` (opcional - por defecto es `true`): Especifica que los cambios de estado de la colección causan incrementos de la versión de la entidad dueña. Para asociaciones uno a muchos, es posible que quiera deshabilitar esta opción.
14. `mutable` (opcional - por defectos es `true`): Un valor `false` especifica que los elementos de la colección nunca cambian. En algunos casos, esto permite una pequeña optimización de rendimiento.

After exploring the basic mapping of collections in the preceding paragraphs we will now focus details like physical mapping considerations, indexed collections and collections of value types.

## 7.2.1. Claves foráneas de colección

On the database level collection instances are distinguished by the foreign key of the entity that owns the collection. This foreign key is referred to as the *collection key column*, or columns, of the collection table. The collection key column is mapped by the `@JoinColumn` annotation respectively the `<key>` XML element.

There can be a nullability constraint on the foreign key column. For most collections, this is implied. For unidirectional one-to-many associations, the foreign key column is nullable by default, so you may need to specify

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

or

```
<key column="productSerialNumber" not-null="true"/>
```

The foreign key constraint can use `ON DELETE CASCADE`. In XML this can be expressed via:

```
<key column="productSerialNumber" on-delete="cascade"/>
```

In annotations the Hibernate specific annotation @OnDelete has to be used.

```
@OnDelete(action=OnDeleteAction.CASCADE)
```

See *Sección 5.1.11.3, “Key”* for more information about the `<key>` element.

## 7.2.2. Colecciones indexadas

In the following paragraphs we have a closer at the indexed collections `List` and `Map` how the their index can be mapped in Hibernate.

### 7.2.2.1. Lists

Lists can be mapped in two different ways:

- as ordered lists, where the order is not materialized in the database
- as indexed lists, where the order is materialized in the database

To order lists in memory, add `@javax.persistence.OrderBy` to your property. This annotation takes as parameter a list of comma separated properties (of the target entity) and orders the collection accordingly (eg `firstname asc, age desc`), if the string is empty, the collection will be ordered by the primary key of the target entity.

**Ejemplo 7.7. Ordered lists using `@OrderBy`**

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @OrderBy("number")
   public List<Order> getOrders() { return orders; }
   public void setOrders(List<Order> orders) { this.orders = orders; }
   private List<Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------|
| Order       | | Customer |
```

```
|-------------| |----------|
| id          | | id       |
| number      | |----------|
| customer_id |
|-------------|
```

To store the index value in a dedicated column, use the `@javax.persistence.OrderColumn` annotation on your property. This annotations describes the column name and attributes of the column keeping the index value. This column is hosted on the table containing the association foreign key. If the column name is not specified, the default is the name of the referencing property, followed by underscore, followed by `ORDER` (in the following example, it would be `orders_ORDER`).

### Ejemplo 7.8. Explicit index column using `@OrderColumn`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @OrderColumn(name="orders_index")
   public List<Order> getOrders() { return orders; }
   public void setOrders(List<Order> orders) { this.orders = orders; }
   private List<Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|--------------| |----------|
| Order        | | Customer |
|--------------| |----------|
| id           | | id       |
| number       | |----------|
| customer_id  |
| orders_order |
|--------------|
```

**Nota**

We recommend you to convert the legacy `@org.hibernate.annotations.IndexColumn` usages to `@OrderColumn` unless you are making use of the base property. The `base` property lets you define the index value of the first element (aka as base index). The usual value is `0` or `1`. The default is 0 like in Java.

Looking again at the Hibernate mapping file equivalent, the index of an array or list is always of type `integer` and is mapped using the `<list-index>` element. The mapped column contains sequential integers that are numbered from zero by default.

### Ejemplo 7.9. index-list element for indexed collections in xml mapping

```
<list-index
        column="column_name"                                    (1)
        base="0|1|..."/>
```

(1) `column_name` (requerido): el nombre de la columna que tiene los valores del índice de la colección.

(1) `base` (opcional - por defecto es `0`): el valor de la columna índice que corresponde al primer elemento de la lista o el array.

Si su tabla no tiene una columna índice y todavía desea utilizar `List` como tipo de propiedad, puede mapear la propiedad como un *<bag>* de Hibernate. Un bag (bolsa) no retiene su orden al ser recuperado de la base de datos, pero puede ser ordenado o clasificado de manera opcional.

## 7.2.2.2. Maps

The question with `Maps` is where the key value is stored. There are everal options. Maps can borrow their keys from one of the associated entity properties or have dedicated columns to store an explicit key.

To use one of the target entity property as a key of the map, use `@MapKey(name="myProperty")`, where `myProperty` is a property name in the target entity. When using `@MapKey` without the name attribuate, the target entity primary key is used. The map key uses the same column as the property pointed out. There is no additional column defined to hold the map key, because the map key represent a target property. Be aware that once loaded, the key is no longer kept in sync with the property. In other words, if you change the property value, the key will not change automatically in your Java model.

### Ejemplo 7.10. Use of target entity property as map key via `@MapKey`

```
@Entity
```

```
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @MapKey(name="number")
   public Map<String,Order> getOrders() { return orders; }
   public void setOrders(Map<String,Order> order) { this.orders = orders; }
   private Map<String,Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------|
| Order       | | Customer |
|-------------| |----------|
| id          | | id       |
| number      | |----------|
| customer_id |
|-------------|
```

Alternatively the map key is mapped to a dedicated column or columns. In order to customize the mapping use one of the following annotations:

- `@MapKeyColumn` if the map key is a basic type. If you don't specify the column name, the name of the property followed by underscore followed by `KEY` is used (for example `orders_KEY`).
- `@MapKeyEnumerated` / `@MapKeyTemporal` if the map key type is respectively an enum or a `Date`.
- `@MapKeyJoinColumn`/`@MapKeyJoinColumns` if the map key type is another entity.
- `@AttributeOverride`/`@AttributeOverrides` when the map key is a embeddable object. Use `key.` as a prefix for your embeddable object property names.

You can also use `@MapKeyClass` to define the type of the key if you don't use generics.

### Ejemplo 7.11. Map key as basic type using `@MapKeyColumn`

```
@Entity
```

```
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany @JoinTable(name="Cust_Order")
   @MapKeyColumn(name="orders_number")
   public Map<String,Order> getOrders() { return orders; }
   public void setOrders(Map<String,Order> orders) { this.orders = orders; }
   private Map<String,Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------| |---------------|
| Order       | | Customer | | Cust_Order    |
|-------------| |----------| |---------------|
| id          | | id       | | customer_id   |
| number      | |----------| | order_id      |
| customer_id |              | orders_number |
|-------------|              |---------------|
```

### Nota

We recommend you to migrate from `@org.hibernate.annotations.MapKey` / `@org.hibernate.annotation.MapKeyManyToMany` to the new standard approach described above

Using Hibernate mapping files there exists equivalent concepts to the descibed annotations. You have to use `<map-key>`, `<map-key-many-to-many>` and `<composite-map-key>`. `<map-key>` is used for any basic type, `<map-key-many-to-many>` for an entity reference and `<composite-map-key>` for a composite type.

### Ejemplo 7.12. map-key xml mapping element

```
<map-key
```

```
        column="column_name"                    (1)
        formula="any SQL expression"            (2)
        type="type_name"                        (3)
        node="@attribute-name"
        length="N"/>
```

(1) `column` (opcional): el nombre de la columna que tiene los valores del índice de colecciones.
(2) `formula` (opcional): una fórmula SQL que se usa para evaluar la clave del mapa.
(3) `type` (requerido): el tipo de las claves del mapa.

### Ejemplo 7.13. map-key-many-to-many

```
<map-key-many-to-many
        column="column_name"                    (1)
        formula="any SQL expression"            (2)(3)
        class="ClassName"
/>
```

(1) `column` (opcional): el nombre de la columna de la clave foránea para los valores del índice de la colección.
(2) `formula` (opcional): una fórmula SQ utilizada para evaluar la clave foránea de la clave de mapeos.
(3) `class` (requerido): La clase de entidad que se usa como la clave mapeada.

## 7.2.3. Collections of basic types and embeddable objects

In some situations you don't need to associate two entities but simply create a collection of basic types or embeddable objects. Use the `@ElementCollection` for this case.

### Ejemplo 7.14. Collection of basic types mapped via `@ElementCollection`

```
@Entity
public class User {
   [...]
   public String getLastname() { ...}

   @ElementCollection
   @CollectionTable(name="Nicknames", joinColumns=@JoinColumn(name="user_id"))
   @Column(name="nickname")
   public Set<String> getNicknames() { ... }
}
```

The collection table holding the collection data is set using the `@CollectionTable` annotation. If omitted the collection table name defaults to the concatenation of the name of the containing

entity and the name of the collection attribute, separated by an underscore. In our example, it would be `User_nicknames`.

The column holding the basic type is set using the `@Column` annotation. If omitted, the column name defaults to the property name: in our example, it would be `nicknames`.

But you are not limited to basic types, the collection type can be any embeddable object. To override the columns of the embeddable object in the collection table, use the `@AttributeOverride` annotation.

### Ejemplo 7.15. @ElementCollection for embeddable objects

```
@Entity
public class User {
   [...]
   public String getLastname() { ...}

   @ElementCollection
   @CollectionTable(name="Addresses", joinColumns=@JoinColumn(name="user_id"))
   @AttributeOverrides({
      @AttributeOverride(name="street1", column=@Column(name="fld_street"))
   })
   public Set<Address> getAddresses() { ... }
}

@Embeddable
public class Address {
   public String getStreet1() {...}
   [...]
}
```

Such an embeddable object cannot contains a collection itself.

#### Nota

in `@AttributeOverride`, you must use the `value.` prefix to override properties of the embeddable object used in the map value and the `key.` prefix to override properties of the embeddable object used in the map key.

```
@Entity
public class User {
   @ElementCollection
   @AttributeOverrides({
      @AttributeOverride(name="key.street1", column=@Column(name="fld_street")),
      @AttributeOverride(name="value.stars", column=@Column(name="fld_note"))
   })
   public Map<Address,Rating> getFavHomes() { ... }
```

**Nota**

We recommend you to migrate from `@org.hibernate.annotations.CollectionOfElements` to the new `@ElementCollection` annotation.

Using the mapping file approach a collection of values is mapped using the `<element>` tag. For example:

**Ejemplo 7.16. <element> tag for collection values using mapping files**

```
<element
        column="column_name"                    (1)
        formula="any SQL expression"            (2)
        type="typename"                         (3)
        length="L"
        precision="P"
        scale="S"
        not-null="true|false"
        unique="true|false"
        node="element-name"
/>
```

1. `column` (opcional): el nombre de la columna que tiene los valores de los elementos de la colección.
2. `formula` (opcional): una fórmula SQL utilizada para evaluar el elemento.
3. `type` (requerido): el tipo del elemento de colección.

## 7.3. Mapeos de colección avanzados

### 7.3.1. Colecciones ordenadas

Hibernate supports collections implementing `java.util.SortedMap` and `java.util.SortedSet`. With annotations you declare a sort comparator using `@Sort`. You chose between the comparator types unsorted, natural or custom. If you want to use your own comparator implementation, you'll also have to specify the implementation class using the `comparator` attribute. Note that you need to use either a `SortedSet` or a `SortedMap` interface.

**Ejemplo 7.17. Sorted collection with @Sort**

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Sort(type = SortType.COMPARATOR, comparator = TicketComparator.class)
public SortedSet<Ticket> getTickets() {
```

```
    return tickets;
}
```

Using Hibernate mapping files you specify a comparator in the mapping file with `<sort>`:

### Ejemplo 7.18. Sorted collection using xml mapping

```
<set name="aliases"
            table="person_aliases"
            sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

Los valores permitidos del atributo `sort` son `unsorted`, `natural` y el nombre de una clase que implemente `java.util.Comparator`.

**Sugerencia**

Las colecciones ordenadas realmente se comportan como `java.util.TreeSet` o `java.util.TreeMap`.

If you want the database itself to order the collection elements, use the `order-by` attribute of `set`, `bag` or `map` mappings. This solution is implemented using `LinkedHashSet` or `LinkedHashMap` and performs the ordering in the SQL query and not in the memory.

### Ejemplo 7.19. Sorting in database using order-by

```
<set name="aliases" table="person_aliases" order-by="lower(name) asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

**Nota**

El valor del atributo `order-by` es una ordenación SQL, no una ordenación HQL.

Las asociaciones pueden incluso ser ordenadas por algún criterio arbitrario en tiempo de ejecución utilizando un `filter()` de colección:

**Ejemplo 7.20. Sorting via a query filter**

```
sortedUsers = s.createFilter( group.getUsers(), "order by this.name" ).list();
```

## 7.3.2. Asociaciones bidireccionales

Una *asociación bidireccional* permite la navegación desde ambos "extremos" de la asociación. Se soportan dos tipos de asociación bidireccional:

uno-a-muchos
: conjunto o bag valorados en un lado, monovaluados en el otro

muchos-a-muchos
: set o bag valorados en ambos extremos

Often there exists a many to one association which is the owner side of a bidirectional relationship. The corresponding one to many association is in this case annotated by `@OneToMany(mappedBy=...)`

**Ejemplo 7.21. Bidirectional one to many with many to one side as association owner**

```
@Entity
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk")
    public Troop getTroop() {
    ...
}
```

`Troop` has a bidirectional one to many relationship with `Soldier` through the `troop` property. You don't have to (must not) define any physical mapping in the `mappedBy` side.

To map a bidirectional one to many, with the one-to-many side as the owning side, you have to remove the `mappedBy` element and set the many to one `@JoinColumn` as insertable and updatable to false. This solution is not optimized and will produce additional UPDATE statements.

### Ejemplo 7.22. Bidirectional associtaion with one to many side as owner

```
@Entity
public class Troop {
    @OneToMany
    @JoinColumn(name="troop_fk") //we need to duplicate the physical information
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk", insertable=false, updatable=false)
    public Troop getTroop() {
    ...
}
```

How does the mappping of a bidirectional mapping look like in Hibernate mapping xml? There you define a bidirectional one-to-many association by mapping a one-to-many association to the same table column(s) as a many-to-one association and declaring the many-valued end `inverse="true"`.

### Ejemplo 7.23. Bidirectional one to many via Hibernate mapping files

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <set name="children" inverse="true">
        <key column="parent_id"/>
        <one-to-many class="Child"/>
    </set>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

Mapear un extremo de una asociación con `inverse="true"` no afecta la operación de cascadas ay que éstos son conceptos ortogonales.

A many-to-many association is defined logically using the `@ManyToMany` annotation. You also have to describe the association table and the join conditions using the `@JoinTable` annotation. If the association is bidirectional, one side has to be the owner and one side has to be the inverse end (ie. it will be ignored when updating the relationship values in the association table):

### Ejemplo 7.24. Many to many association via @ManyToMany

```
@Entity
public class Employer implements Serializable {
    @ManyToMany(
        targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
        cascade={CascadeType.PERSIST, CascadeType.MERGE}
    )
    @JoinTable(
        name="EMPLOYER_EMPLOYEE",
        joinColumns=@JoinColumn(name="EMPER_ID"),
        inverseJoinColumns=@JoinColumn(name="EMPEE_ID")
    )
    public Collection getEmployees() {
        return employees;
    }
    ...
}
```

```
@Entity
public class Employee implements Serializable {
    @ManyToMany(
        cascade = {CascadeType.PERSIST, CascadeType.MERGE},
        mappedBy = "employees",
        targetEntity = Employer.class
    )
    public Collection getEmployers() {
        return employers;
    }
}
```

In this example `@JoinTable` defines a `name`, an array of join columns, and an array of inverse join columns. The latter ones are the columns of the association table which refer to the `Employee` primary key (the "other side"). As seen previously, the other side don't have to (must not) describe the physical mapping: a simple `mappedBy` argument containing the owner side property name bind the two.

As any other annotations, most values are guessed in a many to many relationship. Without describing any physical mapping in a unidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table name, _ and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

### Ejemplo 7.25. Default values for `@ManyToMany` (uni-directional)

```
@Entity
public class Store {
    @ManyToMany(cascade = CascadeType.PERSIST)
    public Set<City> getImplantedIn() {
        ...
    }
}

@Entity
public class City {
    ... //no bidirectional relationship
}
```

A `Store_City` is used as the join table. The `Store_id` column is a foreign key to the `Store` table. The `implantedIn_id` column is a foreign key to the `City` table.

Without describing any physical mapping in a bidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the other side property name, _, and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

### Ejemplo 7.26. Default values for `@ManyToMany` (bi-directional)

```
@Entity
public class Store {
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    public Set<Customer> getCustomers() {
        ...
    }
}

@Entity
public class Customer {
    @ManyToMany(mappedBy="customers")
    public Set<Store> getStores() {
        ...
    }
}
```

A `Store_Customer` is used as the join table. The `stores_id` column is a foreign key to the `Store` table. The `customers_id` column is a foreign key to the `Customer` table.

Using Hibernate mapping files you can map a bidirectional many-to-many association by mapping two many-to-many associations to the same database table and declaring one end as *inverse*.

**Nota**

You cannot select an indexed collection.

*Ejemplo 7.27, "Many to many association using Hibernate mapping files"* shows a bidirectional many-to-many association that illustrates how each category can have many items and each item can be in many categories:

### Ejemplo 7.27. Many to many association using Hibernate mapping files

```
<class name="Category">
    <id name="id" column="CATEGORY_ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="Item">
    <id name="id" column="ITEM_ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

Los cambios realizados sólamente al extremo inverso de la asociación *no* son persistidos. Esto significa que Hibernate tiene dos representaciones en memoria para cada asociación bidireccional: un enlace de A a B y otro enlace de B a A. Esto es más fácil de entender si piensa en el modelo de objetos de Java y cómo creamos una relación muchos-a-muchos en Java:

### Ejemplo 7.28. Effect of inverse vs. non-inverse side of many to many associations

```
category.getItems().add(item);          // The category now "knows" about the relationship
item.getCategories().add(category);     // The item now "knows" about the relationship

session.persist(item);                  // The relationship won't be saved!
session.persist(category);              // The relationship will be saved
```

El lado no-inverso se utiliza para guardar la representación en memoria a la base de datos.

### 7.3.3. Asociaciones bidireccionales con colecciones indexadas

There are some additional considerations for bidirectional mappings with indexed collections (where one end is represented as a `<list>` or `<map>`) when using Hibernate mapping files. If there is a property of the child class that maps to the index column you can use `inverse="true"` on the collection mapping:

**Ejemplo 7.29. Bidirectional association with indexed collection**

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children" inverse="true">
        <key column="parent_id"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <property name="name"
        not-null="true"/>
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

Si no existe tal propiedad en la clase hija, no podemos considerar la asociación como verdaderamente bidireccional. Es decir, hay información en un extremo de la asociación que no está disponible en el otro extremo. En este caso, no puede mapear la colección con `inverse="true"`. En cambio, puede usar el siguiente mapeo:

**Ejemplo 7.30. Bidirectional association with indexed collection, but no index column**

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children">
        <key column="parent_id"
            not-null="true"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>
```

```
<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        insert="false"
        update="false"
        not-null="true"/>
</class>
```

Note que en este mapeo, el extremo de la asociación valuado en colección es responsable de las actualizaciones de la clave foránea.

## 7.3.4. Asociaciones ternarias

Hay tres enfoques posibles para mapear una asociación ternaria. Una es utilizar un `Map` con una asociación como su índice:

### Ejemplo 7.31. Ternary association mapping

```
@Entity
public class Company {
   @Id
   int id;
   ...
   @OneToMany // unidirectional
   @MapKeyJoinColumn(name="employee_id")
   Map<Employee, Contract> contracts;
}

// or

<map name="contracts">
    <key column="employer_id" not-null="true"/>
    <map-key-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>
```

A second approach is to remodel the association as an entity class. This is the most common approach. A final alternative is to use composite elements, which will be discussed later.

## 7.3.5. Using an <idbag>

The majority of the many-to-many associations and collections of values shown previously all map to tables with composite keys, even though it has been suggested that entities should have synthetic identifiers (surrogate keys). A pure association table does not seem to benefit much from a surrogate key, although a collection of composite values *might*. For this reason Hibernate provides a feature that allows you to map many-to-many associations and collections of values to a table with a surrogate key.

El elemento `<idbag>` le permite mapear una `List` (o `Collection`) con semántica de bag. Por ejemplo:

```
<idbag name="lovers" table="LOVERS">
    <collection-id column="ID" type="long">
        <generator class="sequence"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Person" fetch="join"/>
</idbag>
```

Un `<idbag>` tiene un generador de id sintético, al igual que una clase de entidad. Se asigna una clave delegada diferente a cada fila de la colección. Sin embargo, Hibernate no proporciona ningún mecanismo para descubrir el valor de la clave delegada de una fila en particular.

El rendimiento de actualización de un `<idbag>` es mucho mejor que el de un `<bag>` normal. Hibernate puede localizar filas individuales eficientemente y actualizarlas o borrarlas individualmente, al igual que si fuese una lista, mapa o conjunto.

En la implementación actual, la estrategia de generación de identificador `native` no se encuentra soportada para identificadores de colecciones `<idbag>`.

## 7.4. Ejemplos de colección

Esta sección cubre los ejemplos de colección.

La siguiente clase tiene una colección de instancias `Child`:

### Ejemplo 7.32. Example classes `Parent` and `Child`

```
public class Parent {
    private long id;
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   private long id;
   private String name


   // getter/setter
   ...
}
```

Si cada hijo tiene como mucho un padre, el mapeo más natural es una asociación uno-a-muchos:

### Ejemplo 7.33. One to many unidirectional `Parent-Child` relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;
   private String name;


   // getter/setter
   ...
}
```

### Ejemplo 7.34. One to many unidirectional `Parent-Child` relationship using mapping files

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Esto mapea a las siguientes definiciones de tabla:

### Ejemplo 7.35. Table definitions for unidirectional `Parent-Child` relationship

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

Si el padre es *requerido*, utilice una asociación bidireccional uno-a-muchos:

### Ejemplo 7.36. One to many bidirectional `Parent-Child` relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany(mappedBy="parent")
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;

   private String name;

   @ManyToOne
   private Parent parent;


   // getter/setter
   ...
}
```

### Ejemplo 7.37. One to many bidirectional `Parent-Child` relationship using mapping files

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true">
            <key column="parent_id"/>
```

```
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="Parent" column="parent_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

Observe la restricción `NOT NULL`:

### Ejemplo 7.38. Table definitions for bidirectional `Parent-Child` relationship

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

Alternatively, if this association must be unidirectional you can enforce the `NOT NULL` constraint.

### Ejemplo 7.39. Enforcing NOT NULL constraint in unidirectional relation using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany(optional=false)
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;
   private String name;


   // getter/setter
   ...
```

```
}
```

### Ejemplo 7.40. Enforcing NOT NULL constraint in unidirectional relation using mapping files

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id" not-null="true"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

On the other hand, if a child has multiple parents, a many-to-many association is appropriate.

### Ejemplo 7.41. Many to many `Parent-Child` relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @ManyToMany
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;

   private String name;


   // getter/setter
```

```
   ...
}
```

### Ejemplo 7.42. Many to many `Parent-Child` relationship using mapping files

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" table="childset">
            <key column="parent_id"/>
            <many-to-many class="Child" column="child_id"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Definiciones de tabla:

### Ejemplo 7.43. Table definitions for many to many releationship

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child
```

For more examples and a complete explanation of a parent/child relationship mapping, see *Capítulo 24, Ejemplo: Padre/Hijo* for more information. Even more complex association mappings are covered in the next chapter.

# Mapeos de asociación

## 8.1. Introducción

Los mapeos de asociación son frecuentemente lo más difícil de implementar correctamente. En esta sección revisaremos algunos casos canónicos uno por uno, comenzando con los mapeos unidireccionales y luego considerando los casos bidireccionales. Vamos a utilizar `Person` y `Address` en todos los ejemplos.

Vamos a clasificar las asociaciones en cuanto su multiplicidad y a si mapean o no a una tabla de unión interviniente.

Las claves foráneas que aceptan valores nulos no se consideran como una buena práctica en el modelado tradicional de datos, así que todos nuestros ejemplos utilizan claves foráneas no nulas. Esto no es un requisito de Hibernate y todos los mapeos funcionarán si quita las restricciones de nulabilidad.

## 8.2. Asociaciones Unidireccionales

### 8.2.1. Many-to-one

Una *asociación unidireccional muchos-a-uno* es el tipo de asociación unidireccional más común.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

### 8.2.2. Uno-a-uno

Una *asociación unidireccional uno-a-uno en una clave foránea* es casi idéntica. La única diferencia es la restricción de unicidad de la columna.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

Usualmente, una *asociación unidireccional uno-a-uno en una clave principal* utiliza un generador de id especial. Sin embargo, hemos invertido la dirección de la asociación:

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property"
>person</param>
        </generator>
    </id>
    <one-to-one name="person" constrained="true"/>
</class
>
```

```
create table Person ( personId bigint not null primary key )
```

```
create table Address ( personId bigint not null primary key )
```

### 8.2.3. Uno-a-muchos

Una *asociación unidireccional uno-a-muchos en una clave foránea* es un caso muy inusual y no se recomienda.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses">
        <key column="personId"
            not-null="true"/>
        <one-to-many class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, personId bigint not null )
```

En lugar debe utilizar una tabla de unión para esta clase de asociación.

## 8.3. Asociaciones unidireccionales con tablas de unión

### 8.3.1. Uno-a-muchos

Se prefiere una *asociación unidireccional uno-a-muchos en una tabla de unión* . El especificar `unique="true"`, cambia la multiplicidad de muchos-a-muchos a uno-a-muchos.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
```

```
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

### 8.3.2. Many-to-one

Una *asociación unidireccional muchos-a-uno en una tabla de unión* es común cuando la asociación es opcional. Por ejemplo:

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

### 8.3.3. Uno-a-uno

Una *asociación unidireccional uno-a-uno en una tabla de unión* es extremadamente inusual, pero es posible.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null
 unique )
create table Address ( addressId bigint not null primary key )
```

### 8.3.4. Muchos-a-muchos

Finalmente, este es un ejemplo de una *asociación unidireccional muchos-a-muchos*.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
```

```
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key
 (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

## 8.4. Asociaciones bidireccionales

### 8.4.1. uno-a-muchos / muchos-a-uno

Una *asociación bidireccional muchos-a-uno* es el tipo de asociación más común. El siguiente ejemplo ilustra la relación estándar padre/hijo.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
    </set>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

Si utiliza un `List`, u otra colección con índice, configure la columna `key` de la clave foránea como `not null`. Hibernate administrará la asociación del lado de las colecciones para

mantener el índice de cada elemento, haciendo del otro lado virtualmente inverso al establecer `update="false"` y `insert="false"`:

```
<class name="Person">
   <id name="id"/>
   ...
   <many-to-one name="address"
      column="addressId"
      not-null="true"
      insert="false"
      update="false"/>
</class>

<class name="Address">
   <id name="id"/>
   ...
   <list name="people">
      <key column="addressId" not-null="true"/>
      <list-index column="peopleIdx"/>
      <one-to-many class="Person"/>
   </list>
</class
>
```

Es importante que defina `not-null="true"` en el elemento `<key>` del mapeo de la colección si la columna de la clave foránea es `NOT NULL`. No declare sólamente `not-null="true"` en un elemento `<column>` posiblemente anidado sino en el elemento `<key>`.

## 8.4.2. Uno-a-uno

Una *asociación bidireccional uno-a-uno en una clave foránea* es común:

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
   <one-to-one name="person"
        property-ref="address"/>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

Una *asociación bidireccional uno-a-uno en una clave primaria* utiliza el generador de id especial:

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property"
>person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

## 8.5. Asociaciones bidireccionales con tablas de unión

### 8.5.1. uno-a-muchos / muchos-a-uno

El siguiente es un ejemplo de una *asociación bidireccional uno-a-muchos en una tabla de unión*. El `inverse="true"` puede ir en cualquier lado de la asociación, en la colección, o en la unión.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses"
        table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
```

```
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"/>
    </join>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

### 8.5.2. uno a uno

Una *asociación bidireccional uno-a-uno en una tabla de unión* es extremadamente inusual, pero es posible.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true"
        inverse="true">
        <key column="addressId"
            unique="true"/>
```

```
        <many-to-one name="person"
            column="personId"
            not-null="true"
            unique="true"/>
    </join>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null
 unique )
create table Address ( addressId bigint not null primary key )
```

### 8.5.3. Muchos-a-muchos

Este es un ejemplo de una *asociación bidireccional muchos-a-muchos*.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true" table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key
 (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

## 8.6. Mapeos de asociación más complejos

Uniones de asociación más complejas son *extremadamente* raras. Hibernate maneja situaciones más complejas utilizando fragmentos SQL incluidos en el documento de mapeo. Por ejemplo, si una tabla con datos históricos de información de cuenta define las columnas `accountNumber`, `effectiveEndDate` y `effectiveStartDate`, se mapearían así:

```
<properties name="currentAccountKey">
    <property name="accountNumber" type="string" not-null="true"/>
    <property name="currentAccount" type="boolean">
        <formula
>case when effectiveEndDate is null then 1 else 0 end</formula>
    </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/>
```

Entonces puede mapear una asociación a la instancia *actual*, la que tiene `effectiveEndDate` nulo, utilizando:

```
<many-to-one name="currentAccountInfo"
        property-ref="currentAccountKey"
        class="AccountInfo">
    <column name="accountNumber"/>
    <formula
>'1'</formula>
</many-to-one
>
```

En un ejemplo más complejo, imagínese que la asociación entre `Employee` y `Organization` se mantienen en una tabla `Employment` llena de datos históricos de empleo. Entonces se puede mapear una asociación al empleador *más reciente* del empleado, el que tiene la `startDate` más reciente, de esta manera:

```
<join>
    <key column="employeeId"/>
    <subselect>
        select employeeId, orgId
        from Employments
        group by orgId
        having startDate = max(startDate)
    </subselect>
    <many-to-one name="mostRecentEmployer"
            class="Organization"
            column="orgId"/>
</join
>
```

Esta funcionalidad le permite cierto grado de creatividad y flexibilidad, pero usualmente es más práctico manejar esta clase de casos utilizando HQL o una petición de criterio.

# Mapeo de componentes

La noción de un *componente* se reutiliza en muchos contextos diferentes, para propósitos diferentes a través de Hibernate.

## 9.1. Objetos dependientes

Un componente es un objeto contenido que es persistido como un tipo de valor, no una referencia de entidad. El término "componente" hace referencia a la noción orientada a objetos de composición y no a componentes a nivel de arquitectura. Por ejemplo, puede modelar una persona así:

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
```

```
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

Ahora `Name` puede ser persistido como un componente de `Person`. `Name` define métodos getter y setter para sus propiedades persistentes, pero no necesita declarar ninguna interfaz ni propiedades identificadoras.

Nuestro mapeo de Hibernate se vería así:

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"
> <!-- class attribute optional -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class
>
```

La tabla person tendría las columnas `pid`, `birthday`, `initial`, `first` y `last`.

Como todos los tipos de valor, los componentes no soportan referencias compartidas. En otras palabras, dos personas pueden tener el mismo nombre, pero los dos objetos persona contendrían dos objetos nombre independientes, sólamente "iguales" en valor. La semántica de valor nulo de un componente es *ad hoc*. Cuando se recargue el objeto contenedor, Hibernate asumirá que si todas las columnas del componente son nulas, el componente entero es nulo. Esto es apropiado para la mayoría de propósitos.

Las propiedades de un componente pueden ser de cualquier tipo de Hibernate (colecciones, asociaciones muchos-a-uno, otros componentes, etc). Los componentes anidados *no* deben ser considerados como un uso exótico. Hibernate está concebido para soportar un modelo de objetos muy detallado.

El elemento `<component>` permite un subelemento `<parent>` que mapea una propiedad de la clase del componente como una referencia a la entidad contenedora.

```
<class name="eg.Person" table="person">
```

```
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name" unique="true">
        <parent name="namedPerson"/> <!-- reference back to the Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class
>
```

## 9.2. Colecciones de objetos dependientes

Las colecciones de componentes se encuentran soportadas (por ejemplo, un array de tipo `Name`). Declare su colección de componentes remplazando la etiqueta `<element>` por una etiqueta `<composite-element>`:

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="eg.Name"
> <!-- class attribute required -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </composite-element>
</set
>
```

**Importante**

Si define un `Set` de elementos compuestos, es muy importante implementar `equals()` y `hashCode()` de manera correcta.

Los elementos compuestos pueden contener componentes pero no colecciones. Si su elemento compuesto contiene a su vez componentes, use la etiqueta `<nested-composite-element>`. Este es un caso de una colección de componentes que a su vez tienen componentes. Se debe estar preguntando si una asociación uno-a-muchos es más apropiada. Remodele el elemento compuesto como una entidad - pero observe que aunque el modelo Java es el mismo, el modelo relacional y la semántica de persistencia siguen siendo ligeramente diferentes.

Un mapeo de elemento compuesto no soporta propiedades nulables si está utilizando un `<set>`. No hay una columna clave principal separada en la tabla del elemento compuesto. Hibernate utiliza el valor de cada columna para identificar un registro al borrar objetos, lo cual es imposible con valores nulos. Tiene que usar sólo propiedades no nulas en un elemento compuesto o elegir un `<list>`, `<map>`, `<bag>` o `<idbag>`.

Un caso especial de un elemento compuesto es un elemento compuesto con un elemento anidado `<many-to-one>`. Este mapeo le permite mapear columnas extra de una tabla de asociación muchos-a-muchos a la clase del elemento compuesto. La siguiente es una asociación muchos-a-muchos de `Order` a `Item`, donde `purchaseDate`, `price` y `quantity` son propiedades de la asociación:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- class attribute is optional -->
        </composite-element>
    </set>
</class
>
```

No puede haber una referencia a la compra del otro lado para la navegación bidireccional de la asociación. Los componentes son tipos de valor y no permiten referencias compartidas. Una sola `Purchase` puede estar en el conjunto de una `Order`, pero no puede ser referenciada por el `Item` al mismo tiempo.

Incluso son posibles las asociaciones ternarias (o cuaternarias, etc):

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails class="eg.Purchase"/>
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class
>
```

Los elementos compuestos pueden aparecer en consultas usando la misma sintáxis que las asociaciones a otras entidades.

## 9.3. Componentes como índices de Mapeo

El elemento `<composite-map-key>` le permite mapear una clase componente como la clave de un `Map`. Asegúrese de sobrescribir `hashCode()` y `equals()` correctamente en la clase componente.

## 9.4. Componentes como identificadores compuestos

Puede utilizar un componente como un identidicador de una clase entidad. Su clase componente tiene que satisfacer ciertos requerimientos:

- Tiene que implementar `java.io.Serializable`.
- Tiene que re-implementar `equals()` y `hashCode()`, consistentemente con la noción de la base de datos de igualdad de clave compuesta.

**Nota**

En Hibernate3, aunque el segundo requerimiento no es un requerimiento absolutamente rígido de Hibernate, en todo caso se recomienda.

No puede utilizar un `IdentifierGenerator` para generar claves compuestas. En cambio, la aplicación debe asignar sus propios identificadores.

Use la etiqueta `<composite-id>`, con elementos anidados `<key-property>`, en lugar de la declaración usual `<id>`. Por ejemplo, la clase `OrderLine` tiene una clave principal que depende de la clave principal (compuesta) de `Order`.

```
<class name="OrderLine">

    <composite-id name="id" class="OrderLineId">
        <key-property name="lineId"/>
        <key-property name="orderId"/>
        <key-property name="customerId"/>
    </composite-id>

    <property name="name"/>

    <many-to-one name="order" class="Order"
            insert="false" update="false">
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-one>
    ....

</class
>
```

Cualquier clave foránea que referencie la tabla de `OrderLine` también es compuesta. Declare esto en sus mapeos de otras clases. Una asociación a `OrderLine` se mapea así:

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
    <column name="lineId"/>
```

```
    <column name="orderId"/>
    <column name="customerId"/>
</many-to-one
>
```

### Sugerencia

El elemento `column` es una alternativa al atributo `column` en cualquier lugar. El uso del elemento `column` simplemente le da más opciones de declaración, las cuales son útiles al utilizar `hbm2ddl`.

Una asociación `muchos-a-muchos` a `OrderLine` también usa la clave foránea compuesta:

```
<set name="undeliveredOrderLines">
    <key column name="warehouseId"/>
    <many-to-many class="OrderLine">
        <column name="lineId"/>
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-many>
</set
>
```

La colección de `OrderLines` en `Order` utilizaría:

```
<set name="orderLines" inverse="true">
    <key>
        <column name="orderId"/>
        <column name="customerId"/>
    </key>
    <one-to-many class="OrderLine"/>
</set
>
```

El elemento `<one-to-many>` declara ninguna columna.

Si `OrderLine` posee una colección por sí misma, tiene también una clave foránea compuesta.

```
<class name="OrderLine">
    ....
    ....
    <list name="deliveryAttempts">
        <key
>   <!-- a collection inherits the composite key type -->
            <column name="lineId"/>
            <column name="orderId"/>
            <column name="customerId"/>
```

```
        </key>
        <list-index column="attemptId" base="1"/>
        <composite-element class="DeliveryAttempt">
            ...
        </composite-element>
    </set>
</class
>
```

## 9.5. Componentes dinámicos

También puede mapear una propiedad del tipo `Map`:

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO" type="string"/>
    <property name="bar" column="BAR" type="integer"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</dynamic-component
>
```

La semántica de un mapeo `<dynamic-component>` es ídentica a la de `<component>`. La ventaja de este tipo de mapeos es la habilidad para determinar las propiedades reales del bean en tiempo de despliegue, sólo con editar el documento de mapeo. La manipulación del documento de mapeo en tiempo de ejecución también es posible, usando un analizador DOM. También puede acceder y cambiar el metamodelo de tiempo de configuración de Hibernate por medio del objeto `Configuration`.

# Mapeo de herencias

## 10.1. Las tres estrategias

Hibernate soporta las tres estrategias básicas de mapeo de herencia:

- tabla por jerarquía de clases
- table per subclass
- tabla por clase concreta

Además, Hibernate soporta un cuarto, un tipo ligeramente diferente de polimorfismo:

- polimorfismo implícito

Es posible utilizar estrategias de mapeo diferentes para diferentes ramificaciones de la misma jerarquía de herencia. Luego puede usar un polimorfismo implícito para conseguir polimorfismo a través de toda la jerarquía. Sin embargo, Hibernate no soporta la mezcla de mapeos `<subclass>`, `<joined-subclass>` y `<union-subclass>` bajo el mismo elemento `<class>` raíz. Es posible mezclar las estrategias de tabla por jerarquía y tabla por subclase bajo el mismo elemento `<class>`, combinando los elementos `<subclass>` y `<join>` (a continuación encontrará un ejemplo).

Es posible definir los mapeos `subclass`, `union-subclass`, y `joined-subclass` en documentos de mapeo separados, directamente debajo de `hibernate-mapping`. Esto le permite extender une jerarquía de clase sólamente añadiendo un nuevo archivo de mapeo. Tiene que especificar un atributo `extends` en la subclase de mapeo, nombrando una superclase mapeada previamente. Nota: Anteriormente esta característica hacia que el orden de los documentos de mapeo fuera importante. Desde Hibernate3, el orden de los archivos de mapeo no tiene relevancia cuando se utiliza la palabra clave extends. El orden dentro de un sólo archivo de mapeo todavía necesita ser definido como superclases antes de subclases.

```
 <hibernate-mapping>
     <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
          <property name="name" type="string"/>
     </subclass>
 </hibernate-mapping
>
```

### 10.1.1. Tabla por jerarquía de clases

Suponga que tenemos una interfaz `Payment`, con los implementadores `CreditCardPayment`, `CashPayment`, `ChequePayment`. El mapeo de tabla por jerarquía se vería así:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class
>
```

Se requiere exactamente una tabla. Hay una limitación de esta estrategia de mapeo: las columnas declaradas por las subclases tal como `CCTYPE`, no pueden tener restricciones `NOT NULL`.

### 10.1.2. Tabla por subclase

Un mapeo de tabla por subclase se vería así:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class
>
```

Se necesitan cuatro tablas. Las tres tablas de subclase tienen asociaciones de clave principal a la tabla de superclase de modo que en el modelo relacional es realmente una asociación uno-a-uno.

### 10.1.3. Tabla por subclase: utilizando un discriminador

La implementación de Hibernate de tabla por subclase no requiere ninguna columna discriminadora. Otros mapeadores objeto/relacional usan una implementación diferente de tabla por subclase que necesita una columna discriminadora de tipo en la tabla de superclase. Este enfoque es mucho más difícil de implementar pero discutiblemente más correcto desde un punto de vista relacional. Si quisiere utilizar una columna discriminadora con la estrategia de tabla por subclase, puede combinar el uso de `<subclass>` y `<join>`, así:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class
>
```

La declaración opcional `fetch="select"` le dice a Hibernate que no recupere los datos de la subclase `ChequePayment` utilizando una unión externa (outer join) al consultar la superclase.

### 10.1.4. Mezcla de tabla por jerarquía de clases con tabla por subclase

Incluso puede mezclar las estrategias de tabla por jerarquía y tabla por subclase utilizando este enfoque:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
```

```
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class
>
```

Para cualquiera de estas estrategias de mapeo, una asociación polimórfica a la clase raíz `Payment` es mapeada usando `<many-to-one>`.

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

### 10.1.5. Tabla por clase concreta

Hay dos maneras de mapear la tabla por estrategia de clase concreta. La primera es utilizar `<union-subclass>`.

```
<class name="Payment">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        ...
    </union-subclass>
</class
>
```

Hay tres tablas involucradas. Cada tabla define columnas para todas las propiedades de la clase, incluyendo las propiedades heredadas.

La limitación de este enfoque es que si una propiedad se mapea en la superclase, el nombre de la columna debe ser el mismo en todas las tablas de subclase. La estrategia del generador de identidad no está permitida en la herencia de unión de subclase. La semilla de la clave principal tiene que compartirse a través de todas las subclases unidas de una jerarquía.

Si su superclase es abstracta, mapeéla con `abstract="true"`. Si no es abstracta, se necesita una tabla adicional (en el ejemplo anterior, por defecto es `PAYMENT`) para mantener las instancias de la superclase.

## 10.1.6. Tabla por clase concreta utilizando polimorfismo implícito

Un enfoque alternativo es para hacer uso del polimorfismo implícito:

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class
>
```

Observe que la interfaz `Payment` no se menciona explícitamente. También note que las propiedades de `Payment` se mapean en cada una de las subclases. Si quiere evitar la duplicación, considere el usar entidades XML (por ejemplo, `[ <!ENTITY allproperties SYSTEM "allproperties.xml"> ]` en la declaración `DOCTYPE` y `&allproperties;` en el mapeo).

La desventaja de este enfoque es que Hibernate no genera `UNION`es de SQL al realizar consultas polimórficas.

Para esta estrategia de mapeo, una asociación polimórfica a `Payment` es mapeada generalmente utilizando `<any>`.

```
<any name="payment" meta-type="string" id-type="long">
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any
>
```

### 10.1.7. Mezcla de polimorfismo implícito con otros mapeos de herencia

Ya que las subclases se mapean cada una en su propio elemento `<class>` y debido a que `Payment` es sólo una interfaz, cada una de las subclases podría ser fácilmente parte de otra jerarquía de herencia. Todavía puede seguir usando consultas polimórficas contra la interfaz `Payment`.

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class
>
```

Una vez más, no mencionamos a `Payment` explícitamente. Si ejecutamos una consulta frente a la interfaz `Payment` - por ejemplo, `from Payment`, Hibernate retorna automáticamente instancias de `CreditCardPayment` (y sus subclases, ya que ellas también implementan `Payment`), `CashPayment` y `ChequePayment` pero no las instancias de `NonelectronicTransaction`.

## 10.2. Limitaciones

Existen ciertas limitaciones al enfoque de "polimorfismo implícito" en la estrategia de mapeo de tabla por clase concreta. Existen limitaciones un poco menos restrictivas a los mapeos `<union-subclass>`.

La siguiente tabla muestra las limitaciones de los mapeos de tabla por clase concreta y del polimorfismo implícito en Hibernate.

### Tabla 10.1. Funcionalidades de los mapeos de herencia

| Estrategi de herencia | Polimórfi muchos-a-uno | Polimórfi uno-a-uno | Polimórfi uno-a-muchos | Polimórfi muchos-a-muchos | Polimórfi `load()/get()` | Consulta polimórfi | Uniones polimórfi | Recuperación por unión externa |
|---|---|---|---|---|---|---|---|---|
| tabla por jerarquía de clases | `<many-to-one>` | `<one-to-one>` | `<one-to-many>` | `<many-to-many>` | `s.get(Payment.class, id)` | `from Payment p` | `from Order o join o.payment p` | *supported* |
| table per subclass | `<many-to-one>` | `<one-to-one>` | `<one-to-many>` | `<many-to-many>` | `s.get(Payment.class, id)` | `from Payment p` | `from Order o join o.payment p` | *supported* |
| tabla por clase concreta (union-subclass) | `<many-to-one>` | `<one-to-one>` | `<one-to-many>` (solo para `inverse="true"`) | `<many-to-many>` | `s.get(Payment.class, id)` | `from Payment p` | `from Order o join o.payment p` | *supported* |
| tabla por clase concreta (polimorfismo implícito) | `<any>` | *not supported* | *not supported* | `<many-to-any>` | `s.createCriteria(Payment.class).add( Restrictions.` | `from Payment p` | *not supported* | *not supported* |

# Trabajo con objetos

Hibernate es una solución completa de mapeo objeto/relacional que no sólo proteje al desarrollador de los detalles del sistema de administración de la base datos subyacente, sino que además ofrece *administración de estado* de objetos. Contrario a la administración de `declaraciones` SQL en capas comunes de persistencia JDBC/SQL, esta es una vista natural orientada a objetos de la persistencia en aplicaciones Java.

En otras palabras, los desarrolladores de aplicaciones de Hibernate siempre deben pensar en el *estado* de sus objetos, y no necesariamente en la ejecución de declaraciones SQL. Hibernate se ocupa de esto y es sólamente relevante para el desarrollador de la aplicación al afinar el rendimiento del sistema.

## 11.1. Estados de objeto de Hibernate

Hibernate define y soporta los siguientes estados de objeto:

- *Transitorio* - un objeto es transitorio si ha sido recién instanciado utilizando el operador `new`, y no está asociado a una `Session` de Hibernate. No tiene una representación persistente en la base de datos y no se le ha asignado un valor identificador. Las instancias transitorias serán destruídas por el recolector de basura si la aplicación no mantiene más una referencia. Utiliza la `Session` de Hibernate para hacer un objeto persistente (y deja que Hibernate se ocupe de las declaraciones SQL que necesitan ejecutarse para esta transición).
- *Persistente* - una instancia persistente tiene una representación en la base de datos y un valor identificador. Puede haber sido guardado o cargado, sin embargo, por definición, se encuentra en el ámbito de una `Session`. Hibernate detectará cualquier cambio realizado a un objeto en estado persistente y sincronizará el estado con la base de datos cuando se complete la unidad de trabajo. Los desarrolladores no ejecutan declaraciones `UPDATE` manuales, o declaraciones `DELETE` cuando un objeto se debe poner como transitorio.
- *Separado* - una instancia separada es un objeto que se ha hecho persistente, pero su `Session` ha sido cerrada. La referencia al objeto todavía es válida, por supuesto, y la instancia separada podría incluso ser modificada en este estado. Una instancia separada puede ser re-unida a una nueva `Session` más tarde, haciéndola persistente de nuevo (con todas las modificaciones). Este aspecto habilita un modelo de programación para unidades de trabajo de ejecución larga que requieren tiempo-para-pensar por parte del usuario. Las llamamos *transaccciones de aplicación*, por ejemplo, una unidad de trabajo desde el punto de vista del usuario.

Discutiremos ahora los estados y transiciones de estados (y los métodos de Hibernate que disparan una transición) en más detalle.

## 11.2. Haciendo los objetos persistentes

Las instancias recién instanciadas de una clase persistente, Hibernate las considera como *transitorias*. Podemos hacer una instancia transitoria *persistente* asociándola con una sesión:

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

Si `Cat` tiene un identificador generado, el identificador es generado y asignado al `cat` cuando se llama a `save()`. Si `Cat` tiene un identificador `assigned`, o una clave compuesta, el identificador debe ser asignado a la instancia de `cat` antes de llamar a `save()`. También puede utilizar `persist()` en vez de `save()`, con la semántica definida en el borrador de EJB3.

- `persist()` hace que una instancia transitoria sea persistente. Sin embargo, no garantiza que el valor identificador sea asignado a la instancia persistente inmediatamente, la tarea puede tener lugar durante el vaciado. `persist()` también garantiza que no ejecutará una declaración `INSERT` si se llama por fuera de los límites de una transacción. Esto es útil en conversaciones largas con un contexto extendido sesión/persistencia.
- `save()` sí garantiza el retorno de un identificador. Si se tiene que ejecutar un INSERT para obtener el identificador ( por ejemplo, generador "identidad", no "secuencia"), este INSERT tiene lugar inmediatamente sin importar si se encuentra dentro o fuera de una transacción. Esto es problemático en una conversación larga con un contexto extendido sesión/persistencia.

Opcionalmente, puede asignar el identificador utilizando una versión sobrecargada de `save()`.

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

Si el objeto que hace persistente tiene objetos asociados (por ejemplo, la colección `kittens` en el ejemplo anterior), estos objetos pueden ser hechos persistentes en cualquier orden que quiera a menos de que tenga una restricción `NOT NULL` sobre una columna clave foránea. Nunca hay riesgo de violar restricciones de clave foránea. Sin embargo, puede que usted viole una restricción `NOT NULL` si llama a `save()` sobre los objetos en el orden equivocado.

Usualmente no se preocupe de este detalle, pues muy probablemente utilizará la funcionalidad de *persistencia transitiva* de Hibernate para guardar los objetos asociados automáticamente. Entonces, ni siquiera tienen lugar violaciones de restricciones `NOT NULL` - Hibernate se ocupará de todo. Más adelante en este capítulo se discute la persistencia transitiva.

## 11.3. Cargando un objeto

Los métodos `load()` de `Session` le proporcionan una forma de recuperar una instancia persistente si ya conoce su identificador. `load()` toma un objeto clase y carga el estado dentro de una instancia recién instanciada de esa clase, en un estado persistente.

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );
```

Alternativamente, puede cargar estado dentro de una instancia dada:

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

Note que `load()` lanzará una excepción irrecuperable si no hay una fila correspondiente en la base de datos. Si la clase se mapea con un proxy, `load()` sólo retorna un proxy no inicializado y no llamará realmente a la base de datos hasta que invoque un método del proxy. Este comportamiento es muy útil si desea crear una asociación a un objeto sin cargarlo realmente de la base de datos. Además permite que múltiples instancias sean cargadas como un lote si se define `batch-size` para el mapeo de la clase.

Si no tiene la certeza de que existe una fila correspondiente, debe utilizar el método `get()`, que llama a la base de datos inmediatamente y devuelve nulo si no existe una fila correspondiente.

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

Incluso puede cargar un objeto utilizando un `SELECT ... FOR UPDATE` de SQL, usando un `LockMode`. Consulte la documentación de la API para obtener más información.

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

*Ninguna* instancia asociada o colección contenida es seleccionada para actualizacion - `FOR UPDATE`, a menos de que decida especificar `lock` o `all` como un estilo de cascada para la asociación.

Es posible volver a cargar un objeto y todas sus colecciones en cualquier momento, utilizando el método `refresh()`. Esto es útil cuando se usan disparadores de base de datos para inicializar algunas de las propiedades del objeto.

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

How much does Hibernate load from the database and how many SQL `SELECT`s will it use? This depends on the *fetching strategy*. This is explained in *Sección 21.1, "Estrategias de recuperación"*.

## 11.4. Consultas

Si no conoce los identificadores de los objetos que está buscando, necesita una consulta. Hibernate soporta un lenguaje de consulta orientado a objetos (HQL) fácil de usar pero potente a la vez. Para la creación de consultas programáticas, Hibernate soporta una funcionalidad sofisticada de consulta de Criteria y Example (QBC y QBE). También puede expresar su consulta en el SQL nativo de su base de datos, con soporte opcional de Hibernate para la conversión del conjunto de resultados a objetos.

### 11.4.1. Ejecución de consultas

Las consultas HQL y SQL nativas son representadas con una instancia de `org.hibernate.Query`. Esta interfaz ofrece métodos para ligar parámetros, manejo del conjunto resultado, y para la ejecución de la consulta real. Siempre obtiene una `Query` utilizando la `Session` actual:

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
```

```
    .uniqueResult();]]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

Una consulta se ejecuta usualmente invocando a `list()`. El resultado de la consulta será cargado completamente dentro de una colección en memoria. Las instancias de entidad recuperadas por una consulta se encuentran en estado persistente. El método `uniqueResult()` ofrece un atajo si sabe que su consulta retornará sólamente un objeto. Las consultas que hacen uso de una recuperación temprana de colecciones usualmente retornan duplicados de los objetos raíz, pero con sus colecciones inicializadas. Puede filtrar estos duplicados a través de un `Set`.

#### 11.4.1.1. Iteración de resultados

Ocasionalmente, puede lograr un mejor rendimiento al ejecutar la consulta utilizando el método `iterate()`. Esto ocurrirá usualmente si espera que las instancias reales de entidad retornadas por la consulta estén ya en la sesión o en el caché de segundo nivel. Si todavía no están en caché, `iterate()` será más lento que `list()` y podría requerir muchas llamadas a la base de datos para una consulta simple, usualmente *1* para la selección inicial que sólamente retorna identificadores y *n* selecciones adicionales para inicializar las instancias reales.

```
// fetch ids
Iterator iter = sess.createQuery("from eg.Qux q order by q.likeliness").iterate();
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}
```

#### 11.4.1.2. Consultas que devuelven tuplas

Las consultas de Hibernate a veces retornan tuplas de objetos. Cada tupla se retorna como un array:

```
Iterator kittensAndMothers = sess.createQuery(
            "select kitten, mother from Cat kitten join kitten.mother mother")
            .list()
            .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten = (Cat) tuple[0];
    Cat mother = (Cat) tuple[1];
```

```
    ....
}
```

### 11.4.1.3. Resultados escalares

Las consultas pueden especificar una propiedad de una clase en la cláusula `select`. Pueden incluso llamar a funciones de agregación SQL. Las propiedades o agregaciones son considerados resultados "escalares" y no entidades en estado persistente.

```
Iterator results = sess.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color")
        .list()
        .iterator();

while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

### 11.4.1.4. Ligado de parámetros

Los métodos en `Query` se proporcionan para enlazar valores a los parámetros con nombre o parámetros `?` de estilo JDBC. *Al contrario de JDBC, Hibernate numera los parámetros desde cero.* Los parámetros con nombre son identificadores de la forma `:name` en la cadena de la consulta. Las ventajas de los parámetros con nombre son las siguientes:

- los parámetros con nombre son insensibles al orden en que aparecen en la cadena de consulta
- pueden aparecer múltiples veces en la misma petición
- son auto-documentados

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//named parameter list
```

```
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

### 11.4.1.5. Paginación

Si necesita especificar enlaces sobre su conjunto de resultados, el número máximo de filas que quiere recuperar y/o la primera fila que quiere recuperar, puede utilizar los métodos de la interfaz `Query`:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Hibernate sabe cómo traducir este límite de consulta al SQL nativo de su DBMS.

### 11.4.1.6. Iteración deslizable

Si su controlador JDBC soporta `ResultSets` deslizables, la interfaz `Query` se puede utilizar para obtener un objeto `ScrollableResults` que permite una navegación flexible de los resultados de consulta.

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
                           "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabetical list of cats by name
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );

}
cats.close()
```

Note que se requiere una conexión de base de datos abierta y un cursor para esta funcionalidad. Utilice `setMaxResult()`/`setFirstResult()` si necesita la funcionalidad de paginación fuera de línea.

### 11.4.1.7. Externalización de consultas con nombre

Queries can also be configured as so called named queries using annotations or Hibernate mapping documents. `@NamedQuery` and `@NamedQueries` can be defined at the class level as seen in *Ejemplo 11.1, “Defining a named query using @NamedQuery”* . However their definitions are global to the session factory/entity manager factory scope. A named query is defined by its name and the actual query string.

**Ejemplo 11.1. Defining a named query using `@NamedQuery`**

```
@Entity
@NamedQuery(name="night.moreRecentThan", query="select n from Night n where n.date >= :date")
public class Night {
    ...
}

public class MyDao {
    doStuff() {
        Query q = s.getNamedQuery("night.moreRecentThan");
        q.setDate( "date", aMonthAgo );
        List results = q.list();
        ...
    }
    ...
}
```

Using a mapping document can be configured using the `<query>` node. Remember to use a `CDATA` section if your query contains characters that could be interpreted as markup.

**Ejemplo 11.2. Defining a named query using `<query>`**

```
<query name="ByNameAndMaximumWeight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

Parameter binding and executing is done programatically as seen in *Ejemplo 11.3, “Parameter binding of a named query”*.

**Ejemplo 11.3. Parameter binding of a named query**

```
Query q = sess.getNamedQuery("ByNameAndMaximumWeight");
```

```
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

El código real del programa es independiente del lenguaje de consulta utilizado. También puede definir consultas SQL nativas en metadatos, o migrar consultas existentes a Hibernate colocándolas en archivos de mapeo.

Observe además que una declaración de consulta dentro de un elemento `<hibernate-mapping>` necesita de un nombre único global para la consulta, mientras que una declaración de consulta dentro de un elemento `<class>` se hace única automáticamente al agregar el nombre completamente calificado de la clase. Por ejemplo, `eg.Cat.ByNameAndMaximumWeight`.

## 11.4.2. Filtración de colecciones

Un *filtro* de colección es un tipo especial de consulta que puede ser aplicado a una colección persistente o array. La cadena de consulta puede referirse a `this`, lo que quiere decir el elemento de la colección actual.

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.class) )
    .list()
);
```

La colección devuelta es considerada un bag, y es una copia de la colección dada. La colección original no es modificada. Esto es lo opuesto a lo que implica el nombre "filtro", pero es consistente con el comportamiento esperado.

Observe que los filtros no requieren una cláusula `from` aunque pueden tener una si se necesita. Los filtros no están limitados a devolver los elementos de colección por sí mismos.

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue")
    .list();
```

Incluso una consulta de filtro vacío es útil, por ejemplo, para cargar un subconjunto de elementos en una colección enorme:

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
    .setFirstResult(0).setMaxResults(10)
    .list();
```

### 11.4.3. Consultas de criterios

HQL es extremadamente potente pero algunos desarrolladores prefieren construir consultas dinámicamente utilizando una API orientada a objetos, en vez de construir cadenas de consulta. Hibernate brinda una API intuitiva de consulta `Criteria` para estos casos:

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Restrictions.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

The `Criteria` and the associated `Example` API are discussed in more detail in *Capítulo 17, Consultas por criterios*.

### 11.4.4. Consultas en SQL nativo

Puede expresar una consulta en SQL, utilizando `createSQLQuery()` y dejar que Hibernate administre el mapeo de los conjuntos de resultados a objetos. Puede llamar en cualquier momento a `session.connection()` y utilizar la `Connection` JDBC directamente. Si elige usar la API de Hibernate, tiene que encerrar los alias de SQL entre llaves:

```
List cats = session.createSQLQuery("SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10")
    .addEntity("cat", Cat.class)
.list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
           "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10")
    .addEntity("cat", Cat.class)
.list()
```

SQL queries can contain named and positional parameters, just like Hibernate queries. More information about native SQL queries in Hibernate can be found in *Capítulo 18, SQL Nativo*.

## 11.5. Modificación de objetos persistentes

Las *instancias persistentes transaccionales* (por ejemplo, los objetos cargados, creados o consultados por la `Session`) pueden ser manipulados por la aplicación y cualquier cambio al estado persistente será persistido cuando se *vacie* la `Session`. Esto se discute más adelante en este capítulo. No hay necesidad de llamar a un método en particular (como `update()`, que tiene un propósito diferente) para hacer persistentes sus modificaciones. De modo que la forma más directa de actualizar el estado de un objeto es cargarlo con `load()` y luego manipularlo directamente, mientras la `Session` está abierta:

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush();  // changes to cat are automatically detected and persisted
```

A veces este modelo de programación es ineficiente pues requiere un `SELECT` de SQL para cargar un objeto y un `UPDATE` de SQL para hacer persistente su estado actualizado en la misma sesión. Por lo tanto, Hibernate ofrece un enfoque opcional, utilizando instancias separadas.

## 11.6. Modificación de objetos separados

Muchas aplicaciones necesitan recuperar un objeto en una transacción, enviarla a la capa de UI para su manipulación, y entonces guardar los cambios en una nueva transacción. Las aplicaciones que usan este tipo de enfoque en un entorno de alta concurrencia usualmente utilizan datos versionados para asegurar el aislamiento de la unidad de trabajo "larga".

Hibernate soporta este modelo al proveer re-unión de instancias separadas utilizando los métodos `Session.update()` o `Session.merge()`:

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat);  // update cat
secondSession.update(mate); // update mate
```

Si el `Cat` con identificador `catId` ya hubiera sido cargado por `secondSession` cuando la aplicación intentó volver a unirlo, se habría lanzado una excepción.

Utilice `update()` si está seguro de que la sesión no tiene una instancia ya persistente con el mismo identificador. Utilice `merge()` si quiere fusionar sus modificaciones en cualquier momento sin consideración del estado de la sesión. En otras palabras, `update()` usualmente es el primer método que usted llamaría en una sesión actualizada, asegurando que la re-unión de sus instancias separadas es la primera operación que se ejecuta.

The application should individually `update()` detached instances that are reachable from the given detached instance *only* if it wants their state to be updated. This can be automated using *transitive persistence*. See *Sección 11.11, "Persistencia transitiva"* for more information.

El método `lock()` también le permite a una aplicación reasociar un objeto con una sesión nueva. Sin embargo, la instancia separada no puede haber sido modificada.

```
//just reassociate:
```

```
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassociate:
sess.lock(pk, LockMode.UPGRADE);
```

Note que `lock()` se puede utilizar con varios `LockMode`s. Consulte la documentación de la API y el capítulo sobre el manejo de transacciones para obtener mayor información. La re-unión no es el único caso de uso para `lock()`.

Other models for long units of work are discussed in *Sección 13.3, "Control de concurrencia optimista"*.

## 11.7. Detección automática de estado

Los usuarios de Hibernate han pedido un método de propósito general que bien guarde una instancia transitoria generando un identificador nuevo, o bien actualice/reúna las instancias separadas asociadas con su identificador actual. El método `saveOrUpdate()` implementa esta funcionalidad.

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat);   // update existing state (cat has a non-null id)
secondSession.saveOrUpdate(mate);  // save the new instance (mate has a null id)
```

La utilización y semántica de `saveOrUpdate()` parece ser confuso para los usuarios nuevos. Primero, en tanto no esté tratando de utilizar instancias de una sesión en otra sesión nueva, no debe necesitar usar `update()`, `saveOrUpdate()`, o `merge()`. Algunas aplicaciones enteras nunca usarán ninguno de estos métodos.

Usualmente `update()` o `saveOrUpdate()` se utilizan en el siguiente escenario:

- la aplicación carga un objeto en la primera sesión
- el objeto se pasa a la capa de UI
- se realizan algunas modificaciones al objeto
- el objeto se pasa abajo de regreso a la capa lógica de negocios
- la aplicación hace estas modificaciones persistentes llamando a `update()` en una segunda sesión

`saveOrUpdate()` hace lo siguiente:

- si el objeto ya es persistente en esta sesión, no haga nada

- si otro objeto asociado con la sesión tiene el mismo identificador, lance una excepción
- si el objeto no tiene ninguna propiedad identificadora, guárdelo llamando a `save()`
- si el identificador del objeto tiene el valor asignado a un objeto recién instanciado, guárdelo llamando a `save()`
- si el objeto está versionado por un `<version>` o `<timestamp>`, y el valor de la propiedad de versión es el mismo valor asignado a un objeto recién instanciado, guárdelo llamando a `save()`
- de otra manera actualice el objeto llamando a `update()`

y `merge()` es muy diferente:

- si existe una instancia persistente con el mismo identificador asignado actualmente con la sesión, copie el estado del objeto dado en la instancia persistente
- si no existe ninguna instancia persistente asociada a la sesión actualmente, intente cargarla desde la base de datos, o cree una nueva instancia persistente
- la instancia persistente es devuelta
- la instancia dada no se asocia a la sesión, permanece separada

## 11.8. Borrado de objetos persistentes

`Session.delete()` borrará el estado de un objeto de la base de datos. Sin embargo, su aplicación puede tener todavía una referencia a un objeto borrado. Lo mejor es pensar en `delete()` al hacer transitoria una instancia persistente.

```
sess.delete(cat);
```

Puede borrar objetos en el orden que quiera, sin riesgo de violaciones de restricción de clave foránea. Aún es posible violar una restricción `NOT NULL` sobre una columna de clave foránea borrando objetos en un orden erróneo, por ejemplo, si borra el padre, pero olvida borrar los hijos.

## 11.9. Replicación de objetos entre dos almacenamientos de datos diferentes

A veces es útil poder tomar un grafo de la instancias persistentes y hacerlas persistentes en un almacenamiento de datos diferente, sin regenerar los valores identificadores.

```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
```

```
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

El `ReplicationMode` determina cómo `replicate()` tratará los conflictos con filas existentes en la base de datos:

- `ReplicationMode.IGNORE`: ignora el objeto cuando existe una fila de la base de datos con el mismo identificador
- `ReplicationMode.OVERWRITE`: sobrescribe cualquier fila de la base de datos existente con el mismo identificador
- `ReplicationMode.EXCEPTION`: lanza una excepción si existe una fila de la base de datos con el mismo identificador
- `ReplicationMode.LATEST_VERSION`: sobrescribe la fila si su número de versión es anterior al número de versión del objeto, o de lo contrario ignora el objeto

Los casos de uso para esta funcionalidad incluyen reconciliar datos ingresados en instancias diferentes de bases de datos, actualizar información de configuración del sistema durante actualizaciones de producto, deshacer cambios realizados durante transacciones no-ACID y más.

## 11.10. Limpieza (flushing) de la sesión

A veces la `Session` ejecutará las declaraciones SQL necesarias para sincronizar el estado de la conexión JDBC con el estado de los objetos en la menoria. Este proceso, denominado *vaciado (flush)*, ocurre por defecto en los siguientes puntos:

- antes de algunas ejecuciones de consulta
- desde `org.hibernate.Transaction.commit()`
- desde `Session.flush()`

Las declaraciones SQL se emiten en el siguiente orden:

1. todas las inserciones de entidades, en el mismo orden que los objetos correspondientes fueron guardados utilizando `Session.save()`
2. todas las actualizaciones de entidades
3. todas los borrados de colecciones
4. todos los borrados, actualizaciones e inserciones de elementos de colección
5. todas las inserciones de colecciones
6. todos los borrados de entidades, en el mismo orden que los objetos correspondientes fueron borrados usando `Session.delete()`

Una excepción es que los objetos que utilizan generación de ID `native` se insertan cuando se guardan.

Excepto cuando llama explícitamente a `flush()`, no hay en absoluto garantías sobre *cuándo* la `Session` ejecuta las llamadas JDBC, sólamente sobre el *orden* en que se ejecutan. Sin embargo, Hibernate garantiza que los métodos `Query.list(..)` nunca devolverán datos desactualizados o incorrectos.

It is possible to change the default behavior so that flush occurs less frequently. The `FlushMode` class defines three different modes: only flush at commit time when the Hibernate `Transaction` API is used, flush automatically using the explained routine, or never flush unless `flush()` is called explicitly. The last mode is useful for long running units of work, where a `Session` is kept open and disconnected for a long time (see *Sección 13.3.2, "Sesión extendida y versionado automático"*).

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // allow queries to return stale state

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

// might return stale data
sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();
```

During flush, an exception might occur (e.g. if a DML operation violates a constraint). Since handling exceptions involves some understanding of Hibernate's transactional behavior, we discuss it in *Capítulo 13, Transacciones y concurrencia*.

## 11.11. Persistencia transitiva

Es absolutamente incómodo guardar, borrar, o reunir objetos individuales, especialmente si trata con un grafo de objetos asociados. Un caso común es una relación padre/hijo. Considere el siguiente ejemplo:

Si los hijos en una relación padre/hijo pudieran ser tipificados en valor (por ejemplo, una colección de direcciones o cadenas), sus ciclos de vida dependerían del padre y no se requeriría ninguna otra acción para el tratamiento apropiado en "cascada" de los cambios de estado. Cuando se guarda el padre, los objetos hijo tipificados en valor también se guardan, cuando se borra el padre, se borran los hijos, etc. Esto funciona incluso para operaciones tales como el retiro de un hijo de la colección. Hibernate detectará esto y ya que los objetos tipificados en valor no pueden tener referencias compartidas entonces borrará el hijo de la base de datos.

Ahora considere el mismo escenario con los objetos padre e hijos siendo entidades, no tipos de valor (por ejemplo, categorías e ítems, o gatos padres e hijos). Las entidades tienen su propio ciclo de vida y soportan referencias compartidas. El eliminar una entidad de una colección no significa que se pueda borrar, y no hay por defecto ningún tratamiento en "cascada" del estado

de una entidad a otras entidades asociadas. Hibernate no implementa por defecto la *persistencia por alcance*.

Para cada operación básica de la sesión de Hibernate - incluyendo `persist()`, `merge()`, `saveOrUpdate()`, `delete()`, `lock()`, `refresh()`, `evict()`, `replicate()` - existe un estilo de cascada correspondiente. Respectivamente, los estilos de cascada se llaman `create`, `merge`, `save-update`, `delete`, `lock`, `refresh`, `evict`, `replicate`. Si quiere que una operación sea tratada en cascada a lo largo de una asociación, debe indicar eso en el documento de mapeo. Por ejemplo:

```
<one-to-one name="person" cascade="persist"/>
```

Los estilos de cascada pueden combinarse:

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

Incluso puede utilizar `cascade="all"` para especificar que *todas* las operaciones deben ser tratadas en cascada a lo largo de la asociación. La `cascade="none"` predeterminada especifica que ninguna operación se tratará en cascada.

In case you are using annotatons you probably have noticed the `cascade` attribute taking an array of `CascadeType` as a value. The cascade concept in JPA is very is similar to the transitive persistence and cascading of operations as described above, but with slightly different semantics and cascading types:

- `CascadeType.PERSIST`: cascades the persist (create) operation to associated entities persist() is called or if the entity is managed
- `CascadeType.MERGE`: cascades the merge operation to associated entities if merge() is called or if the entity is managed
- `CascadeType.REMOVE`: cascades the remove operation to associated entities if delete() is called
- `CascadeType.REFRESH`: cascades the refresh operation to associated entities if refresh() is called
- `CascadeType.DETACH`: cascades the detach operation to associated entities if detach() is called
- `CascadeType.ALL`: all of the above

**Nota**

CascadeType.ALL also covers Hibernate specific operations like save-update, lock etc...

A special cascade style, `delete-orphan`, applies only to one-to-many associations, and indicates that the `delete()` operation should be applied to any child object that is removed from the association. Using annotations there is no `CascadeType.DELETE-ORPHAN` equivalent. Instead you can use the attribute `orphanRemoval as seen in` *Ejemplo 11.4, "@OneToMany with orphanRemoval"*. If an entity is removed from a `@OneToMany` collection or an associated entity is dereferenced from a `@OneToOne` association, this associated entity can be marked for deletion if `orphanRemoval` is set to true.

### Ejemplo 11.4. `@OneToMany` with `orphanRemoval`

```
@Entity
public class Customer {
   private Set<Order> orders;

   @OneToMany(cascade=CascadeType.ALL, orphanRemoval=true)
   public Set<Order> getOrders() { return orders; }

   public void setOrders(Set<Order> orders) { this.orders = orders; }

   [...]
}

@Entity
public class Order { ... }

Customer customer = em.find(Customer.class, 1l);
Order order = em.find(Order.class, 1l);
customer.getOrders().remove(order); //order will be deleted by cascade
```

Recomendaciones:

- It does not usually make sense to enable cascade on a many-to-one or many-to-many association. In fact the `@ManyToOne` and `@ManyToMany` don't even offer a `orphanRemoval` attribute. Cascading is often useful for one-to-one and one-to-many associations.
- If the child object's lifespan is bounded by the lifespan of the parent object, make it a *life cycle object* by specifying `cascade="all,delete-orphan"(@OneToMany(cascade=CascadeType.ALL, orphanRemoval=true))`.
- En otro caso, puede que usted no necesite tratamiento en cascada en absoluto. Pero si piensa que va a estar trabajando frecuentemente con padre e hijos juntos en la misma transacción, y quiere ahorrarse algo de escritura en computador, considere el utilizar `cascade="persist,merge,save-update"`.

Mapear una asociación (ya sea una asociación monovaluada, o una colección) con `cascade="all"` marca la asociación como una relación del estilo *padre/hijo* en donde guardar/actualizar/borrar (save/update/delete) el padre causa el guardar/actualizar/borrar del hijo o hijos.

Furthermore, a mere reference to a child from a persistent parent will result in save/update of the child. This metaphor is incomplete, however. A child which becomes unreferenced by its

parent is *not* automatically deleted, except in the case of a one-to-many association mapped with `cascade="delete-orphan"`. The precise semantics of cascading operations for a parent/child relationship are as follows:

- Si un padre pasa a `persist()`, se pasan todos los hijos a `persist()`
- Si un padre pasa a `merge()`, se pasan todos los hijos a `merge()`
- Si se pasa un padre a `save()`, `update()` o `saveOrUpdate()`, todos los hijos pasan a `saveOrUpdate()`
- Si un hijo transitorio o separado se vuelve referenciado por un padre persistente, le es pasado a `saveOrUpdate()`
- Si se borra un padre, se pasan todos los hijos a `delete()`
- Si un hijo deja de ser referenciado por un padre persistente, *no ocurre nada especial* - la aplicación debe borrar explícitamente el hijo de ser necesario - a menos que `cascade="delete-orphan"`, en cuyo caso se borra el hijo "huérfano".

Finalmente, note que las operaciones en cascadas se pueden aplicar a un grafo de objeto en *tiempo de llamada* o en *tiempo de vaciado*. Todas las operaciones, si se encuentran activadas se tratan en cascadas en entidades asociadas alcanzables cuando se ejecuta la operación. Sin embargo, `save-upate` y `delete-orphan` son transitivos para todas las entidades asociadas alcanzables durante el vaciado de la `Session`.

## 11.12. Utilización de metadatos

Hibernate requiere de un modelo de meta-nivel muy rico de todas las entidades y tipos de valor. Este modelo puede ser útil para la aplicación misma. Por ejemplo, la aplicación podría utilizar los metadatos de Hibernate para implementar un algoritmo "inteligente" de copia en profundidad que entienda qué objetos deben ser copiados (por ejemplo, tipos de valor mutables) y cuáles no (por ejemplo, tipos de valor inmutables y posiblemente las entidades asociadas).

Hibernate expone los metadatos por medio de las interfaces `ClassMetadata` y `CollectionMetadata` y la jerarquía `Type`. Las instancias de las interfaces de metadatos se pueden obtener de la `SessionFactory`.

```
Cat fritz = ......;
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);

Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();

// get a Map of all properties which are not collections or associations
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

# Read-only entities

**Importante**

Hibernate's treatment of *read-only* entities may differ from what you may have encountered elsewhere. Incorrect usage may cause unexpected results.

When an entity is read-only:

- Hibernate does not dirty-check the entity's simple properties or single-ended associations;
- Hibernate will not update simple properties or updatable single-ended associations;
- Hibernate will not update the version of the read-only entity if only simple properties or single-ended updatable associations are changed;

In some ways, Hibernate treats read-only entities the same as entities that are not read-only:

- Hibernate cascades operations to associations as defined in the entity mapping.
- Hibernate updates the version if the entity has a collection with changes that dirties the entity;
- A read-only entity can be deleted.

Even if an entity is not read-only, its collection association can be affected if it contains a read-only entity.

For details about the affect of read-only entities on different property and association types, see *Sección 12.2, “Read-only affect on property type”*.

For details about how to make entities read-only, see *Sección 12.1, “Making persistent entities read-only”*

Hibernate does some optimizing for read-only entities:

- It saves execution time by not dirty-checking simple properties or single-ended associations.
- It saves memory by deleting database snapshots.

## 12.1. Making persistent entities read-only

Only persistent entities can be made read-only. Transient and detached entities must be put in persistent state before they can be made read-only.

Hibernate provides the following ways to make persistent entities read-only:

- you can map an entity class as *immutable*; when an entity of an immutable class is made persistent, Hibernate automatically makes it read-only. see *Sección 12.1.1, "Entities of immutable classes"* for details
- you can change a default so that entities loaded into the session by Hibernate are automatically made read-only; see *Sección 12.1.2, "Loading persistent entities as read-only"* for details
- you can make an HQL query or criteria read-only so that entities loaded when the query or criteria executes, scrolls, or iterates, are automatically made read-only; see *Sección 12.1.3, "Loading read-only entities from an HQL query/criteria"* for details
- you can make a persistent entity that is already in the in the session read-only; see *Sección 12.1.4, "Making a persistent entity read-only"* for details

## 12.1.1. Entities of immutable classes

When an entity instance of an immutable class is made persistent, Hibernate automatically makes it read-only.

An entity of an immutable class can created and deleted the same as an entity of a mutable class.

Hibernate treats a persistent entity of an immutable class the same way as a read-only persistent entity of a mutable class. The only exception is that Hibernate will not allow an entity of an immutable class to be changed so it is not read-only.

## 12.1.2. Loading persistent entities as read-only

**Nota**

Entities of immutable classes are automatically loaded as read-only.

To change the default behavior so Hibernate loads entity instances of mutable classes into the session and automatically makes them read-only, call:

```
Session.setDefaultReadOnly( true );
```

To change the default back so entities loaded by Hibernate are not made read-only, call:

```
Session.setDefaultReadOnly( false );
```

You can determine the current setting by calling:

```
Session.isDefaultReadOnly();
```

If Session.isDefaultReadOnly() returns true, entities loaded by the following are automatically made read-only:

- Session.load()
- Session.get()
- Session.merge()
- executing, scrolling, or iterating HQL queries and criteria; to override this setting for a particular HQL query or criteria see *Sección 12.1.3, "Loading read-only entities from an HQL query/criteria"*

Changing this default has no effect on:

- persistent entities already in the session when the default was changed
- persistent entities that are refreshed via Session.refresh(); a refreshed persistent entity will only be read-only if it was read-only before refreshing
- persistent entities added by the application via Session.persist(), Session.save(), and Session.update() Session.saveOrUpdate()

## 12.1.3. Loading read-only entities from an HQL query/criteria

**Nota**

Entities of immutable classes are automatically loaded as read-only.

If Session.isDefaultReadOnly() returns false (the default) when an HQL query or criteria executes, then entities and proxies of mutable classes loaded by the query will not be read-only.

You can override this behavior so that entities and proxies loaded by an HQL query or criteria are automatically made read-only.

For an HQL query, call:

```
Query.setReadOnly( true );
```

`Query.setReadOnly( true )` must be called before `Query.list()`, `Query.uniqueResult()`, `Query.scroll()`, or `Query.iterate()`

For an HQL criteria, call:

```
Criteria.setReadOnly( true );
```

`Criteria.setReadOnly( true )` must be called before `Criteria.list()`, `Criteria.uniqueResult()`, or `Criteria.scroll()`

Entities and proxies that exist in the session before being returned by an HQL query or criteria are not affected.

Uninitialized persistent collections returned by the query are not affected. Later, when the collection is initialized, entities loaded into the session will be read-only if Session.isDefaultReadOnly() returns true.

Using `Query.setReadOnly( true )` or `Criteria.setReadOnly( true )` works well when a single HQL query or criteria loads all the entities and intializes all the proxies and collections that the application needs to be read-only.

When it is not possible to load and initialize all necessary entities in a single query or criteria, you can temporarily change the session default to load entities as read-only before the query is executed. Then you can explicitly initialize proxies and collections before restoring the session default.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

setDefaultReadOnly( true );
Contract contract =
   ( Contract ) session.createQuery(
           "from Contract where customerName = 'Sherman'" )
           .uniqueResult();
Hibernate.initialize( contract.getPlan() );
Hibernate.initialize( contract.getVariations() );
Hibernate.initialize( contract.getNotes() );
setDefaultReadOnly( false );
...
tx.commit();
session.close();
```

If Session.isDefaultReadOnly() returns true, then you can use Query.setReadOnly( false ) and Criteria.setReadOnly( false ) to override this session setting and load entities that are not read-only.

### 12.1.4. Making a persistent entity read-only

**Nota**

Persistent entities of immutable classes are automatically made read-only.

To make a persistent entity or proxy read-only, call:

```
Session.setReadOnly(entityOrProxy, true)
```

To change a read-only entity or proxy of a mutable class so it is no longer read-only, call:

```
Session.setReadOnly(entityOrProxy, false)
```

**Importante**

When a read-only entity or proxy is changed so it is no longer read-only, Hibernate assumes that the current state of the read-only entity is consistent with its database representation. If this is not true, then any non-flushed changes made before or while the entity was read-only, will be ignored.

To throw away non-flushed changes and make the persistent entity consistent with its database representation, call:

```
session.refresh( entity );
```

To flush changes made before or while the entity was read-only and make the database representation consistent with the current state of the persistent entity:

```
// evict the read-only entity so it is detached
session.evict( entity );

// make the detached entity (with the non-flushed changes) persistent
session.update( entity );

// now entity is no longer read-only and its changes can be flushed
s.flush();
```

## 12.2. Read-only affect on property type

The following table summarizes how different property types are affected by making an entity read-only.

### Tabla 12.1. Affect of read-only entity on property types

| Property/Association Type | Changes flushed to DB? |
|---|---|
| Simple | no* |

| Property/Association Type | Changes flushed to DB? |
|---|---|
| (*Sección 12.2.1, "Simple properties"*) | |
| Unidirectional one-to-one<br><br>Unidirectional many-to-one<br><br>(*Sección 12.2.2.1, "Unidirectional one-to-one and many-to-one"*) | no*<br><br>no* |
| Unidirectional one-to-many<br><br>Unidirectional many-to-many<br><br>(*Sección 12.2.2.2, "Unidirectional one-to-many and many-to-many"*) | yes<br><br>yes |
| Bidirectional one-to-one<br><br>(*Sección 12.2.3.1, "Bidirectional one-to-one"*) | only if the owning entity is not read-only* |
| Bidirectional one-to-many/many-to-one<br><br>inverse collection<br><br>non-inverse collection<br><br>(*Sección 12.2.3.2, "Bidirectional one-to-many/ many-to-one"*) | only added/removed entities that are not read-only*<br><br>yes |
| Bidirectional many-to-many<br><br>(*Sección 12.2.3.3, "Bidirectional many-to-many"*) | yes |

* Behavior is different when the entity having the property/association is read-only, compared to when it is not read-only.

## 12.2.1. Simple properties

When a persistent object is read-only, Hibernate does not dirty-check simple properties.

Hibernate will not synchronize simple property state changes to the database. If you have automatic versioning, Hibernate will not increment the version if any simple properties change.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

// get a contract and make it read-only
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );

// contract.getCustomerName() is "Sherman"
```

```
contract.setCustomerName( "Yogi" );
tx.commit();

tx = session.beginTransaction();

contract = ( Contract ) session.get( Contract.class, contractId );
// contract.getCustomerName() is still "Sherman"
...
tx.commit();
session.close();
```

## 12.2.2. Unidirectional associations

### 12.2.2.1. Unidirectional one-to-one and many-to-one

Hibernate treats unidirectional one-to-one and many-to-one associations in the same way when the owning entity is read-only.

We use the term *unidirectional single-ended association* when referring to functionality that is common to unidirectional one-to-one and many-to-one associations.

Hibernate does not dirty-check unidirectional single-ended associations when the owning entity is read-only.

If you change a read-only entity's reference to a unidirectional single-ended association to null, or to refer to a different entity, that change will not be flushed to the database.

**Nota**

If an entity is of an immutable class, then its references to unidirectional single-ended associations must be assigned when that entity is first created. Because the entity is automatically made read-only, these references can not be updated.

If automatic versioning is used, Hibernate will not increment the version due to local changes to unidirectional single-ended associations.

In the following examples, Contract has a unidirectional many-to-one association with Plan. Contract cascades save and update operations to the association.

The following shows that changing a read-only entity's many-to-one association reference to null has no effect on the entity's database representation.

```
// get a contract with an existing plan;
// make the contract read-only and set its plan to null
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
```

```
session.setReadOnly( contract, true );
contract.setPlan( null );
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );

// contract.getPlan() still refers to the original plan;

tx.commit();
session.close();
```

The following shows that, even though an update to a read-only entity's many-to-one association has no affect on the entity's database representation, flush still cascades the save-update operation to the locally changed association.

```
// get a contract with an existing plan;
// make the contract read-only and change to a new plan
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );
Plan newPlan = new Plan( "new plan"
contract.setPlan( newPlan);
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );
newPlan = ( Contract ) session.get( Plan.class, newPlan.getId() );

// contract.getPlan() still refers to the original plan;
// newPlan is non-null because it was persisted when
// the previous transaction was committed;

tx.commit();
session.close();
```

### 12.2.2.2. Unidirectional one-to-many and many-to-many

Hibernate treats unidirectional one-to-many and many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

Hibernate dirty-checks unidirectional one-to-many and many-to-many associations;

The collection can contain entities that are read-only, as well as entities that are not read-only.

Entities can be added and removed from the collection; changes are flushed to the database.

If automatic versioning is used, Hibernate will update the version due to changes in the collection if they dirty the owning entity.

## 12.2.3. Bidirectional associations

### 12.2.3.1. Bidirectional one-to-one

If a read-only entity owns a bidirectional one-to-one association:

- Hibernate does not dirty-check the association.
- updates that change the association reference to null or to refer to a different entity will not be flushed to the database.
- If automatic versioning is used, Hibernate will not increment the version due to local changes to the association.

**Nota**

If an entity is of an immutable class, and it owns a bidirectional one-to-one association, then its reference must be assigned when that entity is first created. Because the entity is automatically made read-only, these references cannot be updated.

When the owner is not read-only, Hibernate treats an association with a read-only entity the same as when the association is with an entity that is not read-only.

### 12.2.3.2. Bidirectional one-to-many/many-to-one

A read-only entity has no impact on a bidirectional one-to-many/many-to-one association if:

- the read-only entity is on the one-to-many side using an inverse collection;
- the read-only entity is on the one-to-many side using a non-inverse collection;
- the one-to-many side uses a non-inverse collection that contains the read-only entity

When the one-to-many side uses an inverse collection:

- a read-only entity can only be added to the collection when it is created;
- a read-only entity can only be removed from the collection by an orphan delete or by explicitly deleting the entity.

### 12.2.3.3. Bidirectional many-to-many

Hibernate treats bidirectional many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

Hibernate dirty-checks bidirectional many-to-many associations.

The collection on either side of the association can contain entities that are read-only, as well as entities that are not read-only.

Entities are added and removed from both sides of the collection; changes are flushed to the database.

If automatic versioning is used, Hibernate will update the version due to changes in both sides of the collection if they dirty the entity owning the respective collections.

# Transacciones y concurrencia

El punto más importante sobre Hibernate y el control de concurrencia es que es fácil de comprender. Hibernate usa directamente conexiones JDBC y recursos JTA sin agregar ningún comportamiento de bloqueo adicional. Le recomendamos bastante que tome algo de tiempo con la especificación de JDBC, ANSI y el aislamiento de transacciones de su sistema de gestión de base de datos.

Hibernate no bloquea objetos en la memoria. Su aplicación puede esperar el comportamiento definido por el nivel de aislamiento de sus transacciones de las bases de datos. Gracias a la `Session`, la cual también es un caché con alcance de transacción, Hibernate proporciona lecturas repetidas para búsquedas del identificador y consultas de entidad y no consultas de reporte que retornan valores escalares.

Además del versionado del control de concurrencia optimista automático, Hibernate también ofrece una API (menor) para bloqueo pesimista de filas, usando la sintáxis `SELECT FOR UPDATE`. Esta API y el control de concurrencia optimista se discuten más adelante en este capítulo.

Comenzamos la discusión del control de concurrencia en Hibernate con la granularidad de `Configuration`, `SessionFactory` y `Session`, así como las transacciones de la base de datos y las conversaciones largas.

## 13.1. Ámbitos de sesión y de transacción

Una `SessionFactory` es un objeto seguro entre hilos y costoso de crear pensado para que todas las hebras de la aplicación lo compartan. Se crea una sola vez, usualmente en el inicio de la aplicación, a partir de una instancia `Configuration`.

Una `Session` es un objeto de bajo costo, inseguro entre hilos que se debe utilizar una sola vez y luego se debe descartar: para un sólo pedido, una sola conversación o una sóla unidad de trabajo. Una `Session` no obtendrá una `Connection` JDBC o un `Datasource` a menos de que sea necesario. No consumirá recursos hasta que se utilice.

Una transacción de la base de datos tiene que ser tan corta como sea posible para reducir la contención de bloqueos en la base de datos. Las transacciones largas de la base de datos prevendrán a su aplicación de escalar a una carga altamente concurrente. Por lo tanto, no se recomienda que mantenga una transacción de la base de datos abierta durante el tiempo para pensar del usuario, hasta que la unidad de trabajo se encuentre completa.

¿Cuál es el ámbito de una unidad de trabajo? ¿Puede una sola `Session` de Hibernate extenderse a través de varias transacciones de la base de datos o ésta es una relación uno-a-uno de ámbitos? ¿Cuándo debe abrir y cerrar una `Session`? y ¿cómo demarca los límites de la transacción de la base de datos? En las siguientes secciones abordaremos estas preguntas.

### 13.1.1. Unidad de trabajo

First, let's define a unit of work. A unit of work is a design pattern described by Martin Fowler as " [maintaining] a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems. "[PoEAA] In other words, its a series of operations we wish to carry out against the database together. Basically, it is a transaction, though fulfilling a unit of work will often span multiple physical database transactions (see *Sección 13.1.2, "Conversaciones largas"*). So really we are talking about a more abstract notion of a transaction. The term "business transaction" is also sometimes used in lieu of unit of work.

Primero, no use el antipatrón *sesión-por-operación*: no abra y cierre una `Session` para cada llamada simple a la base de datos en un solo hilo. Lo mismo aplica para las transacciones de base de datos. Las llamadas a la base de datos en una aplicación se hacen usando una secuencia planeada; estas se agrupan dentro de unidades de trabajo atómicas. Esto también significa que el auto-commit después de cada una de las declaraciones SQL es inútil en una aplicación ya que este modo está pensado para trabajo ad-hoc de consola SQL. Hibernate deshabilita, o espera que el servidor de aplicaciones lo haga, el modo auto-commit inmediatamente. Las transacciones de las bases de datos nunca son opcionales. Toda comunicación con una base de datos tiene que ocurrir dentro de una transacción. El comportamiento auto-commit para leer datos se debe evitar, ya que hay muy poca probabilidad de que las transacciones pequeñas funcionen mejor que una unidad de trabajo definida claramente. La última es mucho más sostenible y extensible.

El patrón más común en una aplicación multiusuario cliente/servidor es *sesión-por-petición*. En este modelo, una petición del cliente se envia al servidor, en donde se ejecuta la capa de persistencia de Hibernate. Se abre una nueva `Session` de Hibernate y todas las operaciones de la base de datos se ejecutan en esta unidad de trabajo. Una vez completado el trabajo, y una vez se ha preparado la respuesta para el cliente, se limpia la sesión y se cierra. Use una sóla transacción de la base de datos para servir la petición del cliente, dándole inicio y guardándola cuando abre y cierra la `Session`. La relación entre las dos es uno-a-uno y este modelo es a la medida perfecta de muchas aplicaciones.

El reto se encuentra en la implementación. Hibernate brinda administración incorporada de la "sesión actual" para simplificar este patrón. Inicie una transacción cuando se tiene que procesar un pedido del servidor y termine la transacción antes de que se envie la respuesta al cliente. Las soluciones más comunes son `ServletFilter`, un interceptor AOP con un punto de corte en los métodos del servicio o un contenedor proxy/intercepción. Un contenedor EJB es una manera estandarizada de implementar aspectos de doble filo como demarcación de transacción en beans de sesión EJB, declarativamente con CMT. Si decide utilizar la demarcación de transacción programática, use el API `Transaction` de Hibernate de fácil uso y portable que se muestra más adelante en este capítulo.

Your application code can access a "current session" to process the request by calling `sessionFactory.getCurrentSession()`. You will always get a `Session` scoped to the current database transaction. This has to be configured for either resource-local or JTA environments, see *Sección 2.3, "Sesiones contextuales"*.

Puede extender el ámbito de una `Session` y transacción de la base de datos hasta que "se ha presentado la vista". Esto es bastante útil en aplicaciones de servlet que utilizan una fase de entrega separada después de que se ha procesado el pedido. El extender la transacción de la base de datos hasta que la entrega de la vista se encuentre completa es fácil de lograr si implementa su propio interceptor. Sin embargo, no se logra fácilmente si depende de EJBs con transacciones administradas por el contenedor. Una transacción se completará cuando un método EJB retorna, antes de que pueda empezar la entrega de cualquier vista. Vea el sitio web de Hibernate y el foro para encontrar consejos y ejemplos sobre este patrón de *sesión abierta en vista*.

### 13.1.2. Conversaciones largas

El patrón sesión-por-petición no es la única forma de diseñar unidades de trabajo. Muchos procesos empresariales requieren una serie completa de interacciones con el usuario intercaladas con accesos a la base de datos. En aplicaciones empresariales y web no es aceptable que una transacción de la base de datos abarque la interacción de un usuario. Considere el siguiente ejemplo:

- Se abre la primera pantalla de un diálogo. Los datos que ve el usuario han sido cargados en una `Session` en particular y en una transacción de la base de datos. El usuario es libre de modificar los objetos.
- El usuario hace click en "Guardar" después de 5 minutos y espera que sus modificaciones se hagan persistentes. También espera que él sea la única persona editando esta información y que no ocurra ningún conflicto en la modificación.

Desde el punto de vista del usuario, llamamos a esta unidad de trabajo, una larga *conversación* o *transacción de aplicación*. Hay muchas formas de implementar esto en su aplicación.

Una primera implementación ingenua podría mantener abierta la `Session` y la transacción de la base de datos durante el tiempo para pensar del usuario, con bloqueos en la base de datos para prevenir la modificación simultánea y para garantizar el aislamiento y la atomicidad. Esto es un antipatrón, ya que la contención de bloqueo no permitiría a la aplicación escalar con el número de usuarios simultáneos.

Tiene que usar varias transacciones de la base de datos para implementar la conversación. En este caso, mantener el aislamiento de los procesos empresariales se vuelve una responsabilidad parcial de la capa de la aplicación. Una sóla conversación usualmente abarca varias transacciones de la base de datos. Será atómica si sólo una de estas transacciones de la base de datos (la última) almacena los datos actualizados. Todas las otras simplemente leen datos (por ejemplo, en un diálogo de estilo-asistente abarcando muchos ciclos petición/respuesta). Esto es más fácil de implementar de lo que suena, especialmente si usa las funcionalidades de Hibernate:

- *Versionado automático* - Hibernate puede realizar un control automático de concurrencia optimista por usted .Puede detectar automáticamente si ha ocurrido una modificación simultánea durante el tiempo para pensar del usuario. Chequee esto al final de la conversación.

- *Objetos separados*: Si decide usar el patrón *sesión-por-petición*, todas las instancias cargadas estarán en estado separado durante el tiempo para pensar del usuario. Hibernate le permite volver a unir los objetos y hacer persistentes las modificaciones. El patrón se llama *sesión-por-petición-con-objetos-separados*. Se usa el versionado automático para aislar las modificaciones simultáneas.
- *Sesión extendida (o larga)* - La `Session` de Hibernate puede ser desconectada de la conexión JDBC subyacente después de que haya guardado la transacción de la base de datos y haya reconectado cuando ocurra una nueva petición del cliente. Este patrón se conoce como *sesión-por-conversación* y hace la re-unión innecesaria. Para aislar las modificaciones simultáneas se usa el versionado automático y usualmente no se permite que se limpie la `Session` automáticamente sino explícitamente.

Tanto la *sesión-por-petición-con-objetos-separados* como la *sesión-por-conversación* tienen ventajas y desventajas. Estas desventajas las discutimos más adelante en este capítulo en el contexto del control optimista de concurrencia.

### 13.1.3. Consideración de la identidad del objeto

Una aplicación puede acceder simultáneamente al mismo estado persistente en dos `Session`es diferentes. Sin embargo, una instancia de una clase persistente nunca se comparte entre dos instancias de `Session`. Por lo tanto, existen dos nociones diferentes de identidad:

Identidad de Base de Datos
```
foo.getId().equals( bar.getId() )
```

Identidad JVM
```
foo==bar
```

Para los bjetos unidos a una `Session` *en particular* (por ejemplo, en el ámbito de una `Session`) las dos nociones son equivalentes y la identidad de la MVJ para la identidad de la base de datos se encuentra garantizada por Hibernate. Mientras la aplicación acceda simultáneamente al "mismo" objeto empresarial (identidad persistente) en dos sesiones diferentes, las dos instancias serán realmente "diferentes" (identidad MVJ). Los conflictos se resuelven usando un enfoque optimista y el versionado automático en tiempo de vaciado/ al guardar.

Este enfoque deja que Hibernate y la base de datos se preocupen de la concurrencia. Además provee la mejor escalabilidad, ya que garantizando la identidad en unidades de trabajo monohilo no se necesitan bloqueos caros u otros medios de sincronización. La aplicación no necesita sincronizar sobre ningún objeto empresarial, siempre que se mantenga un solo hilo por `Session`. Dentro de una `Session` la aplicación puede usar con seguridad `==` para comparar objetos.

Sin embargo, una aplicación que usa `==` fuera de una `Session`, podría ver resultados inesperados. Esto podría ocurrir incluso en sitios algo inesperados. Por ejemplo, si pone dos instancias separadas dentro del mismo `Set` ambas podrían tener la misma identidad de la base de datos (por ejemplo, representar la misma fila). Sin embargo, la identidad MVJ, por definición, no está garantizada para las instancias en estado separado. El desarrollador tiene que sobrescribir los métodos `equals()` y `hashCode()` en las clases persistentes e implementar su propia noción

de igualdad de objetos. Hay una advertencia: nunca use el identificador de la base de datos para implementar la igualdad. Use una clave de negocio, una combinación de atributos únicos, usualmente inmutables. El identificador de la base de datos cambiará si un objeto transitorio es hecho persistente. Si la instancia transitoria (usualmente junto a las instancias separadas) es mantenida en un `Set`, cambiar el código hash rompe el contrato del `Set`. Los atributos para las claves empresariales no tienen que ser tan estables como las claves principales de la base de datos, sólo tiene que garantizar estabilidad en tanto los objetos estén en el mismo `Set`. Mire el sitio web de Hibernate para obetener una discusión más profunda de este tema. Note también que éste no es problema de Hibernate, sino que simplemente se tiene que implementar la identidad y la igualdad de los objetos Java.

### 13.1.4. Temas comúnes

No use los antipatrones *sesión-por-sesión-de-usuario* o *sesión-por-aplicación* (hay excepciones raras a esta regla). Algunos de los siguientes temas también podrían aparecer con los patrones recomendados así que asegúrese de que entiende las implicaciones antes de tomar una decisión de diseño:

- Una `Session` no es segura entre hilos. Las cosas que funcionan de manera simultánea, como las peticiones HTTP, beans de sesión, o workers de Swing, provocarán condiciones de competencia si una instancia de `Session` se comparte. Si guarda su `Session` de Hibernate en su `HttpSession` (se discute más adelante), debe considerar el sincronizar el acceso a su sesión HTTP. De otro modo, un usuario que hace click lo suficientemente rápido puede llegar a usar la misma `Session` en dos hilos ejecutándose simultáneamente.
- Una excepción lanzada por Hibernate significa que tiene que deshacer su transacción de la base de datos y cerrar la `Session` inmediatamente (se discute en más detalle más adelante en este capítulo). Si su `Session` está vinculada a la aplicación, tiene que parar la aplicación. Deshacer la transacción de la base de datos no pone a sus objetos de vuelta al estado en que estaban al comienzo de la transacción. Esto significa que el estado de la base de datos y los objetos empreariales quedan fuera de sincronía. Usualmente esto no es un problema, pues las excepciones no son recuperables y tendrá que volver a comenzar después de deshacer de todos modos.
- The `Session` caches every object that is in a persistent state (watched and checked for dirty state by Hibernate). If you keep it open for a long time or simply load too much data, it will grow endlessly until you get an OutOfMemoryException. One solution is to call `clear()` and `evict()` to manage the `Session` cache, but you should consider a Stored Procedure if you need mass data operations. Some solutions are shown in *Capítulo 15, Procesamiento por lotes*. Keeping a `Session` open for the duration of a user session also means a higher probability of stale data.

## 13.2. Demarcación de la transacción de la base de datos

Los límites de las transacciones de la base de datos o el sistema son siempre necesarios. Ninguna comunicación con la base de datos puede darse fuera de una transacción de la base de datos

(esto parece confundir a muchos desarrolladores acostumbrados al modo auto-commit). Siempre use límites de transacción claros, incluso para las operaciones de sólo lectura. Dependiendo del nivel de aislamiento y las capacidades de la base de datos, esto podría requerirse o no, pero no hay inconvenientes si siempre demarca explícitamente las transacciones. Con seguridad, una transacción única de base de datos va a funcionar mejor que muchas transacciones pequeñas, inclusive para leer datos.

Una aplicación Hibernate puede ejecutarse en entornos no administrados (por ejemplo, aplicaciones simples Web o Swing autónomas) y entornos administrados por J2EE. En un entorno no administrado, Hibernate es usualmente responsable de su propio pool de conexiones de la base de datos. El desarrollador de aplicaciones tiene que establecer manualmente los límites de transacción (inicar, guardar o deshacer las transacciones de la base de datos) por sí mismo. Un entorno administrado usualmente proporciona transacciones gestionadas por contenedor, con el ensamble de transacción definido declarativamente (por ejemplo, en descriptores de despliegue de beans de sesión EJB). La demarcación programática de transacciones ya no es necesaria.

Sin embargo, comúnmente se quiere mantener su capa de persistencia portátil entre entornos locales- de recursos no-administrados y sistemas que pueden confiar en JTA, pero utilizar BMT en vez de CMT. En ambos casos utilizaría la demarcación de transacción programática. Hibernate ofrece una API de envoltura llamada `Transaction` que se traduce al sistema de transacciones nativo de su entorno de despliegue. Esta API es de hecho opcional, pero le recomendamos bastante su uso salvo que esté en un bean de sesión CMT.

Usualmente, el finalizar una `Session` implica cuatro fases distintas:

- limpiar la sesión
- someter la transacción
- cerrar la sesión
- manejar excepciones

Anteriormente se discutió el vacido de la sesión así que ahora vamos a mirar más de cerca la demarcación de transacciones y el manejo de excepciones en los dos entornos administrado y no administrado.

### 13.2.1. Entorno no administrado

Si una capa de persistencia Hibernate se ejecuta en un entorno no administrado, las conexiones de la base de datos se manejan usualmente por simples pools de conexión (por ejemplo, no-DataSource) del cual Hibernate obtiene conexiones al ser necesario. El idioma de manejo de sesión/transacción se ve así:

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
```

```
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

No tiene que vaciar con `flush()` la `Session` explícitamente: la llamada a `commit()` automáticamente dispara la sincronización dependiendo del *FlushMode* para la sesión. Una llamada a `close()` marca el final de una sesión. La implicación principal de `close()` es que la conexión JDBC será abandonada por la sesión. Este código Java es portátil y ejecuta en entornos tanto no-administrados como JTA.

Como se mencionó anteriormente, una solución mucho más flexible es la administración de contexto "sesión actual" incorporada en Hibernate:

```
// Non-managed environment idiom with getCurrentSession()
try {
    factory.getCurrentSession().beginTransaction();

    // do some work
    ...

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

No verá estos pedazos de código en una aplicación normal; las excepciones fatales (del sistema) siempre deben ser capturadas en la "cima". En otras palabras, el código que ejecuta las llamadas de Hibernate en la capa de persistencia y el código que maneja `RuntimeException` (y usualmente sólo puede limpiar y salir) se encuentran en capas diferentes. La administración de contexto actual de Hibernate puede simplificar de manera importante este diseño, ya que todo lo que necesita hacer es acceder a `SessionFactory`. El manejo de excepciones se discute más adelante en este capítulo.

Debe seleccionar `org.hibernate.transaction.JDBCTransactionFactory`, el cual es el predeterminado, y para el segundo ejemplo seleccionar `"thread"` como su `hibernate.current_session_context_class`.

### 13.2.2. Utilización de JTA

Si su capa de persistencia se ejecuta en un servidor de aplicaciones (por ejemplo, detrás de los beans de sesión EJB), cada conexión de fuente de datos obtenida por Hibernate será parte de la transacción JTA global de manera automática. También puede instalar una implementación JTA autónoma y utilizarla sin EJB. Hibernate ofrece dos estrategias para esta integración JTA.

Si usa transacciones gestionadas-por-bean (BMT) Hibernate le dirá al servidor de aplicaciones que comience y finalice una transacción BMT si usa la API de `Transaction`. De modo que, el código de gestión de la transacción es idéntico al de un entorno no administrado.

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Si quiere utilizar un vínculo de transacción `Session`, es decir, la funcionalidad `getCurrentSession()` para propagación de contexto de manera fácil, tendrá que utilizar el API `UserTransaction` del JTA directamente:

```
// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
                            .lookup("java:comp/UserTransaction");

    tx.begin();

    // Do some work on Session bound to transaction
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

Con CMT, la demarcación de transacción se realiza en los descriptores de implementacion bean de sesión, no programáticamente. Por lo tanto el código se reduce a:

```
// CMT idiom
 Session sess = factory.getCurrentSession();

 // do some work
 ...
```

En un CMT/EJB incluso el deshacer sucede de forma automática. Un `RuntimeException` lanzado por un método bean de sesión le dice al contenedor que establezca una transacción global para deshacer. *No necesita utilizar el API `Transaction` de Hibernate con BMT o CMT y obtiene la propagación automática de sesión"actual" vinculada a la transacción.*

Al configurar la fábrica de transacciones de Hibernate, escoja `org.hibernate.transaction.JTATransactionFactory` si utiliza JTA directamente (BMT) y `org.hibernate.transaction.CMTTransactionFactory` en una bean de sesión CMT. Recuerde establecer también `hibernate.transaction.manager_lookup_class`. Asegúrese de que su `hibernate.current_session_context_class` no se encuentra configurado (compatibilidad retrasada) o configurada como `"jta"`.

La operación `getCurrentSession()` tiene un inconveniente en un entorno JTA. Hay una desventaja en el uso del modo de liberación de la conección `after_statement`, la cual luego se utiliza por defecto. Debido a una limitación de la especificación JTA, no le es posible a Hibernate limpiar automáticamente cualquier instancia `ScrollableResults` o `Iterator` no cerradas y retornadas por `scroll()` o `iterate()`. *Tiene* que liberar el cursor de la base de datos subyacente llamando a `ScrollableResults.close()` o `Hibernate.close(Iterator)` explícitamente desde un bloque `finally`. La mayoría de las aplicaciones pueden evitar fácilmente el utilizar `scroll()` o `iterate()` del código JTA o CMT.

### 13.2.3. Manejo de excepciones

Si la `Session` lanza una excepción, incluyendo cualquier `SQLException`, debe deshacer inmediatamente la transacción de la base de datos, llamar a `Session.close()` y descartar la instancia de `Session`. Ciertos métodos de `Session` *no* dejarán la sesión en un estado consistente. Ninguna excepción lanzada por Hibernate puede ser tratada como recuperable. Asegúrese de que la `Session` se cierre llamando a `close()` en un bloque `finally`.

La `HibernateException`, que envuelve a la mayoría de los errores que pueden ocurrir en la capa de persistencia de Hibernate, es una excepción no chequeada. No lo era en versiones anteriores de Hibernate. En nuestra opinión, no debemos forzar al desarrollador de aplicaciones a capturar una excepción irrecuperable en una capa baja. En la mayoría de los sistemas, las excepciones no chequeadas y fatales son manejadas en uno de los primeros cuadros de la pila de llamadas a métodos (por ejemplo, en las capas más altas) y presenta un mensaje de error al usuario de la aplicación o se toma alguna otra acción apropiada. Note que Hibernate podría

también lanzar otras excepciones no chequeadas que no sean una `HibernateException`. Estas no son recuperables y debe tomarse una acción apropiada.

Hibernate envuelve `SQLException`s lanzadas mientras se interactúa con la base de datos en una `JDBCException`. De hecho, Hibernate intentará convertir la excepción en una subclase de `JDBCException` más significativa. La `SQLException` subyacente siempre está disponible por medio de `JDBCException.getCause()`. Hibernate convierte la `SQLException` en una subclase de `JDBCException` apropiada usando el `SQLExceptionConverter` adjunto a la `SessionFactory`. Por defecto, el `SQLExceptionConverter` está definido por el dialecto configurado. Sin embargo, también es posible enchufar una implementación personalizada . Consulte los javadocs de la clase `SQLExceptionConverterFactory` para obtener más detalles. Los subtipos estándar de `JDBCException` son:

- `JDBCConnectionException`: indica un error con la comunicación JDBC subyacente.
- `SQLGrammarException`: indica un problema de gramática o sintáxis con el SQL publicado.
- `ConstraintViolationException`: indica alguna forma de violación de restricción de integridad.
- `LockAcquisitionException`: indica un error adquiriendo un nivel de bloqueo necesario para realizar una operación solicitada.
- `GenericJDBCException`: una excepción genérica que no encajó en ninguna de las otras categorías.

## 13.2.4. Tiempo de espera de la transacción

Una característica importante proporcionada por un entorno administrado como EJB que nunca es proporcionado para un código no-administrado, es el tiempo de espera de la transacción. Estos tiempos de espera se aseguran de que ninguna transacción que se comporte inapropiadamente pueda vincular recursos mientras no devuelva una respuesta al usuario. Fuera de un entorno administrado (JTA), Hibernate no puede proporcionar completamente esta funcionalidad. Sin embargo, Hibernate puede por lo menos controlar las operaciones de acceso de datos, asegurándose de que los bloqueos a nivel de base de datos y las consultas con grandes grupos de resultados se encuentran limitados por un tiempo de espera definido. En un entorno administrado, Hibernate puede delegar el tiempo de espera de la transacción a JTA. Esta funcionalidad es abstraída por el objeto `Transaction` de Hibernate.

```
Session sess = factory.openSession();
try {
    //set transaction timeout to 3 seconds
    sess.getTransaction().setTimeout(3);
    sess.getTransaction().begin();

    // do some work
    ...

    sess.getTransaction().commit()
}
```

```
catch (RuntimeException e) {
    sess.getTransaction().rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

`setTimeout()` no se puede llamar en un bean CMT, en donde se deben definir declarativamente los tiempos de espera de las transacciones.

## 13.3. Control de concurrencia optimista

El único enfoque consistente con una alta concurrencia y una alta escalabilidad es el control de concurrencia optimista con versionamiento. El chequeo de versión utiliza números de versión, o sellos de fecha (timestamps), para detectar actualizaciones en conflicto y para prevenir la pérdida de actualizaciones. Hibernate proporciona tres enfoques posibles de escribir código de aplicación que utilice concurrencia optimista. Los casos de uso que mostramos se encuentran en el contexto de conversaciones largas, pero el chequeo de versiones tiene además el beneficio de prevenir la pérdida de actualizaciones en transacciones individuales de la base de datos.

### 13.3.1. Chequeo de versiones de la aplicación

En una implementación que no tiene mucha ayuda de Hibernate, cada interacción con la base de datos ocurre en una nueva `Session` y el desarrollador es el responsable de recargar todas las intancias persistentes desde la base de datos antes de manipularlas. Este enfoque fuerza a la aplicación a realizar su propio chequeo de versiones para asegurar el aislamiento de transacciones de conversaciones. Este enfoque es el menos eficiente en términos de acceso a la base de datos. Es el enfoque más similar a los EJBs de entidad.

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();

int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion != foo.getVersion() ) throw new StaleObjectStateException();
foo.setProperty("bar");

t.commit();
session.close();
```

La propiedad `version` se mapea utilizando `<version>`, e Hibernate la incrementará automáticamente durante la limpieza si la entidad está desactualizada.

Si está operando un entorno de baja-concurrencia-de-datos y no requiere chequeo de versiones, puede usar este enfoque y simplemente saltarse el chequeo de versiones. En ese caso, *el último que guarda gana* y será la estrategia por defecto para conversaciones largas. Tenga en mente

que esto podría confundir a los usuarios de la aplicación, pues podrían experimentar pérdidas de actualizaciones sin mensajes de error ni oportunidad de fusionar los cambios conflictivos.

El chequeo manual de versiones es factible sólamente en circunstancias muy triviales y no es práctico para la mayoría de las aplicaciones. Con frecuencia se tienen que chequear no sólamente las intancias sólas, sino también grafos completos de objetos modificados. Hibernate ofrece el chequeo de versiones automático con el paradigma de diseño de `Session` larga o de instancias separadas.

## 13.3.2. Sesión extendida y versionado automático

Una sóla instancia de `Session` y sus instancias persistentes se utilizan para toda la convervsación conocida como *sesión-por-conversación*. Hibernate chequea las versiones de instancia en el momento de vaciado, lanzando una excepción si se detecta una modificación concurrente. Le concierne al desarrollador capturar y manejar esta excepción. Las opciones comunes son la oportunidad del usuario de fusionar los cambios, o de recomenzar el proceso empresarial sin datos desactualizados.

La `Session` se desconecta de cualquier conexión JDBC subyacente a la espera de una interacción del usuario. Este enfoque es el más eficiente en términos de acceso a la base de datos. La aplicación no necesita por sí misma tratar con el chequeo de versiones, ni re-unir instancias separadas, ni tiene que recargar instancias en cada transacción de la base de datos.

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC connection, start transaction

foo.setProperty("bar");

session.flush();    // Only for last transaction in conversation
t.commit();         // Also return JDBC connection
session.close();    // Only for last transaction in conversation
```

El objeto `foo` sabe en qué `Session` fue cargado. El dar inicio a una nueva base de datos en una sesión vieja obtiene una nueva conexión y reanuda la sesión. El guardar una transacción de la base de datos desconecta una sesión de la conexion JDBC y devuelve la conexión al pool. Después de la reconexión, para poder forzar una verificación de versión sobre datos que usted no está actalizando, puede llamar a `Session.lock()` con `LockMode.READ` en cualquier objeto que pueda haber sido actualizado por otra transacción. No necesita bloquear ningún dato que *sí esté* actualizando. Usualmente configuraría `FlushMode.MANUAL` en una `Session` extendida, de manera que de hecho sólamente se permite persistir el último ciclo de transacción de la base de datos de todas las modificaciones realizadas en esta conversación. Sólamente esta última transacción de la base de datos incluiría la operación `flush()` y luego cierra `-close()-` la sesión para dar fin a la conversación.

Este patrón es problemático si la `Session` es demasiado grande para almacenarla durante el tiempo para pensar del usuario, por ejemplo, una `HttpSession` se debe mantener tan pequeña como sea posible. Como la `Session` también lo es el caché de primer nivel (obligatorio)

y comprende todos los objetos cargados, probablemente podemos utilizar esta estrategia sólamente para unos pocos ciclos de pedido/respuesta. Debe utilizar una `Session` sólamente para una conversación única ya que pronto también tendrá datos añejos.

**Nota**

Las versiones anteriores de Hibernate necesitaban desconexión explícita y reconexión de una `Session`. Estos métodos ya no se aprueban ya que tienen el mismo efecto que dar inicio o finalizar a una transacción.

Mantenga la `Session` desconectada cerca a la capa de persistencia. Use un bean de sesión EJB con estado para mantener la `Session` en un entorno de tres capas . No la transfiera a la capa web ni la serialice en una capa separada para almacenarla en la `HttpSession`.

El patrón de sesión extendido, o *sesión-por-conversación*, es más dificil de implementar con la administración de contexto de sesión actual. Necesita proporcionar su propia implementación de la `CurrentSessionContext` para esto, vea el Wiki de Hibernate para obtener más ejemplos.

## 13.3.3. Objetos separados y versionado automático

Cada interacción con el almacenamiento persistente ocurre en una nueva `Session`. Sin embargo, las mismas instancias persistentes son reutilizadas para cada interacción con la base de datos. La aplicación manipula el estado de las instancias separadas cargadas originalmente en otra `Session` y luego las readjunta usando `Session.update()`, `Session.saveOrUpdate()`, o `Session.merge()`.

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have been loaded already
t.commit();
session.close();
```

De nuevo, Hibernate chequeará las versiones de la instancia durante el vaciado, lanzando una excepción si tuvieron lugar conflictos en las actualizaciones.

También puede llamar a `lock()` en lugar de `update()` y utilizar `LockMode.READ` (realizando un chequeo de versión, evitando todos los cachés) si está seguro de que el objeto no ha sido modificado.

## 13.3.4. Personalización del versionado automático

Puede deshabilitar el incremento de la versión automática de Hibernate para ciertas propiedades y colecciones en particular estableciendo el atributo de mapeo `optimistic-lock` como `false`.

Hibernate entonces ya no incrementará más las versiones si la propiedad se encuentra desactualizada.

Los esquemas heredados de la base de datos con frecuencia son estáticos y no pueden ser modificados. Inclusive otras aplicaciones podrían también acceder la misma base de datos y no saber cómo manejar los números de versión ni los sellos de fecha. En ambos casos, el versionado no puede confiarse a una columna en particular en una tabla. Para forzar un chequeo de versiones sin un mapeo de propiedad de versión o sello de fecha, con una comparación del estado de todos los campos en una fila, active `optimistic-lock="all"` en el mapeo de `<class>`. Esto funciona conceptualmente sólamente si Hibernate puede comparar el estado viejo y el nuevo, es decir, si usa una sóla `Session` larga y no sesión-por-petición-con-instancias-separadas.

Las modificaciones simultáneas pueden permitirse en instancias en tanto los cambios que se hayan realizado no se superpongan. Si establece `optimistic-lock="dirty"` al mapear la `<class>`, Hibernate sólo comparará los campos desactualizados durante el vaciado.

En ambos casos, con columnas de versión/sello de fecha dedicadas o con comparación de campos completos/desactualizados, Hibernate utiliza una sóla declaración `UPDATE` (con una cláusula `WHERE` apropiada) por entidad para ejecutar el chequeo de versiones y actualizar la información. Si utiliza una persistencia transitiva para la re-unión en cascada de entidades asociadas, Hibernate podría ejecutar actualizaciones innecesarias. Esto usualmente no es problema, pero podrían ejecutarse disparadores (triggers) *enactualizazción* en la base de datos incluso cuando no se haya hecho ningún cambio a las instancias separadas. Puede personalizar este comportamiento estableciendo `select-before-update="true"` en el mapeo de `<class>`, forzando a Hibernate a `SELECT` la instancia para asegurar que las actualizaciones realmente ocurran, antes de actualizar la fila.

## 13.4. Bloqueo pesimista

No se pretende que los usuarios tomen mucho tiempo preocupándose de las estrategias de bloqueo. Usualmente es suficiente con especificar un nivel de aislamiento para las conexiones JDBC y entonces simplemente dejar que la base de datos haga todo el trabajo. Sin embargo, los usuarios avanzados a veces pueden obtener bloqueos exclusivos pesimistas, o reobtener bloqueos al comienzo de una nueva transacción.

Hibernate siempre usará el mecanismo de bloqueo de la base de datos, nunca el bloqueo de objetos en memoria.

La clase `LockMode` define los diferentes niveles de bloqueo que Hibernate puede adquirir. Un bloqueo se obtiene por medio de los siguientes mecanismos:

- `LockMode.WRITE` se adquiere automáticamente cuando Hibernate actualiza o inserta una fila.
- `LockMode.UPGRADE` se puede ser adquirir bajo petición explícita del usuario usando `SELECT ... FOR UPDATE` en bases de datos que soporten esa sintáxis.
- `LockMode.UPGRADE_NOWAIT` se puede adquirir bajo petición explícita del usuario usando un `SELECT ... FOR UPDATE NOWAIT` bajo Oracle.

- `LockMode.READ` se adquiere automáticamente cuando Hibernate lee los datos bajo los niveles de aislamiento de lectura repetible o serializable. Se puede readquirir por pedido explícito del usuario.
- `LockMode.NONE` representa la ausencia de un bloqueo. Todos los objetos se pasan a este modo de bloqueo al final de una `Transaction`. Los objetos asociados con una sesión por medio de una llamada a `update()` o `saveOrUpdate()` también comienzan en este modo de bloqueo.

La "petición explícita del usuario" se expresa en una de las siguientes formas:

- Una llamada a `Session.load()`, especificando un `LockMode`.
- Una llamada a `Session.lock()`.
- Una llamada a `Query.setLockMode()`.

Si se llama a `Session.load()` con `UPGRADE` o `UPGRADE_NOWAIT`, y el objeto pedido no ha sido cargado todavía por la sesión, el objeto es cargado usando `SELECT ... FOR UPDATE`. Si se llama a `load()` para un objeto que ya esté cargado con un bloqueo menos restrictivo que el pedido, Hibernate llama a `lock()` para ese objeto.

`Session.lock()` realiza un chequeo de número de versión si el modo de bloqueo especificado es `READ`, `UPGRADE` o `UPGRADE_NOWAIT`. En el caso de `UPGRADE` o `UPGRADE_NOWAIT`, se usa `SELECT ... FOR UPDATE`.

Si la base de datos no soporta el modo de bloqueo solicitado, Hibernate usa un modo opcional apropiado en lugar de lanzar una excepción. Esto asegura que las aplicaciones serán portátiles.

## 13.5. Modos de liberación de la conexión

La herencia (2x) de Hibernate en relación con la administración de la conexion JDBC fue que una `Session` obtendría una conexión cuando se necesitara por primera vez y luego la mantendría hasta que se cerrara la sesión. Hibernate 3.x introdujo la noción de modos de liberación de conexión para decirle a la sesión como manejar sus conexiones JDBC. La siguiente discusión sólamente es pertinente para las conexiones provistas por medio de un `ConnectionProvider` configurado. Las conexiones provistas por el usuario no se discuten aquí. Los diferentes modos de liberación se identifican por los valores numerados de `org.hibernate.ConnectionReleaseMode`:

- `ON_CLOSE`: es el comportamiento heredado descrito anteriormente. La sesión de Hibernate obtiene una conexión cuando necesita acceder a JDBC la primera vez y mantiene esa conexión hasta que se cierra la sesión.
- `AFTER_TRANSACTION`: libera las conecciones después de que se ha completado una `org.hibernate.Transaction`.
- `AFTER_STATEMENT` (también se conoce como una liberación agresiva): libera conexiones después de cada ejecución de una declaración. Se salta esta liberación agresiva si la declaración deja abiertos recursos asociados con la sesión dada. Actualmente la única situación donde ocurre esto es por medio del uso de `org.hibernate.ScrollableResults`.

El parámetro de configuración `hibernate.connection.release_mode` se utiliza para especificar el modo de liberación a utilizar. Los valores posibles son los siguientes:

- `auto` (predeterminado): esta opción delega al modo de liberación devuelto por el método `org.hibernate.transaction.TransactionFactory.getDefaultReleaseMode()`. Para JTATransactionFactory, esto devuelve ConnectionReleaseMode.AFTER_STATEMENT; para JDBCTransactionFactory, esto devuelve ConnectionReleaseMode.AFTER_TRANSACTION. No cambie este comportamiento predeterminado ya que las fallas debido a este valor de esta configuración tienden a indicar errores y/o suposiciones en el código del usuario.
- `on_close`: usa ConnectionReleaseMode.ON_CLOSE. Esta configuración se deja para la compatibilidad con versiones anteriores, pero no se recomienda para nada su utilización.
- `after_transaction`: utiliza ConnectionReleaseMode.AFTER_TRANSACTION. Esta configuración no se debe utilizar en entornos JTA. También note que con ConnectionReleaseMode.AFTER_TRANSACTION, si se considera que una sesión se encuentra en modo auto-commit, las conexiones serán liberada como si el modo de liberación fuese AFTER_STATEMENT.
- `after_statement`: usa ConnectionReleaseMode.AFTER_STATEMENT. Además se consulta la `ConnectionProvider` configurada para ver si soporta esta característica `supportsAggressiveRelease()`. Si no, el modo de liberación se vuelve a establecer como ConnectionReleaseMode.AFTER_TRANSACTION. Esta configuración sólamente es segura en entornos en donde podemos re-adquirir la misma conexión JDBC subyacente cada vez que llamamos a `ConnectionProvider.getConnection()` o en entornos auto-commit, en donde no importa si recibimos la misma conexión.

# Interceptores y eventos

Es útil para la aplicación reaccionar a ciertos eventos que ocurren dentro de Hibernate. Esto permite la implementación de funcionalidades genéricas y la extensión de la funcionalidad de Hibernate.

## 14.1. Interceptores

La interfaz `Interceptor` brinda callbacks desde la sesión a la aplicación, permitiendole a ésta última inspeccionar y/o manipular las propiedades de un objeto persistente antes de que sea guardado, actualizado, borrado o cargado. Un uso posible de esto es seguir la pista de la información de auditoría. Por ejemplo, el siguiente `Interceptor` establece automáticamente el `createTimestamp` cuando se crea un `Auditable` y se actualiza la propiedad `lastUpdateTimestamp` cuando se actualiza un `Auditable`.

Puede implementar el `Interceptor` directamente o extender el `EmptyInterceptor`.

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
```

```
                    currentState[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        return false;
    }

    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) {
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
         System.out.println("Creations: " + creates + ", Updates: " + updates, "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

Hay dos clases de interceptores: incluído en `Session`- e incluído en `SessionFactory`.

Se especifica un interceptor incluído `Session` cuando se abre una sesión utilizando uno de los métodos SessionFactory.openSession() sobrecargados aceptando un `Interceptor`.

```
Session session = sf.openSession( new AuditInterceptor() );
```

Un interceptor incluido en `SessionFactory` se encuentra registrado con el objeto `Configuration` antes de construir el `SessionFactory`. En este caso, el interceptor proveido será aplicado a todas las sesiones abiertas desde ese `SessionFactory`; a menos de que se abra una sesión especificando explícitamente el interceptor a utilizar. Los interceptores `SessionFactory` incluidos deben ser a prueba de hilos. Asegúrese de no almacenar un estado especifico a la sesión ya que múltiples sesiones utilizarán este interceptor potencialmente de manera concurrente.

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

## 14.2. Sistema de eventos

Si tiene que reaccionar a eventos particulares en su capa de persistencia, también puede utilizar la arquitectura de *eventos* de Hibernate3. El sistema de eventos se puede ser utilizar además de o como un remplazo para los interceptores.

Todos los métodos de la interfaz `Session` se correlacionan con un evento. Tiene un `LoadEvent`, un `FlushEvent`, etc. Consulte el DTD del archivo de configuración XML o el paquete `org.hibernate.event` para ver la lista completa de los tipos de eventos definidos. Cuando se realiza una petición de uno de estos métodos, la `Session` de Hibernate genera un evento apropiado y se lo pasa al escucha (listener) de eventos configurado para ese tipo. Tal como vienen, estos escuchas implementan el mismo procesamiento en aquellos métodos donde siempre resultan . Sin embargo, usted es libre de implementar una personalización de una de las interfaces escuchas (por ejemplo, el `LoadEvent` es procesado por la implementación registrada de la interfaz `LoadEventListener`), en cuyo caso su implementación sería responsable de procesar cualquier petición `load()` realizada a la `Session`.

Los escuchas se deben considerar como singletons. Esto significa que son compartidos entre las peticiones y por lo tanto, no deben guardar ningún estado como variables de instancia.

Un escucha personalizado implementa la interfaz apropiada para el evento que quiere procesar y/o extender una de las clases base de conveniencia (o incluso los escuchas de eventos predeterminados utilizados por Hibernate de fábrica al declararlos como no-finales para este propósito). Los escuchas personalizados pueden ser registrados programáticamente a través del objeto `Configuration`, o especificados en el XML de configuración de Hibernate. No se soporta la configuración declarativa a través del archivo de propiedades. Este es un ejemplo de un escucha personalizado de eventos load:

```
public class MyLoadListener implements LoadEventListener {
    // this is the single method defined by the LoadEventListener interface
    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
            throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(), event.getEntityId() ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
```

```
}
```

También necesita una entrada de configuración diciéndole a Hibernate que utilice el oyente en vez del oyente por defecto:

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="com.eg.MyLoadListener"/>
            <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration
>
```

En cambio, puede registrarlo programáticamente:

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new DefaultLoadEventListener() };
cfg.EventListeners().setLoadEventListeners(stack);
```

Los oyentes registrados declarativamente no pueden compartir instancias. Si se utiliza el mismo nombre de clase en múltiples elementos `<listener/>`, cada referencia resultará en una instancia separada de esa clase. Si necesita compartir instancias de oyentes entre tipos de oyentes debe usar el enfoque de registración programática.

¿Por qué implementar una interfaz y definir el tipo específico durante la configuración? Una implementación de escucha podría implementar múltiples interfaces de escucha de eventos. Teniendo el tipo definido adicionalmente durante la registración hace más fácil activar o desactivar escuchas personalizados durante la configuración.

## 14.3. Seguridad declarativa de Hibernate

Usualmente, la seguridad declarativa en aplicaciones Hibernate se administra en una capa de fachada de sesión. Hibernate3 permite que ciertas acciones se permitan por medio de JACC y las autoriza por medio de JAAS. Esta es una funcionalidad opcional construída encima de la arquitectura de eventos.

Primero, tiene que configurar los oyentes de eventos apropiados, para habilitar la utilización de autorización JAAS.

```
<listener type="pre-delete" class="org.hibernate.secure.JACCPreDeleteEventListener"/>
<listener type="pre-update" class="org.hibernate.secure.JACCPreUpdateEventListener"/>
<listener type="pre-insert" class="org.hibernate.secure.JACCPreInsertEventListener"/>
```

```
<listener type="pre-load" class="org.hibernate.secure.JACCPreLoadEventListener"/>
```

Note que `<listener type="..." class="..."/>` es la abreviatura para `<event type="..."><listener class="..."/></event>` cuando hay exactamente un escucha para un tipo de evento en particular.

A continuación, todavía en `hibernate.cfg.xml`, enlace los permisos a los roles:

```
<grant role="admin" entity-name="User" actions="insert,update,read"/>
<grant role="su" entity-name="User" actions="*"/>
```

Los nombres de los roles son comprendidos por su proveedor de JACC.

# Procesamiento por lotes

Un enfoque ingenuo para insertar 100.000 filas en la base de datos utilizando Hibernate puede verse así:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

Esto podría caer dentro de una `OutOfMemoryException` en algún sitio cerca de la fila 50.000. Esto se debe a que Hibernate tiene en caché todas las instancias de `Customer` recién insertadas en el caché de nivel de sesión. En este capítulo le vamos a mostrar cómo evitar este problema.

Si está realizando un procesamiento por lotes (batch processing), es necesario que habilite el uso del lote JDBC. Esto es esencial si quiere lograr un rendimiento óptimo. Establezca el tamaño de lote JDBC con un número razonable (por ejemplo, 10-50):

```
hibernate.jdbc.batch_size 20
```

Hibernate desactiva el lote de inserción a nivel de JDBC de forma transparente si usted utiliza un generador de identificador `identiy`.

También puede realizar este tipo de trabajo en un proceso en donde la interacción con el caché de segundo nivel se encuentre completamente desactivado:

```
hibernate.cache.use_second_level_cache false
```

Sin embargo, esto no es absolutamente necesario ya que podemos establecer explícitamente el `CacheMode` para descativar la interacción con el caché de segundo nivel.

## 15.1. Inserciones de lotes

Al hacer persistentes los objetos nuevos es necesario que realice `flush()` y luego `clear()` en la sesión regularmente para controlar el tamaño del caché de primer nivel.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
```

```
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

## 15.2. Actualizaciones de lotes

Para recuperar y actualizar datos se aplican las mismas ideas. Además, necesita utilizar `scroll()` para sacar ventaja de los cursores del lado del servidor en consultas que retornen muchas filas de datos.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

## 15.3. La interfaz de Sesión sin Estado

Opcionalmente, Hibernate proporciona una API orientada a comandos que se puede utilizar para datos que concurren desde y hacia la base de datos en forma de objetos separados. Un `StatelessSession` no tiene un contexto de persistencia asociado con él y no proporciona mucha de la semántica a un alto nivel de ciclo de vida. En particular, una sesión sin estado no implementa un caché en primer nivel y tampoco interactúa con cachés de segundo nivel o de peticiones. No implementa escritura-retrasada transaccional o chequeo de desactualizaciones automático. Las operaciones realizadas con la utilización de una sesión sin estado nunca usan cascadas para las instancias asociadas. La sesión sin estado ignora las colecciones. Las operaciones llevadas a cabo por una sesión sin estado ignoran el modelo de evento y los interceptores de Hibernte.

Las sesiones sin estado son vulnerables a efectos de sobrenombamiento de datos debido a la falta de un caché de primer nivel. Una sesión sin estado es una abstracción en un nivel más bajo, mucho más cerca del JDBC subyacente.

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

tx.commit();
session.close();
```

En este código de ejemplo, las instancias `Customer` retornadas por la petición se separan inmediatamente. Nunca se asocian con ningún contexto de persistencia.

Las operaciones `insert(), update()` y `delete()` definidas por la interfaz `StatelessSession` son consideradas como operaciones directas a nivel de filas de la base de datos. Esto resulta en una ejecución inmediata de un `INSERT, UPDATE` SQL o `DELETE` respectivamente. Tienen una semántica diferente a la de las operaciones `save(), saveOrUpdate()` y `delete()` definidas por la interfaz `Session`.

## 15.4. Operaciones de estilo DML

Como se discutió anteriormente, el mapeo objeto/relacional transparente se refiere a la administración del estado de objetos. El estado del objeto está disponible en la memoria. Esto significa que el manipular datos directamente en la base de datos (utilizando DML (del inglés `Data Manipulation Language`) las declaraciones: `INSERT`, `UPDATE`, `DELETE`) no afectarán el estado en la memoria. Sin embargo, Hibernate brinda métodos para la ejecución de declaraciones en masa DML del estilo de SQL, las cuales se realizan por medio del Lenguaje de Consulta de Hibernate (*HQL*).

La pseudo-sintáxis para las declaraciones `UPDATE` y `DELETE` es: `( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?`.

Algunos puntos a observar:

- En la cláusula-from, la palabra clave FROM es opcional
- Sólamente puede haber una entidad mencionada en la cláusula-from y puede tener un alias. Si el nombre de la entidad tiene un alias entonces cualquier referencia a la propiedad tiene que ser calificada utilizando ese alias. Si el nombre de la entidad no tiene un alias entonces es ilegal calificar cualquier referencia de la propiedad.

- No se puede especificar ninguna *unión* ya sea implícita o explícita, en una consulta masiva de HQL. Se pueden utilizar subconsultas en la cláusula-where y en donde las subconsultas puedan contener uniones en sí mismas.
- La cláusula-where también es opcional.

Como ejemplo, para ejecutar un `UPDATE` de HQL, utilice el método `Query.executeUpdate()`. El método es nombrado para aquellos familiarizados con el `PreparedStatement.executeUpdate()` de JDBC:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";
// or String hqlUpdate = "update Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Para mantenerse de acuerdo con la especificación de EJB3, las declaraciones `UPDATE` de HQL, por defecto no afectan la *versión* o los valores de la propiedad *sello de fecha* para las entidades afectadas. Sin embargo, puede obligar a Hibernate a poner en cero apropiadamente los valores de las propiedades `versión` o `sello de fecha` por medio de la utilización de una `actualización con versión`. Esto se logra agregando la palabra clave `VERSIONED` después de la palabra clave `UPDATE`.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Observe que los tipos de versiones personalizados (`org.hibernate.usertype.UserVersionType`) no están permitidos en conjunto con una declaración `update versioned`.

Para ejecutar un `DELETE` HQL, utilice el mismo método `Query.executeUpdate()`:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Customer c where c.name = :oldName";
```

```
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

El valor `int` retornado por el método `Query.executeUpdate()` indica el número de entidades afectadas por la operación. Considere que esto puede estar correlacionado o no con el número de filas afectadas en la base de datos. Una operación masiva de HQL puede llegar a causar que se ejecuten múltiples declaraciones SQL reales, por ejemplo, para una subclase-joined. El número retornado indica el número de entidades realmente afectadas por la declaración. De vuelta al ejemplo de la subclase joined, un borrado contra una de las subclases puede resultar, de hecho, en borrados de no sólamente la tabla a la cual esa subclase esta mapeada, sino también la tabla "raíz" y potencialmente las tablas de subclases joined hasta la jerarquía de herencia.

La pseudo-sintáxis para las declaraciones `INSERT` es: `INSERT INTO EntityName properties_list select_statement`. Algunos puntos que se deben observar son:

- Sólamente se soporta la forma INSERT INTO ... SELECT ..., no la forma INSERT INTO ... VALUES ...

  La lista de propiedades (properties_list) es análoga a la `column speficiation` en la declaración `INSERT` de SQL. Para las entidades involucradas en la herencia mapeada, sólamente las propiedades definidas directamente en ese nivel de clase dado se pueden utlizar en la lista de propiedades. Las propiedades de la superclase no están permitidas, y las propiedaeds de la subclase no tienen sentido. Es decir, las declaraciones `INSERT` son inherentemente no-polimórficas.
- select_statement puede ser cualquier consulta select de HQL válida con la advertencia de que los tipos de retorno coincidan con los tipos esperados por el insert. Actualmente, esto se verifica durante la compilación de la consulta en vez de permitir que se relegue la verificación a la base de datos. Sin embargo, esto puede crear problemas entre los `Type`s de Hibernate, los cuales son *equivalentes* y no *iguales*. Esto puede crear problemas con las uniones mal hechas entre una propiedad definida como un `org.hibernate.type.DateType` y una propiedad definida como una `org.hibernate.type.TimestampType`, aunque puede que la base de datos no distinga o no pueda manejar la conversión.
- Para la propiedad id, la declaración insert le da dos opciones. Puede especificar explícitamente la propiedad id en la lista de propiedades (properties_list ) (en tal caso su valor se toma de la expresión de selección correspondiente) o se omite de la lista de propiedades (en tal caso se utiliza un valor generado). Esta última opción sólamente está disponible cuando se utilizan generadores de id que operan en la base de datos, intentando utilizar esta opción con cualquier generador de tipo "en memoria" provocará una excepción durante el análisis sintáctico. Note que para los propósitos de esta discusión, los generadores en la base de datos son considerados `org.hibernate.id.SequenceGenerator` (y sus subclases) y cualquier implementador de `org.hibernate.id.PostInsertIdentifierGenerator`. La excepción más

importante aquí es `org.hibernate.id.TableHiLoGenerator`, la cual no se puede utilizar ya que no expone una manera selectiva de obtener sus valores.

- Para las propiedades mapeadas como `version` o `timestamp`, la declaración insert le da dos opciones. Puede especificar la propiedad en la lista de propiedades (en tal caso su valor se toma de las expresiones de selección correspondientes) o se omite de la lista de propiedades (en tal caso se utiliza el `seed value` definido por el `org.hibernate.type.VersionType`).

Un ejemplo de la ejecución de la declaración `INSERT` de HQL:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer
 c where ...";
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

# HQL: El lenguaje de consulta de Hibernate

Hibernate utiliza un lenguaje de consulta potente (HQL) que se parece a SQL. Sin embargo, comparado con SQL, HQL es completamente orientado a objetos y comprende nociones como herencia, polimorfismo y asociación.

## 16.1. Sensibilidad a mayúsculas

Las consultas no son sensibles a mayúsculas, a excepción de los nombres de las clases y propiedades Java. De modo que `SeLeCT` es lo mismo que `sELEct` e igual a `SELECT`, pero `org.hibernate.eg.FOO` no es lo mismo que `org.hibernate.eg.Foo` y `foo.barSet` no es igual a `foo.BARSET`.

Este manual utiliza palabras clave HQL en minúsculas. Algunos usuarios encuentran que las consultas con palabras clave en mayúsculas son más fáciles de leer, pero esta convención no es apropiada para las peticiones incluidas en código Java.

## 16.2. La cláusula from

La consulta posible más simple de Hibernate es de esta manera:

```
from eg.Cat
```

Esto retorna todas las instancias de la clase `eg.Cat`. Usualmente no es necesario calificar el nombre de la clase ya que `auto-import` es el valor predeterminado. Por ejemplo:

```
from Cat
```

Con el fin de referirse al `Cat` en otras partes de la petición, necesitará asignar un *alias*. Por ejemplo:

```
from Cat as cat
```

Esta consulta asigna el alias `cat` a las instancias `Cat`, de modo que puede utilizar ese alias luego en la consulta. La palabra clave `as` es opcional. También podría escribir:

```
from Cat cat
```

Pueden aparecer múltiples clases, lo que causa un producto cartesiano o una unión "cruzada" (cross join).

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

Se considera como una buena práctica el nombrar los alias de consulta utilizando una inicial en minúsculas, consistente con los estándares de nombrado de Java para las variables locales (por ejemplo, `domesticCat`).

## 16.3. Asociaciones y uniones (joins)

También puede asignar alias a entidades asociadas o a elementos de una colección de valores utilizando una `join`. Por ejemplo:

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
from Cat as cat left join cat.mate.kittens as kittens
```

```
from Formula form full join form.parameter param
```

Los tipos de uniones soportadas se tomaron prestados de ANSI SQL

- `inner join`
- `left outer join`
- `right outer join`
- `full join` (no es útil usualmente)

Las construcciones `inner join`, `left outer join` y `right outer join` se pueden abreviar.

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

Puede proveer condiciones extras de unión utilizando la palabra clave `with` de HQL.

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight
> 10.0
```

A "fetch" join allows associations or collections of values to be initialized along with their parent objects using a single select. This is particularly useful in the case of a collection. It effectively overrides the outer join and lazy declarations of the mapping file for associations and collections. See *Sección 21.1, "Estrategias de recuperación"* for more information.

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

Usualmente no se necesita asignársele un alias a una unión de recuperación ya que los objetos asociados no se deben utilizar en la cláusula `where` (ni en cualquier otra cláusula). Los objetos asociados no se retornan directamente en los resultados de la consulta. En cambio, se pueden acceder por medio del objeto padre. La única razón por la que necesitaríamos un alias es si estamos uniendo recursivamente otra colección:

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

La construcción `fetch` no puede utilizarse en consultas llamadas que usen `iterate()` (aunque se puede utilizar `scroll()`). `Fetch` se debe usar junto con `setMaxResults()` o `setFirstResult()` ya que estas operaciones se basan en las filas de resultados, las cuales usualmente contienen duplicados para la recuperación de colección temprana, por lo tanto, el número de filas no es lo que se esperaría. `Fetch` no se debe usar junto con una condición `with` improvisadas. Es posible crear un producto cartesiano por medio de una recuperación por union más de una colección en una consulta, así que tenga cuidado en este caso. La recuperación por unión de múltiples roles de colección también da resultados a veces inesperados para mapeos de bag, así que tenga cuidado de cómo formular sus consultas en este caso. Finalmente, observe que `full join fetch` y `right join fetch` no son significativos.

Si está utilizando una recuperación perezosa a nivel de propiedad (con instrumentación de código byte), es posible forzar a Hibernate a traer las propiedades perezosas inmediatamente utilizando `fetch all properties`.

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

## 16.4. Formas de sintaxis unida

HQL soporta dos formas de unión de asociación: `implicit` y `explicit`.

Las consultas que se mostraron en la sección anterior todas utilizan la forma `explicit`, en donde la palabra clave join se utiliza explícitamente en la claúsula from. Esta es la forma recomendada.

La forma `implicit` no utiliza la palabra clave join. Las asociaciones se "desreferencian" utilizando la notación punto. Uniones `implicit` pueden aparecer en cualquiera de las cláusulas HQL. La unión `implicit` causa uniones internas (inner joins) en la declaración SQL que resulta.

```
from Cat as cat where cat.mate.name like '%s%'
```

## 16.5. Referencia a la propiedad identificadora

Hay dos maneras de referirse a la propiedad identificadora de una entidad:

- La propiedad especial (en minúsculas) `id` se puede utilizar para referenciar la propiedad identificadora de una entidad *dado que la entidad no defina un id del nombre de la propiedad no-identificadora*.
- Si la entidad define una propiedad identificadora nombrada, puede utilizar ese nombre de propiedad.

Las referencias a propiedades identificadoras compuestas siguen las mismas reglas de nombramiento. Si la entidad no tiene un id del nombre de la propiedad no-identificadora, la propiedad identificadora compuesta sólamente puede ser referenciada por su nombre definido. De otra manera se puede utilizar la propiedad `id` especial para referenciar la propiedad identificadora.

**Importante**

Observe que esto ha cambiado bastante desde la version 3.2.2. En versiones previas, `id`*siempre* se refería a la propiedad identificadora sin importar su nombre real. Una ramificación de esa decisión fue que las propiedades no-identificadoras nombradas `id` nunca podrían ser referenciadas en consultas de Hibernate.

## 16.6. La cláusula select

La cláusula `select` escoge qué objetos y propiedades devolver en el conjunto de resultados de la consulta. Considere lo siguiente:

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

La consulta seleccionará `mate`s de otros `Cat`s. Puede expresar esta consulta de una manera más compacta así:

```
select cat.mate from Cat cat
```

Las consultas pueden retornar propiedades de cualquier tipo de valor incluyendo propiedades del tipo componente:

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

Las consultas pueden retornar múltiples objetos y/o propiedades como un array de tipo `Object[]`,

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

O como una `List`:

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

O asumiendo que la clase `Family` tiene un constructor apropiado - como un objeto Java de tipo seguro:

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

Puede asignar alias para expresiones seleccionadas utilizando `as`:

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

Esto es lo más útil cuando se usa junto con `select new map`:

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

Esta consulta devuelve un `Map` de alias a valores seleccionados.

## 16.7. Funciones de agregación

Las consultas HQL pueden incluso retornar resultados de funciones de agregación sobre propiedades:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

Las funciones de agregación soportadas son:

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

Puede utilizar operadores aritméticos, concatenación y funciones SQL reconocidas en la cláusula select:

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

Las palabras clave `distinct` y `all` se pueden utilizar y tienen las misma semántica que en SQL.

```
select distinct cat.name from Cat cat
```

```
select count(distinct cat.name), count(cat) from Cat cat
```

## 16.8. Consultas polimórficas

Una consulta como:

```
from Cat as cat
```

devuelve instancias no sólamente de `Cat`, sino también de subclases como `DomesticCat`. Las consultas de Hibernate pueden nombrar *cualquier* clase o interfaz Java en la cláusula `from`. La consulta retornará instancias de todas las clases persistentes que extiendan esa clase o implementen la interfaz. La siguiente consulta retornaría todos los objetos persistentes.

```
from java.lang.Object o
```

La interfaz `Named` se podría implementar por varias clases persistentes:

```
from Named n, Named m where n.name = m.name
```

Las dos últimas consultas requerirán más de un `SELECT` SQL. Esto significa que la cláusula `order by` no ordenará correctamente todo el conjunto que resulte. También significa que no puede llamar estas consulta usando `Query.scroll()`.

## 16.9. La cláusula where

La cláusula `where` le permite refinar la lista de instancias retornadas. Si no existe ningún alias, puede referirse a las propiedades por nombre:

```
from Cat where name='Fritz'
```

Si existe un alias, use un nombre de propiedad calificado:

```
from Cat as cat where cat.name='Fritz'
```

Esto retorna instancias de `Cat` llamadas 'Fritz'.

La siguiente petición:

```
select foo
```

```
from Foo foo, Bar bar
where foo.startDate = bar.date
```

retornará todas las instancias de `Foo` con una instancia de `bar` con una propiedad `date` igual a la propiedad `startDate` del `Foo`. Las expresiones de ruta compuestas hacen la cláusula `where` extremadamente potente. Tome en consideración lo siguiente:

```
from Cat cat where cat.mate.name is not null
```

Esta consulta se traduce a una consulta SQL con una unión de tabla (interna). Por ejemplo:

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

terminaría con una consulta que requeriría cuatro uniones de tablas en SQL.

El operador = se puede utilizar para comparar no sólamente propiedades sino también instancias:

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

The special property (lowercase) `id` can be used to reference the unique identifier of an object. See *Sección 16.5, “Referencia a la propiedad identificadora ”* for more information.

```
from Cat as cat where cat.id = 123

from Cat as cat where cat.mate.id = 69
```

La segunda consulta es eficiente y no se necesita una unión de tablas.

También se pueden utilizar las propiedades de identificadores compuestos. Considere el siguiente ejemplo en donde `Person` tiene identificadores compuestos que consisten de `country` y `medicareNumber`:

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

Una vez más, la segunda consulta no requiere una unión de tablas.

See *Sección 16.5, "Referencia a la propiedad identificadora "* for more information regarding referencing identifier properties)

La propiedad especial `class` acccede al valor discriminador de una instancia en el caso de persistencia polimórfica. Un nombre de clase Java incluído en la cláusula where será traducido a su valor discriminador.

```
from Cat cat where cat.class = DomesticCat
```

You can also use components or composite user types, or properties of said component types. See *Sección 16.17, "Componentes"* for more information.

Un tipo "any" tiene las propiedades especiales `id` y `class`, permiténdole expresar una unión de la siguiente forma (en donde `AuditLog.item` es una propiedad mapeada con `<any>`).

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

La `log.item.class` y `payment.class` harían referencia a los valores de columnas de la base de datos completamente diferentes en la consulta anterior.

## 16.10. Expresiones

Las expresiones utilizadas en la cláusula `where` incluyen lo siguiente:

- operadores matemáticos: `+, -, *, /`
- operadores de comparación binarios: `=, >=, <=, <>, !=, like`
- operadores lógicos `and, or, not`
- Paréntesis `( )` que indican agrupación
- `in`, `not in`, `between`, `is null`, `is not null`, `is empty`, `is not empty`, `member of` y `not member of`
- Caso "simple", `case ... when ... then ... else ... end`, y caso "buscado", `case when ... then ... else ... end`
- concatenación de cadenas `...||...` o `concat(...,...)`
- `current_date()`, `current_time()` y `current_timestamp()`
- `second(...)`, `minute(...)`, `hour(...)`, `day(...)`, `month(...)`, and `year(...)`
- Cualquier función u operador definido por EJB-QL 3.0: `substring()`, `trim()`, `lower()`, `upper()`, `length()`, `locate()`, `abs()`, `sqrt()`, `bit_length()`, `mod()`

- `coalesce()` y `nullif()`
- `str()` para convertir valores numéricos o temporales a una cadena legible.
- `cast(... as ...)`, donde el segundo argumento es el nombre de un tipo de Hibernate , y `extract(... from ...)` si `cast()` y `extract()` es soportado por la base de datos subyacente.
- la función `index()` de HQL, que se aplica a alias de una colección indexada unida.
- Las funciones de HQL que tomen expresiones de ruta valuadas en colecciones: `size()`, `minelement()`, `maxelement()`, `minindex()`, `maxindex()`, junto con las funciones especiales `elements()` e `indices`, las cuales se pueden cuantificar utilizando `some, all, exists, any, in`.
- Cualquier función escalar SQL soportada por la base de datos como `sign()`, `trunc()`, `rtrim()` y `sin()`
- parámetros posicionales JDBC `?`
- parámetros con nombre `:name`, `:start_date` y `:x1`
- literales SQL `'foo'`, `69`, `6.66E+2`, `'1970-01-01 10:00:01.0'`
- constantes Java `public static finaleg.Color.TABBY`

`in` y `between` pueden utilizarse así:

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

Las formas negadas se pueden escribir así:

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

De manera similar, `is null` y `is not null` se pueden utilizar para probar valores nulos.

Los valores booleanos se pueden utilizar fácilmente en expresiones declarando substituciones de consulta HQL en la configuración de Hibernate:

```
<property name="hibernate.query.substitutions"
>true 1, false 0</property
>
```

Esto remplazará las palabras clave `true` y `false` con los literales `1` y `0` en el SQL traducido de este HQL:

```
from Cat cat where cat.alive = true
```

Puede comprobar el tamaño de una colección con la propiedad especial `size` o la función especial `size()`.

```
from Cat cat where cat.kittens.size
> 0
```

```
from Cat cat where size(cat.kittens)
> 0
```

Para las colecciones indexadas, puede referirse a los índices máximo y mínimo utilizando las funciones `minindex` y `maxindex`. De manera similar, se puede referir a los elementos máximo y mínimo de una colección de tipo básico utilizando las funciones `minelement` y `maxelement`. Por ejemplo:

```
from Calendar cal where maxelement(cal.holidays)
> current_date
```

```
from Order order where maxindex(order.items)
> 100
```

```
from Order order where minelement(order.items)
> 10000
```

Las funciones SQL `any, some, all, exists, in` están soportadas cuando se les pasa el conjunto de elementos o índices de una colección (las funciones `elements` e `indices`) o el resultado de una subconsulta (vea a continuación):

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
from Cat cat where exists elements(cat.kittens)
```

```
from Player p where 3
> all elements(p.scores)
```

```
from Show show where 'fizard' in indices(show.acts)
```

Note que estas construcciones - `size`, `elements`, `indices`, `minindex`, `maxindex`, `minelement`, `maxelement` - solo se pueden utilizar en la cláusula where en Hibernate3.

Los elementos de colecciones indexadas (arrays, listas, mapas) se pueden referir por índice sólamente en una cláusula where:

```
from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

La expresión dentro de `[]` puede incluso ser una expresión aritmética:

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL también proporciona la función incorporada `index()`, para los elementos de una asociación uno-a-muchos o una colección de valores.

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

Se pueden utilizar las funciones SQL escalares soportadas por la base de datos subyacente:

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

Considere qué tan larga y menos leíble sería la siguiente consulta en SQL:

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

*Ayuda:* algo como

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

## 16.11. La cláusula order by

La lista retornada por una consulta se puede ordenar por cualquier propiedad de una clase retornada o componentes:

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

Los `asc` o `desc` opcionales indican ordenamiento ascendente o descendente respectivamente.

## 16.12. La cláusula group by

Una consulta que retorna valores agregados se puede agrupar por cualquier propiedad de una clase retornada o componentes:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

Se permite también una cláusula `having`.

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

Las funciones SQL y las funciones de agregación SQL están permitidas en las cláusulas `having` y `order by`, si están soportadas por la base de datos subyacente (por ejemplo, no lo están en MySQL).

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight)
> 100
order by count(kitten) asc, sum(kitten.weight) desc
```

La cláusula `group by` ni la cláusula `order by` pueden contener expresiones aritméticas. Hibernate tampocoo expande una entidad agrupada así que no puede escribir `group by cat` si todas las propiedades de `cat` son no-agregadas. Tiene que enumerar todas la propiedades no-agregadas explícitamente.

## 16.13. Subconsultas

Para bases de datos que soportan subconsultas, Hibernate soporta subconsultas dentro de consultas. Una subconsulta se debe encerrar entre paréntesis (frecuentemente por una llamada a una función de agregación SQL). Incluso se permiten subconsultas correlacionadas (subconsultas que se refieren a un alias en la consulta exterior).

```
from Cat as fatcat
where fatcat.weight
> (
    select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

Note que las subconsultas HQL pueden ocurrir sólamente en las cláusulas select o where.

Note that subqueries can also utilize `row value constructor` syntax. See *Sección 16.18, "Sintaxis del constructor de valores por fila"* for more information.

## 16.14. Ejemplos de HQL

Las consultas de Hibernate pueden ser bastante potentes y complejas. De hecho, el poder del lenguaje de consulta es uno de las fortalezas principales de Hibernate. He aquí algunos ejemplos de consultas muy similares a las consultas de proyectos recientes. Note que la mayoría de las consultas que escribirá son mucho más simples que los siguientes ejemplos.

La siguiente consulta retorna el order id, número de items y valor total mínimo dado y el valor de la orden para todas las órdenes no pagadas de un cliente en particular. Los resultados se ordenan de acuerdo al valor total. Al determinar los precios, usa el catálogo actual. La consulta SQL resultante, contra las tablas `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG` y `PRICE` tiene cuatro uniones interiores y una subselección (no correlacionada).

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate
>= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount)
> :minAmount
order by sum(price.amount) desc
```

¡Qué monstruo! Realmente, en la vida real, no me gustan mucho las subconsultas, de modo que mi consulta fue realmente algo como esto:

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount)
> :minAmount
order by sum(price.amount) desc
```

La próxima consulta cuenta el número de pagos en cada estado, excluyendo todos los pagos en el estado `AWAITING_APPROVAL` donde el cambio más reciente al estado lo hizo el usuario actual. Se traduce en una consulta SQL con dos uniones interiores y una subselección correlacionada contra las tablas `PAYMENT`, `PAYMENT_STATUS` y `PAYMENT_STATUS_CHANGE`.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <
> PaymentStatus.AWAITING_APPROVAL
    or (
```

```
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <
> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

Si la colección `statusChanges` se mapeara como una lista, en vez de un conjunto, la consulta habría sido mucho más simple de escribir.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <
> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <
> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

La próxima consulta utiliza la función `isNull()` de MS SQL Server para devolver todas las cuentas y pagos aún no cancelados de la organización a la que pertenece el usuario actual. Se traduce como una consulta SQL con tres uniones interiores, una unión exterior y una subselección contra las tablas `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`, `ACCOUNT_TYPE`, `ORGANIZATION` y `ORG_USER`.

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

Para algunas bases de datos, necesitaríamos eliminar la subselección (correlacionada).

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

## 16.15. Declaraciones UPDATE y DELETE masivas

HQL now supports `update`, `delete` and `insert ... select ...` statements. See *Sección 15.4, "Operaciones de estilo DML"* for more information.

## 16.16. Consejos y Trucos

Puede contar el número de resultados de una consulta sin retornarlos:

```
( (Integer) session.createQuery("select count(*) from ....").iterate().next() ).intValue()
```

Para ordenar un resultado por el tamaño de una colección, utilice la siguiente consulta:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

Si su base de datos soporta subselecciones, puede colocar una condición sobre el tamaño de selección en la cláusula where de su consulta:

```
from User usr where size(usr.messages)
>= 1
```

Si su base de datos no soporta subselecciones, utilice la siguiente consulta:

```
select usr.id, usr.name
from User usr
    join usr.messages msg
group by usr.id, usr.name
having count(msg)
>= 1
```

Como esta solución no puede retornar un `User` con cero mensajes debido a la unión interior, la siguiente forma también es útil:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

Las propiedades de un JavaBean pueden ser ligadas a los parámetros de consulta con nombre:

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

Las colecciones son paginables usando la interfaz `Query` con un filtro:

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

Los elementos de colección se pueden ordenar o agrupar usando un filtro de consulta:

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

Puede hallar el tamaño de una colección sin inicializarla:

```
( (Integer) session.createQuery("select count(*) from ....").iterate().next() ).intValue();
```

## 16.17. Componentes

Los componentes se pueden utilizar de la misma manera en que se pueden utilizar los tipos de valores simples en consultas HQL. Pueden aparecer en la cláusula `select` así:

```
select p.name from Person p
```

```
select p.name.first from Person p
```

en donde el nombre de la Persona es un componente. Los componentes también se pueden utilizar en la cláusula `where`:

```
from Person p where p.name = :name
```

```
from Person p where p.name.first = :firstName
```

Los componentes también se pueden utilizar en la cláusula `where`:

```
from Person p order by p.name
```

```
from Person p order by p.name.first
```

Otro uso común de los componentes se encuentra en *row value constructors*.

## 16.18. Sintaxis del constructor de valores por fila

HQL soporta la utilización de la sintaxis `row value constructor` de SQL ANSI que a veces se denomina sintaxis `tuple`, aunque puede que la base de datos subyacentes no soporte esa noción. Aquí estamos refiriéndonos generalmente a las comparaciones multivaluadas que se asocian típicamente con los componentes. Considere una entidad Persona, la cual define un componente de nombre:

```
from Person p where p.name.first='John' and p.name.last='Jingleheimer-Schmidt'
```

Esa es una sintaxis válida aunque un poco verbosa. Puede hacerlo un poco más conciso utilizando la sintaxis `row value constructor`:

```
from Person p where p.name=('John', 'Jingleheimer-Schmidt')
```

También puede ser útil especificar esto en la cláusula `select`:

```
select p.name from Person p
```

También puede ser beneficioso el utilizar la sintaxis `row value constructor` cuando se utilizan subconsultas que necesitan compararse con valores múltiples:

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

Algo que se debe tomar en consideración al decidir si quiere usar esta sintaxis es que la consulta dependerá del orden de las sub-propiedades componentes en los metadatos.

# Consultas por criterios

Acompaña a Hibernate una API de consultas por criterios intuitiva y extensible.

## 17.1. Creación de una instancia `Criteria`

La interfaz `org.hibernate.Criteria` representa una consulta contra una clase persistente en particular. La `Session` es una fábrica de instancias de `Criteria`.

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

## 17.2. Límitando el conjunto de resultados

Un criterio individual de consulta es una instancia de la interfaz `org.hibernate.criterion.Criterion`. La clase `org.hibernate.criterion.Restrictions` define métodos de fábrica para obtener ciertos tipos incorporados de `Criterion`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

Las restricciones se pueden agrupar lógicamente.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

Hay un rango de tipos de criterios incorporados (subclases de `Restrictions`). Uno de los más útiles le permite especificar SQL directamente.

```
List cats = sess.createCriteria(Cat.class)
          .add(   Restrictions.sqlRestriction("lower({alias}.name)   like   lower(?)",   "Fritz
%", Hibernate.STRING) )
    .list();
```

El sitio `{alias}` será remplazado por el alias de fila de la entidad consultada.

También puede obtener un criterio de una instancia `Property`. Puede crear una `Property` llamando a `Property.forName()`.

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" } ) )
    .list();
```

## 17.3. Orden de los resultados

Puede ordenar los resultados usando `org.hibernate.criterion.Order`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

## 17.4. Asociaciones

Al navegar asociaciones usando `createCriteria()` puede especificar restricciones en entidades relacionadas:

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

El segundo `createCriteria()` retorna una nueva instancia de `Criteria`, que se refiere a los elementos de la colección `kittens`.

Hay una alternativa que es útil en ciertas circunstancias:

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` no crea una nueva instancia de `Criteria`.)

Las colecciones de gatitos de las instancias `Cat` retornadas por las dos consultas previas *no* están prefiltradas por los criterios. Si desea recuperar sólo los gatitos que coincidan con los criterios debe usar un `ResultTransformer`.

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

Adicionalmente puede manipular el grupo de resultados utilizando una unión externa izquierda:

```
               List cats = session.createCriteria( Cat.class )
                   .createAlias("mate", "mt", Criteria.LEFT_JOIN, Restrictions.like("mt.name",
 "good%") )
                      .addOrder(Order.asc("mt.age"))
```

```
                        .list();
```

Esto retornará todos los `Cats` -gatos- con una pareja cuyo nombre empiece por "good" ordenado de acuerdo a la edad de la pareja y todos los cats -gatos- que no tengan una pareja. Esto es útil cuando hay necesidad de ordenar o limitar en la base de datos antes de retornar grupos de resultados complejos/grandes y elimina muchas instancias en donde se tendrían que realizar múltiples consultas y unir en memoria los resultados por medio de java.

Sin esta funcionalidad, primero todos los cats sin una pareja tendrían que cargarse en una petición.

Una segunda petición tendría que recuperar los cats -gatos- con las parejas cuyos nombres empiecen por "good" ordenado de acuerdo a la edad de las parejas.

Tercero, en memoria sería necesario unir las listas manualmente.

## 17.5. Recuperación dinámica de asociaciones

Puede especificar la semántica de recuperación de asociaciones en tiempo de ejecución usando `setFetchMode()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

This query will fetch both `mate` and `kittens` by outer join. See *Sección 21.1, “Estrategias de recuperación”* for more information.

## 17.6. Consultas ejemplo

La clase `org.hibernate.criterion.Example` le permite construir un criterio de consulta a partir de una instancia dada.

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

Las propiedades de versión, los identificadores y las asociaciones se ignoran. Por defecto, las propiedades valuadas como nulas se excluyen.

Puede modificar la aplicación del `Example`.

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "color"
    .ignoreCase()              //perform case insensitive string comparisons
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

Puede incluso usar ejemplos para colocar criterios sobre objetos asociados.

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

## 17.7. Proyecciones, agregación y agrupamiento

La clase `org.hibernate.criterion.Projections` es una fábrica de instancias de `Projection`. Puede aplicar una proyección a una consulta llamando a `setProjection()`.

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

No es necesario ningún "agrupamiento por" explícito en una consulta por criterios. Ciertos tipos de proyecciones son definidos para ser *proyecciones agrupadas*, que además aparecen en la cláusula SQL `group by`.

Puede asignar un alias a una proyección de modo que el valor proyectado pueda ser referido en restricciones u ordenamientos. Aquí hay dos formas diferentes de hacer esto:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"), "colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.groupProperty("color").as("colr") )
    .addOrder( Order.asc("colr") )
    .list();
```

Los métodos `alias()` y `as()` simplemente envuelven una instancia de proyección en otra instancia de `Projection` con alias. Como atajo, puede asignar un alias cuando agregue la proyección a una lista de proyecciones:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

```
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

También puede usar `Property.forName()` para expresar proyecciones:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
```

```
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

## 17.8. Consultas y subconsultas separadas

La clase `DetachedCriteria` le permite crear una consulta fuera del ámbito de una sesión y luego ejecutarla usando una `Session` arbitraria.

```
DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName("sex").eq('F') );

Session session = ....;
Transaction txn = session.beginTransaction();
List results = query.getExecutableCriteria(session).setMaxResults(100).list();
txn.commit();
session.close();
```

También puede utilizar una `DetachedCriteria` para expresar una subconsulta. Las instancias de Criterion involucrando subconsultas se pueden obtener por medio de `Subqueries` o `Property`.

```
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight").gt(avgWeight) )
    .list();
```

```
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

Las subconsultas correlacionadas tambie# son posibles:

```
DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.class, "cat2")
    .setProjection( Property.forName("weight").avg() )
    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName("weight").gt(avgWeightForSex) )
    .list();
```

## 17.9. Consultas por identificador natural

Para la mayoría de las consultas, incluyendo las consultas por criterios, el caché de consulta no es muy eficiente debido a que la invalidación del caché de consulta ocurre con demasiada frecuencia. Sin embargo, hay un tipo especial de consulta donde podemos optimizar el algoritmo de invalidación de caché: búsquedas de una clave natural constante. En algunas aplicaciones, este tipo de consulta, ocurre frecuentemente. La API de criterios brinda una provisión especial para este caso.

Primero, mapee la clave natural de su entidad utilizando `<natural-id>` y habilite el uso del caché de segundo nivel.

```
<class name="User">
    <cache usage="read-write"/>
    <id name="id">
        <generator class="increment"/>
    </id>
    <natural-id>
        <property name="name"/>
        <property name="org"/>
    </natural-id>
    <property name="password"/>
</class
>
```

Esta funcionalidad no está pensada para uso con entidades con claves naturales *mutables*.

Una vez que haya habilitado el caché de consulta de Hibernate, `Restrictions.naturalId()` le permite hacer uso del algoritmo de caché más eficiente.

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

# SQL Nativo

También puede expresar sus consultas en el dialecto SQL nativo de su base de datos. Esto es útil si quiere utilizar las características especificas de la base de datos tales como hints de consulta o la palabra clave `CONNECT` en Oracle. También proporciona una ruta de migración limpia desde una aplicación basada en SQL/JDBC a Hibernate.

Hibernate3 le permite especificar SQL escrito a mano, incluyendo procedimientos almacenados para todas las operaciones create, update, delete y load.

## 18.1. Uso de una `SQLQuery`

La ejecución de consultas SQL nativas se controla por medio de la interfaz `SQLQuery`, la cual se obtiene llamando a `Session.createSQLQuery()`. Las siguientes secciones describen cómo utilizar esta API para consultas.

### 18.1.1. Consultas escalares

La consulta SQL más básica es para obtener a una lista de escalares (valores).

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

Estas retornarán una lista de objetos arrays (Object[]) con valores escalares para cada columna en la tabla CATS. Hibernate utilizará ResultSetMetadata para deducir el orden real y los tipos de los valores escalares retornados.

Para evitar los gastos generales de la utilización de `ResultSetMetadata` o simplemente para ser más explícito en lo que se devuelve se puede utilizar `addScalar()`:

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

Se especifica esta consulta:

- la cadena de consulta SQL
- las columnas y tipos que se devuelven

Esto retornará objetos arrays, pero no utilizará `ResultSetMetdata` sino que obtendrá explícitamente las columnas de IDENTIFICACION, NOMBRE y FECHA DE NACIMIENTO

respectivamente como Larga, Cadena y Corta del grupo de resultados subyacente. Esto también significa que sólamente estas tres columnas serán retornadass aunque la consulta este utilizando * y pueda devolver más de las tres columnas enumeradas.

Es posible dejar afuera la información de tipo para todos o algunos de los escalares.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
 .addScalar("BIRTHDATE")
```

Esto es esencialmente la misma consulta que antes, pero ahora se utiliza `ResultSetMetaData` para determinar el tipo de NOMBRE y FECHA DE NACIMIENTO, mientras que el tipo de IDENTIFICACION se especifica explícitamente.

El dialecto controla la manera en que los java.sql.Types retornados de ResultSetMetaData se mapean a los tipos de Hibernate. Si un tipo en especial no se encuentra mapeado o no resulta en el tipo esperado es posible personalizarlo por medio de llamadas a `registerHibernateType` en el dialecto.

### 18.1.2. Consultas de entidades

Todas las consultas anteriores eran sobre los valores escalraes devueltos, basicamente devolviendo los valores "crudos" desde el grupo resultado. Lo siguiente muestra como obtener los objetos entidades desde una consulta sql nativa por medio de `addEntity()`.

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEntity(Cat.class);
```

Se especifica esta consulta:

- la cadena de consulta SQL
- la entidad devuelta por la consulta

Asumiendo que Cat es mapeado como una clase con las columnas IDENTIFICACION, NOMBRE y FECHA DE NACIMIENTO las consultas anteriores devolverán una Lista en donde cada elemento es una entidad Cat.

Si la entidad es mapeada con una `many-to-one` a otra entidad tambien se necesita que devuelva esto cuando realice una consulta nativa, de otra manera, aparecerá un error "no se encontró la columna" específico a la base de datos. Se devolverán automáticamente las columnas adicionales cuando se utiliza la anotación *, pero preferimos ser tan explícitos así como lo muestra el siguiente ejemplo para una `many-to-one` a un `Dog`:

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").addEntity(Cat.class);
```

Esto permitirá que cat.getDog() funcione apropiadamente.

### 18.1.3. Manejo de asociaciones y colecciones

Es posible unir de manera temprana en el `Dog` para evitar el posible viaje de ida y vuelta para iniciar el proxy. Esto se hace por medio del método `addJoin()`, el cual le permite unirse en una asociación o colección.

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d
 WHERE c.DOG_ID = d.D_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

En este ejemplo los `Cats` retornados tendrán su propiedad `dog` completamente iniciada sin ningún viaje extra de ida y vuelta a la base de datos. Observe que agregó un nombre alias ("cat") para poder especificar la ruta de la propiedad de destino de la unión. Es posible hacer la misma unión temprana para colecciones, por ejemplo, si el `Cat` tuviese en lugar un `Dog` uno-a-muchos.

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE
 c.ID = d.CAT_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

En este punto estamos alcanzando los límites de lo que es posible con las consultas nativas sin empezar a mejorar las consultas sql para hacerlas utilizables en Hibernate. Los problemas empiezan a surgir cuando las entidades múltiples retornadas son del mismo tipo o cuando no son suficientes los nombres de las columnas/alias predeterminados.

### 18.1.4. Devolución de entidades múltiples

Hasta ahora se ha asumido que los nombres de las columnas del grupo de resultados son las mismas que los nombres de columnas especificados en el documento de mapeo. Esto puede llegar a ser problemático para las consultas SQL que unen múltiples tablas ya que los mismos nombres de columnas pueden aparecer en más de una tabla.

Se necesita una inyección de alias en las columnas en la siguiente consulta (que con mucha probabilidad fallará):

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

La intención de esta consulta es retornar dos instancias Cat por fila: un gato y su mamá. Sin embargo, esto fallará debido a que hay un conflicto de nombres;las instancias se encuentran mapeadas a los mismos nombres de columna. También en algunas bases de datos los alias de las columnas retornadas serán con mucha probabilidad de la forma "c.IDENTIFICACION", "c.NOMBRE", etc, los cuales no son iguales a las columnas especificadas en los mapeos ("IDENTIFICACION" y "NOMBRE").

La siguiente forma no es vulnerable a la duplicación de nombres de columnas:

```
sess.createSQLQuery("SELECT {cat.*}, {mother.*}  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

Se especifica esta consulta:

- la cadena de consultas SQL, con un espacio reservado para que Hibernate inserte alias de columnas
- las entidades devueltas por la consulta

La anotación {cat.*} y {mother.*} que se utilizó anteriormente es la abreviatura para "todas las propiedades". Opcionalmente puede enumerar las columnas explícitamente, pero inclusive en este caso Hibernate inyecta los alias de columnas SQL para cada propiedad. El espacio para un alias de columna es sólamente el nombre calificado de la propiedad del alias de la tabla. En el siguiente ejemplo, recuperamos Cats y sus madres desde una tabla diferente (cat_log) a la declarada en los meta datos de mapeo. Inclusive puede utilizar los alias de propiedad en la cláusula where.

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
         "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {mother.*} " +
         "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
        .addEntity("cat", Cat.class)
        .addEntity("mother", Cat.class).list()
```

### 18.1.4.1. Referencias de propiedad y alias

Para la mayoría de los casos, se necesita la inyección del alias anterior. Para las consultas relacionadas con mapeos más complejos como propiedades compuestas, discriminadores de herencia, colecciones, etc, existen alias especificos a utilizar para permitir que Hibernate inyecte los alias apropiados.

La siguiente tabla muestra las diferentes maneras de utilizar la inyección de alias. Note que los nombres alias en el resultado son simplemente ejemplos; cada alias tendrá un nombre único y probablemente diferente cuando se utilice.

**Tabla 18.1. Nombres con inyección alias**

| Descripción | Sintaxis | Ejemplo |
|---|---|---|
| Una propiedad simple | `{[aliasname].[propertyname]` | `A_NAME as {item.name}` |
| Una propiedad compuesta | `{[aliasname].[componentname].[propertyname]}` | `CURRENCY as {item.amount.currency}, VALUE as {item.amount.value}` |
| Discriminador de una entidad | `{[aliasname].class}` | `DISC as {item.class}` |
| Todas las propiedades de una entidad | `{[aliasname].*}` | `{item.*}` |
| Una clave de colección | `{[aliasname].key}` | `ORGID as {coll.key}` |
| La identificación -id- de una colección | `{[aliasname].id}` | `EMPID as {coll.id}` |
| El elemento de una colección | `{[aliasname].element}` | `XID as {coll.element}` |
| propiedad del elemento en la colección | `{[aliasname].element.[propertyname]}` | `NAME as {coll.element.name}` |
| Todas las propiedades del elemeto en la colección | `{[aliasname].element.*}` | `{coll.element.*}` |
| Todas las propiedades de la colección | `{[aliasname].*}` | `{coll.*}` |

## 18.1.5. Devolución de entidades no-administradas

Es posible aplicar un ResultTransformer para consultas SQL nativas, permitiéndole retornar entidades no-administradas.

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

Se especifica esta consulta:

- la cadena de consulta SQL

- un transformador de resultado

La consulta anterior devolverá una lista de `CatDTO` a la cual se ha instanciado e inyectado los valores de NOMBRE y FECHA DE NACIMIENTO en su propiedades o campos correspondientes.

### 18.1.6. Manejo de herencias

Las consultas SQL nativas, las cuales consultan por entidades que son mapeadas como parte de una herencia tienen que incluir todas las propiedades para la clase base y todas sus subclases.

### 18.1.7. Parámetros

Las consultas SQL nativas soportan parámetros nombrados así como posicionales:

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like ?").addEntity(Cat.class);
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like :name").addEntity(Cat.class);
List pusList = query.setString("name", "Pus%").list();
```

## 18.2. Consultas SQL nombradas

Named SQL queries can also be defined in the mapping document and called in exactly the same way as a named HQL query (see *Sección 11.4.1.7, "Externalización de consultas con nombre"*). In this case, you do *not* need to call `addEntity()`.

**Ejemplo 18.1. Named sql query using the <sql-query> maping element**

```
<sql-query name="persons">
    <return alias="person" class="eg.Person"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex}
    FROM PERSON person
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

**Ejemplo 18.2. Execution of a named query**

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

El elemento `<return-join>` se utiliza para unir asociaciones y el elemento `<load-collection>` se usa para definir consultas, las cuales dan inicio a colecciones.

### Ejemplo 18.3. Named sql query with association

```
<sql-query name="personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

Una consulta SQL nombrada puede devolver un valor escalar. Tiene que declarar el alias de la columna y el tipo de Hibernate utilizando el elemento `<return-scalar>`:

### Ejemplo 18.4. Named query returning a scalar

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

Puede externalizar el grupo de resultados mapeando información en un elemento `<resultset>`, el cual le permitirá reutilizarlos a través de consultas nombradas o por medio de la API `setResultSetMapping()`.

### Ejemplo 18.5. <resultset> mapping used to externalize mapping information

```
<resultset name="personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
</resultset>

<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
```

```
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

Opcionalmente, puede utilizar el grupo de resultados mapeando la información en sus archivos hbm directamente en código java.

### Ejemplo 18.6. Programmatically specifying the result mapping information

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id"
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

So far we have only looked at externalizing SQL queries using Hibernate mapping files. The same concept is also available with anntations and is called named native queries. You can use `@NamedNativeQuery` (`@NamedNativeQueries`) in conjunction with `@SqlResultSetMapping` (`@SqlResultSetMappings`). Like `@NamedQuery`, `@NamedNativeQuery` and `@SqlResultSetMapping` can be defined at class level, but their scope is global to the application. Lets look at a view examples.

*Ejemplo 18.7, “Named SQL query using @NamedNativeQuery together with @SqlResultSetMapping”* shows how a `resultSetMapping` parameter is defined in `@NamedNativeQuery`. It represents the name of a defined `@SqlResultSetMapping`. The resultset mapping declares the entities retrieved by this native query. Each field of the entity is bound to an SQL alias (or column name). All fields of the entity including the ones of subclasses and the foreign key columns of related entities have to be present in the SQL query. Field definitions are optional provided that they map to the same column name as the one declared on the class property. In the example 2 entities, `Night` and `Area`, are returned and each property is declared and associated to a column name, actually the column name retrieved by the query.

In *Ejemplo 18.8, “Implicit result set mapping”* the result set mapping is implicit. We only describe the entity class of the result set mapping. The property / column mappings is done using the entity mapping values. In this case the model property is bound to the model_txt column.

Finally, if the association to a related entity involve a composite primary key, a `@FieldResult` element should be used for each foreign key column. The `@FieldResult` name is composed of the property name for the relationship, followed by a dot ("."), followed by the name or the field or property of the primary key. This can be seen in *Ejemplo 18.9, “Using dot notation in @FieldResult for specifying associations ”*.

### Ejemplo 18.7. Named SQL query using `@NamedNativeQuery` together with `@SqlResultSetMapping`

```
@NamedNativeQuery(name="night&area", query="select night.id nid, night.night_duration, "
    + " night.night_date, area.id aid, night.area_id, area.name "
    + "from Night night, Area area where night.area_id = area.id",
                  resultSetMapping="joinMapping")
@SqlResultSetMapping(name="joinMapping", entities={
    @EntityResult(entityClass=Night.class, fields = {
        @FieldResult(name="id", column="nid"),
        @FieldResult(name="duration", column="night_duration"),
        @FieldResult(name="date", column="night_date"),
        @FieldResult(name="area", column="area_id"),
        discriminatorColumn="disc"
    }),
    @EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class, fields = {
        @FieldResult(name="id", column="aid"),
        @FieldResult(name="name", column="name")
    })
    }
)
```

### Ejemplo 18.8. Implicit result set mapping

```
@Entity
@SqlResultSetMapping(name="implicit",
                     entities=@EntityResult(entityClass=SpaceShip.class))
@NamedNativeQuery(name="implicitSample",
                  query="select * from SpaceShip",
                  resultSetMapping="implicit")
public class SpaceShip {
    private String name;
    private String model;
    private double speed;

    @Id
    public String getName() {
        return name;
    }

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

    @Column(name="model_txt")
    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
```

```
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }
}
```

### Ejemplo 18.9. Using dot notation in @FieldResult for specifying associations

```
@Entity
@SqlResultSetMapping(name="compositekey",
        entities=@EntityResult(entityClass=SpaceShip.class,
            fields = {
                    @FieldResult(name="name", column = "name"),
                    @FieldResult(name="model", column = "model"),
                    @FieldResult(name="speed", column = "speed"),
                    @FieldResult(name="captain.firstname", column = "firstn"),
                    @FieldResult(name="captain.lastname", column = "lastn"),
                    @FieldResult(name="dimensions.length", column = "length"),
                    @FieldResult(name="dimensions.width", column = "width")
                    }),
        columns = { @ColumnResult(name = "surface"),
                    @ColumnResult(name = "volume") } )

@NamedNativeQuery(name="compositekey",
     query="select name, model, speed, lname as lastn, fname as firstn, length, width, length
 * width as surface from SpaceShip",
    resultSetMapping="compositekey")
} )
public class SpaceShip {
    private String name;
    private String model;
    private double speed;
    private Captain captain;
    private Dimensions dimensions;

    @Id
    public String getName() {
        return name;
    }

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

    @ManyToOne(fetch= FetchType.LAZY)
    @JoinColumns( {
            @JoinColumn(name="fname", referencedColumnName = "firstname"),
            @JoinColumn(name="lname", referencedColumnName = "lastname")
            } )
    public Captain getCaptain() {
        return captain;
    }
```

```
    public void setCaptain(Captain captain) {
        this.captain = captain;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }

    public Dimensions getDimensions() {
        return dimensions;
    }

    public void setDimensions(Dimensions dimensions) {
        this.dimensions = dimensions;
    }
}

@Entity
@IdClass(Identity.class)
public class Captain implements Serializable {
    private String firstname;
    private String lastname;

    @Id
    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    @Id
    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}
```

### Sugerencia

If you retrieve a single entity using the default mapping, you can specify the `resultClass` attribute instead of `resultSetMapping`:

```
@NamedNativeQuery(name="implicitSample", query="select * from
 SpaceShip", resultClass=SpaceShip.class)
public class SpaceShip {
```

In some of your native queries, you'll have to return scalar values, for example when building report queries. You can map them in the `@SqlResultsetMapping` through `@ColumnResult`. You actually can even mix, entities and scalar returns in the same native query (this is probably not that common though).

**Ejemplo 18.10. Scalar values via `@ColumnResult`**

```
@SqlResultSetMapping(name="scalar", columns=@ColumnResult(name="dimension"))
@NamedNativeQuery(name="scalar", query="select length*width as dimension from
 SpaceShip", resultSetMapping="scalar")
```

An other query hint specific to native queries has been introduced: `org.hibernate.callable` which can be true or false depending on whether the query is a stored procedure or not.

## 18.2.1. Utilización de la propiedad return para especificar explícitamente los nombres de columnas/alias

Con `<return-property>` usted puede decirle a Hibernate explícitamente qué alias de columnas se deben utilizar, en vez de utilizar la sintaxis `{}` para dejar que Hibernate inyecte sus propios alias. Por ejemplo:

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
        <return-property name="name" column="myName"/>
        <return-property name="age" column="myAge"/>
        <return-property name="sex" column="mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```

`<return-property>` también funciona con columnas múltiples. Esto resuelve una limitación con la sintaxis `{}`, la cual no puede permitir control muy detallado de propiedades multi-columnas.

```
<sql-query name="organizationCurrentEmployments">
    <return alias="emp" class="Employment">
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
        <return-property name="endDate" column="myEndDate"/>
    </return>
        SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.employer},
        STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDate},
        REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VALUE, CURRENCY
        FROM EMPLOYMENT
        WHERE EMPLOYER = :id AND ENDDATE IS NULL
        ORDER BY STARTDATE ASC
</sql-query>
```

En este ejemplo utilizamos `<return-property>` en combinación junto con la sintaxis `{}` para inyección. Esto le permite a los usuarios escoger cómo quieren referirse a la columna y a las propiedades.

Si su mapeo tiene un discriminador usted tiene que utilizar `<return-discriminator>` para especificar la columna discriminadora.

## 18.2.2. Utilización de procedimientos para consultas

Hibernate 3 brinda soporte para consultas por medio de procedimientos almacenados y funciones. La mayoría de la siguiente documentación es igual para ambos. La función/ procedimiento almacenado tiene que retornar un grupo de resultados como el primer parámetro de salida para poder trabajar con Hibernate. A continuación hay un ejemplo de tal función almacenada en Oracle 9 y posteriores:

```
CREATE OR REPLACE FUNCTION selectAllEmployments
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
 SELECT EMPLOYEE, EMPLOYER,
 STARTDATE, ENDDATE,
 REGIONCODE, EID, VALUE, CURRENCY
 FROM EMPLOYMENT;
      RETURN  st_cursor;
 END;
```

Para utilizar esta consulta en Hibernate u.d necesita mapearla por medio de una consulta nombrada.

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
```

```
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>
        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>
        <return-property name="regionCode" column="REGIONCODE"/>
        <return-property name="id" column="EID"/>
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

Los procedimientos almacenados actualmente sólo retornan escalares y entidades. No se soporta `<return-join>` ni `<load-collection>` .

### 18.2.2.1. Reglas/limitaciones para utilizar procedimientos almacenados

Para utilizar procedimientos almacenados con Hibernate, debe seguir ciertas reglas de funciones/procedimientos. Si no siguen esas reglas entonces no se pueden utilizar con Hibernate. Si todavía quiere utilizar estos procedimientos tiene que ejecutarlos por medio de `session.connection()`. Las reglas son diferentes para cada base de datos debido a que los vendedores de la base de datos tienen diferentes sintaxis/semántica de procedimientos almacenados.

Las consultas de procedimientos almacenados no se pueden llamar con `setFirstResult()/setMaxResults()`.

La forma de la llamada recomendada es SQL92 estándar: `{ ? = call functionName(<parameters>) }` o `{ ? = call procedureName(<parameters>}`. No se soporta la sintaxis de llamadas nativas.

Para Oracle aplican las siguientes reglas:

- Una función tiene que retornar un grupo de resultados. El primer parámetro de un procedimiento tiene que ser un `OUT` que retorna un grupo de resultados. Esto se hace utilizando un tipo `SYS_REFCURSOR` en Oracle 9 o 10. En Oracle necesita definir un tipo `REF CURSOR`. Consulte la documentación de Oracle para obtener mayor información.

Para Sybase o el servidor MS SQL aplican las siguientes reglas:

- El procedimiento tiene que retornar un grupo de resultados. Observe que debido a que estos servidores pueden retornar grupos de resultados múltiples y cuentas actualizadas, Hibernate iterará los resultados y tomará el primer resultado que sea un grupo resultados como su valor retornado. Todo lo demás será descartado.
- Si puede habilitar `SET NOCOUNT ON` en su procedimiento probablemente será más eficiente, pero no es un requerimiento.

## 18.3. Personalice SQL para crear, actualizar y borrar

Hibernate3 can use custom SQL for create, update, and delete operations. The SQL can be overridden at the statement level or inidividual column level. This section describes statement overrides. For columns, see *Sección 5.6, "Column transformers: read and write expressions"*. *Ejemplo 18.11, "Custom CRUD via annotations"* shows how to define custom SQL operatons using annotations.

### Ejemplo 18.11. Custom CRUD via annotations

```
@Entity
@Table(name="CHAOS")
@SQLInsert( sql="INSERT INTO CHAOS(size, name, nickname, id) VALUES(?,upper(?),?,?)")
@SQLUpdate( sql="UPDATE CHAOS SET size = ?, name = upper(?), nickname = ? WHERE id = ?")
@SQLDelete( sql="DELETE CHAOS WHERE id = ?")
@SQLDeleteAll( sql="DELETE CHAOS")
@Loader(namedQuery = "chaos")
@NamedNativeQuery(name="chaos", query="select id, size, name, lower( nickname ) as nickname from
 CHAOS where id= ?", resultClass = Chaos.class)
public class Chaos {
    @Id
    private Long id;
    private Long size;
    private String name;
    private String nickname;
```

`@SQLInsert`, `@SQLUpdate`, `@SQLDelete`, `@SQLDeleteAll` respectively override the INSERT, UPDATE, DELETE, and DELETE all statement. The same can be achieved using Hibernate mapping files and the `<sql-insert>`, `<sql-update>` and `<sql-delete>` nodes. This can be seen in *Ejemplo 18.12, "Custom CRUD XML"*.

### Ejemplo 18.12. Custom CRUD XML

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ? )</sql-insert>
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-update>
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

If you expect to call a store procedure, be sure to set the `callable` attribute to `true`. In annotations as well as in xml.

To check that the execution happens correctly, Hibernate allows you to define one of those three strategies:

- none: no check is performed: the store procedure is expected to fail upon issues
- count: use of rowcount to check that the update is successful
- param: like COUNT but using an output parameter rather that the standard mechanism

To define the result check style, use the `check` parameter which is again available in annoations as well as in xml.

You can use the exact same set of annotations respectively xml nodes to override the collection related statements -see *Ejemplo 18.13, "Overriding SQL statements for collections using annotations"*.

### Ejemplo 18.13. Overriding SQL statements for collections using annotations

```
@OneToMany
@JoinColumn(name="chaos_fk")
@SQLInsert( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = ? where id = ?")
@SQLDelete( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = null where id = ?")
private Set<CasimirParticle> particles = new HashSet<CasimirParticle>();
```

**Sugerencia**

The parameter order is important and is defined by the order Hibernate handles properties. You can see the expected order by enabling debug logging for the `org.hibernate.persister.entity` level. With this level enabled Hibernate will print out the static SQL that is used to create, update, delete etc. entities. (To see the expected sequence, remember to not include your custom SQL through annotations or mapping files as that will override the Hibernate generated static sql)

Overriding SQL statements for secondary tables is also possible using `@org.hibernate.annotations.Table` and either (or all) attributes `sqlInsert`, `sqlUpdate`, `sqlDelete`:

### Ejemplo 18.14. Overriding SQL statements for secondary tables

```
@Entity
@SecondaryTables({
    @SecondaryTable(name = "`Cat nbr1`"),
    @SecondaryTable(name = "Cat2"})
@org.hibernate.annotations.Tables( {
    @Table(appliesTo = "Cat", comment = "My cat table" ),
    @Table(appliesTo = "Cat2", foreignKey = @ForeignKey(name="FK_CAT2_CAT"), fetch = FetchMode.SELECT,
        sqlInsert=@SQLInsert(sql="insert into Cat2(storyPart2, id) values(upper(?), ?)") )
} )
```

```
public class Cat implements Serializable {
```

The previous example also shows that you can give a comment to a given table (primary or secondary): This comment will be used for DDL generation.

**Sugerencia**

The SQL is directly executed in your database, so you can use any dialect you like. This will, however, reduce the portability of your mapping if you use database specific SQL.

Last but not least, stored procedures are in most cases required to return the number of rows inserted, updated and deleted. Hibernate always registers the first statement parameter as a numeric output parameter for the CUD operations:

### Ejemplo 18.15. Stored procedures and their return value

```
CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN VARCHAR2)
    RETURN NUMBER IS
BEGIN

    update PERSON
    set
        NAME = uname,
    where
        ID = uid;

    return SQL%ROWCOUNT;

END updatePerson;
```

## 18.4. Personalice SQL para cargar

You can also declare your own SQL (or HQL) queries for entity loading. As with inserts, updates, and deletes, this can be done at the individual column level as described in *Sección 5.6, "Column transformers: read and write expressions"* or at the statement level. Here is an example of a statement level override:

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

Esta es tan sólo una declaración de consulta nombrada, como se discutió anteriormente. Puede referenciar esta consulta nombrada en un mapeo de clase:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

Esto funciona inclusive con procedimientos almacenados.

Puede incluso definir una consulta para la carga de colección:

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

También puede definir un cargador de entidad que cargue una colección con una unión temprana:

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

The annotation equivalent `<loader>` is the @Loader annotation as seen in *Ejemplo 18.11, "Custom CRUD via annotations"*.

# Filtración de datos

Hibernate3 proporciona un nuevo enfoque innovador para manejar datos con reglas de "visibilidad". Un *filtro Hibernate* es un filtro global, con nombre y parametrizado que puede ser habilitado o deshabilitado para una sesión de Hibernate especifica.

## 19.1. Filtros de Hibernate

Hibernate3 tiene la habilidad de predefinir criterios de filtros y unir esos filtros tanto a nivel de clase como de colección. Un criterio de filtro le permite definir una cláusula de restricción muy similar al atributo existente "where" disponible en el elemento class y en varios elementos de colección. Sin embargo, las condiciones de estos filtros se pueden parametrizar. La aplicación puede tomar la decisión en tiempo de ejecución de si los filtros deben estar habilitados y cuáles deben ser sus parámetros. Los filtros se pueden utilizar como vistas de la base de datos, pero parametrizados dentro de la aplicación.

Using annotatons filters are defined via `@org.hibernate.annotations.FilterDef` or `@org.hibernate.annotations.FilterDefs`. A filter definition has a `name()` and an array of parameters(). A parameter will allow you to adjust the behavior of the filter at runtime. Each parameter is defined by a `@ParamDef` which has a name and a type. You can also define a `defaultCondition()` parameter for a given `@FilterDef` to set the default condition to use when none are defined in each individual `@Filter`. `@FilterDef`(s) can be defined at the class or package level.

We now need to define the SQL filter clause applied to either the entity load or the collection load. `@Filter` is used and placed either on the entity or the collection element. The connection between `@FilterName` and `@Filter` is a matching name.

### Ejemplo 19.1. @FilterDef and @Filter annotations

```
@Entity
@FilterDef(name="minLength", parameters=@ParamDef( name="minLength", type="integer" ) )
@Filters( {
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length"),
    @Filter(name="minLength", condition=":minLength <= length")
} )
public class Forest { ... }
```

When the collection use an association table as a relational representation, you might want to apply the filter condition to the association table itself or to the target entity table. To apply the constraint on the target entity, use the regular `@Filter` annotation. However, if you want to target the association table, use the `@FilterJoinTable` annotation.

### Ejemplo 19.2. Using `@FilterJoinTable` for filterting on the association table

```
@OneToMany
    @JoinTable
    //filter on the target entity table
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length")
    //filter on the association table
    @FilterJoinTable(name="security", condition=":userlevel >= requredLevel")
    public Set<Forest> getForests() { ... }
```

Using Hibernate mapping files for defining filters the situtation is very similar. The filters must first be defined and then attached to the appropriate mapping elements. To define a filter, use the `<filter-def/>` element within a `<hibernate-mapping/>` element:

### Ejemplo 19.3. Defining a filter definition via `<filter-def>`

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

This filter can then be attached to a class or collection (or, to both or multiples of each at the same time):

### Ejemplo 19.4. Attaching a filter to a class or collection using `<filter>`

```
<class name="myClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>

    <set ...>
        <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
    </set>
</class>
```

Los métodos en `Session` son: `enableFilter(String filterName)`, `getEnabledFilter(String filterName)` y `disableFilter(String filterName)`. Por defecto, los filtros *no* están habilitados para una sesión dada. Los filtros deben ser habilitados explícitamente por medio del uso del método `Session.enableFilter()`, el cual retorna una instancia de la interfaz `Filter`. Si se utiliza el filtro simple definido anteriormente, esto se vería así:

```
session.enableFilter("myFilter").setParameter("myFilterParam", "some-value");
```

Los métodos en la interfaz org.hibernate.Filter permiten el encadenamiento de métodos, lo cual es bastante común en gran parte de Hibernate.

Este es un ejemplo completo, utilizando datos temporales con un patrón efectivo de fechas de registro:

```
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
...
    <many-to-one name="department" column="dept_id" class="Department"/>
    <property name="effectiveStartDate" type="date" column="eff_start_dt"/>
    <property name="effectiveEndDate" type="date" column="eff_end_dt"/>
...
    <!--
        Note that this assumes non-terminal records have an eff_end_dt set to
        a max db date for simplicity-sake
    -->
    <filter name="effectiveDate"
            condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
</class>

<class name="Department" ...>
...
    <set name="employees" lazy="true">
        <key column="dept_id"/>
        <one-to-many class="Employee"/>
        <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
    </set>
</class>
```

Con el fin de asegurarse de que siempre recibirá los registros efectivos actualmente, habilite el filtro en la sesión antes de recuperar los datos de los empleados:

```
Session session = ...;
session.enableFilter("effectiveDate").setParameter("asOfDate", new Date());
List results = session.createQuery("from Employee as e where e.salary > :targetSalary")
         .setLong("targetSalary", new Long(1000000))
         .list();
```

En el HQL anterior, aunque sólo mencionamos explícitamente una restricción de salario en los resultados, debido al filtro habilitado la consulta sólo retornará empleados actualmente activos que tengan un salario mayor a un millón de dólares.

Si quiere utilizar filtros con unión externa, ya sea a través de HQL, o bien de recuperación de carga, tenga cuidado en la dirección de expresión de la condición. Lo más seguro es configurar esto para una unión externa izquierda. Coloque el parámetro primero seguido del nombre(s) de la(s) columna(s) después del operador.

Después de definir un filtro, este se puede unir a múltiples entidades y/o colecciones cada una con su propia condición. Esto puede llegar a ser problemático cuando las condiciones son las

mismas. Así que el usar `<filter-def/>` le permite definir una condición por defecto, ya sea como atributo o como CDATA:

```
<filter-def name="myFilter" condition="abc > xyz">...</filter-def>
<filter-def name="myOtherFilter">abc=xyz</filter-def>
```

Esta condición predeterminada se utilizará cuando se una el filtro a algo sin especificar una condición. Esto significa que usted le puede dar una condición especifica como parte del anexo del filtro, el cual substituye la condición por defecto en ese caso en particular.

# Mapeo XML

*XML Mapping is an experimental feature in Hibernate 3.0 and is currently under active development.*

## 20.1. Trabajo con datos XML

Hibernate le permite trabajar con datos XML persistentes en casi de la misma forma que trabaja con POJOs persistentes. Un árbol XML analizado semáticamente se puede considerar como otra manera de representar los datos relacionales a nivel de objetos, en lugar de POJOs.

Hibernate soporta dom4j como API para manipular árboles XML. Puede escribir consultas que recuperen árboles dom4j de la base de datos y puede tener cualquier modificación que realice al árbol sincronizada automáticamente con la base de datos. Incluso puede tomar un documento XML, analizarlo sintácticamente utilizando dom4j, y escribirlo a la base de datos con cualquiera de las operaciones básicas de Hibernate: `persist()`, `saveOrUpdate()`, `merge()`, `delete()`, `replicate()` (merge aún no está soportado).

Esta funcionalidad tiene muchas aplicaciones incluyendo la importación/exportación de datos, externalización de datos de entidad por medio de JMS o SOAP y reportes basados en XSLT.

Un sólo mapeo se puede utilizar para mapear simultáneamente las propiedades de una clase y los nodos de un documento XML a la base de datos, o si no hay ninguna clase a mapear, se puede utilizar para mapear sólo el XML.

### 20.1.1. Especificación de los mapeos de XML y de clase en conjunto

Este es un ejemplo del mapeo de un POJO y XML de manera simultánea:

```
<class name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="accountId"
            column="ACCOUNT_ID"
            node="@id"/>

    <many-to-one name="customer"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"/>

    <property name="balance"
            column="BALANCE"
            node="balance"/>

    ...
```

```
</class
>
```

### 20.1.2. Especificación de sólo un mapeo XML

Este es un ejemplo donde no hay ninguna clase POJO:

```
<class entity-name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="id"
            column="ACCOUNT_ID"
            node="@id"
            type="string"/>

    <many-to-one name="customerId"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"
            entity-name="Customer"/>

    <property name="balance"
            column="BALANCE"
            node="balance"
            type="big_decimal"/>

    ...

</class
>
```

Este mapeo le permite acceder a los datos como un árbol dom4j o como un grafo de parejas nombre/valor de propiedad o`Map`as de Java. Los nombres de propiedades son construcciones puramente lógicas a las que se puede hacer referencia en consultas HQL.

## 20.2. Mapeo de metadatos XML

Muchos elementos de mapeo de Hibernate aceptan el atributo `node`. Esto le permite especificar el nombre de un atributo o elemento XML que contenga los datos de la propiedad o entidad. El formato del atributo `node` tiene que ser uno de los siguientes:

- `"element-name"` - mapea al elemento XML mencionado
- `"@attribute-name"`: mapea al atributo XML mencionado
- `"."` - mapea al elemento padre
- `"element-name/@attribute-name"`: mapea al atributo mencionado del elemento nombrado

Para las colecciones y asociaciones monovaluadas, existe un atributo adicional `embed-xml`. Si `embed-xml="true"`, el cual es el valor por defecto, el árbol XML para la entidad asociada (o colección de tipo de valor) será incluida directamente en el árbol XML para la entidad que

posee la asociación. De otra manera, si `embed-xml="false"`, entonces sólo el valor identificador referenciado aparecerá en el XML para asociaciones de punto único y para las colecciones simplemente no aparecerá.

No deje `embed-xml="true"` para demasiadas asociaciones ya que XML no se ocupa bien de la circularidad.

```
<class name="Customer"
        table="CUSTOMER"
        node="customer">

    <id name="id"
            column="CUST_ID"
            node="@id"/>

    <map name="accounts"
            node="."
            embed-xml="true">
        <key column="CUSTOMER_ID"
                not-null="true"/>
        <map-key column="SHORT_DESC"
                node="@short-desc"
                type="string"/>
        <one-to-many entity-name="Account"
                embed-xml="false"
                node="account"/>
    </map>

    <component name="name"
            node="name">
        <property name="firstName"
                node="first-name"/>
        <property name="initial"
                node="initial"/>
        <property name="lastName"
                node="last-name"/>
    </component>

    ...

</class
>
```

En este caso, la colección de ids de cuenta están incluídos, pero no los datos reales de cuenta. La siguiente consulta HQL:

```
from Customer c left join fetch c.accounts where c.lastName like :lastName
```

retornaría conjuntos de datos como este:

```
<customer id="123456789">
```

```
    <account short-desc="Savings"
>987632567</account>
    <account short-desc="Credit Card"
>985612323</account>
    <name>
        <first-name
>Gavin</first-name>
        <initial
>A</initial>
        <last-name
>King</last-name>
    </name>
    ...
</customer
>
```

Si establece `embed-xml="true"` en el mapeo `<one-to-many>`, puede que los datos se vean así:

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings">
        <customer id="123456789"/>
        <balance
>100.29</balance>
    </account>
    <account id="985612323" short-desc="Credit Card">
        <customer id="123456789"/>
        <balance
>-2370.34</balance>
    </account>
    <name>
        <first-name
>Gavin</first-name>
        <initial
>A</initial>
        <last-name
>King</last-name>
    </name>
    ...
</customer
>
```

## 20.3. Manipulación de datos XML

Puede releer y actualizar documentos XML en la aplicación. Puede hacer esto obteniendo una sesión dom4j:

```
Document doc = ....;

Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

List results = dom4jSession
```

```
    .createQuery("from Customer c left join fetch c.accounts where c.lastName like :lastName")
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
    Element customer = (Element) results.get(i);
    doc.add(customer);
}

tx.commit();
session.close();
```

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

Element cust = (Element) dom4jSession.get("Customer", customerId);
for ( int i=0; i<results.size(); i++ ) {
    Element customer = (Element) results.get(i);
    //change the customer name in the XML and database
    Element name = customer.element("name");
    name.element("first-name").setText(firstName);
    name.element("initial").setText(initial);
    name.element("last-name").setText(lastName);
}

tx.commit();
session.close();
```

Es útil combinar esta funcionalidad con la operación `replicate()` de Hibernate para implementar la importación/exportación de datos basada en XML.

# Mejoramiento del rendimiento

## 21.1. Estrategias de recuperación

Hibernate utiliza una *estrategia de recuperación* para recuperar los objetos asociados cuando la aplicación necesita navegar la asociación. Las estrategias de recuperación se pueden declarar en los metadatos de mapeo O/R, o se pueden sobrescribir por medio de una HQL particular o una petición `Criteria`.

Hibernate3 define las siguientes estrategias de recuperación:

- *Recuperación por unión (join fetching)*: Hibernate recupera la instancia asociada o la colección en el mismo `SELECT`, utilizando un `OUTER JOIN`.
- *Recuperación por selección (select fetching)*: se utiliza un segundo `SELECT` para recuperar la entidad o colección asocidas. A menos que deshabilite explícitamente la recuperación perezosa especificando `lazy="false"`, la segunda selección sólo será ejecutada cuando acceda a la asociación.
- *Recuperación por subselección (subselect fetching)*: se utiliza un segundo `SELECT` para recuperar las colecciones asociadas de todas las entidades recuperadas en una consulta o recuperación previa. A menos de que deshabilite explícitamente la recuperación perezosa especificando `lazy="false"`, esta segunda selección sólo se ejecutará cuando acceda a la asociación.
- *Recuperación en lote*: una estrategia de optimización para la recuperación por selección. Hibernate recupera un lote de instancias de entidad o colecciones en un solo `SELECT`, especificando una lista de claves principales o de claves foráneas.

Hibernate también distingue entre:

- *Recuperación inmediata*: una asociación, colección o atributo se recupera inmediatamente cuando se carga el dueño.
- *Recuperación perezosa de colecciones*: una colección se recupera cuando la aplicación invoca una operación sobre esa colección. Este es el valor predeterminado para las colecciones.
- *Recuperación de colección "extra-perezoza"* : se accede a elementos individuales desde la base de datos cuando se necesita. Hibernate intenta no recuperar toda la colección en la memoria a menos de que sea absolutamente necesario. Esto es apropiado para colecciones muy grandes.
- *Recuperación por proxy*: una asociación monovaluada se recupera cuando se invoca un método que no sea el getter del identificador sobre el objeto asociado.
- *Recuperación "no-proxy"* : una asociación monovaluada se recupera cuando se accede a la variable de la instancia. Comparado con la recuperación por proxy, este enfoque es menos perezozo; la asociación se recupera cuando se accede sólamente al identificador. También

es más transparente ya que para la aplicación no hay proxies visibles. Este enfoque requiere instrumentación del código byte del tiempo estimado de construcción y se necesita muy raramente.

- *Recuperación perezosa de atributos*: un atributo o una asociación monovaluada se recuperan cuando se accede a la variable de la instancia. Este enfoque requiere instrumentación del código byte en tiempo estimado de construcción y se necesita muy raramente.

Aquí tenemos dos nociones ortogonales: *cuándo* se recupera la aplicación, y *cómo* se recupera. Es importante que no las confunda. Utilizamos `fetch` para afinar el rendimiento. Podemos usar `lazy` para definir un contrato sobre qué datos están siempre disponibles en cualquier instancia separada de una clase en particular.

### 21.1.1. Trabajo con asociaciones perezosas

Por defecto, Hibernate3 usa una recuperación perezosa por selección para colecciones y una recuperación por proxy perezosa para asociaciones monovaluadas. Estas políticas predeterminadas tienen sentido para casi todas las asociaciones en la mayoría de las aplicaciones.

Si configura `hibernate.default_batch_fetch_size`, Hibernate utilizará la optimización de recuperación en lotes para recuperación perezosa. Esta optimización también se puede habilitar en un nivel más detallado.

Note que el acceder a una asociación perezosa fuera del contexto de una sesión de Hibernate abierta resultará en una excepción. Por ejemplo:

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userName")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

tx.commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Error!
```

Ya que la colección de permisos no fue inicializada cuando se cerró la `Session`, la colección no será capaz de cargar su estado. *Hibernate no soporta la inicialización perezosa de objetos separados*. La solución es mover el código que lee de la colección a justo antes de que se guarde la transacción.

Opcionalmente puede utilizar una colección no perezosa o asociación, especificando `lazy="false"` para el mapeo de asociación. Sin embargo, el propósito de la inicialización perezosa es que se utilice para casi todas las colecciones y asociaciones. ¡Si define demasiadas asociaciones no perezosas en su modelo de objetos, Hibernate recuperará la base de datos entera en toda transacción.

Por otro lado, puede utilizar la recuperación por unión, la cual no es perezosa por naturaleza, en lugar de la recuperación por selección en una transacción en particular. Veremos ahora cómo personalizar la estrategia de recuperación. En Hibernate3, los mecanismos para elegir una estrategia de recuperación son idénticas para las de las asociaciones monovaluadas y las colecciones.

### 21.1.2. Afinación de las estrategias de recuperación

La recuperación por selección (la preestablecida) es extremadamente vulnerable a problemas de selección N+1, de modo que puede que queramos habilitar la recuperación por unión (join fetching) en el documento de mapeo:

```
<set name="permissions"
            fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set
```

```
<many-to-one name="mother" class="Cat" fetch="join"/>
```

La estrategia de `recuperación` definida en el documento de mapeo afecta a:

- las recuperaciones por medio de `get()` o `load()`
- las recuperaciones que ocurren implícitamente cuando se navega una asociación (recuperación perezosa)
- las consultas de `Criteria`
- las consultas HQL si se utiliza la recuperación `subselect`

Sin importar que estrategia de recuperación utilice, se garantiza que la gráfica no-perezoza definida será cargada en la memoria. Sin embargo, esto puede causar la utilización de varias selecciones inmediatas para ejecutar una consulta HQL en particular.

Usualmente, no utilizamos el documento de mapeo para personalizar la recuperación. En cambio, mantenemos el comportamiento por defecto y lo sobrescribimos para una transacción en particular, utilizando `left join fetch` en HQL. Esto le dice a Hibernate que recupere la asociación tempranamente en la primera selección, usando una unión externa. En la API de consulta de `Criteria`, usted utilizaría `setFetchMode(FetchMode.JOIN)`.

Si quiere cambiar la estrategia de recuperación utilizada por `get()` o `load()`; utilice una consulta `Criteria`. Por ejemplo:

```
User user = (User) session.createCriteria(User.class)
                .setFetchMode("permissions", FetchMode.JOIN)
                .add( Restrictions.idEq(userId) )
```

```
                .uniqueResult();
```

Esto es el equivalente de Hibernate de lo que otras soluciones ORM denominan un "plan de recuperación".

Un enfoque completamente diferente de evitar problemas con selecciones N+1 es usar el caché de segundo nivel.

### 21.1.3. Proxies de asociaciones de un sólo extremo

La recuperación perezosa de colecciones está implementada utilizando la implementación de colecciones persistentes propia de Hibernate. Sin embargo, se necesita un mecanismo diferente para un comportamiento perezoso en las asociaciones de un sólo extremo. La entidad destino de la asociación se debe tratar con proxies. Hibernate implementa proxies de inicialización perezosa para objetos persistentes utilizando la mejora del código byte en tiempo de ejecución por medio de la biblioteca CGLIB).

En el arranque, Hibernate3 genera proxies por defecto para todas las clases persistentes y los usa para habilitar la recuperación perezosa de asociaciones `muchos-a-uno` y `uno-a-uno`.

El archivo de mapeo puede declarar una interfaz a utilizar como interfaz de proxy para esa clase, con el atributo `proxy`. Por defecto, Hibernate usa una subclase de la clase. *La clase tratada con proxies debe implementar un constructor por defecto con al menos visibilidad de paquete. Recomendamos este constructor para todas las clases persistentes*.

Hay problemas potenciales que se deben tener en cuenta al extender este enfoque a las clases polimórficas. Por ejemplo:

```
<class name="Cat" proxy="Cat">
    ......
    <subclass name="DomesticCat">
        .....
    </subclass>
</class>
```

Primero, las instancias de `Cat` nunca serán objeto de un cast a `DomesticCat`, incluso aunque la instancia subyacente sea una instancia de `DomesticCat`:

```
Cat cat = (Cat) session.load(Cat.class, id);  // instantiate a proxy (does not hit the db)
if ( cat.isDomesticCat() ) {                  // hit the db to initialize the proxy
    DomesticCat dc = (DomesticCat) cat;       // Error!
    ....
}
```

Segundo, es posible romper el proxy ==:

```
Cat cat = (Cat) session.load(Cat.class, id);            // instantiate a Cat proxy
DomesticCat dc =
        (DomesticCat) session.load(DomesticCat.class, id);  // acquire new DomesticCat proxy!
System.out.println(cat==dc);                            // false
```

Sin embargo, la situación no es en absoluto tan mala como parece. Aunque tenemos ahora dos referencias a objetos proxy diferentes, la instancia subyacente será aún el mismo objeto:

```
cat.setWeight(11.0);  // hit the db to initialize the proxy
System.out.println( dc.getWeight() );  // 11.0
```

Tercero, no puede usar un proxy CGLIB para una clase `final` o una clase con algún método `final`.

Finalmente, si su objeto persistente adquiere cualquier recurso bajo instanciación (por ejemplo, en inicializadores o constructores por defecto), entonces esos recursos serán adquiridos también por el proxy. La clase del proxy es una subclase real de la clase persistente.

Estos problemas se deben a limitaciones fundamentales en el modelo de herencia única de Java. Si desea evitar estos problemas cada una de sus clases persistentes deben implementar una interfaz que declare sus métodos de negocio. Debe especificar estas interfaces en el archivo de mapeo en donde `CatImpl` implementa la interfaz `Cat` y `DomesticCatImpl` implementa la interfaz `DomesticCat`. Por ejemplo:

```
<class name="CatImpl" proxy="Cat">
    ......
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
```

Entonces los proxies para las instancias de `Cat` y `DomesticCat` pueden ser retornadas por `load()` o `iterate()`.

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
Iterator iter = session.createQuery("from CatImpl as cat where cat.name='fritz'").iterate();
Cat fritz = (Cat) iter.next();
```

**Nota**

`list()` usualmente no retorna proxies.

Las relaciones también son inicializadas perezosamente. Esto significa que debe declarar cualquier propiedad como de tipo `Cat`, no `CatImpl`.

Ciertas operaciones *no* requieren inicialización de proxies:

- `equals()`, si la clase persistente no sobrescribe `equals()`
- `hashCode()`, si la clase persistente no sobrescribe `hashCode()`
- El método getter del identificador

Hibernate detectará las clases persistentes que sobrescriban `equals()` o `hashCode()`.

Al escoger `lazy="no-proxy"` en vez del `lazy="proxy"` predeterminado, podemos evitar los problemas asociados con conversión de tipos (typecasting). Sin embargo, requiere la instrumentación de código byte en tiempo estimado de construcción y todas las operaciones resultarán en una inicialización de proxies inmediata.

### 21.1.4. Inicialización de colecciones y proxies

Hibernate lanzará una `LazyInitializationException` si se accede a una colección o proxy sin acceder fuera del ámbito de la `Session`, por ejemplo, cuando la entidad que posee la colección o que tiene la referencia al proxy esté en el estado separado.

A veces es necesario inicializar un proxy o una colección antes de cerrar la `Session`. Puede forzar la inicialización llamando a `cat.getSex()` o `cat.getKittens().size()`, por ejemplo. Sin embargo, esto puede ser confuso para los lectores del código y no es conveniente para el código genérico.

Los métodos estáticos `Hibernate.initialize()` y `Hibernate.isInitialized()` proporcionan a la aplicación una forma conveniente de trabajar con colecciones o proxies inicializados perezosamente. `Hibernate.initialize(cat)` forzará la inicialización de un proxy, `cat`, en tanto su `Session` esté todavía abierta. `Hibernate.initialize( cat.getKittens() )` tiene un efecto similar para la colección de gatitos.

Otra opción es mantener la `Session` abierta hasta que todas las colecciones y proxies necesarios hayan sido cargados. En algunas arquitecturas de aplicación, particularmente en aquellas donde el código que accede a los datos usando Hibernate, y el código que los utiliza están en capas de aplicación diferentes o procesos físicos diferentes, puede ser un problema asegurar que la `Session` esté abierta cuando se inicializa una colección. Existen dos formas básicas para abordar este tema:

- En una aplicación basada en la web se puede utilizar un filtro de servlets para cerrar la `Session` sólamente al final de una petición del usuario, una vez que la entrega de la vista esté completa (el patrón *sesión abierta en vista (open session in view)*). Por supuesto, estos sitios requieren una fuerte demanda de corrección del manejo de excepciones de la infraestructura de su aplicación. Es de una vital importancia que la `Session` esté cerrada y la transacción terminada antes de volver al usuario, incluso cuando ocurra una excepción durante le entrega de la vista. Refiérase a la Wiki de Hibernate para ver ejemplos de este patrón "Open Session in View" (sesión abierta en vista).

- En una aplicación con una capa de negocios separada, la lógica empresarial tiene que "preparar" todas las colecciones que la capa web va a necesitar antes de retornar. Esto significa que la capa empresarial debe cargar todos los datos y devolver a la capa web/presentación todos los datos ya inicializados que se requieran para un caso de uso en particular. Usualmente, la aplicación llama a `Hibernate.initialize()` para cada colección que se necesitará en la capa web (esta llamada debe tener lugar antes de que se cierre la sesión) o recupera la colección tempranamente utilizando una consulta de Hibernate con una cláusula `FETCH` o una `FetchMode.JOIN` en `Criteria`. Usualmente, esto es más fácil si adopta el patrón *Comando* en vez de una *Fachada de Sesión*.
- También puede adjuntar un objeto cargado previamente a una nueva `Session` con `merge()` o `lock()` antes de acceder a colecciones no inicializadas u otros proxies. Hibernate no y ciertamente *no debe* hacer esto automáticamente ya que introduciría semánticas de transacción improvisadas.

A veces no quiere inicializar una colección grande, pero todavía necesita alguna información sobre ella como por ejemplo, su tamaño o un subconjunto de los datos.

Puede utilizar un filtro de colecciones para obtener el tamaño de una colección sin inicializarla:

```
( (Integer) s.createFilter( collection, "select count(*)" ).list().get(0) ).intValue()
```

El método `createFilter()` también se utiliza para recuperar eficientemente subconjuntos de una colección sin necesidad de inicializar toda la colección:

```
s.createFilter( lazyCollection, "").setFirstResult(0).setMaxResults(10).list();
```

## 21.1.5. Utilización de recuperación de lotes

Usando la recuperación por lotes, Hibernate puede cargar varios proxies sin inicializar si se accede a un proxy. La recuperación en lotes es una optimización de la estrategia de recuperación por selección perezosa. Hay dos formas en que puede configurar la recuperación en lotes: a nivel de la clase y a nivel de colección.

La recuperación en lotes para clases/entidades es más fácil de entender. Considere el siguiente ejemplo: en tiempo de ejecución tiene 25 instancias de `Cat` cargadas en una `Session` y cada `Cat` tiene una referencia a su `owner`, una `Person`. La clase `Person` está mapeada con un proxy, `lazy="true"`. Si ahora itera a través de todos los cats y llama a `getOwner()` para cada uno, Hibernate por defecto, ejecutará 25 declaraciones `SELECT` para recuperar los dueños proxies. Puede afinar este comportamiento especificando un `batch-size` en el mapeo de `Person`:

```
<class name="Person" batch-size="10">...</class>
```

Hibernate ahora ejecutará sólamente tres consultas: el patrón es 10, 10, 5.

También puede habilitar la recuperación en lotes para colecciones. Por ejemplo, si cada `Person` tiene una colección perezosa de `Cats` y hay 10 personas actualmente cargadas en la `Session`, iterar a través de las 10 personas generará 10 `SELECTs`, uno para cada llamada a `getCats()`. Si habilita la recuperación en lotes para la colección de `cats` en el mapeo de `Person`, Hibernate puede recuperar por adelantado las colecciones:

```
<class name="Person">
    <set name="cats" batch-size="3">
        ...
    </set>
</class>
```

Con un `batch-size` de 3, Hibernate cargará las colecciones 3, 3, 3, 1 en cuatro `SELECTs`. Una vez más, el valor del atributo depende del número esperado de colecciones sin inicializar en una `Session` en particular.

La recuperación de colecciones en lotes es particularmente útil si tiene un árbol anidado de ítems, por ejemplo, el típico patrón de cuenta de materiales. Sin embargo, un *conjunto anidado* o una *ruta materializada* podría ser una mejor opción para árboles que sean de lectura en la mayoría de los casos.

### 21.1.6. Utilización de la recuperación por subselección

Si una colección perezosa o proxy monovaluado tiene que ser recuperado, Hibernate los carga a todos, volviendo a ejecutar la consulta original en una subselección. Esto funciona de la misma forma que la recuperación en lotes, sin carga fragmentaria.

### 21.1.7. Perfiles de recuperación

Another way to affect the fetching strategy for loading associated objects is through something called a fetch profile, which is a named configuration associated with the `org.hibernate.SessionFactory` but enabled, by name, on the `org.hibernate.Session`. Once enabled on a `org.hibernate.Session`, the fetch profile will be in affect for that `org.hibernate.Session` until it is explicitly disabled.

So what does that mean? Well lets explain that by way of an example which show the different available approaches to configure a fetch profile:

#### Ejemplo 21.1. Specifying a fetch profile using `@FetchProfile`

```
@Entity
@FetchProfile(name = "customer-with-orders", fetchOverrides = {

  @FetchProfile.FetchOverride(entity = Customer.class, association = "orders", mode = FetchMode.JOIN)
})
public class Customer {
```

```
    @Id
    @GeneratedValue
    private long id;

    private String name;

    private long customerNumber;

    @OneToMany
    private Set<Order> orders;

    // standard getter/setter
    ...
}
```

### Ejemplo 21.2. Specifying a fetch profile using `<fetch-profile>` outside `<class>` node

```
<hibernate-mapping>
    <class name="Customer">
        ...
        <set name="orders" inverse="true">
            <key column="cust_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>
    <class name="Order">
        ...
    </class>
    <fetch-profile name="customer-with-orders">
        <fetch entity="Customer" association="orders" style="join"/>
    </fetch-profile>
</hibernate-mapping>
```

### Ejemplo 21.3. Specifying a fetch profile using `<fetch-profile>` inside `<class>` node

```
<hibernate-mapping>
    <class name="Customer">
        ...
        <set name="orders" inverse="true">
            <key column="cust_id"/>
            <one-to-many class="Order"/>
        </set>
        <fetch-profile name="customer-with-orders">
            <fetch association="orders" style="join"/>
        </fetch-profile>
    </class>
    <class name="Order">
        ...
    </class>
```

```
</hibernate-mapping>
```

Now normally when you get a reference to a particular customer, that customer's set of orders will be lazy meaning we will not yet have loaded those orders from the database. Normally this is a good thing. Now lets say that you have a certain use case where it is more efficient to load the customer and their orders together. One way certainly is to use "dynamic fetching" strategies via an HQL or criteria queries. But another option is to use a fetch profile to achieve that. The following code will load both the customer *and* their orders:

### Ejemplo 21.4. Activating a fetch profile for a given `Session`

```
Session session = ...;
session.enableFetchProfile( "customer-with-orders" );  // name matches from mapping
Customer customer = (Customer) session.get( Customer.class, customerId );
```

**Nota**

`@FetchProfile` definitions are global and it does not matter on which class you place them. You can place the `@FetchProfile` annotation either onto a class or package (package-info.java). In order to define multiple fetch profiles for the same class or package `@FetchProfiles` can be used.

Actualmente solo se soportan los perfiles de recuperación de estilo unido pero se planear soportar estilos adicionales. Consulte *HHH-3414* [http://opensource.atlassian.com/projects/hibernate/browse/HHH-3414] para obtener mayores detalles.

## 21.1.8. Utilización de la recuperación perezosa de propiedades

Hibernate3 soporta la recuperación perezosa de propiedades individuales. Esta técnica de optimización también es conocida como *grupos de recuperación (fetch groups)*. Por favor, note que éste es principalmente un aspecto de marketing, ya que en la práctica, optimizar las lecturas de filas es mucho más importante que la optimización de lectura de columnas. Sin embargo, cargar sólo algunas propiedades de una clase podría ser útil en casos extremos. Por ejemplo, cuando las tablas heredadas tienen cientos de columnas y el modelo de datos no puede ser mejorado.

Para habilitar la carga perezosa de propiedades, establezca el atributo `lazy` en sus mapeos de propiedades:

```
<class name="Document">
       <id name="id">
        <generator class="native"/>
    </id>
```

```
    <property name="name" not-null="true" length="50"/>
    <property name="summary" not-null="true" length="200" lazy="true"/>
    <property name="text" not-null="true" length="2000" lazy="true"/>
</class>
```

La carga perezosa de propiedades requiere la instrumentación del código byte en tiempo de construcción. Si sus clases persistentes no se mejoran, Hibernate ignorará la configuración perezosa de propiedades y retornará a la recuperación inmediata.

Para la instrumentación del código byte, utilice la siguiente tarea Ant:

```
<target name="instrument" depends="compile">
    <taskdef name="instrument" classname="org.hibernate.tool.instrument.InstrumentTask">
        <classpath path="${jar.path}"/>
        <classpath path="${classes.dir}"/>
        <classpath refid="lib.class.path"/>
    </taskdef>

    <instrument verbose="true">
        <fileset dir="${testclasses.dir}/org/hibernate/auction/model">
            <include name="*.class"/>
        </fileset>
    </instrument>
</target>
```

Una forma diferente de evitar lecturas innecesarias de columnas, al menos para transacciones de sólo lectura es utilizar las funcionalidades de proyección de consultas HQL o Criteria. Esto evita la necesidad de procesar el código byte en tiempo de construcción y ciertamente es la solución preferida.

Puede forzar la usual recuperación temprana de propiedades utilizando `fetch all properties` en HQL.

## 21.2. El Caché de Segundo Nivel

Una `Session` de Hibernate es un caché de datos persistentes a nivel de transacción. Es posible configurar un clúster o caché a nivel de MVJ (a nivel de `SessionFactory`) sobre una base de clase-por-clase o colección-por-colección. Incluso puede enchufar un caché en clúster. Tenga en cuenta de que los cachés nunca están al tanto de los cambios que otra aplicación haya realizado al almacén persistente. Sin embargo, se pueden configurar para que los datos en caché expiren regularmente.

You have the option to tell Hibernate which caching implementation to use by specifying the name of a class that implements `org.hibernate.cache.CacheProvider` using the property `hibernate.cache.provider_class`. Hibernate is bundled with a number of built-in integrations with the open-source cache providers that are listed in *Tabla 21.1, “Proveedores de Caché”*. You can also implement your own and plug it in as outlined above. Note that versions prior to Hibernate 3.2 use EhCache as the default cache provider.

**Tabla 21.1. Proveedores de Caché**

| Caché | Clase del Provedor | Tipo | Clúster Seguro | Caché de Consultas Soportado |
|---|---|---|---|---|
| Hashtable (no fue pensado para la utilización en producción) | `org.hibernate.cache.HashtableCacheProvider` | memoria | | yes |
| EHCache | `org.hibernate.cache.EhCacheProvider` | memoria, disco | | yes |
| OSCache | `org.hibernate.cache.OSCacheProvider` | memoria, disco | | yes |
| SwarmCache | `org.hibernate.cache.SwarmCacheProvider` | clúster (ip multicast) | sí (invalidación en clúster) | |
| JBoss Cache 1.x | `org.hibernate.cache.TreeCacheProvider` | clúster (ip multicast), transaccional | sí (replicación) | sí (requiere sincronización de reloj) |
| JBoss Cache 2 | `org.hibernate.cache.jbc.JBossCacheRegionFactory` | clúster (ip multicast), transaccional | sí (replicación o invalidación) | sí (requiere sincronización de reloj) |

## 21.2.1. Mapeos de caché

As we have done in previous chapters we are looking at the two different possibiltites to configure caching. First configuration via annotations and then via Hibernate mapping files.

By default, entities are not part of the second level cache and we recommend you to stick to this setting. However, you can override this by setting the `shared-cache-mode` element in your `persistence.xml` file or by using the `javax.persistence.sharedCache.mode` property in your configuration. The following values are possible:

- `ENABLE_SELECTIVE` (Default and recommended value): entities are not cached unless explicitly marked as cacheable.
- `DISABLE_SELECTIVE`: entities are cached unless explicitly marked as not cacheable.
- `ALL`: all entities are always cached even if marked as non cacheable.

- `NONE`: no entity are cached even if marked as cacheable. This option can make sense to disable second-level cache altogether.

The cache concurrency strategy used by default can be set globaly via the `hibernate.cache.default_cache_concurrency_strategy` configuration property. The values for this property are:

- `read-only`
- `read-write`
- `nonstrict-read-write`
- `transactional`

**Nota**

It is recommended to define the cache concurrency strategy per entity rather than using a global one. Use the `@org.hibernate.annotations.Cache` annotation for that.

### Ejemplo 21.5. Definition of cache concurrency strategy via `@Cache`

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
```

Hibernate also let's you cache the content of a collection or the identifiers if the collection contains other entities. Use the `@Cache` annotation on the collection property.

### Ejemplo 21.6. Caching collections using annotations

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

*Ejemplo 21.7, "@Cache annotation with attributes"* shows the `@org.hibernate.annotations.Cache` annotations with its attributes. It allows you to define the caching strategy and region of a given second level cache.

### Ejemplo 21.7. `@Cache` annotation with attributes

```
@Cache(
    CacheConcurrencyStrategy usage();
    String region() default "";
    String include() default "all";
)
```

1. usage: the given cache concurrency strategy (NONE, READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL)
2. region (optional): the cache region (default to the fqcn of the class or the fq role name of the collection)
3. `include` (optional): all to include all properties, non-lazy to only include non lazy properties (default all).

Let's now take a look at Hibernate mapping files. There the `<cache>` element of a class or collection mapping is used to configure the second level cache. Looking at *Ejemplo 21.8, "The Hibernate <cache> mapping element"* the parallels to anotations is obvious.

### Ejemplo 21.8. The Hibernate `<cache>` mapping element

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"
    region="RegionName"
    include="all|non-lazy"
/>
```

1. `usage` especifica la estrategia de caché: `transactional`, `read-write`, `nonstrict-read-write` o `read-only`
2. `region` (opcional: por defecto es el nombre del rol de la clase o colección): especifica el nombre de la región de caché de segundo nivel.
3. `include` (opcional: por defecto es `all`) `non-lazy`: especifica que las propiedades de la entidad mapeadas con `lazy="true"` no se pueden poner en caché cuando se habilita la recuperación perezoza a nivel de atributos.

Alternatively to `<cache>`, you can use `<class-cache>` and `<collection-cache>` elements in `hibernate.cfg.xml`.

Let's now have a closer look at the different usage strategies

### 21.2.2. Estrategia: sólo lectura

Si su aplicación necesita leer pero no modificar las instancias de una clase persistente, puede utilizar un caché `read-only` (de sólo lectura). Esta es la mejor estrategia y la más simple. Incluso es totalmente segura para utilizar en un clúster.

### 21.2.3. Estrategia: lectura/escritura (read/write)

Si la aplicación necesita actualizar datos, un caché `read-write` puede ser apropiado. Esta estrategia de caché nunca se debe utilizar si se requiere un nivel de aislamiento serializable de transacciones. Si el caché se usa en un entorno JTA, tiene que especificar la propiedad `hibernate.transaction.manager_lookup_class`, mencionando una estrategia para obtener el `TransactionManager` de JTA. En otros entornos, debe asegurarse de que la transacción esté completada cuando se llame a `Session.close()` o `Session.disconnect()`. Si desea utilizar esta estrategia en un clúster, debe asegurarse de que la implementación de caché subyacente soporta bloqueos. Los provedores de caché internos *no* soportan bloqueos.

### 21.2.4. Estrategia: lectura/escritura no estricta

Si la aplicación necesita sólo ocasionalmente actualizar datos (es decir, es extremadamente improbable que dos transacciones intenten actualizar el mismo ítem simultáneamente) y no se requiere de un aislamiento de transacciones estricto, un caché `nonstrict-read-write` podría ser apropiado. Si se utiliza el caché en un entorno JTA, tiene que especificar `hibernate.transaction.manager_lookup_class`. En otros entornos, debe asegurarse que se haya completado la transacción cuando se llame a `Session.close()` o `Session.disconnect()`.

### 21.2.5. Estrategia: transaccional

La estrategia de caché `transactional` brinda soporte a provedores de cachés completamente transaccionales como JBoss TreeCache. Un caché así, sólo se puede utilizar en un entorno JTA y tiene que especificar `hibernate.transaction.manager_lookup_class`.

### 21.2.6. Compatibilidad de proveedor de caché/estrategia de concurrencia

**Importante**

Ninguno de los provedores de caché soporta todas las estrategias de concurrencia al caché.

La siguiente tabla muestra qué provedores son compatibles con qué estrategias de concurrencia.

### Tabla 21.2. Soporte a Estrategia de Concurrencia a Caché

| Caché | read-only | nonstrict-read-write | read-write | transactional |
|---|---|---|---|---|
| Hashtable (no fue pensado para la utilización en producción) | yes | yes | yes | |
| EHCache | yes | yes | yes | |
| OSCache | yes | yes | yes | |
| SwarmCache | yes | yes | | |
| JBoss Cache 1.x | yes | | | yes |
| JBoss Cache 2 | yes | | | yes |

## 21.3. Gestión de cachés

Siempre que pase un objeto a `save()`, `update()` o `saveOrUpdate()` y siempre que recupere un objeto utilizando `load()`, `get()`, `list()`, `iterate()` o `scroll()`, ese objeto se agrega al caché interno de la `Session`.

Cuando luego se llame a `flush()`, el estado de ese objeto será sincronizado con la base de datos. Si no quiere que ocurra esta sincronización o si está procesando un número enorme de objetos y necesita gestionar la memoria eficientemente, puede utilizar el método `evict()` para quitar el objeto y sus colecciones del caché de primer nivel.

### Ejemplo 21.9. Explcitly evicting a cached instance from the first level cache using `Session.evict()`

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a huge result set
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

La `Session` también proporciona un método `contains()` para determinar si una instancia pertenece al caché de la sesión.

Para expulsar todos los objetos del caché de sesión, llame a `Session.clear()`.

Para el caché de segundo nivel, hay métodos definidos en `SessionFactory` para explusar el estado en caché de una instancia, clase entera, instancia de colección o rol entero de colección.

### Ejemplo 21.10. Second-level cache eviction via `SessionFactoty.evict()` and `SessionFacyory.evictCollection()`

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class);  //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten collections
```

El `CacheMode` controla la manera en que interactúa una sesión en particular con el caché de segundo nivel:

- `CacheMode.NORMAL`: lee ítems desde y escribe ítems hacia el caché del segundo nivel
- `CacheMode.GET`: lee ítems del caché del segundo nivel. No escribe al caché de segundo nivel excepto cuando actualiza datos
- `CacheMode.PUT`: escribe ítems al caché de segundo nivel. No lee del caché de segundo nivel
- `CacheMode.REFRESH`: escribe ítems al caché de segundo nivel. No lee del caché de segundo nivel, saltándose el efecto de `hibernate.cache.use_minimal_puts`, forzando la actualización del caché de segundo nivel para todos los ítems leídos de la base de datos

Para navegar por los contenidos de una región de caché de segundo nivel o de consultas, use la API de `Statistics`:

### Ejemplo 21.11. Browsing the second-level cache entries via the `Statistics` API

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

Necesitará habilitar las estadísticas y, opcionalmente, forzar a Hibernate para que guarde las entradas del caché en un formato más fácil de entender para humanos:

### Ejemplo 21.12. Enabling Hibernate statistics

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

## 21.4. El Caché de Consultas

Los conjuntos de resultados de peticiones también pueden ponerse en caché. Esto sólamente es útil para consultas que se ejecutan frecuentemente con los mismos parámetros.

### 21.4.1. Habilitación del caché de peticiones

El poner en caché los resultados de una petición introduce algunos sobrecostos en términos del procesamiento transaccional normal de sus aplicaciones. Por ejemplo, si pone en caché los resultados de una petición frente a Person, Hibernate necesitará rastrear cuando se deben invalidar esos resultados debido a los cambios que se han guardado en Person. Eso más el hecho de que la mayoría de las aplicaciones simplemente no ganan beneficio de poner los resultados en caché, lleva a Hibernate a deshabilitar el caché de los resultados de una petición por defecto. Para utilizar el caché de peticiones primero necesita habilitar el caché de peticiones:

```
hibernate.cache.use_query_cache true
```

Esta configuración crea dos nuevas regiones de caché:

- `org.hibernate.cache.StandardQueryCache`, mantiene los resultados de la petición en caché
- `org.hibernate.cache.UpdateTimestampsCache`, mantiene los sellos de fecha de las actualizaciones más recientes a las tablas de peticiones. Estas se utilizan para validar los resultados ya que se sirven desde el caché de peticiones.

**Importante**

If you configure your underlying cache implementation to use expiry or timeouts is very important that the cache timeout of the underlying cache region for the UpdateTimestampsCache be set to a higher value than the timeouts of any of the query caches. In fact, we recommend that the the UpdateTimestampsCache region not be configured for expiry at all. Note, in particular, that an LRU cache expiry policy is never appropriate.

Como lo mencionamos anteriormente, la mayoría de las consultas no se benefician del caché o de sus resultados; de modo que por defecto las consultas individuales no se ponen en caché incluso después de habilitar el caché para peticiones. Para habilitar el caché de resultados para una petición en particular, llame a `org.hibernate.Query.setCacheable(true)`. Esta llamada permite que la consulta busque resultados existentes en caché o que agregue sus resultados al caché cuando se ejecuta.

**Nota**

El caché de peticiones no pone en caché el estado real de las entidades en el caché; pone en caché solo los valores del identificador y los resultados de tipo valor. Por esta razón, el caché de peticiones siempre se debe utilizar en conjunto con el caché de segundo nivel para aquellas entidades que se esperan poner en

caché como parte de un caché de resultados de una petición (así como con el caché de colección).

### 21.4.2. Regiones de caché de consultas

Si necesita un control muy detallado sobre las políticas de expiración del caché de consultas, puede especificar una región de caché con nombre para una consulta en particular llamando a `Query.setCacheRegion()`.

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
        .setEntity("blogger", blogger)
        .setMaxResults(15)
        .setCacheable(true)
        .setCacheRegion("frontpages")
        .list();
```

Si quiere forzar que el caché de peticiones actualice una de sus regiones (olvídese de cualquier resultado en caché que se encuentre allí) puede utilizar `org.hibernate.Query.setCacheMode(CacheMode.REFRESH)`. Junto con la región que ha definido para la petición dada, Hibernate forzará selectivamente los resultados en caché en esa región en particular que se va a actualizar. Esto es particularmente útil en casos donde los datos subyacentes pueden haber sido actualizados por medio de un proceso separado y esta es una alternativa más eficiente que la expulsión en masa de una región por medio de `org.hibernate.SessionFactory.evictQueries()`.

## 21.5. Comprensión del rendimiento de Colecciones

En las secciones anteriores hemos abordado las colecciones y sus aplicaciones. En esta sección exploramos algunos puntos en relación con las colecciones en tiempo de ejecución.

### 21.5.1. Taxonomía

Hibernate define tres tipos básicos de colecciones:

- colecciones de valores
- Asociaciones uno-a-muchos
- Aociaciones muchos-a-muchos

Esta clasificación distingue las varias tablas y relaciones de clave foránea pero no nos dice absolutamente todo lo que necesitamos saber sobre el modelo relacional. Para entender completamente la estructura relacional y las características de rendimiento, debemos considerar la estructura de la clave primaria que Hibernate utiliza para actualizar o borrar filas de colección. Esto sugiere la siguiente clasificación:

- colecciones indexadas
- conjuntos (sets)
- bolsas (bags)

Todas las colecciones indexadas (mapas, listas y arrays) tienen una clave principal que consiste de las columnas `<key>` e `<index>`. En este caso las actualizaciones de colecciones son extremadamente eficientes. La clave principal puede ser indexada eficientemente y una fila en particular puede ser localizada cuando Hibernate intenta actualizarla o borrarla.

Los conjuntos tienen una clave principal que consiste de `<key>` y columnas de elementos. Esto puede ser menos eficiente para algunos tipos de elementos de colección, particularmente elementos compuestos o texto largo o campos binarios ya que la base de datos puede no ser capaz de indexar una clave principal compleja eficientemente. Sin embargo, para asociaciones uno a muchos o muchos a muchos, particularmente en el caso de los identificadores sintéticos, es probable que sólo sea igual de eficiente. Si quiere que `SchemaExport` realmente cree la clave principal de un `<set>`, tiene que declarar todas las columnas como `not-null="true"`.

Los mapeos de `<idbag>` definen una clave delegada, de modo que siempre resulten eficientes de actualizar. De hecho, son el mejor caso.

Los bags son el peor caso ya que un bag permite valores de elementos duplicados y no tiene ninguna columna índice, no puede definirse ninguna clave principal. Hibernate no tiene forma de distinguir entre filas duplicadas. Hibernate resuelve este problema quitando por completo con un sólo `DELETE` y recreando la colección siempre que cambia. Esto puede ser muy ineficiente.

Para una asociación uno-a-muchos, la "clave principal" puede no ser la clave principal física de la tabla de la base de datos. Incluso en este caso, la clasificación anterior es útil todavía. Refleja cómo Hibernate "localiza" filas individuales de la colección.

## 21.5.2. Las listas, mapas, idbags y conjuntos son las colecciones más eficientes de actualizar

De la discusión anterior, debe quedar claro que las colecciones indexadas y los conjuntos permiten una operación más eficiente en términos de agregar, quitar y actualizar elementos.

Discutiblemente, hay una ventaja más de las colecciones indexadas sobre otros conjuntos para las asociaciones muchos a muchos o colecciones de valores. Debido a la estructura de un `Set`, Hibernate ni siquiera actualiza una fila con `UPDATE` cuando se "cambia" un elemento. Los cambios a un `Set` siempre funcionan por medio de `INSERT` y `DELETE` de filas individuales. Una vez más, esta consideración no se aplica a las asociaciones uno a muchos.

Después de observar que los arrays no pueden ser perezosos, podríamos concluir que las listas, mapas e idbags son los tipos más eficientes de colecciones (no inversas), con los conjuntos (sets) no muy atrás. Se espera que los sets sean el tipo más común de colección en las aplicaciones de Hibernate. Esto se debe a que la semántica de los sets es la más natural en el modelo relacional.

Sin embargo, en modelos de dominio de Hibernate bien dieñados, usualmente vemos que la mayoría de las colecciones son de hecho asociaciones uno-a-muchos con `inverse="true"`. Para estas asociaciones, la actualización es manejada por el extremo muchos-a-uno de la asociación, y las consideraciones de este tipo sobre el rendimiento de la actualización de las colecciones simplemente no se aplican.

### 21.5.3. Los Bags y las listas son las colecciones inversas más eficientes

Hay un caso en particular en el que los bags y también las listas son mucho más eficientes que los conjuntos. Para una colección con `inverse="true"`, por ejemplo, el idioma estándar de relaciones uno-a-muchos bidireccionales, podemos agregar elementos a un bag o lista sin necesidad de inicializar (recuperar) los elementos del bag. Esto se debe a que, a manera opuesta de `Collection.add()` o `Collection.addAll()` siempre deben retornar verdadero para un bag o `List` (no como un `Set`). Esto puede hacer el siguiente código común mucho más rápido:

```
Parent p = (Parent) sess.load(Parent.class, id);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);  //no need to fetch the collection!
sess.flush();
```

### 21.5.4. Borrado de un sólo tiro

Borrar los elementos de una colección uno por uno a veces puede ser extremadamente ineficiente. Hibernate sabe que no debe hacer eso, en el caso de una colección nueva-vacía (si ha llamado a `list.clear()`, por ejemplo). En este caso, Hibernate publicará un sólo `DELETE`.

Suponga que agrega un solo elemento a una colección de tamaño veinte y luego quitamos dos elementos. Hibernate publicará una declaración `INSERT` y dos declaraciones `DELETE` a menos que la colección sea un bag. Esto ciertamente es deseable.

Sin embargo, supónga que quitamos dieciocho elementos, dejando dos y luego añadimos tres elementos nuevos. Hay dos formas posibles de proceder

- borrar dieciocho filas una a una y luego insertar tres filas
- quitar toda la colección en un sólo `DELETE` de SQL e insertar todos los cinco elementos actuales uno por uno

Hibernate no sabe que la segunda opción es probablemente la más rápida. Probablemente no sería deseable que Hibernate fuese tan intuitivo ya que tal comportamiento podría confundir a disparadores de la base de datos, etc.

Afortunadamente, puede forzar este comportamiento (por ejemplo, la segunda estrategia) en cualquier momento descartando (por ejemplo, desreferenciando) la colección original y retornando una colección nuevamente instanciada con todos los elementos actuales.

El borrado-de-un-sólo-tiro no se aplica a las colecciones mapeadas `inverse="true"`.

## 21.6. Control del rendimiento

La optimización no es de mucho uso sin el monitoreo y el acceso a números de rendimiento. Hibernate brinda un rango completo de números sobre sus operaciones internas. Las estadísticas en Hibernate están disponibles por `SessionFactory`.

### 21.6.1. Control de una SessionFactory

Puede acceder a las métricas de `SessionFactory` de dos formas. Su primera opción es llamar a `sessionFactory.getStatistics()` y leer o mostrar por pantalla la `Statistics` por sí mismo.

Hibernate también puede utilizar JMX para publicar las métricas si habilita el MBean `StatisticsService`. Puede habilitar un sólo MBean para todas sus `SessionFactory` o una por fábrica. Véa el siguiente código para ver ejemplos de configuración minimalistas:

```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the server
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
server.registerMBean(stats, on); // Register the MBean on the server
```

Puede activar y desactivar el monitoreo de una `SessionFactory`

- en tiempo de configuración, establezca `hibernate.generate_statistics` como `false`
- en tiempo de ejecución: `sf.getStatistics().setStatisticsEnabled(true)` o `hibernateStatsBean.setStatisticsEnabled(true)`

Las estadísticas pueden ser reajustadas programáticamente utilizando el método `clear()`. Puede enviarse un resumen a un registro (a nivel de información) utilizando el método `logSummary()`.

## 21.6.2. Métricas

Hibernate proporciona un número de métricas, desde información muy básica hasta la más especializada sólamente relevante en ciertos escenarios. Todos los contadores disponibles se describen en la API de la interfaz `Statistics`, en tres categorías:

- Métricas relacionadas al uso general de `Session` usage, tales como número de sesiones abiertas, conexiones JDBC recuperadas, etc,
- Métricas relacionadas con las entidades, colecciones, consultas y cachés como un todo (también conocidas como métricas globales).
- Métricas detalladas relacionadas con una entidad, colección, consulta o región de caché en particular.

Por ejemplo, puede comprobar el acceso, pérdida y radio de colecciones de entidades y consultas en el caché, y el tiempo promedio que necesita una consulta. Tenga en cuenta que el número de milisegundos está sujeto a una aproximación en Java. Hibernate está vinculado a la precisión de la MVJ, en algunas plataformas esto podría tener incluso una exactitud de 10 segundos.

Se usan getters simples para acceder a la métrica global (por ejemplo, no vinculadas en particular a una entidad, colección, región de caché, etc). Puede acceder a las métricas de una entidad, colección, región de caché en particular a través de su nombre y a través de su representación HQL o SQL para las consultas. Por favor refiérase al Javadoc de la API de `Statistics`, `EntityStatistics`, `CollectionStatistics`, `SecondLevelCacheStatistics`, y `QueryStatistics` para obtener más información. El siguiente código es un ejemplo sencillo:

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount  = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
  queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
  stats.getEntityStatistics( Cat.class.getName() );
long changes =
        entityStats.getInsertCount()
        + entityStats.getUpdateCount()
        + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times"  );
```

Para trabajar sobre todas las entidades, colecciones, consultas y regiones de cachés, recuperando la lista de nombres de entidades, colecciones, consultas y regiones de cachés con los siguientes métodos: `getQueries()`, `getEntityNames()`, `getCollectionRoleNames()` y `getSecondLevelCacheRegionNames()`.

# Manual del conjunto de herramientas

La ingeniería compatible con Hibernate es posible utilizando un conjunto de plugins de Eclipse, herramientas de la línea de comandos así como tareas Ant.

Las *herramientas de Hibernate* actualmente incluyen plugins la IDE de Eclipse así como tareas Ant para la ingeniería inversa de bases de datos existentes:

- *Editor de Mapeo:* Un editor de archivos de mapeo XML que soporta autocompleción y resaltado de sintáxis. También soporta la autocompleción semántica de nombres de clases y nombres de campos/propiedades, haciéndolo mucho más versátil que un editor normal de XML.
- *Consola:* La consola es una nueva vista en Eclipse. Además de la vista de árbol de sus configuraciones de la consola, también tiene una vista interactiva de sus clases persistentes y sus relaciones. La consola le permite ejecutar consultas HQL en su base de datos y navegar el resultado directamente en Eclipse.
- *Asistentes de desarrollo:* Se proporcionan muchos asistentes junto con las herramientas Eclipse de Hibernate. Puede utilizar un asistente para generar rápidamente los archivos de configuración de Hibernate (cfg.xml), o incluso puede realizar una ingeniería inversa completa de un esquema de la base de datos existente en los archivos de código fuente de POJO y los archivos de mapeo de Hibernate. El asistente de ingeniería inversa soporta plantillas personalizables.
-

Por favor refiérase al paquete de documentación de las *Herramientas de Hibernate* para obtener más información.

Sin embargo, el paquete principal de Hibernate viene con una herramienta integrada: *SchemaExport* también conocida como `hbm2ddl`. Incluso se puede utilizar "dentro" de Hibernate.

## 22.1. Generación automática de esquemas

Una de las funciones de Hibernate puede generar DDL desde sus archivos de mapeo. El esquema generado incluye restricciones de integridad referencial, claves principales y foráneas, para las tablas de entidades y colecciones. También se creen tablas y secuencias para los generadores de identificadores mapeados.

*Tiene* que especificar un `Dialecto` SQL por medio de la propiedad `hibernate.dialect` al usar esta herramienta, ya que el DDL es altamente específico de acuerdo con el vendedor.

Primero, debe personazar sus archivos de mapeo para mejorar el esquema generado. La siguiente sección aborda la personalización del esquema.

### 22.1.1. Personalización del esquema

Muchos elementos de mapeo de Hibernate definen atributos opcionales denominados `length`, `precision` y `scale`. Con estos atributos puede establecer el tamaño, la precisión y la escala de una columna.

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

Algunas etiquetas también aceptan un atributo `not-null` para generar una restricción `NOT NULL` en columnas de tablas y un atributo `unique` para generar restricciones `UNIQUE` en columnas de tablas.

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique="true"/>
```

Se puede usar un atributo `unique-key` para agrupar columnas en una restricción de clave única. Actualmente, el valor especificado del atributo `unique-key`*no* se utiliza para nombrar la restricción en el DDL generado. Sólamente se utiliza para agrupar las columnas en el archivo de mapeo.

```
<many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>
<property name="employeeId" unique-key="OrgEmployee"/>
```

Un atributo `index` especifica el nombre de un índice que se creará utilizando la columa o las columnas mapeadas. Se pueden ser agrupar múltiples columnas bajo el mismo índice, simplemente especificando el mismo nombre de índice.

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

Un atributo `foreign-key` se puede utilizar para sobreescribir el nombre de cualquier restricción de clave foránea generada.

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

Muchos elementos de mapeo también aceptan un elemento `<column>` hijo. Esto es particularmente útil para mapear tipos de multi-columna:

```
<property name="name" type="my.customtypes.Name"/>
    <column name="last" not-null="true" index="bar_idx" length="30"/>
    <column name="first" not-null="true" index="bar_idx" length="20"/>
    <column name="initial"/>
</property
>
```

El atributo `default` le permite especificar un valor por defecto para una columna. Usted le debe asignar el mismo valor a la propiedad mapeada antes de guardar una nueva instancia de la clase mapeada.

```
<property name="credits" type="integer" insert="false">
    <column name="credits" default="10"/>
</property
>
```

```
<version name="version" type="integer" insert="false">
    <column name="version" default="0"/>
</property
>
```

El atributo `sql-type` permite al usuario sobrescribir el mapeo por defecto de tipo Hibernate a tipo de datos SQL.

```
<property name="balance" type="float">
    <column name="balance" sql-type="decimal(13,3)"/>
</property
>
```

El atributo `check` le permite especificar una comprobación de restricción.

```
<property name="foo" type="integer">
    <column name="foo" check="foo
> 10"/>
</property
>
```

```
<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
```

```
</class
>
```

La siguiente tabla resume estos atributos opcionales.

### Tabla 22.1. Resumen

| Atributo | Valores | Interpretación |
|---|---|---|
| `length` | número | longitud de columna/precisión decimal |
| `precision` | número | precisión decimal de columna |
| `scale` | número | escala decimal de columna |
| `not-null` | `true\|false` | especifica que la columna debe ser sin nulos |
| `unique` | `true\|false` | especifica que la columna debe tener una restricción de unicidad |
| `index` | `index_name` | especifica el nombre de un índice (multicolumna) |
| `unique-key` | `unique_key_name` | especifica el nombre de una restricción de unicidad multicolumna |
| `foreign-key` | `foreign_key_name` | especifica el nombre de la restricción de clave foránea generada por una asociación, para un elemento de mapeo `<one-to-one>`, `<many-to-one>`, `<key>`, o `<many-to-many>`. Observe que `SchemaExport` no considerará los lados `inverse="true"`. |
| `sql-type` | `SQL column type` | sobrescribe el tipo de columna por defecto (sólamente el atributo del elemento `<column>`) |
| `default` | expresión SQL | especifica un valor predeterminado para la columna |
| `check` | expresión SQL | crea una restricción de comprobación SQL en columna o tabla |

El elemento `<comment>` le permite especificar un comentario para el esquema generado.

```
<class name="Customer" table="CurCust">
    <comment
>Current customers only</comment>
    ...
</class
>
```

```
<property name="balance">
    <column name="bal">
        <comment
>Balance in USD</comment>
    </column>
```

```
</property
>
```

Esto da como resultado una declaración `comment on table` o `comment on column` en el DDL generado, donde se encuentre soportado.

## 22.1.2. Ejecución de la herramienta

La herramienta `SchemaExport` escribe un script DDL a la salida estándar y/o ejecuta las declaraciones DDL.

La siguiente tabla presenta las opciones de la línea de comandos de `SchemaExport`

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaExport` *options mapping_files*

### Tabla 22.2. Opciones de Línea de Comandos de `SchemaExport`

| Opción | Descripción |
|---|---|
| `--quiet` | no envie el script a la salida estándar |
| `--drop` | sólamente desechar las tablas |
| `--create` | sólamente crear las tablas |
| `--text` | no exportar a la base de datos |
| `--output=my_schema.ddl` | enviar la salida del script ddl a un archivo |
| `--naming=eg.MyNamingStrategy` | seleccione un `NamingStrategy` |
| `--config=hibernate.cfg.xml` | lee la configuración de Hibernate de un archivo XML |
| `--properties=hibernate.properties` | lee las propiedades de base de datos de un archivo |
| `--format` | formatea muy bien el SQL generado en el script |
| `--delimiter=;` | establece un delimitador de fin de línea para el script |

Inclusive puede incluir `SchemaExport` en su aplicación:

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

## 22.1.3. Propiedades

Las propiedades de la base de datos se pueden especificar:

- como propiedades del sistema con `-D`*<property>*
- en `hibernate.properties`
- en un archivo de propiedades nombrado con `--properties`

Las propiedades necesarias son las siguientes:

### Tabla 22.3. Propiedades de Conexión del SchemaExport

| Nombre de la Propiedad | Descripción |
|---|---|
| `hibernate.connection.driver_class` | clase del controlador jdbc |
| `hibernate.connection.url` | url de jdbc |
| `hibernate.connection.username` | usuario de la base de datos |
| `hibernate.connection.password` | contraseña del usuario |
| `hibernate.dialect` | dialecto |

## 22.1.4. Utilización de Ant

Puede llamar a `SchemaExport` desde su script de construcción de Ant:

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target
>
```

## 22.1.5. Actualizaciones incrementales de esquema

La herramienta `SchemaUpdate` actualizará un esquema existente con cambios "incrementales". El `SchemaUpdate` depende de la API de metadatos de JDBC, de modo que no funcionará con todos los controladores JDBC.

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaUpdate` *options mapping_files*

### Tabla 22.4. Opciones de Línea de Comandos de `SchemaUpdate`

| Opción | Descripción |
|---|---|
| `--quiet` | no envie el script a la salida estándar |

| Opción | Descripción |
|---|---|
| `--text` | no exporte el script a la base de datos |
| `--naming=eg.MyNamingStrategy` | seleccione un `NamingStrategy` |
| `--properties=hibernate.properties` | lee las propiedades de base de datos de un archivo |
| `--config=hibernate.cfg.xml` | specifique un archivo `.cfg.xml` |

Puede incluir `SchemaUpdate` en su aplicación:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

### 22.1.6. Utilización de Ant para actualizaciones incrementales de esquema

Puede llamar a `SchemaUpdate` desde el script de Ant:

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target
>
```

### 22.1.7. Validación de Esquema

La herramienta `SchemaValidator` validará que el esquema de la base de datos existente "coincide" con sus documentos de mapeo. El `SchemaValidator` depene bastante de la API de metadatos JDBC así que no funcionará con todos los controladores JDBC. Esta herramienta es extremadamente útil para comprobar.

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaValidator` *options mapping_files*

La siguiente tabla presenta las opciones de la línea de comandos para `SchemaValidator`

### Tabla 22.5. Opciones de la Línea de Comandos `SchemaValidator`

| Opción | Descripción |
|---|---|
| `--naming=eg.MyNamingStrategy` | seleccione un `NamingStrategy` |
| `--properties=hibernate.properties` | lee las propiedades de base de datos de un archivo |
| `--config=hibernate.cfg.xml` | specifique un archivo `.cfg.xml` |

Puede incluir `SchemaValidator` en su aplicación:

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

## 22.1.8. Utilización de Ant para la validación de esquema

Puede llamar `SchemaValidator` desde el scrip de Ant:

```
<target name="schemavalidate">
    <taskdef name="schemavalidator"
        classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTask"
        classpathref="class.path"/>

    <schemavalidator
        properties="hibernate.properties">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemavalidator>
</target
>
```

# Additional modules

Hibernate Core also offers integration with some external modules/projects. This includes Hibernate Validator the reference implementation of Bean Validation (JSR 303) and Hibernate Search.

## 23.1. Bean Validation

Bean Validation standardizes how to define and declare domain model level constraints. You can, for example, express that a property should never be null, that the account balance should be strictly positive, etc. These domain model constraints are declared in the bean itself by annotating its properties. Bean Validation can then read them and check for constraint violations. The validation mechanism can be executed in different layers in your application without having to duplicate any of these rules (presentation layer, data access layer). Following the DRY principle, Bean Validation and its reference implementation Hibernate Validator has been designed for that purpose.

The integration between Hibernate and Bean Validation works at two levels. First, it is able to check in-memory instances of a class for constraint violations. Second, it can apply the constraints to the Hibernate metamodel and incorporate them into the generated database schema.

Each constraint annotation is associated to a validator implementation responsible for checking the constraint on the entity instance. A validator can also (optionally) apply the constraint to the Hibernate metamodel, allowing Hibernate to generate DDL that expresses the constraint. With the appropriate event listener, you can execute the checking operation on inserts, updates and deletes done by Hibernate.

When checking instances at runtime, Hibernate Validator returns information about constraint violations in a set of `ConstraintViolation`s. Among other information, the `ConstraintViolation` contains an error description message that can embed the parameter values bundle with the annotation (eg. size limit), and message strings that may be externalized to a `ResourceBundle`.

### 23.1.1. Adding Bean Validation

To enable Hibernate's Bean Validation integration, simply add a Bean Validation provider (preferably Hibernate Validation 4) on your classpath.

### 23.1.2. Configuration

By default, no configuration is necessary.

The `Default` group is validated on entity insert and update and the database model is updated accordingly based on the `Default` group as well.

You can customize the Bean Validation integration by setting the validation mode. Use the `javax.persistence.validation.mode` property and set it up for example in your `persistence.xml` file or your `hibernate.cfg.xml` file. Several options are possible:

- `auto` (default): enable integration between Bean Validation and Hibernate (callback and ddl generation) only if Bean Validation is present in the classpath.
- `none`: disable all integration between Bean Validation and Hibernate
- `callback`: only validate entities when they are either inserted, updated or deleted. An exception is raised if no Bean Validation provider is present in the classpath.
- `ddl`: only apply constraints to the database schema when generated by Hibernate. An exception is raised if no Bean Validation provider is present in the classpath. This value is not defined by the Java Persistence spec and is specific to Hibernate.

### Nota

You can use both `callback` and `ddl` together by setting the property to `callback, dll`

```
<persistence ...>
  <persistence-unit ...>
    ...
    <properties>
      <property name="javax.persistence.validation.mode"
                value="callback, ddl"/>
    </properties>
  </persistence-unit>
</persistence>
```

This is equivalent to `auto` except that if no Bean Validation provider is present, an exception is raised.

If you want to validate different groups during insertion, update and deletion, use:

- `javax.persistence.validation.group.pre-persist`: groups validated when an entity is about to be persisted (default to `Default`)
- `javax.persistence.validation.group.pre-update`: groups validated when an entity is about to be updated (default to `Default`)
- `javax.persistence.validation.group.pre-remove`: groups validated when an entity is about to be deleted (default to no group)
- `org.hibernate.validator.group.ddl`: groups considered when applying constraints on the database schema (default to `Default`)

Each property accepts the fully qualified class names of the groups validated separated by a comma (,)

**Ejemplo 23.1. Using custom groups for validation**

```
<persistence ...>
  <persistence-unit ...>
    ...
    <properties>
      <property name="javax.persistence.validation.group.pre-update"
                value="javax.validation.group.Default, com.acme.group.Strict"/>
      <property name="javax.persistence.validation.group.pre-remove"
                value="com.acme.group.OnDelete"/>
      <property name="org.hibernate.validator.group.ddl"
                value="com.acme.group.DDL"/>
    </properties>
  </persistence-unit>
</persistence>
```

**Nota**

You can set these properties in `hibernate.cfg.xml`, `hibernate.properties` or programmatically.

### 23.1.3. Catching violations

If an entity is found to be invalid, the list of constraint violations is propagated by the `ConstraintViolationException` which exposes the set of `ConstraintViolations`.

This exception is wrapped in a `RollbackException` when the violation happens at commit time. Otherwise the `ConstraintViolationException` is returned (for example when calling `flush()`. Note that generally, catchable violations are validated at a higher level (for example in Seam / JSF 2 via the JSF - Bean Validation integration or in your business layer by explicitly calling Bean Validation).

An application code will rarely be looking for a `ConstraintViolationException` raised by Hibernate. This exception should be treated as fatal and the persistence context should be discarded (`EntityManager` or `Session`).

### 23.1.4. Database schema

Hibernate uses Bean Validation constraints to generate an accurate database schema:

- `@NotNull` leads to a not null column (unless it conflicts with components or table inheritance)
- `@Size.max` leads to a `varchar(max)` definition for Strings

- `@Min`, `@Max` lead to column checks (like `value <= max`)
- `@Digits` leads to the definition of precision and scale (ever wondered which is which? It's easy now with `@Digits` :) )

These constraints can be declared directly on the entity properties or indirectly by using constraint composition.

For more information check the Hibernate Validator *reference documentation* [http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html/].

## 23.2. Hibernate Search

### 23.2.1. Description

Full text search engines like Apache Lucene™ are a very powerful technology to bring free text/efficient queries to applications. If suffers several mismatches when dealing with a object domain model (keeping the index up to date, mismatch between the index structure and the domain model, querying mismatch...) Hibernate Search indexes your domain model thanks to a few annotations, takes care of the database / index synchronization and brings you back regular managed objects from free text queries. Hibernate Search is using *Apache Lucene* [http://lucene.apache.org] under the cover.

### 23.2.2. Integration with Hibernate Annotations

Hibernate Search integrates with Hibernate Core transparently provided that the Hibernate Search jar is present on the classpath. If you do not wish to automatically register Hibernate Search event listeners, you can set `hibernate.search.autoregister_listeners` to false. Such a need is very uncommon and not recommended.

Check the Hibernate Search *reference documentation* [http://docs.jboss.org/hibernate/stable/search/reference/en-US/html/] for more information.

# Ejemplo: Padre/Hijo

Una de las primeras cosas que los usuarios nuevos intentan hacer con Hibernate es modelar una relación de tipo padre / hijo. Para esto existen dos enfoques diferentes. El enfoque más conveniente, especialmente para los usuarios nuevos, es modelar tanto `Parent` como `Child` como clases de entidad con una asociación `<one-to-many>` desde `Parent` a `Child`. El enfoque opcional es declarar el `Child` como un `<composite-element>`. La semántica prederterminada de una asociación uno-a-muchos en Hibernate es mucho menos cercana a la semántica usual de una relación padre / hijo que la de un mapeo de elementos compuestos. Explicaremos cómo utilizar una *asociación uno-a-muchos bidireccional con tratamiento en cascada* para modelar una relación padre / hijo de manera eficiente y elegante.

## 24.1. Nota sobre las colecciones

Se considera que las colecciones de Hibernate son una parte lógica de la entidad que las posee y no de las entidades contenidas. Note que esta es una diferencia crucial y que esto tiene las siguientes consecuencias:

- Cuando se elimina/agrega un objeto desde/a una colección, se incrementa el número de la versión del dueño de la colección.
- Si un objeto que fue eliminado de una colección es una instancia de un tipo de valor (por ejemplo, un elemento compuesto), ese objeto cesará de ser persistente y su estado será completamente eliminado de la base de datos. Asimismo, añadir una instancia de tipo de valor a la colección causará que su estado sea persistente inmediatamente.
- Por otro lado, si se elimina una entidad de una colección (una asociación uno-a-muchos o muchos-a-muchos), no se borrará por defecto. Este comportamiento es completamente consistente; un cambio en el estado interno de otra entidad no hace desaparecer la entidad asociada. Asimismo, el agregar una entidad a una colección no causa que la entidad se vuelva persistente por defecto.

El comportamiento por defecto es que al agregar una entidad a una colección se crea un enlace entre las dos entidades. Al eliminar la entidad se eliminará el enlace. Esto es muy apropiado para todos los tipos de casos. Sin embargo, no apropiado en el caso de una relación padre / hijo. En este caso la vida del hijo se encuentra vinculada al ciclo de vida del padre.

## 24.2. Uno-a-muchos bidireccional

Supónga que empezamos con una asociación simple `<one-to-many>` desde `Parent` a `Child`.

```
<set name="children">
    <key column="parent_id"/>
```

```
    <one-to-many class="Child"/>
</set
>
```

Si ejecutásemos el siguiente código:

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate publicaría dos declaraciones SQL:

- un `INSERT` para crear el registro de `c`
- un `UPDATE` para crear el enlace desde `p` a `c`

Esto no es sólo ineficiente, sino que además viola cualquier restricción `NOT NULL` en la columna `parent_id`. Puede arreglar la violación de restricción de nulabilidad especificando `not-null="true"` en el mapeo de la colección:

```
<set name="children">
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set
>
```

Sin embargo, esta no es la solución recomendada.

El caso subyacente de este comportamiento es que el enlace (la clave foránea `parent_id`) de `p` a `c` no se considera parte del estado del objeto `Child` y por lo tanto no se crea en el `INSERT`. De modo que la solución es hacer que el enlace sea parte del mapeo del `Child`.

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

También necesita agregar la propiedad `parent` a la clase `Child`.

Ahora que la entidad `Child` está administrando el estado del enlace, le decimos a la colección que no actualice el enlace. Usamos el atributo `inverse` para hacer esto:

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
```

```
</set
>
```

El siguiente código se podría utilizar para agregar un nuevo `Child`:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

Sólo se emitiría un `INSERT` de SQL.

También podría crear un método `addChild()` de `Parent`.

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

El código para agregar un `Child` se ve así:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

## 24.3. Ciclo de vida en cascada

Puede abordar las frustraciones de la llamada explícita a `save()` utilizando cascadas.

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set
>
```

Esto simplifica el código anterior a:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
```

```
session.flush();
```

De manera similar, no necesitamos iterar los hijos al guardar o borrar un `Parent`. Lo siguiente elimina `p` y todos sus hijos de la base de datos.

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

Sin embargo, el siguiente código:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

no eliminará `c` de la base de datos. En este caso, sólo quitará el enlace a `p` y causará una violación a una restricción `NOT NULL`. Necesita borrar el hijo explícitamente llamando a `delete()` en `Child`.

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

En nuestro caso, un `Child` no puede existir realmente sin su padre. De modo que si eliminamos un `Child` de la colección, realmente queremos que sea borrado. Para esto, tenemos que utilizar `cascade="all-delete-orphan"`.

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set
>
```

Aunque el mapeo de la colección especifique `inverse="true"`, el tratamiento en cascada se procesa aún al iterar los elementos de la colección. De modo que si necesita que un objeto se guarde, borre o actualice en cascada, debe añadirlo a la colección. No es suficiente con simplemente llamar a `setParent()`.

## 24.4. Cascadas y `unsaved-value`

Suppose we loaded up a `Parent` in one `Session`, made some changes in a UI action and wanted to persist these changes in a new session by calling `update()`. The `Parent` will contain a collection of children and, since the cascading update is enabled, Hibernate needs to know which children are newly instantiated and which represent existing rows in the database. We will also assume that both `Parent` and `Child` have generated identifier properties of type `Long`. Hibernate will use the identifier and version/timestamp property value to determine which of the children are new. (See *Sección 11.7, "Detección automática de estado"*.) *In Hibernate3, it is no longer necessary to specify an `unsaved-value` explicitly.*

El siguiente código actualizará `parent` y `child` e insertará `newChild`:

```
//parent and child were both loaded in a previous session
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

Todo eso es apropiado para el caso de un identificador generado, pero ¿qué de los identificadores asignados y de los identificadores compuestos? Esto es más difícil, ya que Hibernate no puede usar la propiedad identificadora para distinguir entre un objeto recién instanciado, con un identificador asignado por el usuario y un objeto cargado en una sesión previa. En este caso, Hibernate utilizará la propiedad de versión o sello de fecha, o bien consultará realmente el caché de segundo nivel, o bien, en el peor de los casos, consultará la base de datos, para ver si la fila existe.

## 24.5. Conclusión

Las secciones que acabamos de cubrir pueden parecer un poco confusas. Sin embargo, en la práctica, todo funciona muy bien. La mayoría de las aplicaciones de Hibernate utilizan el patrón padre / hijo en muchos sitios.

Mencionamos una opción en el primer párrafo. Ninguno de los temas anteriores existe en el caso de los mapeos `<composite-element>`, los cuales tienen exactamente la semántica de una relación padre / hijo. Desafortunadamente, existen dos grandes limitaciones para las clases de elementos compuestos: los elementos compuestos no pueden poseer sus propias colecciones y no deben ser el hijo de cualquier otra entidad que no sea su padre único.

# Ejemplo: Aplicación de Weblog

## 25.1. Clases Persistentes

Las clases persistentes aquí representan un weblog, y un ítem publicado en un weblog. Van a ser modelados como una relación padre/hijo estándar, pero usaremos un bag ordenado, en lugar de un conjunto:

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
```

```
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

## 25.2. Mapeos de Hibernate

Los mapeos XML ahora deben ser bastante sencillos. Por ejemplo:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS">

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
```

```
            unique="true"/>

        <bag
            name="items"
            inverse="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping
>
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>
```

```
    </class>

</hibernate-mapping
>
```

## 25.3. Código Hibernate

La siguiente clase demuestra algunos de los tipos de cosas que podemos hacer con estas clases, utilizando Hibernate:

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.persist(blog);
```

```
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
```

```
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text)
                    throws HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public List listAllBlogNamesAndItemCounts(int max)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
```

```
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
            q.setMaxResults(max);
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.uniqueResult();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime
> :minDate"
            );
```

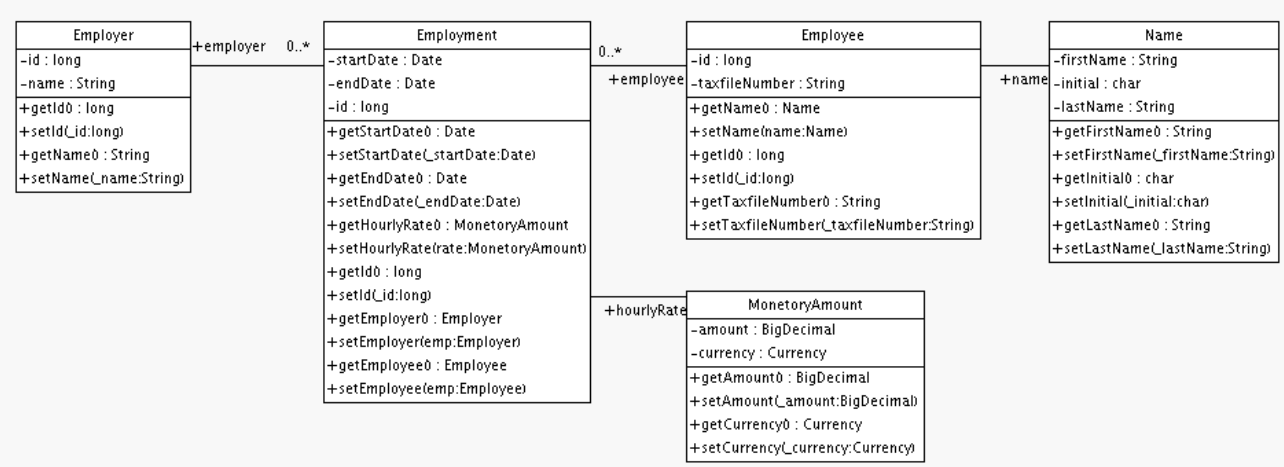
```
            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

# Ejemplo: mapeos varios

Este capítulo explora algunos de los mapeos de asociaciones más complejos.

## 26.1. Empleador/Empleado

El siguiente modelo de la relación entre `Employer` y `Employee` utiliza una clase de entidad (`Employment`) para representar la asociación. Puede hacer esto cuando podría haber más de un período de empleo para los dos mismos participantes. Se utilizan componentes para modelar los valores monetarios y los nombres de los empleados.




He aquí un posible documento de mapeo:

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence"
>employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence"
>employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetaryAmount">
            <property name="amount">
```

```
                <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
            </property>
            <property name="currency" length="12"/>
        </component>

        <many-to-one name="employer" column="employer_id" not-null="true"/>
        <many-to-one name="employee" column="employee_id" not-null="true"/>

    </class>

    <class name="Employee" table="employees">
        <id name="id">
            <generator class="sequence">
                <param name="sequence"
>employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping
>
```

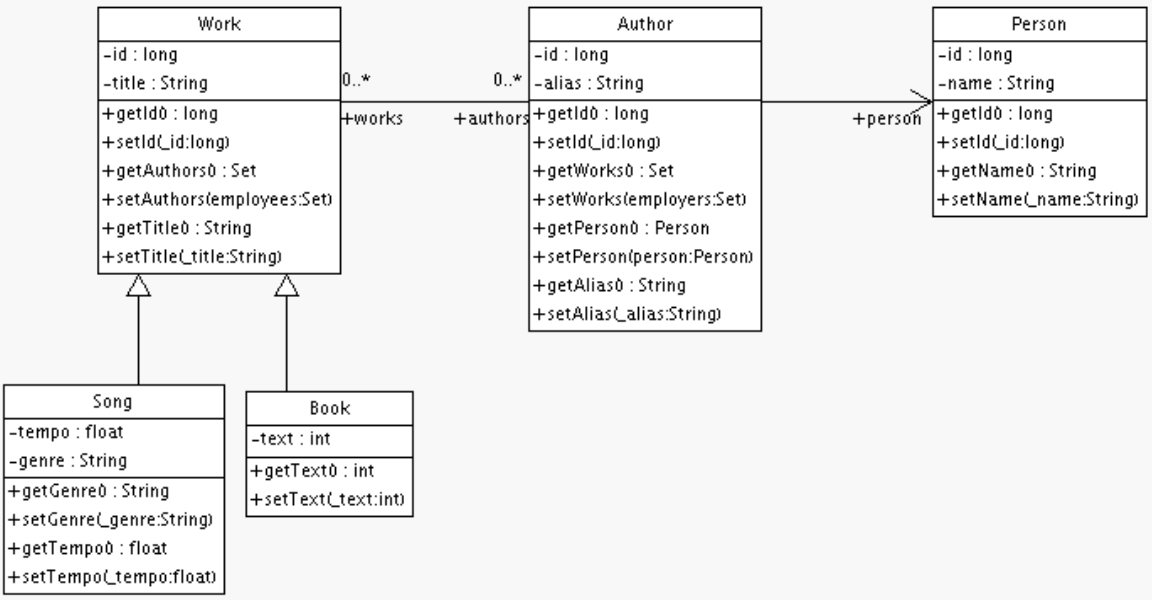
Este es el esquema de tablas generado por `SchemaExport`.

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)
```

```
alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id) references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

## 26.2. Autor/Obra

Considere el siguiente modelo de las relaciones entre `Work`, `Author` y `Person`. En el ejemplo representamos la relación entre `Work` y `Author` como una asociación muchos-a-muchos y la relación entre `Author` y `Person` como una asociación uno-a-uno. Otra posibilidad sería que `Author` extendiera `Person`.




El siguiente documento de mapeo representa estas relaciones de manera correcta:

```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work">
            <key column name="work_id"/>
            <many-to-many class="Author" column name="author_id"/>
        </set>
```

```
        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>

        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
            <property name="genre"/>
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- The Author must have the same identifier as the Person -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
        </set>

    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping
>
```

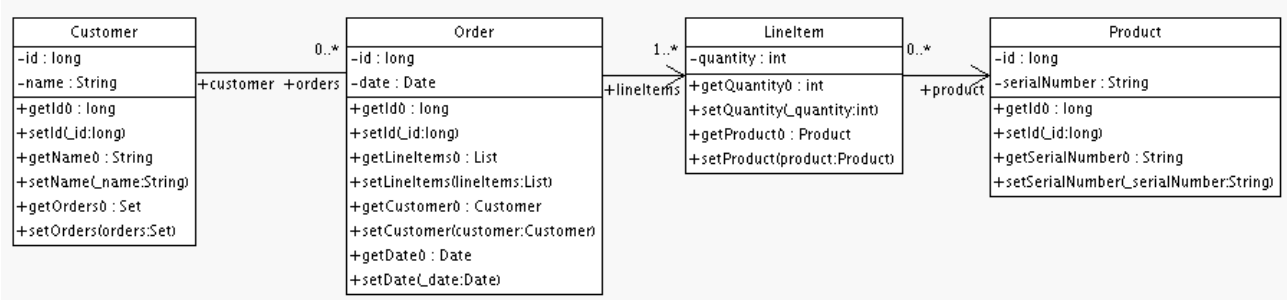
Hay cuatro tablas en este mapeo: `works`, `authors` y `persons` tienen los datos de obra, autor y persona respectivamente. `author_work` es una tabla de asociación enlazando los autores a las obras. Este es el esquema de tablas, tal como fue generado por `SchemaExport`:

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
```

```
    primary key (work_id, author_id)
)

create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references works
```

## 26.3. Cliente/Orden/Producto

En esta sección consideramos un modelo de las relaciones entre `Customer`, `Order`, `Line Item` y `Product`. Hay una asociación uno-a-muchos entre `Customer` y `Order`, pero, ¿cómo deberíamos representar `Order` / `LineItem` / `Product`? En el ejemplo, `LineItem` se mapea como una clase de asociación representando la asociación muchos-a-muchos entre `Order` y `Product`. En Hibernate, esto se llama un elemento compuesto.




El documento de mapeo se verá así:

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>
```

```
    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items">
            <key column="order_id"/>
            <list-index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping
>
```

`customers`, `orders`, `line_items` y `products` tienen los datos de cliente, orden, ítem de línea de orden y producto respectivamente. Además `line_items` también actúa como una tabla de asociación enlazando órdenes con productos.

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
```

```
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references orders
```

## 26.4. Mapeos varios de ejemplo

Estos ejemplos están disponibles en la suite de pruebas de Hibernate. Allí encontrará muchos otros mapeos de ejemplos útiles en la carpeta `test` de la distribución de Hibernate.

### 26.4.1. Asociación uno-a-uno "Tipificada"

```
<class name="Person">
    <id name="name"/>
    <one-to-one name="address"
            cascade="all">
        <formula
>name</formula>
        <formula
>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
            cascade="all">
        <formula
>name</formula>
        <formula
>'MAILING'</formula>
    </one-to-one>
</class>

<class name="Address" batch-size="2"
        check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
    <composite-id>
        <key-many-to-one name="person"
                column="personName"/>
        <key-property name="type"
                column="addressType"/>
    </composite-id>
    <property name="street" type="text"/>
    <property name="state"/>
    <property name="zip"/>
</class
>
```

### 26.4.2. Ejemplo de clave compuesta

```
<class name="Customer">
```

```
    <id name="customerId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="name" not-null="true" length="100"/>
    <property name="address" not-null="true" length="200"/>

    <list name="orders"
            inverse="true"
            cascade="save-update">
        <key column="customerId"/>
        <index column="orderNumber"/>
        <one-to-many class="Order"/>
    </list>

</class>

<class name="Order" table="CustomerOrder" lazy="true">
    <synchronize table="LineItem"/>
    <synchronize table="Product"/>

    <composite-id name="id"
            class="Order$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
    </composite-id>

    <property name="orderDate"
            type="calendar_date"
            not-null="true"/>

    <property name="total">
        <formula>
            ( select sum(li.quantity*p.price)
            from LineItem li, Product p
            where li.productId = p.productId
                and li.customerId = customerId
                and li.orderNumber = orderNumber )
        </formula>
    </property>

    <many-to-one name="customer"
            column="customerId"
            insert="false"
            update="false"
            not-null="true"/>

    <bag name="lineItems"
            fetch="join"
            inverse="true"
            cascade="save-update">
        <key>
            <column name="customerId"/>
            <column name="orderNumber"/>
        </key>
        <one-to-many class="LineItem"/>
    </bag>
```

```
</class>

<class name="LineItem">

    <composite-id name="id"
            class="LineItem$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
        <key-property name="productId" length="10"/>
    </composite-id>

    <property name="quantity"/>

    <many-to-one name="order"
            insert="false"
            update="false"
            not-null="true">
        <column name="customerId"/>
        <column name="orderNumber"/>
    </many-to-one>

    <many-to-one name="product"
            insert="false"
            update="false"
            not-null="true"
            column="productId"/>

</class>

<class name="Product">
    <synchronize table="LineItem"/>

    <id name="productId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="description"
        not-null="true"
        length="200"/>
    <property name="price" length="3"/>
    <property name="numberAvailable"/>

    <property name="numberOrdered">
        <formula>
            ( select sum(li.quantity)
            from LineItem li
            where li.productId = productId )
        </formula>
    </property>

</class
>
```

### 26.4.3. Muchos-a-muchos con atributo compartido de clave compuesta

```
<class name="User" table="`User`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <set name="groups" table="UserGroup">
        <key>
            <column name="userName"/>
            <column name="org"/>
        </key>
        <many-to-many class="Group">
            <column name="groupName"/>
            <formula
>org</formula>
        </many-to-many>
    </set>
</class>

<class name="Group" table="`Group`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <property name="description"/>
    <set name="users" table="UserGroup" inverse="true">
        <key>
            <column name="groupName"/>
            <column name="org"/>
        </key>
        <many-to-many class="User">
            <column name="userName"/>
            <formula
>org</formula>
        </many-to-many>
    </set>
</class>
```

### 26.4.4. Discriminación basada en contenido

```
<class name="Person"
    discriminator-value="P">

    <id name="id"
        column="person_id"
        unsaved-value="0">
        <generator class="native"/>
    </id>


    <discriminator
```

```
        type="character">
        <formula>
            case
                when title is not null then 'E'
                when salesperson is not null then 'C'
                else 'P'
            end
        </formula>
    </discriminator>

    <property name="name"
        not-null="true"
        length="80"/>

    <property name="sex"
        not-null="true"
        update="false"/>

    <component name="address">
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </component>

    <subclass name="Employee"
        discriminator-value="E">
            <property name="title"
                length="20"/>
            <property name="salary"/>
            <many-to-one name="manager"/>
    </subclass>

    <subclass name="Customer"
        discriminator-value="C">
            <property name="comments"/>
            <many-to-one name="salesperson"/>
    </subclass>

</class
>
```

## 26.4.5. Asociaciones sobre claves alternativas

```
<class name="Person">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="name" length="100"/>

    <one-to-one name="address"
        property-ref="person"
        cascade="all"
        fetch="join"/>
```

```
    <set name="accounts"
        inverse="true">
        <key column="userId"
            property-ref="userId"/>
        <one-to-many class="Account"/>
    </set>

    <property name="userId" length="8"/>

</class>

<class name="Address">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="address" length="300"/>
    <property name="zip" length="5"/>
    <property name="country" length="25"/>
    <many-to-one name="person" unique="true" not-null="true"/>

</class>

<class name="Account">
    <id name="accountId" length="32">
        <generator class="uuid"/>
    </id>

    <many-to-one name="user"
        column="userId"
        property-ref="userId"/>

    <property name="type" not-null="true"/>

</class
>
```

# Prácticas recomendadas

Escriba las clases detalladas y mapéelas utilizando `<component>`:

Utilice una clase `Dirección` para encapsular `calle`, `distrito`, `estado`, `código postal`. Esto promueve la reutilización de código y simplifica la refabricación.

Declare las propiedades identificadoras en clases persistentes:

Las propiedades identificadoras son opcionales en Hibernate. Existe todo tipo de razones por las que debe usarlas. Recomendamos que los identificadores sean 'sintéticos', es decir, generados sin ningún significado empresarial.

Identifique las llaves naturales:

Identifique las claves naturales de todas las entidades, y mapéelas usando `<natural-id>`. Implemente `equals()` y `hashCode()` para comparar las propiedades que componen la clave natural.

Coloque cada mapeo de clase en su propio fichero:

No use un sólo documento monolítico de mapeo. Mapee `com.eg.Foo` en el archivo `com/eg/Foo.hbm.xml`. Esto tiene sentido particularmente en un entorno de equipo.

Cargue los mapeos como recursos:

Despliegue los mapeos junto a las clases que mapean.

Considere el externalizar las cadenas de petición:

Esta es una buena práctica si sus consultas llaman a funciones SQL que no son del estándar ANSI. Externalizar las cadenas de consulta a archivos de mapeo hará la aplicación más portátil.

Use variables de vinculación.

Al igual que en JDBC, siempre remplace los valores no constantes con "?". No use la manipulación de cadenas para enlazar un valor no constante en una consulta. También considere utilizar parámetros con nombre en las consultas.

No administre sus propias conexiones JDBC:

Hibernate deja a la aplicación administrar las conexiones JDBC, pero este enfoque debe considerarse como el último recurso. Si no puede utilizar los provedores de conexión incorporados, considere proveer su propia implementación de `org.hibernate.connection.ConnectionProvider`.

Considere utilizar un tipo personalizado:

Supónga que tiene un tipo Java de una biblioteca, que necesita hacerse persistente pero que no provee los métodos de acceso necesarios para mapearlo como un componente. Debe considerar el implementar `org.hibernate.UserType`. Este enfoque libera al código de aplicación de implementar transformaciones a/desde un tipo Hibernate.

Utilice JDBC codificado a mano cuando se encuentre atascado:

En áreas de rendimiento crítico del sistema, algunos tipos de operaciones podrían beneficiarse del JDBC directo. Sin embargo, no asuma que JDBC es necesariamente más rápido. Por favor, espere hasta que *sepa* que se encuentra realmente atascado. Si necesita utilizar JDBC directo, puede abrir una `Session` de Hibernate, envuelva su operación JDBC como un objeto `org.hibernate.jdbc.Work` usando esa conexión JDBC. De esta manera puede usar aún la misma estrategia de transacción y el mismo proveedor de conexiones subyacente.

Comprenda el vaciado de `Session`:

A veces la sesión sincroniza su estado persistente con la base de datos. El rendimiento se verá afectado si este proceso ocurre con demasiada frecuencia. A veces puede minimizar el vaciado innecesario deshabilitando el vaciado automático o incluso cambiando el orden de las consultas u otras operaciones en una transacción en particular.

En una arquitectura con tres niveles considere el utilizar objetos separados:

Al usar una arquitectura de servlet/sesión, puede pasar objetos persistentes en el bean de sesión hacia y desde la capa del servlet/JSP. Use una sesión nueva para atender el servicio de cada petición. Use `Session.merge()` o `Session.saveOrUpdate()` para sincronizar los objetos con la base de datos.

En una arquitectura con dos niveles considere el utilizar contextos largos de persistencia:

Las transacciones de la base de datos tienen que ser tan cortas como sea posible para obtener una mejor escalabilidad. Sin embargo, con frecuencia es necesario implementar *transacciones de aplicación* de larga ejecución, una sola unidad de trabajo desde el punto de vista de un usuario. Una transacción de aplicación puede abarcar muchos ciclos de petición/respuesta del cliente. Es común usar objetos separados para implementar transacciones de aplicación. Una alternativa apropiada en arquitecturas de dos niveles, es mantener una sesión de un sólo contacto de persistencia abierto para todo el ciclo de vida de la transacción de aplicación. Luego simplemente desconectar de la conexión JDBC al final de cada petición y reconectar al comienzo de la petición subsecuente. Nunca comparta una sesión única a través de más de una transacción de aplicación o estará trabajando con datos desactualizados.

No trate las excepciones como recuperables:

Esto es más bien una práctica necesaria más que una práctica "recomendada". Cuando ocurra una excepción, deshaga la `Transaction` y cierre la `Session`. Si no lo hace, Hibernate no puede garantizar que el estado en memoria representa con exactitud el estado persistente. Por ejemplo, no utilice `Session.load()` para determinar si una instancia con el identificador dado existe en la base de datos; en cambio, use `Session.get()` o una consulta.

Prefiera una recuperación perezosa para las asociaciones:

No utilice con frecuencia la recuperación temprana. Use proxies y colecciones perezosas para la mayoría de asociaciones a clases que probablemente no se encuentren en el caché de segundo nivel. Para las asociaciones a clases en caché, donde hay una probabilidad de acceso a caché extremadamente alta, deshabilite explícitamente la recuperación temprana

usando `lazy="false"`. Cuando la recuperación por unión sea apropiada para un caso de uso en particular, utilice una consulta con un `left join fetch`.

Use el patrón de *sesión abierta en vista* o una *fase de ensamblado* disciplinada para evitar problemas con datos no recuperados.

Hibernate libera al desarrollador de escribir tediosos *objetos de transferencia de datos (DTO del inglés Data Transfer Objects)*. En una arquitectura tradicional de EJB, los DTOs tienen un propósito doble: primero, atacan el problema de que los beans de entidad no son serializables. Segundo, definen implícitamente una fase de ensamblado cuando se recuperan y se forman (marshalling) todos los datos a usar por la vista en los DTOs antes de devolver el control al nivel de presentación. Hibernate elimina el primer propósito. Sin embargo, aún necesita una fase de ensamblado a menos de que esté preparado para tener el contexto de persistencia (la sesión) abierto a través del proceso de entrega de la vista. Piense en sus métodos empresariales como si tuviesen un contrato estricto con el nivel de presentación sobre qué datos están disponibles en los objetos separados. Esta no es una limitación de Hibernate. Este es un requerimiento fundamental de acceso seguro a datos transaccionales.

Considere abstraer su lógica empresarial de Hibernate:

Oculte el código de acceso a datos de Hibernate detrás de una interfaz. Combine los patrones *DAO* y *sesión local de hilo*. Incluso puede hacer algunas clases persistentes por medio de JDBC escrito a mano, asociadas a Hibernate por medio de un `UserType`. Sin embargo, este consejo va para las aplicaciones "suficientemente grandes". No es apropiado para una aplicación con cinco tablas.

No utilice mapeos de asociación exóticos:

Son raros los casos de uso de asociaciones reales muchos-a-muchos. La mayor parte del tiempo necesita información adicional almacenada en una "tabla de enlace". En este caso, es mucho mejor usar dos asociaciones uno-a-muchos a una clase de enlace intermedio. De hecho, la mayoría de las asociaciones son uno-a-muchos y muchos-a-uno. Por esta razón, debe tener cuidado al utilizar cualquier otro estilo de asociación.

Prefiera las asociaciones bidireccionales:

Las asociaciones unidireccionales son más difíciles de consultar. En una aplicación grande, casi todas las asociaciones deben ser navegables en ambas direcciones en consultas.

# Consideraciones de la portabilidad de la base de datos

## 28.1. Aspectos básicos de la portabilidad

Uno de los aspectos que más vende de Hibernate (y realmente del mapeo objeto/relacional en sí) es la noción de portabilidad de la base de datos. Podría ser el caso de un administrador de sistemas migrando de una base de datos de un vendedor a otro, o podría ser un marco de trabajo o una aplicación desplegable consumiendo Hibernate para que apunte simultáneamente a múltiples productos de bases de datos. Sin importar el escenario exacto, la idea básica es que quiere que Hibernate le ayude a ejecutar frente a cualquier número de bases de datos sin cambiar el código e idealmente sin cambiar los metadatos de mapeo.

## 28.2. Dialecto

La primera línea de portabilidad para Hibernate es el dialecto, el cual es una especialización del contrato `org.hibernate.dialect.Dialect`. Un dialecto encapsula todas las diferencias en la manera en que Hibernate debe comunicarse con una base de datos en particular para lograr alguna tarea como el obtener un valor de secuencia o el estructurar una petición SELECT. Hibernate reune un gran rango de dialectos para muchas de las bases de datos más populares. Si encuentra que su base de datos en particular no se encuentra entre estos, no es demasiado dificil es escribir el propio.

## 28.3. Resolución del dialecto

Originalmente, Hibernate siempre requería que los usuarios especificaran qué dialecto utilizar. En el caso de aquellos usuarios que buscaban apuntar a múltiples bases de datos de manera simultánea con su construcción eso representaba un problema. Generalmente esto requería que los usuarios configuraran el dialecto de Hibernate o que definieran su propio método para establecer ese valor.

Empezando con la versión 3.2, Hibernate introdujo la noción de detectar automáticamente el dialecto a utilizar con base en los `java.sql.DatabaseMetaData` que se obtuvieron de una `java.sql.Connection` a esa base de datos. Esto era mucho mejor pero esta resolución estaba limitada a las bases de datos que Hibernate conoce por adelantado y de ninguna manera era configurable ni se podía sobreescribir.

Starting with version 3.3, Hibernate has a fare more powerful way to automatically determine which dialect to should be used by relying on a series of delegates which implement the `org.hibernate.dialect.resolver.DialectResolver` which defines only a single method:

```
public Dialect resolveDialect(DatabaseMetaData metaData) throws JDBCConnectionException
```

The basic contract here is that if the resolver 'understands' the given database metadata then it returns the corresponding Dialect; if not it returns null and the process continues to the next resolver. The signature also identifies `org.hibernate.exception.JDBCConnectionException` as possibly being thrown. A JDBCConnectionException here is interpreted to imply a "non transient" (aka non-recoverable) connection problem and is used to indicate an immediate stop to resolution attempts. All other exceptions result in a warning and continuing on to the next resolver.

La parte divertida de estos resolvedores es que los usuarios también pueden registrar sus propios resolvedores personalizados, los cuales se procesarán antes de los incluídos en Hibernate. Esto puede llegar a ser útil en un número de situaciones diferentes: permite una fácil integración para la auto-detección de dialectos más allá de los que se envían junto con Hibernate; le permite especificar el uso de un dialecto personalizado cuando se reconoce una base de datos en particular; etc. Para registrar uno o más resolvedores, simplemente especifiquelos (separados por comas o espacios) usando la configuración 'hibernate.dialect_resolvers' (consulte la constante `DIALECT_RESOLVERS` en `org.hibernate.cfg.Environment`).

## 28.4. Generación del identificador

When considering portability between databases, another important decision is selecting the identifier generation stratagy you want to use. Originally Hibernate provided the *native* generator for this purpose, which was intended to select between a *sequence*, *identity*, or *table* strategy depending on the capability of the underlying database. However, an insidious implication of this approach comes about when targtetting some databases which support *identity* generation and some which do not. *identity* generation relies on the SQL definition of an IDENTITY (or auto-increment) column to manage the identifier value; it is what is known as a post-insert generation strategy becauase the insert must actually happen before we can know the identifier value. Because Hibernate relies on this identifier value to uniquely reference entities within a persistence context it must then issue the insert immediately when the users requests the entitiy be associated with the session (like via save() e.g.) regardless of current transactional semantics.

**Nota**

Hibernate was changed slightly once the implication of this was better understood so that the insert is delayed in cases where that is feasible.

The underlying issue is that the actual semanctics of the application itself changes in these cases.

Starting with version 3.2.3, Hibernate comes with a set of *enhanced* [http://in.relation.to/2082.lace] identifier generators targetting portability in a much different way.

**Nota**

There are specifically 2 bundled *enhanced*generators:

- `org.hibernate.id.enhanced.SequenceStyleGenerator`
- `org.hibernate.id.enhanced.TableGenerator`

The idea behind these generators is to port the actual semantics of the identifer value generation to the different databases. For example, the `org.hibernate.id.enhanced.SequenceStyleGenerator` mimics the behavior of a sequence on databases which do not support sequences by using a table.

## 28.5. Funciones de la base de datos

**Aviso**

Esta es un área en la que Hibernate necesita mejorar. En términos de qué tan portatil puede ser, esta función que se maneja actualmente trabaja bastante bien desde HQL; sin embargo, en otros aspectos le falta mucho.

Los usuarios pueden referenciar las funciones de SQL de muchas maneras. Sin embargo, no todas las bases de datos soportan el mismo grupo de funciones. Hibernate proporciona una manera de mapear un nombre de una función *lógica* a un delegado, el cual sabe cómo entregar esa función en particular, tal vez incluso usando una llamada de función física totalmente diferente.

**Importante**

Técnicamente este registro de función se maneja por medio de la clase `org.hibernate.dialect.function.SQLFunctionRegistry`, la cual tiene el propósito de permitirle a los usuarios el proporcionar definiciones de funciones personalizadas sin tener que brindar un dialecto personalizado. Este comportamiento especifico todavía no está del todo completo.

De cierta manera está implementado para que los usuarios puedan registrar programáticamente las funciones con la `org.hibernate.cfg.Configuration` y aquellas funciones serán reconocidas por HQL.

## 28.6. Mapeos de tipo

Esta sección se completará en un futuro cercano...

# Referencias

[PoEAA] *Patrones de la arquitectura de aplicaciones empresariales*. 0-321-12742-0. por Martin Fowler. Copyright © 2003 Pearson Education, Inc.. Addison-Wesley Publishing Company.

[JPwH] *Persistencia de Java con Hibernate*. Segunda edición de Hibernate en acción. 1-932394-88-5. *http://www.manning.com/bauer2* . por Christian Bauer y Gavin King. Copyright © 2007 Manning Publications Co.. Manning Publications Co..