EBook Gratis

APRENDIZAJE hibernate

Free unaffiliated eBook created from
Stack Overflow contributors.

#hibernate

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: hibernate

It is an unofficial and free hibernate ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official hibernate.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con hibernar

Observaciones

El bean `SessionFactory` es responsable de crear, mantener, cerrar y vaciar todas las sesiones de base de datos que el `TransactionManager` le pide que cree. Es por eso que creamos automáticamente `SessionFactory` en DAO y hacemos que todas las consultas se realicen a través de él.

Una de las mayores preguntas que hacen los nuevos usuarios de Hibernate es "¿Cuándo se comprometen mis cambios?" y la respuesta tiene sentido cuando piensa cómo funciona el `TransactionManager` con `SesisonFactory` . Los cambios en la base de datos se borrarán y se confirmarán cuando salga del método de servicio anotado con `@Transactional` . La razón de esto es que se supone que una transacción representa una única 'unidad' de trabajo ininterrumpido. Si algo sale mal con la unidad, se asume que la unidad falló y todos los cambios deberían revertirse. Por lo tanto, `SessionFactory` vaciará y borrará la sesión cuando salga del método de servicio al que llamó originalmente.

Eso no quiere decir que no se vaciará ni borrará la sesión mientras se realiza la transacción. Por ejemplo, si llamo a un método de servicio para agregar una colección de 5 objetos y devolver el recuento total de objetos en la base de datos, `SessionFactory` daría cuenta de que la consulta (`SELECT COUNT(*)`) requiere un estado actualizado para ser precisa, y así se eliminaría la adición de los 5 objetos antes de ejecutar la consulta de conteo. La ejecución podría verse algo así:

Versiones

Versión	Enlace de documentación	Fecha de lanzamiento
4.2.0	http://hibernate.org/orm/documentation/4.2/	2013-03-01
4.3.0	http://hibernate.org/orm/documentation/4.3/	2013-12-01
5.0.0	http://hibernate.org/orm/documentation/5.0/	2015-09-01

Examples

Usando la configuración XML para configurar Hibernate

Creo un archivo llamado `database-servlet.xml` algún lugar de la ruta de `database-servlet.xml` .

Inicialmente, su archivo de configuración se verá así:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
    http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-
tx-3.2.xsd">

</beans>
```

Notarás que `jdbc` espacios de nombre `tx` y `jdbc` Spring. Esto se debe a que los vamos a utilizar bastante en este archivo de configuración.

Lo primero que debe hacer es habilitar la gestión de transacciones basada en anotaciones (`@Transactional`). La razón principal por la que las personas usan Hibernate en Spring es porque Spring administrará todas sus transacciones por usted. Agregue la siguiente línea a su archivo de configuración:

```
<tx:annotation-driven />
```

Necesitamos crear una fuente de datos. El origen de datos es básicamente la base de datos que Hibernate usará para conservar sus objetos. En general, un administrador de transacciones tendrá un origen de datos. Si desea que Hibernate hable con múltiples fuentes de datos, tiene múltiples administradores de transacciones.

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="" />
    <property name="url" value="" />
    <property name="username" value="" />
    <property name="password" value="" />
</bean>
```

La clase de este bean puede ser cualquier cosa que implemente `javax.sql.DataSource` para que puedas escribir el tuyo. Esta clase de ejemplo es proporcionada por Spring, pero no tiene su propio grupo de subprocesos. Una alternativa popular es el Apache Commons `org.apache.commons.dbcp.BasicDataSource` , pero hay muchos otros. Voy a explicar cada una de las propiedades a continuación:

- *driverClassName* : la ruta a su controlador JDBC. Este es un JAR **específico para la base de datos** que debería estar disponible en su classpath. Asegúrese de tener la versión más actualizada. Si está utilizando una base de datos Oracle, necesitará un OracleDriver. Si tiene una base de datos MySQL, necesitará un MySQLDriver. Vea si puede encontrar el controlador que necesita aquí, pero un google rápido le dará el controlador correcto.

- *url* : La URL de su base de datos. Normalmente esto será algo como `jdbc\:oracle\:thin\:\path\to\your\database` o `jdbc:mysql://path/to/your/database` . Si busca en Google la ubicación predeterminada de la base de datos que está utilizando, debería poder averiguar cuál debería ser. Si obtiene una `HibernateException` con el mensaje `org.hibernate.HibernateException: Connection cannot be null when 'hibernate.dialect' not`

`set` y está siguiendo esta guía, hay un 90% de probabilidad de que su URL sea incorrecta, un 5% de probabilidad que su base de datos no se haya iniciado y un 5% de probabilidad de que su nombre de usuario / contraseña sea incorrecto.

- *nombre de usuario* : el nombre de usuario que se usará cuando se autentique con la base de datos.
- *contraseña* : la contraseña que se utilizará al autenticarse con la base de datos.

Lo siguiente, es configurar el `SessionFactory` . Esto es lo que Hibernate usa para crear y administrar sus transacciones, y en realidad habla con la base de datos. Tiene bastantes opciones de configuración que trataré de explicar a continuación.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="au.com.project />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.use_sql_comments">true</prop>
            <prop key="hibernate.hbm2ddl.auto">validate</prop>
        </props>
    </property>
</bean>
```

- *dataSource:* Su grano de fuente de datos. Si cambió la ID de la fuente de datos, configúrela aquí.
- *packagesToScan* : los paquetes a escanear para encontrar sus objetos anotados JPA. Estos son los objetos que la fábrica de sesiones necesita gestionar, generalmente serán POJO y anotados con `@Entity` . Para obtener más información sobre cómo configurar relaciones de objetos en Hibernate, consulte aquí .
- *annotatedClasses* (no se muestra): también puede proporcionar una lista de clases para que Hibernate las explore si no están todas en el mismo paquete. Debes usar `packagesToScan` o `annotatedClasses` pero no ambos. La declaración se ve así:

```
<property name="annotatedClasses">
    <list>
        <value>foo.bar.package.model.Person</value>
        <value>foo.bar.package.model.Thing</value>
    </list>
</property>
```

- *hibernateProperties* : Hay una gran cantidad de estos documentados aquí con amor. Los principales que utilizará son los siguientes:
- *hibernate.hbm2ddl.auto* : Una de las preguntas más importantes de Hibernate detalla esta propiedad. Véalo para más información . Generalmente uso validar, y configuro mi base de datos usando scripts SQL (para una memoria interna), o creo la base de datos de antemano (base de datos existente).
- *hibernate.show_sql* : indicador booleano, si es verdadero, Hibernate imprimirá todo el SQL

que genere en la `stdout` . También puede configurar su registrador para que le muestre los valores que están vinculados a las consultas configurando `log4j.logger.org.hibernate.type=TRACE log4j.logger.org.hibernate.SQL=DEBUG` en su administrador de registros (yo uso log4j).

- *hibernate.format_sql* : bandera booleana, hará que Hibernate imprima bastante su SQL a la salida estándar.
- *hibernate.dialect* (No se muestra, por una buena razón): muchos de los tutoriales antiguos que hay por ahí le muestran cómo configurar el dialecto de Hibernate que usará para comunicarse con su base de datos. Hibernate **puede** detectar automáticamente qué dialecto usar según el controlador JDBC que está utilizando. Como hay alrededor de 3 dialectos de Oracle diferentes y 5 dialectos de MySQL diferentes, dejaría esta decisión en manos de Hibernate. Para obtener una lista completa de los dialectos compatibles con Hibernate, consulte aquí .

Los 2 últimos frijoles que debes declarar son:

```
<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"
    id="PersistenceExceptionTranslator" />

<bean id="transactionManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

El `PersistenceExceptionTranslator` traduce `HibernateException` o `SQLExceptions` específicas de la base de datos en excepciones Spring que pueden ser comprendidas por el contexto de la aplicación.

El bean `TransactionManager` es lo que controla las transacciones, así como los retrocesos.

Nota: debe auto cablear su bean `SessionFactory` a sus DAO.

Configuración de hibernación sin XML

Este ejemplo ha sido tomado de aquí.

```
package com.reborne.SmartHibernateConnector.utils;

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

public class LiveHibernateConnector implements IHibernateConnector {

    private String DB_DRIVER_NAME = "";
    private String DB_URL = "jdbc:h2:~/liveDB;MV_STORE=FALSE;MVCC=FALSE";
    private String DB_USERNAME = "sa";
    private String DB_PASSWORD = "";
    private String DIALECT = "org.hibernate.dialect.H2Dialect";
    private String HBM2DLL = "create";
    private String SHOW_SQL = "true";
```

```
    private static Configuration config;
    private static SessionFactory sessionFactory;
    private Session session;

    private boolean CLOSE_AFTER_TRANSACTION = false;

    public LiveHibernateConnector() {

        config = new Configuration();

        config.setProperty("hibernate.connector.driver_class",         DB_DRIVER_NAME);
        config.setProperty("hibernate.connection.url",                 DB_URL);
        config.setProperty("hibernate.connection.username",         DB_USERNAME);
        config.setProperty("hibernate.connection.password",         DB_PASSWORD);
        config.setProperty("hibernate.dialect",                     DIALECT);
        config.setProperty("hibernate.hbm2dll.auto",                 HBM2DLL);
        config.setProperty("hibernate.show_sql",                    SHOW_SQL);

        /*
         * Config connection pools
         */

        config.setProperty("connection.provider_class",
"org.hibernate.connection.C3P0ConnectionProvider");
        config.setProperty("hibernate.c3p0.min_size", "5");
        config.setProperty("hibernate.c3p0.max_size", "20");
        config.setProperty("hibernate.c3p0.timeout", "300");
        config.setProperty("hibernate.c3p0.max_statements", "50");
        config.setProperty("hibernate.c3p0.idle_test_period", "3000");


        /**
         * Resource mapping
         */

//        config.addAnnotatedClass(User.class);
//        config.addAnnotatedClass(User.class);
//        config.addAnnotatedClass(User.class);

        sessionFactory = config.buildSessionFactory();
    }


    public HibWrapper openSession() throws HibernateException {
        return new HibWrapper(getOrCreateSession(), CLOSE_AFTER_TRANSACTION);
    }


    public Session getOrCreateSession() throws HibernateException {
        if (session == null) {
            session = sessionFactory.openSession();
        }
        return session;
    }

    public void reconnect() throws HibernateException {
        this.sessionFactory = config.buildSessionFactory();
    }
```

```
}
```

Tenga en cuenta que con la última versión de Hibernate este enfoque no funciona bien (la versión 5.2 de Hibernate todavía permite esta configuración)

Ejemplo de hibernación simple usando XML

Para configurar un proyecto de hibernación simple utilizando XML para las configuraciones, necesita 3 archivos, hibernate.cfg.xml, un POJO para cada entidad y un EntityName.hbm.xml para cada entidad. Aquí hay un ejemplo de cada uno usando MySQL:

hibernate.cfg.xml

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

<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>

   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/DBSchemaName
   </property>
   <property name="hibernate.connection.username">
      testUserName
   </property>
   <property name="hibernate.connection.password">
      testPassword
   </property>

   <!-- List of XML mapping files -->
   <mapping resource="HibernatePractice/Employee.hbm.xml"/>

</session-factory>
</hibernate-configuration>
```

DBSchemaName, testUserName y testPassword serían reemplazados. Asegúrese de usar el nombre completo del recurso si está en un paquete.

Empleado.java

```
package HibernatePractice;

public class Employee {
    private int id;
    private String firstName;
    private String middleName;
    private String lastName;
```

```
    public Employee(){

    }
    public int getId(){
        return id;
    }
    public void setId(int id){
        this.id = id;
    }
    public String getFirstName(){
        return firstName;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public String getMiddleName(){
        return middleName;
    }
    public void setMiddleName(String middleName){
        this.middleName = middleName;
    }
    public String getLastName(){
        return lastName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
}
```

Employee.hbm.xml

```
<hibernate-mapping>
   <class name="HibernatePractice.Employee" table="employee">
      <meta attribute="class-description">
         This class contains employee information.
      </meta>
      <id name="id" type="int" column="empolyee_id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="middleName" column="middle_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
   </class>
</hibernate-mapping>
```

Nuevamente, si la clase está en un paquete, use el nombre completo de la clase packageName.className.

Una vez que tenga estos tres archivos, estará listo para usar la hibernación en su proyecto.

Lea Empezando con hibernar en línea: https://riptutorial.com/es/hibernate/topic/907/empezando-con-hibernar

Capítulo 2: Almacenamiento en caché

Examples

Habilitando el almacenamiento en caché de hibernación en WildFly

Para habilitar el almacenamiento en caché de segundo nivel para Hibernate en WildFly, agregue esta propiedad a su archivo `persistence.xml` :

```
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

También puede habilitar el caché de consultas con esta propiedad:

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```

WildFly no requiere que defina un proveedor de caché cuando habilite la caché de segundo nivel de Hibernate, ya que Infinispan se usa de forma predeterminada. Sin embargo, si desea utilizar un proveedor de caché alternativo, puede hacerlo con la propiedad `hibernate.cache.provider_class` .

Lea Almacenamiento en caché en línea:
https://riptutorial.com/es/hibernate/topic/3462/almacenamiento-en-cache

Capítulo 3: Asignaciones de asociación entre entidades

Examples

Asociación OneToMany

Para ilustrar la relación OneToMany necesitamos 2 Entidades, por ejemplo, País y Ciudad. Un país tiene múltiples ciudades.

En la CountryEntity beloww definimos el conjunto de ciudades para Country.

```
@Entity
@Table(name = "Country")
public class CountryEntity implements Serializable
{
   private static final long serialVersionUID = 1L;

   @Id
   @Column(name = "COUNTRY_ID", unique = true, nullable = false)
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Integer           countryId;

   @Column(name = "COUNTRY_NAME", unique = true, nullable = false, length = 100)
   private String            countryName;

   @OneToMany(mappedBy="country", fetch=FetchType.LAZY)
   private Set<CityEntity> cities = new HashSet<>();

   //Getters and Setters are not shown
}
```

Ahora la entidad de la ciudad.

```
@Entity
@Table(name = "City")
public class CityEntity implements Serializable
{
   private static final long serialVersionUID = 1L;

   @Id
   @Column(name = "CITY_ID", unique = true, nullable = false)
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Integer           cityId;

   @Column(name = "CITY_NAME", unique = false, nullable = false, length = 100)
   private String            cityName;

   @ManyToOne(optional=false, fetch=FetchType.EAGER)
   @JoinColumn(name="COUNTRY_ID", nullable=false)
   private CountryEntity country;

   //Getters and Setters are not shown
```

```
}
```

Una a muchas asociaciones usando XML

Este es un ejemplo de cómo haría una asignación de uno a varios utilizando XML. Usaremos Autor y Libro como nuestro ejemplo y asumiremos que un autor puede haber escrito muchos libros, pero cada libro solo tendrá un autor.

Clase de autor

```
public class Author {
    private int id;
    private String firstName;
    private String lastName;

    public Author(){

    }
    public int getId(){
        return id;
    }
    public void setId(int id){
        this.id = id;
    }
    public String getFirstName(){
        return firstName;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public String getLastName(){
        return lastName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
}
```

Clase de libro

```
public class Book {
    private int id;
    private String isbn;
    private String title;
    private Author author;
    private String publisher;

    public Book() {
        super();
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getIsbn() {
```

```
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public Author getAuthor() {
        return author;
    }
    public void setAuthor(Author author) {
        this.author = author;
    }
    public String getPublisher() {
        return publisher;
    }
    public void setPublisher(String publisher) {
        this.publisher = publisher;
    }
}
```

Author.hbm.xml:

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

<hibernate-mapping>
   <class name="Author" table="author">
      <meta attribute="class-description">
         This class contains the author's information.
      </meta>
      <id name="id" type="int" column="author_id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
   </class>
</hibernate-mapping>
```

Libro.hbm.xml:

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

<hibernate-mapping>
    <class name="Book" table="book_title">
      <meta attribute="class-description">
         This class contains the book information.
      </meta>
        <id name="id" type="int" column="book_id">
            <generator class="native"/>
```

```
        </id>
        <property name="isbn" column="isbn" type="string"/>
        <property name="title" column="title" type="string"/>
         <many-to-one name="author" class="Author" cascade="all">
             <column name="author"></column>
         </many-to-one>
        <property name="publisher" column="publisher" type="string"/>
    </class>
</hibernate-mapping>
```

Lo que hace que la conexión sea única es que la clase Book contiene un Autor y el xml tiene la etiqueta <many-to-one>. El atributo en cascada le permite establecer cómo se guardará / actualizará la entidad secundaria.

Lea Asignaciones de asociación entre entidades en línea:
https://riptutorial.com/es/hibernate/topic/6165/asignaciones-de-asociacion-entre-entidades

Capítulo 4: Asociaciones de mapeo

Examples

Mapeo de Hibernación Uno a Uno

Cada país tiene un capital. Cada capital tiene un país.

País.java

```
package com.entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "countries")
public class Country {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

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

    @Column(name = "national_language")
    private String nationalLanguage;

    @OneToOne(mappedBy = "country")
    private Capital capital;

    //Constructor

    //getters and setters

   }
```

Capital.java

```
package com.entity;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
```

```
@Table(name = "capitals")
public class Capital {

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

    private String name;

    private long population;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "country_id")
    private Country country;

    //Constructor

    //getters and setters

}
```

HibernateDemo.java

```
package com.entity;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateDemo {

public static void main(String ar[]) {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    Country india = new Country();
    Capital delhi = new Capital();
    delhi.setName("Delhi");
    delhi.setPopulation(357828394);
    india.setName("India");
    india.setNationalLanguage("Hindi");
    delhi.setCountry(india);
    session.save(delhi);
    session.close();
  }

}
```

Lea Asociaciones de mapeo en línea: https://riptutorial.com/es/hibernate/topic/6478/asociaciones-de-mapeo

Capítulo 5: Consultas SQL nativas

Examples

Consulta simple

Suponiendo que tiene un controlador en el objeto `Session` hibernación, en este caso llamada `session` :

```
List<Object[]> result = session.createNativeQuery("SELECT * FROM some_table").list();
for (Object[] row : result) {
    for (Object col : row) {
        System.out.print(col);
    }
}
```

Esto recuperará todas las filas en `some_table` y las colocará en la variable de `result` e imprimirá cada valor.

Ejemplo para obtener un resultado único.

```
Object pollAnswered = getCurrentSession().createSQLQuery(
        "select * from TJ_ANSWERED_ASW where pol_id = "+pollId+" and prf_log =
'"+logid+"'").uniqueResult();
```

con esta consulta, obtiene un resultado único cuando sabe que el resultado de la consulta siempre será único.

Y si la consulta devuelve más de un valor, obtendrá una excepción

    org.hibernate.NonUniqueResultException

También puedes ver los detalles en este enlace aquí con más descripción.

Entonces, asegúrese de saber que la consulta devolverá un resultado único

Lea Consultas SQL nativas en línea: https://riptutorial.com/es/hibernate/topic/6978/consultas-sql-nativas

Capítulo 6: Criterios y Proyecciones

Examples

Lista usando restricciones

Suponiendo que tenemos una tabla TravelReview con nombres de ciudades como columna "título"

```
 Criteria criteria =
    session.createCriteria(TravelReview.class);
  List review =
    criteria.add(Restrictions.eq("title", "Mumbai")).list();
  System.out.println("Using equals: " + review);
```

Podemos agregar restricciones a los criterios encadenándolos de la siguiente manera:

```
List reviews = session.createCriteria(TravelReview.class)
   .add(Restrictions.eq("author", "John Jones"))
   .add(Restrictions.between("date",fromDate,toDate))
   .add(Restrictions.ne("title","New York")).list();
```

Utilizando proyecciones

Si deseamos recuperar solo unas pocas columnas, podemos usar la clase Proyecciones para hacerlo. Por ejemplo, el siguiente código recupera la columna de título

```
 // Selecting all title columns
  List review = session.createCriteria(TravelReview.class)
        .setProjection(Projections.property("title"))
    .list();
  // Getting row count
  review = session.createCriteria(TravelReview.class)
    .setProjection(Projections.rowCount())
    .list();
  // Fetching number of titles
  review = session.createCriteria(TravelReview.class)
    .setProjection(Projections.count("title"))
    .list();
```

Utilizar filtros

`@Filter` se usa como un campamento `WHERE` , aquí algunos ejemplos

Entidad estudiantil

```
@Entity
@Table(name = "Student")
public class Student
```

```
{
    /*...*/

    @OneToMany
    @Filter(name = "active", condition = "EXISTS(SELECT * FROM Study s WHERE state = true and
s.id = study_id)")
    Set<StudentStudy> studies;

    /* getters and setters methods */
}
```

Entidad de estudio

```
@Entity
@Table(name = "Study")
@FilterDef(name = "active")
@Filter(name = "active", condition="state = true")
public class Study
{
    /*...*/

    @OneToMany
    Set<StudentStudy> students;

    @Field
    boolean state;

    /* getters and setters methods */
}
```

Estudiante entidad de estudio

```
@Entity
@Table(name = "StudentStudy")
@Filter(name = "active", condition = "EXISTS(SELECT * FROM Study s WHERE state = true and s.id
= study_id)")
public class StudentStudy
{
    /*...*/

    @ManytoOne
    Student student;

    @ManytoOne
    Study study;

    /* getters and setters methods */
}
```

De esta manera, cada vez que se activa el filtro "activo",

-Todas las consultas que hacemos sobre la entidad estudiantil devolverán **TODOS los** estudiantes **SOLAMENTE** con su `state = true` estudios `state = true`

-Cada consulta que hagamos en la entidad de Estudio devolverá **TODOS** `state = true` estudios de `state = true`

-Todas las consultas que hacemos en la entidad StudentStudy devolverán **SOLAMENTE** las que tengan una relación de estudio `state = true`

Tenga en cuenta que study_id es el nombre del campo en la tabla de StudentStudy de sql

Lea Criterios y Proyecciones en línea: https://riptutorial.com/es/hibernate/topic/3939/criterios-y-proyecciones

Capítulo 7: Estrategia de nomenclatura personalizada

Examples

Creación y uso de una ImplicitNamingStrategy personalizada

La creación de una `ImplicitNamingStrategy` personalizada le permite modificar la forma en que Hibernate asignará nombres a los atributos de `Entity` no explícitamente nombrados, incluidas las claves externas, las claves únicas, las columnas identificadoras, las columnas básicas y mucho más.

Por ejemplo, de forma predeterminada, Hibernate generará claves foráneas que están en hash y son similares a:

```
FKe6hidh4u0qh8y1ijy59s2ee6m
```

Si bien esto no suele ser un problema, es posible que desee que el nombre sea más descriptivo, como:

```
FK_asset_tenant
```

Esto se puede hacer fácilmente con un `ImplicitNamingStrategy` personalizado.

Este ejemplo amplía `ImplicitNamingStrategyJpaCompliantImpl` , sin embargo, puede elegir implementar `ImplicitNamingStrategy` si lo desea.

```
import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.ImplicitForeignKeyNameSource;
import org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl;

public class CustomNamingStrategy extends ImplicitNamingStrategyJpaCompliantImpl {

    @Override
    public Identifier determineForeignKeyName(ImplicitForeignKeyNameSource source) {
        return toIdentifier("FK_" + source.getTableName().getCanonicalName() + "_" +
source.getReferencedTableName().getCanonicalName(), source.getBuildingContext());
    }

}
```

Para indicar a Hibernate qué `ImplicitNamingStrategy` debe usar, defina la propiedad `hibernate.implicit_naming_strategy` en su archivo `persistence.xml` o `hibernate.cfg.xml` como se muestra a continuación:

```
<property name="hibernate.implicit_naming_strategy"
                  value="com.example.foo.bar.CustomNamingStrategy"/>
```

O puede especificar la propiedad en el archivo `hibernate.properties` la siguiente manera:

```
hibernate.implicit_naming_strategy=com.example.foo.bar.CustomNamingStrategy
```

En este ejemplo, todas las Claves foráneas que no tienen un `name` definido explícitamente ahora obtendrán su nombre de `CustomNamingStrategy` .

Estrategia de nomenclatura física personalizada

Al asignar nuestras entidades a nombres de tablas de bases de datos, nos basamos en una anotación de `@Table` . Pero si tenemos una convención de nomenclatura para los nombres de nuestras tablas de la base de datos, podemos implementar una estrategia de denominación física personalizada para indicar a Hibernate que calcule los nombres de las tablas basándose en los nombres de las entidades, sin indicar explícitamente esos nombres con la anotación `@Table` . Lo mismo ocurre con la asignación de atributos y columnas.

Por ejemplo, el nombre de nuestra entidad es:

```
ApplicationEventLog
```

Y nuestro nombre de mesa es:

```
application_event_log
```

Nuestra estrategia de nomenclatura física necesita convertir los nombres de entidades que son casos de camellos a los nombres de nuestras tablas db que son casos de serpientes. Podemos lograr esto extendiendo `PhysicalNamingStrategyStandardImpl` de hibernate:

```
import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl;
import org.hibernate.engine.jdbc.env.spi.JdbcEnvironment;

public class PhysicalNamingStrategyImpl extends PhysicalNamingStrategyStandardImpl {

    private static final long serialVersionUID = 1L;
    public static final PhysicalNamingStrategyImpl INSTANCE = new
PhysicalNamingStrategyImpl();

    @Override
    public Identifier toPhysicalTableName(Identifier name, JdbcEnvironment context) {
        return new Identifier(addUnderscores(name.getText()), name.isQuoted());
    }

    @Override
    public Identifier toPhysicalColumnName(Identifier name, JdbcEnvironment context) {
        return new Identifier(addUnderscores(name.getText()), name.isQuoted());
    }

    protected static String addUnderscores(String name) {
        final StringBuilder buf = new StringBuilder(name);
        for (int i = 1; i < buf.length() - 1; i++) {
            if (Character.isLowerCase(buf.charAt(i - 1)) &&
```

```
                    Character.isUpperCase(buf.charAt(i)) &&
                    Character.isLowerCase(buf.charAt(i + 1))) {
                buf.insert(i++, '_');
            }
        }
        return buf.toString().toLowerCase(Locale.ROOT);
    }
}
```

Estamos anulando el comportamiento predeterminado de los métodos `toPhysicalTableName` y `toPhysicalColumnName` para aplicar nuestra convención de nomenclatura de db.

Para utilizar nuestra implementación personalizada, necesitamos definir la propiedad `hibernate.physical_naming_strategy` y darle el nombre de nuestra clase `PhysicalNamingStrategyImpl` .

```
hibernate.physical_naming_strategy=com.example.foo.bar.PhysicalNamingStrategyImpl
```

De esta manera podemos aliviar nuestro código de las anotaciones `@Table` y `@Column` , por lo que nuestra clase de entidad:

```
@Entity
public class ApplicationEventLog {
    private Date startTimestamp;
    private String logUser;
    private Integer eventSuccess;

    @Column(name="finish_dtl")
    private String finishDetails;
}
```

será correctamente asignado a la tabla db:

```
CREATE TABLE application_event_log (
  ...
  start_timestamp timestamp,
  log_user varchar(255),
  event_success int(11),
  finish_dtl varchar(2000),
  ...
)
```

Como se vio en el ejemplo anterior, aún podemos indicar explícitamente el nombre del objeto db si no lo está, por alguna razón, de acuerdo con nuestra convención general de nomenclatura: `@Column(name="finish_dtl")`

Lea Estrategia de nomenclatura personalizada en línea:
https://riptutorial.com/es/hibernate/topic/3051/estrategia-de-nomenclatura-personalizada

Capítulo 8: Habilitar / deshabilitar el registro de SQL

Observaciones

El registro de estas consultas es **lento** , incluso más lento de lo que suele ser Hibernate. También utiliza una gran cantidad de espacio de registro. No utilice el inicio de sesión en situaciones en las que se requiera rendimiento. Use esto solo cuando pruebe las consultas que Hibernate realmente genera.

Examples

Usando un archivo de configuración de registro

En el archivo de configuración de registro de su elección, establezca el registro de los siguientes paquetes en los niveles mostrados:

```
# log the sql statement
org.hibernate.SQL=DEBUG
# log the parameters
org.hibernate.type=TRACE
```

Probablemente habrá algunos prefijos específicos del registrador que se requieren.

Log4j config:

```
log4j.logger.org.hibernate.SQL=DEBUG
log4j.logger.org.hibernate.type=TRACE
```

`application.properties` inicio de primavera.propiedades:

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type=TRACE
```

Logback `logback.xml` :

```
<logger name="org.hibernate.SQL" level="DEBUG"/>
<logger name="org.hibernate.type" level="TRACE"/>
```

Usando las propiedades de Hibernate

Esto le mostrará el SQL generado, pero no le mostrará los valores contenidos en las consultas.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
```

```
    <property name="hibernateProperties">
        <props>
            <!-- show the sql without the parameters -->
            <prop key="hibernate.show_sql">true</prop>
            <!-- format the sql nice -->
            <prop key="hibernate.format_sql">true</prop>
            <!-- show the hql as comment -->
            <prop key="use_sql_comments">true</prop>
        </props>
    </property>
</bean>
```

Habilitar / deshabilitar el registro de SQL en la depuración

Algunas aplicaciones que utilizan Hibernate generan una gran cantidad de SQL cuando se inicia la aplicación. A veces es mejor habilitar / deshabilitar el registro de SQL en puntos específicos al depurar.

Para habilitarlo, simplemente ejecute este código en su IDE cuando esté depurando la aplicación:

```
org.apache.log4j.Logger.getLogger("org.hibernate.SQL")
    .setLevel(org.apache.log4j.Level.DEBUG)
```

Deshabilitar:

```
org.apache.log4j.Logger.getLogger("org.hibernate.SQL")
    .setLevel(org.apache.log4j.Level.OFF)
```

Lea Habilitar / deshabilitar el registro de SQL en línea:
https://riptutorial.com/es/hibernate/topic/3548/habilitar---deshabilitar-el-registro-de-sql

Capítulo 9: Hibernate y JPA

Examples

Relación entre Hibernate y JPA

Hibernate es una implementación del estándar JPA . Como tal, todo lo dicho también es cierto para Hibernate.

Hibernate tiene algunas extensiones para JPA. Además, la forma de configurar un proveedor de JPA es específica del proveedor. Esta sección de documentación solo debe contener lo que es específico de Hibernate.

Lea Hibernate y JPA en línea: https://riptutorial.com/es/hibernate/topic/6313/hibernate-y-jpa

Capítulo 10: HQL

Introducción

HQL es Hibernate Query Language, basado en SQL y tras bambalinas se cambia a SQL pero la sintaxis es diferente. Utiliza nombres de entidades / clases, no nombres de tablas y nombres de campos, ni nombres de columnas. También permite muchas taquigrafías.

Observaciones

Lo principal a recordar cuando se usa hql es usar el nombre de la clase y los nombres de campo en lugar de los nombres de tabla y columna a los que estamos acostumbrados en SQL.

Examples

Seleccionando una mesa entera

```
hql = "From EntityName";
```

Seleccionar columnas específicas

```
hql = "Select id, name From Employee";
```

Incluir una cláusula Where

```
hql = "From Employee where id = 22";
```

Unirse

```
hql = "From Author a, Book b Where a.id = book.author";
```

Lea HQL en línea: https://riptutorial.com/es/hibernate/topic/9388/hql

Capítulo 11: La optimización del rendimiento

Examples

No utilice el tipo de búsqueda EAGER

Hibernate puede usar dos tipos de recuperación cuando mapea la relación entre dos entidades: `EAGER` y `LAZY` .

En general, el tipo de recuperación `EAGER` no es una buena idea, porque le dice a JPA que *siempre* obtenga los datos, incluso cuando estos datos no son necesarios.

Por ejemplo, si tiene una entidad de `Person` y la relación con `Address` como esta:

```
@Entity
public class Person {

  @OneToMany(mappedBy="address", fetch=FetchType.EAGER)
  private List<Address> addresses;

}
```

Cada vez que consulte a una `Person` , la lista de `Address` de esta `Person` también se devolverá.

Entonces, en lugar de mapear su entidad con:

```
@ManyToMany(mappedBy="address", fetch=FetchType.EAGER)
```

Utilizar:

```
@ManyToMany(mappedBy="address", fetch=FetchType.LAZY)
```

Otra cosa a la que hay que prestar atención son las relaciones `@OneToOne` y `@ManyToOne` . Ambos son EAGER *por defecto* . Por lo tanto, si le preocupa el rendimiento de su aplicación, debe establecer la búsqueda para este tipo de relación:

```
@ManyToOne(fetch=FetchType.LAZY)
```

Y:

```
@OneToOne(fetch=FetchType.LAZY)
```

Usar composición en lugar de herencia.

Hibernate tiene algunas estrategias de herencia. El tipo de herencia `JOINED` hace una unión entre la entidad secundaria y la entidad principal.

El problema con este enfoque es que Hibernate **siempre** trae los datos de todas las tablas involucradas en la herencia.

Por ejemplo, si tiene las entidades `Bicycle` y `MountainBike` usando el tipo de herencia `JOINED` :

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Bicycle {

}
```

Y:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class MountainBike extends Bicycle {

}
```

Cualquier consulta de JPQL que llegue a `MountainBike` traerá los datos de `Bicycle` , creando una consulta SQL como:

```
select mb.*, b.* from MountainBike mb JOIN Bicycle b ON b.id = mb.id WHERE ...
```

Si tiene otro padre para `Bicycle` (como `Transport` , por ejemplo), esta consulta anterior también trae los datos de este padre, haciendo una ÚNICA ADICIONAL.

Como puede ver, este es un tipo de mapeo de `EAGER` también. No tiene la opción de traer solo los datos de la tabla `MountainBike` usando esta estrategia de herencia.

Lo mejor para el rendimiento es la composición de uso en lugar de la herencia.

Para lograr esto, puede mapear la entidad `MountainBike` para tener una `bicycle` campo:

```
@Entity
public class MountainBike {

    @OneToOne(fetchType = FetchType.LAZY)
    private Bicycle bicycle;

}
```

Y `Bicycle`

```
@Entity
public class Bicycle {

}
```

Cada consulta ahora traerá solo los datos de `MountainBike` por defecto.

Lea La optimización del rendimiento en línea: https://riptutorial.com/es/hibernate/topic/2326/la-

optimizacion-del-rendimiento

Capítulo 12: Lazy Loading vs Eager Loading

Examples

Lazy Loading vs Eager Loading

La obtención o carga de datos se puede clasificar principalmente en dos tipos: ansiosos y perezosos.

Para utilizar Hibernate, asegúrese de agregar la última versión a la sección de dependencias de su archivo pom.xml:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.1.Final</version>
</dependency>
```

1. Carga impaciente y carga perezosa

Lo primero que debemos discutir aquí es qué carga perezosa y carga ansiosa son:

Eager Loading es un patrón de diseño en el que la inicialización de datos se produce en el lugar. Esto significa que las colecciones se recuperan por completo en el momento en que se recuperan los padres (se recupera inmediatamente)

La carga perezosa es un patrón de diseño que se utiliza para diferir la inicialización de un objeto hasta el punto en el que se necesita. Esto puede contribuir efectivamente al rendimiento de la aplicación.

2. Usando los diferentes tipos de carga

La carga diferida se puede habilitar usando el siguiente parámetro XML:

```
lazy="true"
```

Vamos a profundizar en el ejemplo. Primero tenemos una clase de usuario:

```
public class User implements Serializable {

    private Long userId;
    private String userName;
    private String firstName;
    private String lastName;
    private Set<OrderDetail> orderDetail = new HashSet<>();

    //setters and getters
    //equals and hashcode
    }
```

Mira el Set de orderDetail que tenemos. Ahora echemos un vistazo a la **clase OrderDetail** :

```
public class OrderDetail implements Serializable {

    private Long orderId;
    private Date orderDate;
    private String orderDesc;
    private User user;

    //setters and getters
    //equals and hashcode
}
```

La parte importante que interviene en la configuración de la carga diferida en el `UserLazy.hbm.xml` :

```
<set name="orderDetail" table="USER_ORDER" inverse="true" lazy="true" fetch="select">
    <key>
        <column name="USER_ID" not-null="true" />
    </key>
   <one-to-many class="com.baeldung.hibernate.fetching.model.OrderDetail" />
</set>
```

Así es como se habilita la carga lenta. Para deshabilitar la carga perezosa, simplemente podemos usar: `lazy = "false"` y esto a su vez habilitará la carga impaciente. El siguiente es el ejemplo de configurar una carga impaciente en otro archivo User.hbm.xml:

```
<set name="orderDetail" table="USER_ORDER" inverse="true" lazy="false" fetch="select">
    <key>
        <column name="USER_ID" not-null="true" />
    </key>
   <one-to-many class="com.baeldung.hibernate.fetching.model.OrderDetail" />
</set>
```

Alcance

Para aquellos que no han jugado con estos dos diseños, el alcance de los perezosos y ávidos está dentro de una **sesión** específica de *SessionFactory* . *Eager* carga todo al instante, lo que significa que no hay necesidad de llamar a nada para obtenerlo. Pero la recuperación perezosa suele exigir alguna acción para recuperar la colección / objeto asignado. A veces, esto es problemático, ya que la recuperación fuera de la *sesión es lenta* . Por ejemplo, tiene una vista que muestra los detalles de algunos POJO asignados.

```
@Entity
public class User {
    private int userId;
    private String username;
    @OneToMany
    private Set<Page> likedPage;

    // getters and setters here
}

@Entity
```

```
public class Page{
    private int pageId;
    private String pageURL;

    // getters and setters here
}

public class LazzyTest{
    public static void main(String...s){
        SessionFactory sessionFactory = new SessionFactory();
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();

        User user = session.get(User.class, 1);
        transaction.commit();
        session.close();

        // here comes the lazy fetch issue
        user.getLikedPage();
    }
}
```

Cuando intente obtener **perezosos** fuera de la *sesión* , obtendrá la *lazyinitializeException* . Esto se debe a que, por defecto, la estrategia de búsqueda para todos los oneToMany o cualquier otra relación es *perezosa* (llamar a DB a pedido) y cuando se cierra la sesión, no tiene poder para comunicarse con la base de datos. por lo tanto, nuestro código intenta obtener una colección de *likedPage* y *genera* una excepción porque no hay una sesión asociada para la representación de la base de datos.

La solución para esto es usar:

1. Abrir sesión en vista : en la que mantiene la sesión abierta incluso en la vista renderizada.
2. `Hibernate.initialize(user.getLikedPage())` antes de cerrar la sesión: esto le indica a Hibernate que inicialice los elementos de la colección.

Lea Lazy Loading vs Eager Loading en línea: https://riptutorial.com/es/hibernate/topic/7249/lazy-loading-vs-eager-loading

Capítulo 13: Recogiendo en hibernación

Introducción

La captura es realmente importante en JPA (Java Persistence API). En JPA, HQL (Hibernate Query Language) y JPQL (Java Persistence Query Language) se usan para obtener las entidades en función de sus relaciones. Aunque es mucho mejor que usar tantas consultas de unión y subconsultas para obtener lo que queremos mediante el uso de SQL nativo, la estrategia de cómo obtenemos las entidades asociadas en JPA sigue afectando esencialmente el rendimiento de nuestra aplicación.

Examples

Se recomienda utilizar FetchType.LAZY. Únete a buscar las columnas cuando sean necesarias.

A continuación se muestra una clase de entidad de empleador que se asigna a la tabla de empleadores. Como puede ver, utilicé **fetch = FetchType.LAZY en** lugar de fetch = FetchType.EAGER. La razón por la que estoy usando LAZY es porque el Empleador puede tener muchas propiedades más adelante y cada vez no necesito saber todos los campos de un Empleador, por lo que cargarlos todos llevará un mal desempeño, entonces el empleador está cargado.

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

        @Column(name = "name")
        private String Name;

        @OneToMany(mappedBy = "employer", fetch = FetchType.LAZY,
               cascade = { CascadeType.ALL }, orphanRemoval = true)
        private List<Employee> employees;

        public Long getId() {
            return id;
        }

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

        public String getName() {
            return name;
        }

        public void setName(String name) {
```

```
            this.name = name;
        }

        public List<Employee> getEmployees() {
            return employees;
        }

        public void setEmployees(List<Employee> employees) {
            this.employees = employees;
        }
    }
```

Sin embargo, para las asociaciones recuperadas LAZY, los proxies no inicializados a veces llevan a LazyInitializationException. En este caso, simplemente podemos usar JOIN FETCH en el HQL / JPQL para evitar LazyInitializationException.

```
SELECT Employer employer FROM Employer
       LEFT JOIN FETCH employer.name
       LEFT JOIN FETCH employer.employee employee
       LEFT JOIN FETCH employee.name
       LEFT JOIN FETCH employer.address
```

Lea Recogiendo en hibernación en línea:
https://riptutorial.com/es/hibernate/topic/9475/recogiendo-en-hibernacion

Capítulo 14: Relaciones de entidad de hibernación usando anotaciones

Parámetros

Anotación	Detalles
`@OneToOne`	Especifica una relación uno a uno con un objeto correspondiente.
`@OneToMany`	Especifica un solo objeto que se asigna a muchos objetos.
`@ManyToOne`	Especifica una colección de objetos que se asignan a un solo objeto.
`@Entity`	Especifica un objeto que se asigna a una tabla de base de datos.
`@Table`	Especifica qué tabla de base de datos se asigna también este objeto.
`@JoinColumn`	Especifica en qué columna se almacena una clave foregin.
`@JoinTable`	Especifica una tabla intermedia que almacena claves externas.

Examples

Bidireccional de muchos a muchos utilizando el objeto de tabla de combinación administrada por el usuario

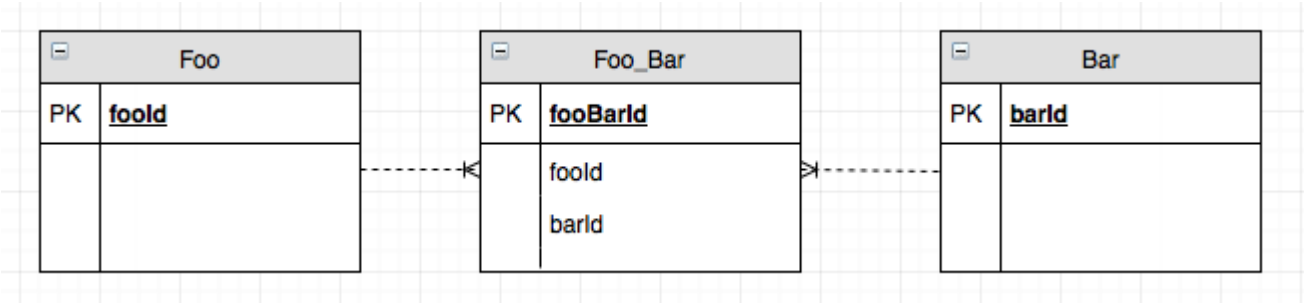
```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany(mappedBy = "bar")
    private List<FooBar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToMany(mappedBy = "foo")
    private List<FooBar> foos;
}

@Entity
@Table(name="FOO_BAR")
public class FooBar {
    private UUID fooBarId;
```

```
    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;

    @ManyToOne
    @JoinColumn(name = "barId")
    private Bar bar;

    //You can store other objects/fields on this table here.
}
```

Especifica una relación bidireccional entre muchos objetos `Foo` y muchos objetos `Bar` utilizando una tabla de unión intermedia que el usuario administra.

Los objetos `Foo` se almacenan como filas en una tabla llamada `FOO` . Los objetos `Bar` se almacenan como filas en una tabla llamada `BAR` . Las relaciones entre los objetos `Foo` y `Bar` se almacenan en una tabla llamada `FOO_BAR` . Hay un objeto `FooBar` como parte de la aplicación.

Se usa comúnmente cuando se desea almacenar información adicional en el objeto de unión, como la fecha en que se creó la relación.



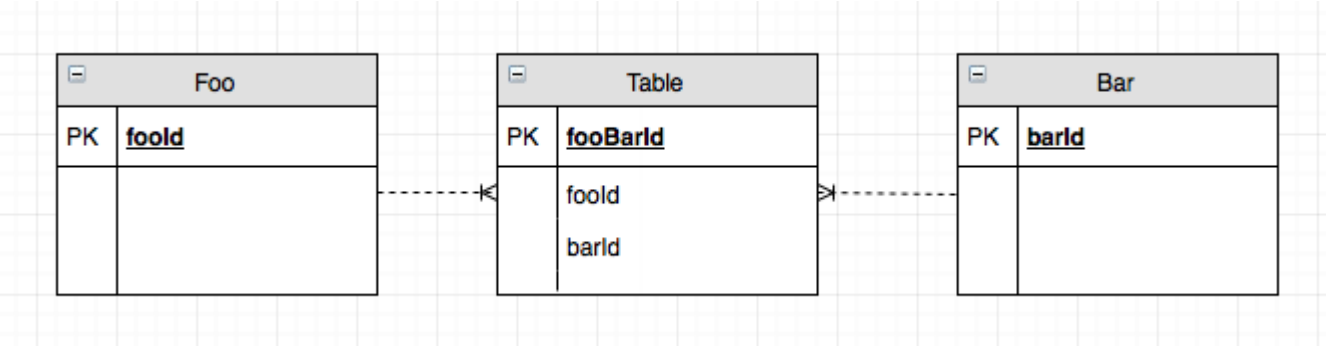
Bidireccional Muchos a muchos usando la tabla de unión administrada de Hibernate

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="fooId"),
        inverseJoinColumns = @JoinColumn(name="barId"))
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="barId"),
        inverseJoinColumns = @JoinColumn(name="fooId"))
    private List<Foo> foos;
```

```
}
```

Especifica una relación entre muchos objetos `Foo` y muchos objetos `Bar` utilizando una tabla de unión intermedia que Hibernate administra.

Los objetos `Foo` se almacenan como filas en una tabla llamada `FOO` . Los objetos `Bar` se almacenan como filas en una tabla llamada `BAR` . Las relaciones entre los objetos `Foo` y `Bar` se almacenan en una tabla llamada `FOO_BAR` . Sin embargo, esto implica que no hay ningún objeto `FooBar` como parte de la aplicación.

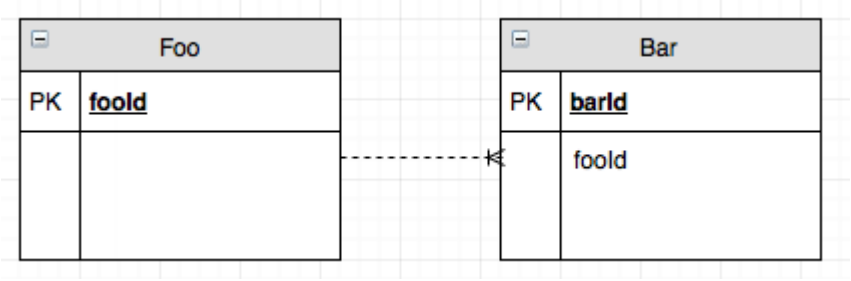


Relación bidireccional de uno a muchos mediante mapeo de clave externa

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany(mappedBy = "bar")
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;
}
```

Especifica una relación bidireccional entre un objeto `Foo` y muchos objetos `Bar` utilizando una clave externa.

Los objetos `Foo` se almacenan como filas en una tabla llamada `FOO` . Los objetos `Bar` se almacenan como filas en una tabla llamada `BAR` . La clave externa se almacena en la tabla `BAR` en una columna llamada `fooId` .



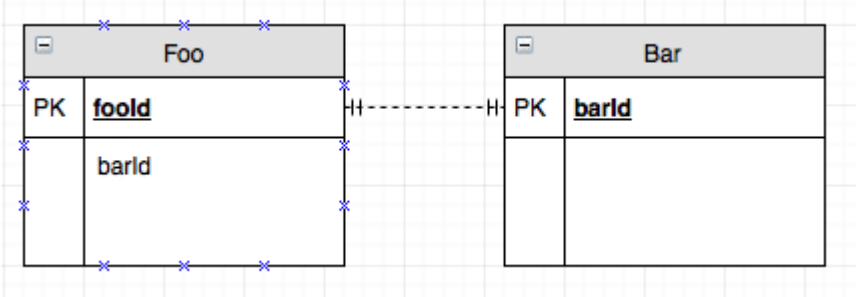
Relación bidireccional uno a uno gestionada por Foo.class

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "barId")
    private Bar bar;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToOne(mappedBy = "bar")
    private Foo foo;
}
```

Especifica una relación bidireccional entre un objeto `Foo` y un objeto `Bar` utilizando una clave externa.

Los objetos `Foo` se almacenan como filas en una tabla llamada `FOO` . Los objetos `Bar` se almacenan como filas en una tabla llamada `BAR` . La clave externa se almacena en la tabla `FOO` en una columna llamada `barId` .

Tenga en cuenta que el valor `mappedBy` es el nombre del campo en el objeto, no el nombre de la columna.



Relación unidireccional de uno a muchos mediante la tabla de unión administrada por el usuario

```
@Entity
```

```
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="fooId"),
        inverseJoinColumns = @JoinColumn(name="barId", unique=true))
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    //No Mapping specified here.
}

@Entity
@Table(name="FOO_BAR")
public class FooBar {
    private UUID fooBarId;

    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;

    @ManyToOne
    @JoinColumn(name = "barId", unique = true)
    private Bar bar;

    //You can store other objects/fields on this table here.
}
```

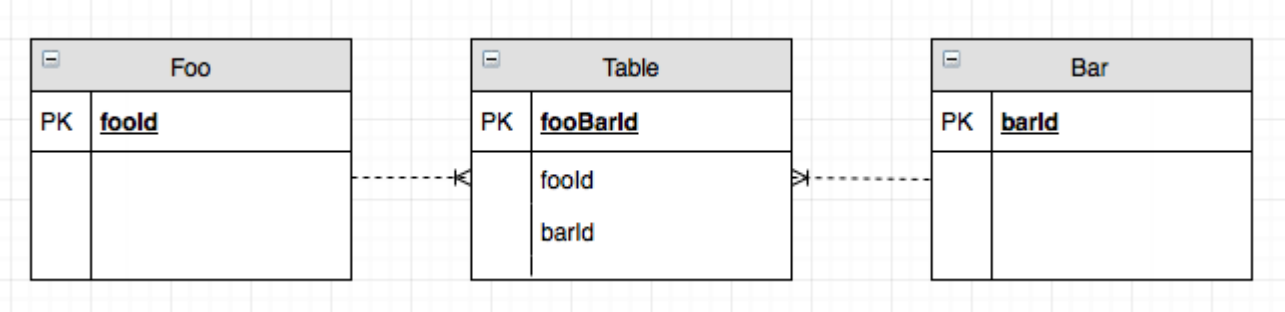
Especifica una relación unidireccional entre un objeto `Foo` y muchos objetos `Bar` utilizando una tabla de unión intermedia que el usuario administra.

Esto es similar a una relación `ManyToMany` , pero si agrega una restricción `unique` a la clave externa de destino, puede imponer que es `OneToMany` .

Los objetos `Foo` se almacenan como filas en una tabla llamada `FOO` . Los objetos `Bar` se almacenan como filas en una tabla llamada `BAR` . Las relaciones entre los objetos `Foo` y `Bar` se almacenan en una tabla llamada `FOO_BAR` . Hay un objeto `FooBar` como parte de la aplicación.

Tenga en cuenta que no existe una asignación de objetos `Bar` objetos `Foo` . `Bar` objetos de `Bar` se pueden manipular libremente sin afectar los objetos de `Foo` .

Muy comúnmente usado con Spring Security cuando se configura un objeto de `User` que tiene una lista de `Role` que pueden realizar. Puede agregar y quitar roles a un usuario sin tener que preocuparse por las cascadas de eliminación de `Role` 's.



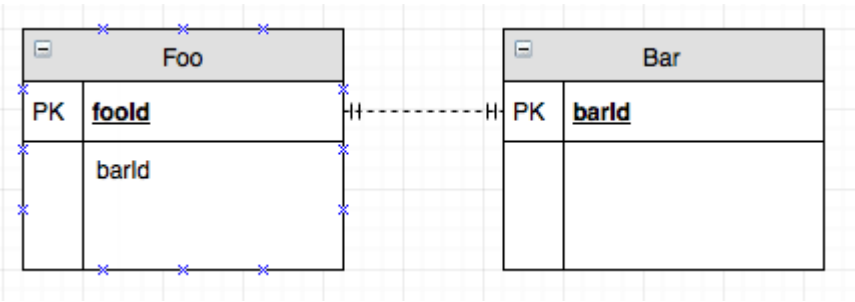
Relación unidireccional uno a uno

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToOne
    private Bar bar;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;
    //No corresponding mapping to Foo.class
}
```

Especifica una relación unidireccional entre un objeto `Foo` y un objeto `Bar` .

Los objetos `Foo` se almacenan como filas en una tabla llamada `FOO` . Los objetos `Bar` se almacenan como filas en una tabla llamada `BAR` .

Tenga en cuenta que no existe una asignación de objetos `Bar` objetos `Foo` . `Bar` objetos de `Bar` se pueden manipular libremente sin afectar los objetos de `Foo` .



Lea Relaciones de entidad de hibernación usando anotaciones en línea: https://riptutorial.com/es/hibernate/topic/5742/relaciones-de-entidad-de-hibernacion-usando-anotaciones

Creditos

S. No	Capítulos	Contributors
1	Empezando con hibernar	Community, JamesENL, Michael Piefel, Naresh Kumar, Reborn, user7491506
2	Almacenamiento en caché	Mitch Talmadge
3	Asignaciones de asociación entre entidades	StanislavL, user7491506
4	Asociaciones de mapeo	Dherik, omkar sirra
5	Consultas SQL nativas	Daniel Käfer, Nathaniel Ford, Sandeep Kamath
6	Criterios y Proyecciones	Saifer, Sameer Srivastava
7	Estrategia de nomenclatura personalizada	Mitch Talmadge, Naresh Kumar, veljkost
8	Habilitar / deshabilitar el registro de SQL	Daniel Käfer, Dherik, JamesENL, Michael Piefel
9	Hibernate y JPA	Michael Piefel
10	HQL	Daniel Käfer, user7491506
11	La optimización del rendimiento	Dherik, Michael Piefel
12	Lazy Loading vs Eager Loading	BELLIL, Pramod, Pritam Banerjee, vicky
13	Recogiendo en hibernación	rObOtAndChalie
14	Relaciones de entidad de hibernación usando	Aleksei Loginov, JamesENL

	anotaciones