Primeros pasos y conceptos detallados

J J a a v v a a PAG PAG mAi PnAiG rPsAG l

METROB l onorte okrte l l B

Java Code Geeks

JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Hebert Coelho

Byron Kiourtzoglou

Tabla de contenido

Razones que llevaron a la creación de JPA

Uno de los problemas de la Orientación a objetos es cómo mapear los objetos según lo requiera la base de datos. Es posible tener una clase con el nombre Car pero sus datos se conservan en una tabla llamada TB_CAR. El nombre de la tabla es solo el comienzo del problema, ¿y si la clase Car tiene el atributo " *nombre* "Pero en la base de datos encuentra la columna STR_NAME_CAR?

El marco básico de Java para acceder a la base de datos es JDBC. Desafortunadamente, con JDBC, se necesita mucho trabajo manual para convertir el resultado de una consulta de base de datos en clases Java.

En el siguiente fragmento de código, demostramos cómo transformar el resultado de una consulta JDBC en objetos Car:

```
importar java.sql. *;

importar java.util.LinkedList;
importar java.util.List;

clase pública MainWithJDBC {
	vacío estático público main (String [] args) lanza Excepción {
		Class.forName ( "org.hsqldb.jdbcDriver" );

		Conexión conexión = // obtener una conexión válida

		Declaración declaración = connection.createStatement (); ResultSet rs = statement.executeQuery ( "SELECCIONAR \" Id \

		", \" Nombre \ "DE \" Coche \ "" ); Lista <Coche> coches = nuevo LinkedList <Car> ();



		mientras ( rs.next ()) {
			Coche coche = nuevo Auto(); car.setId (rs.getInt ( "Identificación"
			));
			car.setName (rs.getString ( "Nombre" ));
			cars.add (coche);
		}

		por ( Coche coche: coches) {
			System.out.println ( "Identificación del coche:" + car.getId () + "Nombre del coche:" + car.getName ());
		}

		connection.close ();
	}
}
```

Como puede comprender, hay una gran cantidad de código repetitivo involucrado en el uso de este enfoque. Solo imagina que la clase Car no tiene dos, sino treinta atributos ... Para empeorar las cosas, imagina una clase con 30 atributos y con una relación con otra clase que tiene 30 atributos también, por ejemplo, una clase Car puede mantener una lista de clases Person

representando a los conductores del automóvil específico donde la clase Person puede tener 30 atributos para ser poblados.

Otras desventajas de JDBC es su portabilidad. La sintaxis de la consulta cambiará de una base de datos a otra. Por ejemplo, con la base de datos Oracle, el comando ROWNUM se usa para limitar la cantidad de líneas devueltas, mientras que con SqlServer el comando es TOP.

La portabilidad de la aplicación es un tema problemático cuando se utilizan consultas nativas de bases de datos. Hay varias soluciones para este tipo de problema, por ejemplo, un archivo separado con todo el código de consulta podría almacenarse fuera del archivo desplegable de la aplicación. Con este enfoque para cada proveedor de base de datos, se requeriría un archivo específico de SQL.

Es posible encontrar otros problemas al desarrollar con JDBC como: actualizar una tabla de base de datos y sus relaciones, no dejar registros huérfanos o una forma fácil de usar la jerarquía.

¿Qué es JPA? ¿Qué es una implementación de JPA?

JPA fue creado como una solución a los problemas mencionados anteriormente.

JPA nos permite trabajar con clases de Java ya que proporciona una capa transparente a los detalles específicos de cada base de datos; JPA hará el arduo trabajo de mapear la tabla a la estructura de clases y la semántica para el desarrollador.

Una definición fácil de JPA es: “Un grupo de especificaciones (muchos textos, regularizaciones e interfaces Java) para definir cómo debe comportarse una implementación de JPA”. Hay muchas implementaciones de JPA disponibles tanto gratuitas como de pago, por ejemplo, Hibernate, OpenJPA, EclipseLink y el "recién nacido" Batoo, etc.

La mayoría de las implementaciones de JPA son libres de agregar códigos adicionales, anotaciones que no están presentes en la especificación JPA, pero que deben cumplir con la escemántica de la especificación JPA.

La característica principal de JPA para abordar la portabilidad de la aplicación es la capacidad de asignar tablas de base de datos a las clases. En las siguientes secciones, demostraremos cómo es posible mapear la columna de una tabla en un campo de clase Java independientemente de los nombres tanto de la columna de la base de datos como del campo de la clase Java.

JPA creó un lenguaje de base de datos llamado JPQL para realizar consultas a la base de datos. La ventaja de JPQL es que la consulta se puede ejecutar en todas las bases de datos.

```
SELECCIONE identificación, nombre , color, edad, puertas DESDE Auto
```

La consulta anterior podría traducirse al JPQL a continuación:

```
SELECCIONE C DESDE Coche c
```

Observe que el resultado de la consulta anterior es "c", es decir, un objeto de automóvil y no los campos / valores que se encuentran en la tabla de la base de datos. JPA creará el objeto automáticamente.

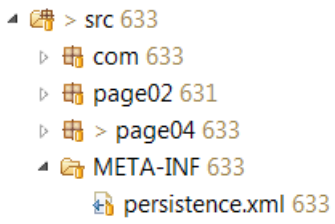
Si desea ver varias formas de ejecutar un consulta de base de datos con JPA haga clic aquí .
JPA será responsable de traducir la consulta JPQL a la consulta nativa de la base de datos; el desarrollador no tendrá que preocuparse por cuál es la sintaxis de la base de datos SQL requerida.

¿Para qué se utiliza el archivo persistence.xml? ¿Y sus configuraciones?

El archivo persistence.xml es responsable de toda la configuración del entorno JPA; puede contener la configuración específica de la implementación de la base de datos, la aplicación y JPA.

En el código persistence.xml presentado aquí usamos la configuración específica de implementación de EclipseLink JPA, pero el concepto y la teoría aplicados a las clases de Java en esta publicación se pueden aplicar cuando se usa cualquier marco de implementación de JPA disponible.

El archivo persistence.xml debe estar ubicado en una carpeta META-INF en la misma ruta que las clases de Java. A continuación se muestra una imagen que muestra dónde debe colocarse el archivo:



Esta imagen es válida cuando se usa Eclipse IDE. Para Netbeans es necesario verificar su documentación a partir de dónde está el lugar correcto para colocar el archivo.

Si recibió el mensaje de error: "No se pudo encontrar ningún archivo META-INF / persistence.xml en la ruta de clase", aquí hay algunos consejos que puede verificar:

- Abra el archivo WAR generado y verifique si persistence.xml se encuentra en “/ WEBINF / classes / META-INF /”. Si el archivo se genera automáticamente y no está allí, su error está en la creación de WAR, debe colocar el archivo persistence.xml donde espera el IDE. Si el artefacto

la creación se realiza manualmente, verifique el script de creación (ant, maven).

• Si su proyecto se implementa con un archivo EAR, verifique si el archivo persistence.xml está en la raíz del jar EJB. Si el archivo se genera automáticamente y no está allí, su error está en la creación de WAR, debe colocar el archivo persistence.xml donde espera el IDE. Si la creación del artefacto es manualmente, verifique el script de creación (ant, maven).

• Si su proyecto es un proyecto JSE (escritorio), debe verificar si el archivo no está en la carpeta “METAINF”. Como comportamiento predeterminado, el JPA buscará en la raíz JAR la carpeta y el archivo: “/METAINF/persistence.xml”.

• Compruebe si el archivo tiene el nombre: "persistence.xml". El archivo debe tener el nombre con todas las letras en minúsculas.

Si JPA no puede encontrar el archivo, se mostrará el error anterior. Como regla general, es una buena práctica colocar el archivo como se muestra arriba.

Consulte a continuación una muestra de persistence.xml:

```
<? xml versión = "1.0" codificacion = "UTF-8" ?>

< persistencia versión = "2.0"
      xmlns = "http://java.sun.com/xml/ns/persistence" xmlns: xsi = "http://www.w3.org/2001/XMLSchema-instance"
      xsi: schemaLocation = "http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" >

      < unidad de persistencia nombre = "MyPU" tipo de transacción = "RESOURCE_LOCAL" > < proveedor > org.eclipse.persistence.jpa.PersistenceProvider
            </ proveedor >

            < clase > página20.Persona </ clase >
            < clase > página20 Celular </ clase >
            < clase > page20.Call </ clase >
            < clase > página20.Perro </ clase >

            < excluir-clases-no listadas > verdadero </ excluir-clases-no listadas >

            < propiedades >
                  < propiedad nombre = "javax.persistence.jdbc.driver" valor = "org.hsqldb.jdbcDriver" /> < propiedad nombre = "javax.persistence.jdbc.url"
                  valor = "jdbc: hsqldb: mem: myDataBase" /> < propiedad nombre = "javax.persistence.jdbc.user" valor = "sa" /> < propiedad nombre
                  = "javax.persistence.jdbc.password" valor = "" /> < propiedad nombre = "eclipselink.ddl-generation" valor = "crear-tablas" /> < propiedad
                  nombre = "eclipselink.logging.level" valor = "MEJOR" />


            </ propiedades >
      </ unidad de persistencia >

      < unidad de persistencia nombre = "PostgresPU" tipo de transacción = "RESOURCE_LOCAL" > < proveedor > org.eclipse.persistence.jpa.PersistenceProvider
            </ proveedor >

            < clase > página26.Car </ clase >
            < clase > página26.Perro </ clase >
```

```
            < clase > página26.Persona </ clase >

            < excluir-clases-no listadas > verdadero </ excluir-clases-no listadas >

            < propiedades >
                  < propiedad nombre = "javax.persistence.jdbc.url" valor = "jdbc: postgresql: // localhost / JpaRelationships"
/>
                  < propiedad nombre = "javax.persistence.jdbc.driver" valor = "org.postgresql.Driver" /> < propiedad nombre = "javax.persistence.jdbc.user"
                  valor = "postgres" /> < propiedad nombre = "javax.persistence.jdbc.password" valor = "postgres" />

                  <! - <property name = "eclipselink.ddl-generation" value = "drop-and-create-tables" /> ->
                  < propiedad nombre = "eclipselink.ddl-generation" valor = "crear-tablas" />
                  <! - <nombre de propiedad = "eclipselink.logging.level" value = "MEJOR" /> ->
            </ propiedades >
      </ unidad de persistencia >
</ persistencia >
```

Acerca del código anterior:

• <persistence-unit name = ”MyPU” => con esta configuración es posible definir el nombre de la unidad de persistencia. La Unidad de persistencia puede entenderse como el universo JPA de su aplicación. Contiene información sobre todas las clases, relaciones, claves y otras configuraciones que relacionan la base de datos con su aplicación. Es posible agregar más de una unidad de persistencia en el mismo archivo persistence.xml como se muestra en el código anterior.

• transaction-type = ”RESOURCE_LOCAL” => Defina el tipo de transacción. Aquí se permiten dos valores: RESOURCE_LOCAL y JTA. Para aplicaciones de escritorio, se debe utilizar RESOURCE_LOCAL; para la aplicación web se pueden usar ambos valores, el valor correcto depende del diseño de la aplicación.
• <provider> org.eclipse.persistence.jpa.PersistenceProvider </provider> => Define el proveedor de implementación de JPA. El proveedor es la implementación de la aplicación JPA. Si su aplicación usa Hibernate, el valor del proveedor debe ser "org.hibernate.ejb.HibernatePersistence" y para OpenJPA "org.apache.openjpa.persistence.PersistenceProviderImpl".

• <class> </class> => Se usa para declarar las clases de Java. En un JEE / JSE normalmente esto no es necesario;
Por ejemplo, para una aplicación de escritorio Hibernate, no es necesario declarar las clases con estas etiquetas, pero con EclipseLink y OpenJPA se requiere la presencia de la etiqueta de clase.
• <exclude-unlisted-classes> true </exclude-unlisted-classes> => esta configuración define que si una clase no está listada en persistence.xml no debe manejarse como una entidad (veremos más sobre entidad en el página siguiente) en la Unidad de persistencia. Esta configuración es muy útil para aplicaciones con más de una Unidad de Persistencia, donde una Entidad debería aparecer en una base de datos pero no en la otra.

• Es posible comentar un código usando la etiqueta <! - ->
• <properties> => Es posible agregar configuraciones de implementaciones JPA específicas. Valores como controlador, contraseña y usuario es normal encontrarlo en todas las implementaciones; por lo general, estos valores se escriben en el archivo persistence.xml para una aplicación RESOUCE_LOCAL. Para las aplicaciones JTA, los contenedores utilizan las fuentes de datos. Se puede especificar una fuente de datos con las etiquetas <jta-data-

fuente> </jta-data-source> <non-jta-data-source> </non-jta-data-source>. Algunas versiones de JBoss requieren que la declaración de la fuente de datos esté presente incluso si la conexión es local. Vale la pena mencionar dos configuraciones:

Configuraciones	Implementación	Usado para
eclipselink.ddl-generation	EclipseLink	Activará la creación automática de la tabla de la base de datos, o simplemente validará el esquema de la base de datos contra el JPA configuración.
hibernate.hbm2ddl.auto	Hibernar	
openjpa.jdbc.SynchronizeMappings	OpenJPA	
eclipselink.logging.level	EclipseLink	Definirá el nivel LOG de la implementación de JPA.
org.hibernate.SQL.level = MEJOR org.hibernate.type.level = MEJOR	Hibernar	
openjpa.Log	OpenJPA	
En Internet encontrará los valores permitidos para cada implementación.		

Definiciones de entidades. ¿Qué son las anotaciones lógicas y físicas?

Para que JPA mapee correctamente las tablas de la base de datos en las clases de Java, se creó el concepto de Entidad. Se debe crear una entidad para admitir la misma estructura de tabla de base de datos; por lo tanto, JPA maneja la modificación de datos de la tabla a través de una entidad.

Para que una clase Java sea considerada una Entidad, debe seguir las siguientes reglas:

- Para ser anotado con @Entity
- Un constructor público sin argumentos
- La clase Java deberá tener un campo con la anotación @Id

La siguiente clase sigue todos los requisitos para ser considerada una entidad:

```
importar javax.persistence.Entity;
importar javax.persistence.Id;

@Entidad
clase pública Auto {
```

```
        público Auto(){

        }

        público Auto( En t identificación){
                esta . id = id;
        }

        // Solo para mostrar que no es necesario tener get / set cuando hablamos de JPA Id
        @Identificación
        int privado identificación;

        privado Nombre de cadena;

        público String getName () {}
                regreso nombre;


        vacío público setName (nombre de la cadena) {
                esta . nombre = nombre;
        }
}
```

Acerca del código anterior:

- La clase tiene la anotación @Entity encima de su nombre
- Hay un atributo que se considera el id de la clase que está anotado con @Id. *Cada entidad debe tener una identificación.* Por lo general, este campo es un campo entero secuencial, pero puede ser una cadena u otros valores permitidos
- Tenga en cuenta que no hay get / set para el atributo id. Para JPA se considera que la identificación de una entidad es inmutable, por lo que no es necesario editar el valor de la identificación.
- La presencia de un constructor público sin argumentos es obligatoria. Se pueden agregar otros constructores

Como se muestra en el fragmento de código anterior, solo usamos dos anotaciones, una para definir una entidad y otra para declarar el campo "id"; por defecto, JPA buscará una tabla llamada CAR en la base de datos con columnas denominadas ID y NAME. Por defecto, JPA usará el nombre de la clase y los nombres de los atributos de la clase para encontrar las tablas y sus estructuras.

Según el libro “Pro JPA 2” podemos definir las anotaciones JPA de dos formas: anotaciones lógicas y anotaciones físicas. Las anotaciones físicas mapearán la configuración de la base de datos en la clase. Las anotaciones lógicas definirán el modelado de la aplicación. Verifique el código a continuación:

```
importar java.util.List;

importar javax.persistence. *;

@Entidad
@Mesa (nombre = "TB_PERSON_02837" )
```

```
clase pública Persona {

	@Identificación
	@GeneratedValue (estrategia = GenerationType.AUTO)
	int privado identificación;

	@Básico (buscar = FetchType.LAZY)
	@Columna (nombre = "PERSON_NAME" , longitud = 100 , único = cierto , anulable = falso )
	privado Nombre de cadena;

	@Uno a muchos
	privado Lista de coches <Coche>;

	público String getName () {}
		regreso nombre;


	vacío público setName (nombre de la cadena) {
		esta . nombre = nombre;
	}
}
```

En el fragmento de código de arriba se muestran las siguientes anotaciones: @Entity, @Id y @OneToMany (veremos sobre esta anotación más adelante); estos se consideran anotaciones lógicas. Tenga en cuenta que estas anotaciones no definen nada relacionado con la base de datos, pero definen cómo una clase Java se comportará como una entidad.

Además, en el código anterior demostramos otro conjunto de anotaciones: @Table, @Column y @Basic. Estas anotaciones crearán la relación entre la tabla de la base de datos y la entidad. Es posible definir el nombre de la tabla, el nombre de la columna y otra información relacionada con la base de datos. Este tipo de anotaciones se conocen como anotaciones físicas, su trabajo es "conectar" la base de datos al código JPA.

Está fuera del alcance de este mini libro presentar todas las anotaciones respaldadas por JPA, pero es muy fácil encontrar esta información en Internet; p.ej: *@ Columna (nombre = "PERSON_NAME", longitud = 100, único = verdadero, anulable = falso).*
Las anotaciones físicas son más fáciles de entender porque se parecen a la configuración de la base de datos.

Id Generation: Definición, usando identidad o secuencia

Como se dijo en capítulos anteriores, toda entidad debe tener una identificación. JPA tiene la opción de generar automáticamente la identificación de la entidad.

Hay tres opciones para la generación automática de identificaciones:

- Identidad
- Secuencia
- TableGenerator

Debemos tener en cuenta que cada base de datos tiene su propio mecanismo de generación de id. Las bases de datos de Oracle y Postgres usan el enfoque Sequence, SqlServer y MySQL usan el enfoque Identity. No es posible utilizar un enfoque de generación de ID en un servidor cuando el servidor no lo admite.

Se permite el uso de los siguientes tipos de Java para un atributo de identificación: byte / Byte, int / Integer, short / Short, long / Long, char / Character, String, BigInteger, java.util.Date y java.sql.Date.

Identidad

Este es el enfoque de generación de id más simple. Simplemente anote el campo de identificación como se muestra a continuación:

```
importar javax.persistence.Entity;
importar javax.persistence.GeneratedValue;
importar javax.persistence.GenerationType;
importar javax.persistence.Id;

@Entidad
clase pública Persona {

	@Identificación
	@GeneratedValue (estrategia = GenerationType.IDENTITY)
	int privado identificación;

	privado Nombre de cadena;

	público String getName () {}
		regreso nombre;


	vacío público setName (nombre de la cadena) {
		esta . nombre = nombre;
	}
}
```

La base de datos controla la generación de ID, JPA no actúa en absoluto sobre la ID. Por lo tanto, para recuperar la identificación de la base de datos, una entidad debe persistir primero y, después de que la transacción se confirma, se ejecuta una consulta para recuperar la identificación generada para la entidad específica. Este procedimiento conduce a una pequeña degradación del rendimiento, pero no a algo problemático.

El enfoque de generación de id antes mencionado no permite la asignación de id en la memoria. Examinaremos lo que significa esta asignación de identificadores en los siguientes capítulos.

Secuencia

El enfoque de secuencia se puede configurar como se muestra a continuación:

```
importar javax.persistence.Entity;
importar javax.persistence.GeneratedValue;
importar javax.persistence.GenerationType;
importar javax.persistence.Id;
importar javax.persistence.SequenceGenerator;

@Entidad
@SequenceGenerator (nombre = Car.CAR_SEQUENCE_NAME, sequenceName = Car.CAR_SEQUENCE_NAME, initialValue = 10 , tamaño de asignación = 53 )

clase pública Auto {

        final estática pública Cadena CAR_SEQUENCE_NAME = "CAR_SEQUENCE_ID" ;

        @Identificación
        @GeneratedValue (estrategia = GenerationType.SEQUENCE, generator = CAR_SEQUENCE_NAME)
        int privado identificación;

        privado Nombre de cadena;

        público String getName () {}
                regreso nombre;


        vacío público setName (nombre de la cadena) {
                esta . nombre = nombre;
        }
}
```

Acerca del código anterior:

- La anotación @SequenceGenerator definirá que habrá una secuencia en la base de datos con el nombre especificado (atributo sequenceName). La misma secuencia podría compartirse entre entidades, pero no se recomienda. Si usamos la secuencia Car en la entidad House, los valores de id en las tablas no serían secuenciales; cuando se persiste el primer automóvil, el id sería 1, si se persiste otro automóvil, el id sería 2; cuando se persiste la primera casa, su id sería 3, esto sucederá si la entidad Casa usa la misma secuencia que la entidad Coche.

- El atributo "nombre = Car.CAR_SEQUENCE_NAME" define el nombre de la secuencia dentro de la aplicación. La declaración de secuencia debe realizarse solo una vez. Todas las entidades que usarán la misma secuencia solo tendrán que usar el nombre de secuencia específico. Es por eso que asignamos el valor del nombre de la secuencia a un atributo estático en la entidad específica.

- El valor "sequenceName = Car.CAR_SEQUENCE_NAME" refleja el nombre de la secuencia en el nivel de la base de datos.

- El "initialValue = 10" define el primer valor de identificación de la secuencia. Un desarrollador debe tener cuidado

con esta configuración; si después de insertar la primera fila se reinicia la aplicación, JPA intentará insertar una nueva entidad con el mismo valor definido en el atributo initialValue. Se mostrará un mensaje de error que indica que la identificación ya está en uso.

• "allocationSize = 53" representa la cantidad de identificadores que JPA almacenará en la caché. Funciona así: cuando se inicia la aplicación, JPA asignará en la memoria el número especificado de identificadores y compartirá estos valores con cada nueva entidad persistente. En el código anterior, los identificadores irían desde 10 (valor inicial) hasta 63 (valor inicial + tamaño de asignación). Cuando finalice el número de identificadores asignados, JPA solicitará de la base de datos y asignará en la memoria 53 identificadores más. Este acto de asignar identificadores en la memoria es un buen enfoque para optimizar la memoria del servidor, ya que JPA no necesitará activar la base de datos con cada inserción para obtener el identificador creado al igual que con el enfoque @Identity.

• El @GeneratedValue (estrategia = GenerationType.SEQUENCE, generator = CAR_SEQUENCE_NAME) define que el tipo de generación es SEQUENCE y el nombre del generador.

Generación de identificación: TableGenerator y Auto

TableGenerator

El texto a continuación describe cómo funciona TableGenerator:

• Se utiliza una tabla para almacenar los valores de id.

• Esta tabla tiene una columna que almacenará el nombre de la tabla y el valor de identificación real

• Hasta ahora, este es el único enfoque de Generation ID que permite la portabilidad de la base de datos, sin la necesidad de alterar el enfoque de generación de id. Imagine una aplicación que se ejecuta con Postgres y usa una secuencia. Si tuviéramos que usar la misma aplicación con SqlServer también, tendríamos que crear dos artefactos distintos (WAR / JAR / EAR), uno para cada base de datos, ya que SqlServer no admite Secuencias. Con el enfoque de TableGenerator, se puede usar el mismo artefacto para ambas bases de datos.

Verifique el código a continuación para ver cómo se puede usar el enfoque de TableGenerator:

```
importar javax.persistence. *;

@Entidad
clase pública Persona {

    @Identificación
    @TableGenerator (nombre = "TABLE_GENERATOR" , tabla = "ID_TABLE" , pkColumnName = "ID_TABLE_NAME" ,
pkColumnValue = "PERSON_ID" , valueColumnName = "ID_TABLE_VALUE" )
    @GeneratedValue (estrategia = GenerationType.TABLE, generator = "TABLE_GENERATOR" )
```

```
        int privado identificación;

        privado Nombre de cadena;

        público String getName () {}
                regreso nombre;


        vacío público setName (nombre de la cadena) {
                esta . nombre = nombre;
        }
}
```

El código anterior usará la tabla en la imagen a continuación (es posible configurar JPA para crear la tabla específica automáticamente, simplemente agregue la configuración necesaria como se presenta en la sección sobre persistence.xml):

	id_table_name [PK] character varying(255)	id_table_value bigint
1	PERSON_ID	101
*		

Acerca del código anterior:

- " name "=> Es el ID de TableGenerator dentro de la aplicación.
- " table "=> Nombre de la tabla que contendrá los valores.
- " pkColumnName "=> Nombre de la columna que contendrá el nombre de identificación. En el código de arriba, se utilizará el nombre de la columna "id_table_name".
- " valueColumnName "=> Nombre de la columna que contendrá el valor de id.
- " pkColumnValue "=> Nombre de la tabla que persistirá en la tabla. El valor predeterminado es el nombre de la entidad + id (atributo de entidad definido como id). En el código anterior estará PERSON_ID que es el mismo valor descrito anteriormente.

- initialValue, deploymentSize => Estas opciones también se pueden usar en la anotación @TableGenerator. Consulte la sección Método de secuencia anterior para ver cómo se deben usar.
- Es posible declarar TableGenerator en diferentes entidades sin el problema de perder identificadores secuenciados (como el problema Sequence que vimos antes).

La mejor práctica para usar la declaración del generador de tablas es en un archivo orm.xml. Este archivo se utiliza para anular la configuración de JPA a través de anotaciones y está fuera del alcance de este mini libro.

Auto

El enfoque automático (automáticamente) permite a JPA elegir un enfoque a utilizar. Este es el valor predeterminado y se puede utilizar como se muestra a continuación:

```
@Identificación
@GeneratedValue (estrategia = GenerationType.AUTO) // o simplemente @GeneratedValue
int privado identificación;
```

Con el enfoque automático, JPA puede utilizar cualquier estrategia. JPA elegirá entre los 3 enfoques discutidos anteriormente.

Clave compuesta simple

Una clave simple es cuando la identificación usa solo un campo. Se usa una clave simple como a continuación:

```
@Id
private int id;
```

Se necesita una clave compuesta cuando se requiere más de un atributo como identificación de entidad. Es posible encontrar una clave compuesta simple y una clave compuesta compleja. Con una clave compuesta simple, use solo atributos simples de Java para la identificación (por ejemplo, String, int,…). En las siguientes secciones, analizaremos todo lo relacionado con la semántica de claves compuestas complejas.

Hay dos formas de asignar una clave compuesta simple, con @IdClass o @EmbeddedId.

@IdClass

Verifique el código a continuación:

```
importar javax.persistence. *;

@Entidad
@IdClass (CarId. clase )
clase pública Auto {

    @Identificación
    int privado de serie;

    @Identificación
    privado Marca de cuerda;
```

```
        privado Nombre de cadena;

        público String getName () {}
                regreso nombre;


        vacío público setName (nombre de la cadena) {
                esta . nombre = nombre;
        }

        público int getSerial () {
                regreso de serie;
        }

        vacío público setSerial ( En t serial) {
                esta . serial = serial;
        }

        público String getBrand () {}
                regreso marca;


        vacío público setBrand (marca de cadena) {
                esta . marca = marca;
        }
}
```

Sobre el código anterior;

- @IdClass (CarId.class) => esta anotación indica que el CarId tiene dentro de él los atributos de identificación que se encuentran en la entidad Car.
- Todos los campos anotados con @Id deben encontrarse en IdClass.
- Es posible utilizar la anotación @GeneratedValue con este tipo de clave compuesta. Por ejemplo, en el código anterior sería posible utilizar @GeneratedValue con el atributo "serial".

Compruebe a continuación el código de clase CarId:

```
importar java.io.Serializable;

clase pública CarId implementos Serializable {

        privado estático final largo serialVersionUID = 343L;

        int privado de serie;
        privado Marca de cuerda;

        // debe tener un construcot predeterminado
        público CarId () {

        }
```

```
        público CarId ( En t serial, marca String) {
                esta . serial = serial;
                esta . marca = marca;
        }

        público int getSerial () {
                regreso de serie;
        }

        público String getBrand () {}
                regreso marca;


        // Debe tener un método hashCode
        @Anular
        público int código hash() {
                regreso serial + brand.hashCode (); }


        // Debe tener un método igual
        @Anular
        booleano público es igual a (Objeto obj) {
                si ( obj en vez de CarId) {
                        CarId carId = (CarId) obj;
                        regreso carId.serial == esta . serial && carId.brand.equals ( esta . marca);
                }

                falso retorno ;
        }
}
```

La clase CarId tiene los campos listados como @Id en la entidad Car. Para usar una clase

como ID, debe seguir las siguientes reglas:

- Se debe encontrar un constructor público sin argumentos.
- Implementa la interfaz serializable
- Sobrescribir el método hashCode / equals

Para hacer una consulta en una base de datos para encontrar una entidad dada una clave compuesta simple, simplemente haga lo siguiente:

```
EntityManager em = // obtener un administrador de entidad válido

CarId carId = nuevo CarId ( 33 , "Vado" ); Car persistedCar = em.find (Car. clase , carId); System.out.println

(persistedCar.getName () + "-" + persistedCar.getSerial ());
```

Para utilizar el método de búsqueda es necesario proporcionar la clase de identificación con la información requerida.

@Embeddable

El otro enfoque para usar la clave compuesta se presenta a continuación:

```
importar javax.persistence. *;

@Entidad
clase pública Auto {

	@EmbeddedId
	privado CarId carId;

	privado Nombre de cadena;

	público String getName () {}
		regreso nombre;


	vacío público setName (nombre de la cadena) {
		esta . nombre = nombre;
	}

	público CarId getCarId () {}
		regreso carId;


	vacío público setCarId (CarId carId) {
		esta . carId = carId;
	}
}
```

Acerca del código anterior:

- La clase de identificación se escribió dentro de la clase Coche.
- El @EmbeddedId se usa para definir la clase como una clase de identificación.
- Ya no es necesario utilizar la anotación @Id.

La identificación de la clase será la siguiente:

```
importar java.io.Serializable;

importar javax.persistence.Embeddable;

@Embeddable
clase pública CarId implementos Serializable {

	privado estático final largo serialVersionUID = 343L;

	int privado de serie;
	privado Marca de cuerda;
```

```
        // debe tener un construcot predeterminado
        público CarId () {

        }

        público CarId ( En t serial, marca String) {
                esta . serial = serial;
                esta . marca = marca;
        }

        público int getSerial () {
                regreso de serie;
        }

        público String getBrand () {}
                regreso marca;


        // Debe tener un método hashCode
        @Anular
        público int código hash() {
                regreso serial + brand.hashCode (); }


        // Debe tener un método igual
        @Anular
        booleano público es igual a (Objeto obj) {
                si ( obj en vez de CarId) {
                        CarId carId = (CarId) obj;
                        regreso carId.serial == esta . serial && carId.brand.equals ( esta . marca);
                }

                falso retorno ;
        }
}
```

Acerca del código anterior:

- La anotación @Embeddable permite que la clase se use como id.
- Los campos dentro de la clase se usarán como identificadores.

Para usar una clase como ID, debe seguir las siguientes reglas:

- Se debe encontrar un constructor público sin argumentos.
- Implementa la interfaz serializable
- Sobrescribir el método hashCode / equals

Es posible realizar consultas con este tipo de clave compuesta como la @IdClass presentada anteriormente.

Clave compuesta compleja

Una clave compuesta compleja se compone de otras entidades, no de atributos simples de Java.

Imagina una entidad DogHouse donde usa el perro como identificación. Eche un vistazo al código a continuación:

```
importar javax.persistence. *;

@Entidad
clase pública Perro {
	@Identificación
	int privado identificación;

	privado Nombre de cadena;

	público int getId () {
		regreso identificación;
	}

	vacío público Pon la identificacion( En t identificación) {
		esta . id = id;
	}

	público String getName () {}
		regreso nombre;


	vacío público setName (nombre de la cadena) {
		esta . nombre = nombre;
	}
}
```

```
importar javax.persistence. *;

@Entidad
clase pública Casa de perro {

	@Identificación
	@Doce y cincuenta y nueve de la noche
	@UnirseColumna (nombre = "DOG_ID" )
	privado Perro perro;

	privado Marca de cuerda;

	público Perro getDog () {}
		regreso perro;


	vacío público setDog (perro perro) {
		esta . perro = perro;
	}

	público String getBrand () {
```

```
            regreso marca;
        }

        vacío público setBrand (marca de cadena) {
            esta . marca = marca;
        }
}
```

Acerca del código anterior:

- La anotación @Id se utiliza en la entidad DogHouse para informar a JPA que DogHouse tendrá la misma identificación que el Dog.

- Podemos combinar la anotación @Id con la anotación @OneToOne para dictar que existe una relación explícita entre las clases. Más información sobre la anotación @OneToOne estará disponible más adelante.

Imagine un escenario en el que se requiere acceder a la identificación de DogHouse sin pasar por la clase Dog (dogHouse.getDog (). GetId ()). JPA tiene una forma de hacerlo sin la necesidad del patrón de la Ley de Deméter :

```
importar javax.persistence. *;

@Entidad
clase pública DogHouseB {

        @Identificación
        int privado dogId;

        @MapsId
        @Doce y cincuenta y nueve de la noche
        @UnirseColumna (nombre = "DOG_ID" )
        privado Perro perro;

        privado Marca de cuerda;

        público Perro getDog () {}
            regreso perro;


        vacío público setDog (perro perro) {
            esta . perro = perro;
        }

        público String getBrand () {}
            regreso marca;


        vacío público setBrand (marca de cadena) {
            esta . marca = marca;
        }

        público int getDogId () {
            regreso dogId;
```

```
        }

        vacío público setDogId ( En t dogId) {
                esta . dogId = dogId;
        }
}
```

Acerca del código anterior:

- Hay un campo explícito mapeado con @Id.
- La entidad Dog se asigna con la anotación @MapsId. Esta anotación indica que JPA usará Dog.Id como DogHouse.DogId (como antes); el atributo dogId tendrá el mismo valor que dog.getId () y este valor se atribuirá en tiempo de ejecución.
- El campo dogId no necesita asignarse explícitamente a una columna de la tabla de la base de datos. Cuando la aplicación se inicia, JPA atribuirá dog.getId () al atributo dogId.

Para terminar este tema, veamos un tema más. ¿Cómo podemos mapear una identificación de entidad con más de una entidad?

Verifique el código a continuación:

```
importar javax.persistence. *;

@Entidad
@IdClass (DogHouseId. clase )
clase pública Casa de perro {

        @Identificación
        @Doce y cincuenta y nueve de la noche
        @UnirseColumna (nombre = "DOG_ID" )
        privado Perro perro;

        @Identificación
        @Doce y cincuenta y nueve de la noche
        @UnirseColumna (nombre = "PERSON_ID" )
        privado Persona persona;

        privado Marca de cuerda;

        // obtener y configurar
}
```

Acerca del código anterior:

- Observe que ambas entidades (Perro y Persona) se anotaron con @Id.
- La anotación @IdClass se usa para indicar la necesidad de una clase para mapear el id.

```
importar java.io.Serializable;

clase pública DogHouseId implementos Serializable {
```

```
    privado estático final largo serialVersionUID = 1L;

    int privado persona;
    int privado perro;

    público int getPerson () {
        regreso persona;
    }

    vacío público setPerson ( En t persona) {
        esta . person = persona;
    }

    público int getDog () {
        regreso perro;
    }

    vacío público setDog ( En t perro) {
        esta . perro = perro;
    }

    @Anular
    público int código hash() {
        regreso persona + perro; }


    @Anular
    booleano público es igual a (Objeto obj) {
        si ( obj en vez de DogHouseId) {
            DogHouseId dogHouseId = (DogHouseId) obj;
            regreso dogHouseId.dog == perro && dogHouseId.person == persona;
        }

        falso retorno ;
    }
}
```

Acerca del código anterior:

- La clase tiene la misma cantidad de atributos que la cantidad de atributos en la clase DogHouse anotado con @Id

- Observe que los atributos dentro de DogHouseId tienen el mismo nombre que los atributos dentro de DogHouse anotados con @Id. Esto es obligatorio para que JPA utilice correctamente la funcionalidad de identificación. Por ejemplo, si nombramos el atributo de tipo Person dentro de la clase DogHouse como "dogHousePerson", el nombre del atributo de tipo Person dentro de la clase DogHouseId tendría que cambiar también a "dogHousePerson".

Para usar una clase como ID, debe seguir las siguientes reglas:

- Se debe encontrar un constructor público sin argumentos.

- Implementa la interfaz serializable
- Sobrescribir el método hashCode / equals

Cómo obtener un EntityManager

Hay dos formas de obtener un EntityManager. Uno es con inyección y el otro a través de una fábrica.

La forma más fácil de obtener un EntityManager es mediante inyección, el contenedor inyectará el EntityManager. A continuación se muestra cómo funciona el código de inyección:

```
@PersistenceContext (unitName = "PERSISTENCE_UNIT_MAPPED_IN_THE_PERSISTENCE_XML" )
privado EntityManager entityManager;
```

Es necesario anotar el campo EntityManager con: "@PersistenceContext (unitName = “PERSISTENCE_UNIT_MAPPED_IN_THE_PERSISTENCE_XML”) ”. La opción de inyección solo funcionará para aplicaciones JEE, ejecutándose dentro de servidores de aplicaciones como JBoss, Glassfish… Para lograr la inyección sin problemas el persistence.xml debe estar en el lugar correcto, y debe tener (si es necesario) una fuente de datos definida.

La inyección EntityManager, hasta hoy, funcionará solo con un servidor que admita un contenedor EJB. Tomcat y otros contenedores solo WEB / Servlet no lo inyectarán.

Cuando la aplicación es una aplicación JSE (escritorio) o cuando una aplicación web quiere manejar la conexión de la base de datos manualmente, simplemente use el código a continuación:

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory ( "PERSISTENCE_UNIT_MAPPED_IN_THE_PERSISTENCE_XML" ); EntityManager
entityManager = emf.createEntityManager ();

entityManager.getTransaction (). begin ();

// hacer algo

entityManager.getTransaction (). commit ();
entityManager.close ();
```

Tenga en cuenta que es necesario obtener primero una instancia de EntityManagerFactory, que se vinculará a una PersistenceUnit creada en el archivo persistence.xml. A través de EntityManagerFactory es posible obtener una instancia de EntityManager.

Mapeo de dos o más tablas en una entidad

Una clase puede tener información en más de una tabla.

Para mapear una entidad que tiene sus datos en más de una tabla, simplemente haga lo siguiente:

```
importar javax.persistence. *;

@Entidad
@Mesa (nombre = "PERRO" )
@SecondaryTables ({
        @SecondaryTable (nombre = "DOG_SECONDARY_A" , pkJoinColumns = { @PrimaryKeyJoinColumn (nombre = "DOG_ID" )}),
        @SecondaryTable (nombre = "DOG_SECONDARY_B" , pkJoinColumns = { @PrimaryKeyJoinColumn (nombre = "DOG_ID" )})
})
clase pública Perro {
        @Identificación
        @GeneratedValue (estrategia = GenerationType.AUTO)
        int privado identificación;

        privado Nombre de cadena;
        int privado edad;
        doble privada peso;

        // obtener y configurar

}
```

Acerca del código anterior:

- La anotación @SecundaryTable se usa para indicar en qué tabla se pueden encontrar los datos de la entidad. Si los datos se encuentran en solo una tabla secundaria, solo la anotación @SecundaryTable es suficiente.

- La anotación @SecundaryTables se usa para agrupar varias tablas en una clase. Agrupa varias anotaciones de @SecundaryTable.

Jerarquía de mapeo: MappedSuperclass

A veces es necesario compartir métodos / atributos entre clases; así crear una jerarquía de clases. En caso de que la superclase, la que tiene los métodos / atributos compartidos, no sea una entidad, debe estar marcada como MappedSuperclass.

Compruebe a continuación cómo se puede aplicar el concepto MappedSuperclass:

```
importar javax.persistence.MappedSuperclass;

@SuperclaseMapa
```

```
clase abstracta pública DogFather {
	privado Nombre de cadena;

	público String getName () {}
		regreso nombre;


	vacío público setName (nombre de la cadena) {
		esta . nombre = nombre;
	}

}
```

```
@Entidad
@Mesa (nombre = "PERRO" )
clase pública Perro se extiende DogFather {

	@Identificación
	@GeneratedValue (estrategia = GenerationType.AUTO)
	int privado identificación;

	privado Color de la cuerda;

	público int getId () {
		regreso identificación;
	}

	vacío público Pon la identificacion( En t identificación) {
		esta . id = id;
	}

	público String getColor () {}
		regreso color;


	vacío público setColor (String color) {
		esta . color = color;
	}
}
```

Acerca del código anterior:

- La clase DogFather está anotada con la anotación @MappedSuperclass. Con esta anotación, todas las clases secundarias de DogFather conservarán sus atributos en la base de datos, pero DogFather no se asignará a una tabla de la base de datos.

- Una MappedSuperclass puede ser una clase abstracta o concreta.
- La clase Dog tiene el generador de ID, solo Dog es una entidad. DogFather no es una entidad.

Algunas recomendaciones sobre MappedSuperclass:

- Una MappedSuperclass no se puede anotar con @Entity \ @Table. No es una clase que será

persistió. Sus atributos / métodos se reflejarán en sus clases secundarias.

• Siempre es una buena práctica crearlo como abstracto. No se utilizará una MappedSuperclass en las consultas.

• No se puede conservar, no es una entidad.

Cuándo lo usamos?

Si no necesitamos usar la superclase en consultas de bases de datos, sería una buena idea usar una MappedSuperclass. En el caso contrario, es una buena idea utilizar la jerarquía de entidades (consulte las siguientes secciones para obtener más detalles).

Jerarquía de asignación: tabla única

Con JPA es posible encontrar diferentes enfoques para persistir jerarquías de clases. En un lenguaje orientado a objetos como Java, es muy fácil encontrar jerarquías entre clases cuyos datos se conservarán.

La estrategia de tabla única almacenará todos los datos de la jerarquía en una tabla. Verifique el código a continuación:

```
importar javax.persistence. *;

@Entidad
@Mesa (nombre = "PERRO" )
@Herencia (estrategia = InheritanceType.SINGLE_TABLE)
@DiscriminadorColumna (nombre = "DOG_CLASS_NAME" )
clase abstracta pública Perro {
	@Identificación
	@GeneratedValue (estrategia = GenerationType.AUTO)
	int privado identificación;

	privado Nombre de cadena;

	// obtener y configurar
}
```

```
importar javax.persistence.Entity;

@Entidad
@DiscriminatorValue ( "PERRO PEQUEÑO" )
clase pública Perro pequeño se extiende Perro {
	privado String littleBark;

	público String getLittleBark () {
		regreso littleBark;
	}

	vacío público setLittleBark (String littleBark) {
		esta . littleBark = littleBark;
	}
```

```
}
```

```
importar javax.persistence. *;

@Entidad
@DiscriminatorValue ( "HUGE_DOG" )
clase pública HugeDog se extiende Perro {
        int privado hugePooWeight;

        público int getHugePooWeight () {
                regreso hugePooWeight;
        }

        vacío público setHugePooWeight ( En t hugePooWeight) {
                esta . enormePooWeight = enormePooWeight;
        }
}
```

Acerca del código anterior:

- @Inheritance (estrategia = InheritanceType.SINGLE_TABLE) => esta anotación debe colocarse en el lugar más alto de la jerarquía (la clase "padre"), también conocida como "raíz". Esta anotación definirá el patrón de jerarquía a seguir, en la anotación sobre la estrategia de mapeo de jerarquía se establece en Tabla única.

- @DiscriminatorColumn (name = “DOG_CLASS_NAME”) => definirá el nombre de la columna que vinculará una fila de la tabla de la base de datos a una clase. Compruebe en la imagen de abajo cómo se almacenan los datos.
- @DiscriminatorValue => Establecerá el valor que se persistirá en la columna definida en la anotación @DiscriminatorColumn. Compruebe en la imagen de abajo cómo se almacenan los datos.
- Tenga en cuenta que la identificación solo se define en la clase raíz. No está permitido que una clase secundaria declare una identificación.

id [PK] integer	dog_class_name character varying(31)	name character varyi	littlebark character v	hugepoowe integer
1	SMALL DOG	Red	hau	
2	SMALL DOG	Green	hiu	
3	SMALL DOG	Black	hie	
4	HUGE DOG	Yellow		3
5	HUGE DOG	Brown		3
6	HUGE DOG	Snow		3

También es posible definir la columna del discriminador de clases como un número entero:

- @DiscriminatorColumn (name = "DOG_CLASS_NAME", discriminatorType = DiscriminatorType.INTEGER) => definir el campo como entero

• @DiscriminatorValue (“1 ″) => El valor que se persistirá en la Entidad debe cambiar, se persistirá un número en lugar de un texto.

Jerarquía de mapeo: unido

Cuando una jerarquía utiliza la estrategia Unida, cada entidad tendrá sus datos almacenados en la tabla respectiva. En lugar de utilizar una sola tabla para todas las clases de jerarquía, cada entidad tendrá su propia tabla.

Consulte el código a continuación para ver cómo se puede utilizar la estrategia conjunta:

```
importar javax.persistence. *;

@Entidad
@Mesa (nombre = "PERRO" )
@Herencia (estrategia = InheritanceType.JOINED)
@DiscriminadorColumna (nombre = "DOG_CLASS_NAME" )
clase abstracta pública Perro {

        @Identificación
        @GeneratedValue (estrategia = GenerationType.AUTO)
        int privado identificación;

        privado Nombre de cadena;

        // obtener y configurar
}
```

```
importar javax.persistence. *;

@Entidad
@DiscriminatorValue ( "HUGE_DOG" )
clase pública HugeDog se extiende Perro {
        int privado hugePooWeight;

        público int getHugePooWeight () {
                regreso hugePooWeight;
        }

        vacío público setHugePooWeight ( En t hugePooWeight) {
                esta . enormePooWeight = enormePooWeight;
        }
}
```

```
importar javax.persistence. *;

@Entidad
@DiscriminatorValue ( "PERRO PEQUEÑO" )
clase pública Perro pequeño se extiende Perro {
        privado String littleBark;
```

```
        público String getLittleBark () {
                regreso littleBark;
        }

        vacío público setLittleBark (String littleBark) {
                esta . littleBark = littleBark;
        }
}
```

Acerca del código anterior:

- La anotación @Inheritance (estrategia = InheritanceType.JOINED) ahora tiene el valor Unido.
- Comprueba a continuación cómo están las tablas:

Mesa para perros

id [PK] integer	dog_class_name character varying(31)	name character varying(255)
1	SMALL DOG	Red
2	SMALL DOG	Green
3	SMALL DOG	Black
4	HUGE DOG	Yellow
5	HUGE DOG	Brown
6	HUGE DOG	Snow

Mesa HugeDog

id [PK] integer	hugepooweight integer
4	6
5	3
6	4

Mesa SmallDog

id [PK] integer	littlebark character varying(255)
1	hau
2	hiu
3	hie

Observe en las imágenes de arriba cómo se conservan los datos en cada tabla. Cada entidad tiene su información persistente en tablas únicas; para esta estrategia, JPA utilizará una tabla por entidad, independientemente de que la entidad sea concreta o abstracta.

La tabla Dog mantiene todos los datos comunes a todas las clases de la jerarquía; La tabla Dog mantiene una columna que indica a qué entidad pertenece una fila.

Jerarquía de asignación: tabla por clase concreta

La estrategia Table Per Concrete creará una tabla por entidad concreta. Si se encuentra una entidad abstracta en la jerarquía, estos datos se conservarán en las tablas de la base de datos de entidades secundarias concretas.

Verifique el código a continuación:

```
importar javax.persistence. *;

@Entidad
@Mesa (nombre = "PERRO" )
@Herencia (estrategia = InheritanceType.TABLE_PER_CLASS)
clase abstracta pública Perro {

        @Identificación
        @GeneratedValue (estrategia = GenerationType.AUTO)
        int privado identificación;

        privado Nombre de cadena;

        // obtener y configurar
}
```

```
importar javax.persistence.Entity;

@Entidad
clase pública HugeDog se extiende Perro {
        int privado hugePooWeight;

        público int getHugePooWeight () {
                regreso hugePooWeight;
        }

        vacío público setHugePooWeight ( En t hugePooWeight) {
                esta . enormePooWeight = enormePooWeight;
        }
}
```

```
importar javax.persistence.Entity;

@Entidad
```

```
clase pública Perro pequeño se extiende Perro {
        privado String littleBark;

        público String getLittleBark () {
                regreso littleBark;
        }

        vacío público setLittleBark (String littleBark) {
                esta . littleBark = littleBark;
        }
}
```

Acerca del código anterior:

- @Inheritance (estrategia = InheritanceType.TABLE_PER_CLASS) => define el tipo de jerarquía como tabla por clase.

- La anotación @DiscriminatorColumn (nombre = “DOG_CLASS_NAME”) ya no se usa en las clases de niños. Cada clase concreta tendrá sus propios datos, JPA no distribuirá los datos de la entidad en las tablas.

- No es necesaria la anotación @DiscriminatorValue para el mismo principal que el anterior.

- Consulte a continuación las tablas de la base de datos:

Mesa HugeDog

id [PK] integer	hugepooweight integer	name character varying(255)
4	6	Yellow
5	3	Brown
6	4	Snow

Mesa SmallDog

id [PK] integer	littlebark character varying(255)	name character varying(255)
1	hau	Red
2	hiu	Green
3	hie	Black

Observe que los atributos de la entidad Dog se conservan en la tabla HugeDog y SmallDog.

Ventajas y desventajas de cada enfoque de mapeo jerárquico

Desafortunadamente, no existe un enfoque "mejor" a seguir, cada enfoque tiene sus ventajas y desventajas. Es necesario analizar los pros / contras de cada enfoque y decidir cuál es mejor para la aplicación:

Acercarse	Pros	Contras
SINGLE_TABLE	Más fácil de entender el modelo de mesa. Solo se requiere una mesa.	No puede tener campos "no nulos". Imagine que SmallDog tiene el atributo hairColor no nulo en la base de datos. Cuando HugeDog persiste, aparece un mensaje de error del Se recibirá la base de datos, informará que hairColor no puede ser nulo.
	Los atributos de las entidades se pueden encontrar todos en una tabla.	
	Por regla general tiene un buen desempeño.	
UNIDO	Cada entidad tendrá su propia tabla de base de datos para almacenar los datos.	El inserto tiene un costo de rendimiento más alto. Se realizará una inserción en cada tabla de la base de datos utilizada en la jerarquía. Imagine una jerarquía como C extiende B extiende A, cuando C persiste se ejecutarán 3 inserciones, una para cada entidad mapeada.
	Seguirá los patrones OO aplicados en el código de la aplicación.	El número de combinación ejecutado en las consultas de la base de datos aumentará en mayor profundidad jerarquías.
TABLE_PER_CLASS	Cuando la consulta se ejecuta para traer solo una entidad, el rendimiento es mejor. Una tabla de base de datos tendrá solo un dato de entidad.	Las columnas se repetirán. Los atributos que se encuentran en las entidades abstractas se repetirán en las entidades secundarias concretas.
		Cuando una consulta trae más de una entidad del jerarquía esta consulta tendrá un costo mayor. Se utilizará UNION o una consulta por tabla.

Objetos incrustados

Objetos incrustados es una forma de organizar entidades que tienen diferentes datos en la misma tabla. Imagine una tabla de base de datos que mantiene información de la "persona" (por ejemplo, nombre, edad) y datos de dirección (nombre de la calle, número de casa, ciudad, etc.).

Mira la siguiente imagen:

id [PK] integer	name character varying(255)	age integer	house_address character varying(255)	house_number integer	house_color character varying(255)
1	John	22	Street A	22	Red
2	Mary	33	Street B	33	Black
3	Joseph	44	Street C	44	Green

Es posible ver datos relacionados con una persona y su correspondiente dirección también. Consulte el código a continuación para ver un ejemplo de cómo implementar el concepto de objetos incrustados con las entidades Persona y Dirección:

```
importar javax.persistence. *;

@Embeddable
clase pública Dirección {
	@Columna (nombre = "dirección de la casa" )
	privado Dirección de cadena;

	@Columna (nombre = "house_color" )
	privado Color de la cuerda;

	@Columna (nombre = "número de casa" )
	int privado número;

	público String getAddress () {
		regreso dirección;
	}

	vacío público setAddress (dirección de cadena) {
		esta . dirección = dirección;
	}

	// obtener y configurar
}
```

```
importar javax.persistence. *;

@Entidad
@Mesa (nombre = "persona" )
clase pública Persona {

	@Identificación
	int privado identificación;
	privado Nombre de cadena;
	int privado edad;
```

```
        @Incrustado
        privado Dirección Dirección;

        público Dirección getAddress () {
                regreso dirección;
        }

        vacío público setAddress (Dirección dirección) {
                esta . dirección = dirección;
        }

        // obtener y configurar
}
```

Acerca del código anterior:

- La anotación @Embeddable (clase de dirección) permite que la clase se use dentro de una entidad, tenga en cuenta que la dirección no es una entidad. Es solo una clase para ayudar a organizar los datos de la base de datos.
- Se utiliza una anotación @Column en la clase Address para indicar el nombre de la columna de la base de datos de la tabla.

- La anotación @Embedded (entidad Persona) indica que JPA mapeará todos los campos que están dentro de la clase Dirección como pertenecientes a la entidad Persona.
- La clase de dirección se puede utilizar en otras entidades. Hay formas de anular la anotación @Column en tiempo de ejecución.

ElementCollection - cómo mapear una lista de valores en una clase

A veces es necesario asignar una lista de valores a una entidad, pero esos valores no son entidades en sí mismas, por ejemplo, una persona tiene correos electrónicos, un perro tiene apodos, etc.

Verifique el código a continuación que demuestra esta situación:

```
importar java.util.List;
importar java.util.Set;

importar javax.persistence. *;

@Entidad
@Mesa (nombre = "persona" )
clase pública Persona {

        @Identificación
        @GeneratedValue
        int privado identificación;
```

```
        privado Nombre de cadena;

        @ElementCollection
        @CollectionTable (nombre = "person_has_emails" )
        privado Establecer correos electrónicos de <String>;

        @ElementCollection (targetClass = CarBrands. clase )
        @Enumerado (EnumType.STRING)
        privado Enumere las marcas <CarBrands>;

        // obtener y configurar
}
```

```
enumeración pública CarBrands {
        FORD, FIAT, SUZUKI
}
```

Acerca del código anterior:

- Observe que se utilizan dos listas de datos: Set <String>, List <CarBrand>. La anotación @ElementCollection no se usa con entidades sino con atributos "simples" (por ejemplo, String, Enum, etc.).
- La anotación @ElementCollection se utiliza para permitir que un atributo se repita varias veces.
- La anotación @Enumerated (EnumType.STRING) se usa con la anotación @ElementCollection. Define cómo se conservará la enumeración en la base de datos, como String o como Ordinal (haga clic aquí para obtener más información).

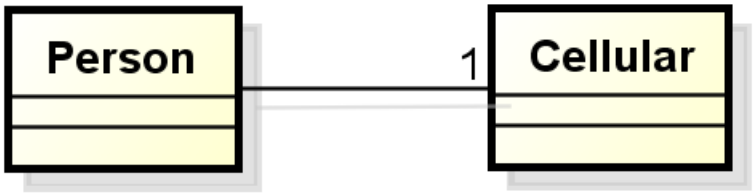
- @CollectionTable (name = “person_has_emails”) => configura JPA en qué tabla se almacenará la información. Cuando esta anotación no está presente, JPA creará una tabla de base de datos con el nombre predeterminado de la clase y el atributo. Por ejemplo, con el atributo “List <CarBrand> marcas”, la tabla de la base de datos se denominaría "person_brands".

OneToOne unidireccional y bidireccional

Es muy fácil encontrar entidades con relaciones. Una persona tiene perros, los perros tienen pulgas, pulgas tiene ... hum ... no importa.

Unidireccional

Una relación uno a uno es la más fácil de entender. Imagine que una persona tiene solo un celular y solo la persona “verá” el celular, el celular no verá a la persona. Mira la imagen a continuación:



Comprueba cómo será la clase Person:

```
importar javax.persistence. *;

@Entidad
clase pública Persona {

        @Identificación
        @GeneratedValue
        int privado identificación;

        privado Nombre de cadena;

        @Doce y cincuenta y nueve de la noche
        @UnirseColumna (nombre = "cell_id" )
        privado Celular celular;

        // obtener y configurar
}
```

```
importar javax.persistence. *;

@Entidad
clase pública Celular {

        @Identificación
        @GeneratedValue
        int privado identificación;

        int privado número;

        // obtener y configurar
}
```

Acerca del código anterior:

- En una relación unidireccional, sólo un lado de la relación conoce ("ve") al otro. Observe que Person conoce Cellular pero Cellular no conoce Person. Es posible hacer person.getCellular () pero no es posible hacer cell.getPerson ().

- En la entidad Persona es posible utilizar la anotación @OneToOne. Esta anotación le indica a JPA que existe una relación entre las entidades.

Toda relación necesita que una de las entidades sea el "propietario de la relación". Ser el propietario de la relación no es más que tener la clave externa en la tabla de la base de datos. En el código anterior, puede ver que se ha utilizado la anotación @JoinColumn. Esta anotación indica que la clave externa se ubicará en la tabla de la base de datos de personas, lo que hará que la entidad Persona sea propietaria de la relación.

Bidireccional

Para transformar esta relación en una bidireccional solo tenemos que editar la entidad Cellular. Compruebe la clase a continuación:

```
importar javax.persistence. *;

@Entidad
clase pública Celular {

        @Identificación
        @GeneratedValue
        int privado identificación;

        int privado número;

        @Doce y cincuenta y nueve de la noche (mappedBy = "celular" )
        privado Persona persona;

        // obtener y configurar
}
```

Acerca del código anterior:

- La misma anotación @OneToOne para el atributo Persona se usa en la entidad Celular.
- El parámetro " *mappedBy* "Se utilizó en la anotación @OneToOnea. Este parámetro indica que la entidad Persona es la propietaria de la relación; la clave externa debe existir dentro de la tabla de personas y no en la tabla celular.

Un desarrollador debe tener en cuenta que para que JPA funcione de manera óptima es una buena práctica dejar un lado de la relación como propietario. Si la anotación @OneToOne que se encuentra en la entidad Cellular no tiene el parámetro "mappedBy", JPA también manejaría la entidad Cellular como propietaria de la relación. No es una buena idea dejar ninguno de los lados de una relación sin definir "mappedBy", o ambos con "mappedBy" configurado.

No existe tal "relación automática"

Para que una relación bidireccional funcione correctamente es necesario hacer lo siguiente:

```
person.setCellular (celular);
cell.setPerson (persona);
```

JPA usa el concepto Java de referencia de clase, una clase debe mantener una referencia a otra si habrá una unión entre ellas. JPA no creará una relación automáticamente; para tener la relación en ambos lados es necesario hacer lo anterior.

OneToMany / ManyToOne unidireccional y bidireccional

La relación de uno a muchos se utiliza cuando una entidad tiene una relación con una lista de otras entidades, por ejemplo, un celular puede tener varias llamadas, pero una llamada solo puede tener un celular. La relación OneToMany se representa con una lista de valores, hay más de una entidad asociada a ella.

Comencemos con el lado ManyToOne:

```
importar javax.persistence. *;

@Entidad
clase pública Llamada {

        @Identificación
        @GeneratedValue
        int privado identificación;

        @ManyToOne
        @UnirseColumna (nombre = "cell_id" )
        privado Celular celular;

        privado largo duración;

        // obtener y configurar
}
```

Acerca del código anterior:

- Se utiliza la anotación @ManyToOne.
- Observe que la anotación @JoinColumn se usa para definir quién es el propietario de la relación.
- El lado @ManyToOne siempre será el dueño de la relación. No hay forma de usar el atributo mappedBy dentro de la anotación @ManyToOne.

Para hacer una relación bidireccional necesitamos editar la entidad Celular (creada en las secciones anteriores). Verifique el código a continuación:

```
importar javax.persistence. *;
```

```
@Entidad
clase pública Celular {

	@Identificación
	@GeneratedValue
	int privado identificación;

	@Doce y cincuenta y nueve de la noche (mappedBy = "celular" )
	privado Persona persona;

	@Uno a muchos (mappedBy = "celular" )
	privado Lista de llamadas de <Call>;

	int privado número;

	// obtener y configurar
}
```

Acerca del código anterior:

- Se utiliza la anotación @OneToMany. Esta anotación debe colocarse en una colección.
- La directiva mappedBy se utiliza para definir la entidad Call como el propietario de la relación.

Toda relación necesita que una de las entidades sea el "propietario de la relación". Ser el propietario de la relación no es más que tener la clave externa en la tabla de la base de datos. En el código anterior, puede ver que se ha utilizado la anotación @JoinColumn. Esta anotación indica que la clave externa se ubicará en la tabla de la base de datos de llamadas, lo que hará que la entidad de llamada sea la propietaria de la relación.

No existe tal "relación automática"

Para que una relación bidireccional funcione correctamente es necesario hacer lo siguiente:

```
call.setCellular (celular);
Cellular.setCalls (llamadas);
```

JPA usa el concepto Java de referencia de clase, una clase debe mantener una referencia a otra si habrá una unión entre ellas. JPA no creará una relación automáticamente; para tener la relación en ambos lados es necesario hacer lo anterior.

ManyToMany unidireccional y bidireccional

En un ejemplo de ManyToMany, una persona puede tener varios perros y un perro puede tener varias personas (imagina un perro que vive en una casa con 15 personas).

En un enfoque ManyToMany es necesario usar una tabla de base de datos adicional para almacenar los identificadores que relacionan la base de datos

tablas de cada entidad de la relación. Por lo tanto, para el ejemplo específico tendremos una tabla de personas, una tabla de perros y una tabla de relaciones llamada person_dog. La tabla person_dog solo mantendría los valores person_id y dog_id que representan qué perro pertenece a qué persona.

Vea las imágenes de la tabla de la base de datos a continuación:

Tabla de personas

id [PK] integer	name character v	cellular_id integer
1	Mary	2

Mesa para perros

id [PK] integer	name character v
4	Spike
5	Snow

mesa person_dog

person_id [PK] integer	dog_id [PK] integer
1	4
1	5

Observe que person_dog solo tiene identificadores.

Marque la entidad Persona a continuación:

```
diablillo ort java.util.List;

importar javax.persistence. *;

@Entidad
clase pública Persona {

    @Identificación
    @GeneratedValue
    int privado identificación;
```

```
        privado Nombre de cadena;

        @Muchos a muchos
        @JoinTable (nombre = "persona_perro" , joinColumns = @UnirseColumna (nombre = "person_id" ), inverseJoinColumns =
@UnirseColumna (nombre = "dog_id" ))
        privado Lista de perros <Perro>;

        @Doce y cincuenta y nueve de la noche
        @UnirseColumna (nombre = "cell_id" )
        privado Celular celular;

        // obtener y configurar
}
```

Acerca del código anterior:

- Se utiliza la anotación @ManyToMany.
- La anotación @JoinTable se utiliza para establecer la tabla de relaciones entre las entidades; "Nombre" establece el nombre de la tabla; "JoinColumn" define el nombre de la columna en la tabla del propietario de la relación; "InverseJoinColumns" define el nombre de la columna en la tabla del propietario sin relación.

La entidad Persona tiene una relación unidireccional con la entidad Perro. Compruebe cómo se vería la entidad Perro en una relación bidireccional:

```
importar java.util.List;

importar javax.persistence. *;

@Entidad
clase pública Perro {

        @Identificación
        @GeneratedValue
        int privado identificación;

        privado Nombre de cadena;

        @Muchos a muchos (mappedBy = "perros" )
        privado Enumere <Persona> personas;

        // obtener y configurar
}
```

Como puede ver, la anotación @ManyToMany está configurada con la opción "mappedBy", que establece la entidad Person como propietaria de la relación.

Toda relación necesita que una de las entidades sea el "propietario de la relación". Para la asociación ManyToMany, la entidad propietaria de la relación es la que está dictada por la opción "mappedBy", normalmente configurada con la

@ManyToMany anotación en la entidad no propietaria de la relación. Si la opción "mappedBy" no se encuentra en ninguna de las entidades relacionadas, JPA definirá a ambas entidades como propietarios de la relación. La opción "mappedBy" debe apuntar al nombre del atributo de la entidad asociada y no al nombre de la entidad asociada.

No existe tal "relación automática"

Para que una relación bidireccional funcione correctamente es necesario hacer lo siguiente:

```
person.setDog (perros);
dog.setPersons (personas);
```

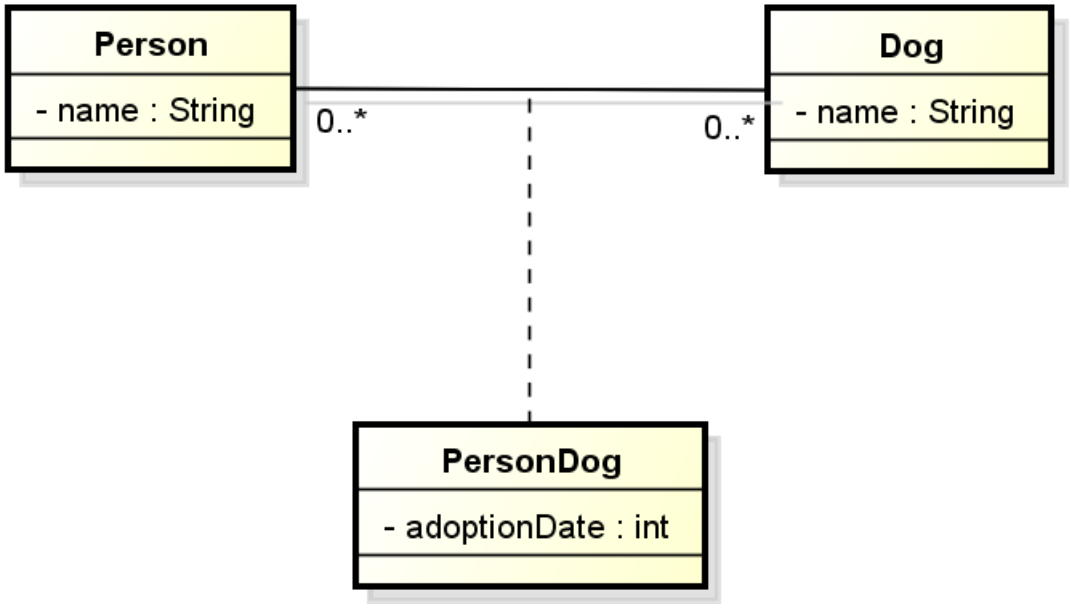
JPA usa el concepto Java de referencia de clase, una clase debe mantener una referencia a otra si habrá una unión entre ellas. JPA no creará una relación automáticamente; para tener la relación en ambos lados es necesario hacer lo anterior.

ManyToMany con campos adicionales

Imagine una entidad Person que tiene una relación ManyToMany con la entidad Dog; Cada vez que un perro es adoptado por una persona, la aplicación debe registrar la fecha de adopción. Este valor debe almacenarse en la relación y no un atributo de la persona ni del perro.

Para manejar este tipo de situación usamos el enfoque de "Clase asociativa" alias "Entidad asociativa". Con este enfoque, es posible almacenar datos adicionales cuando se crea la relación ManyToMany.

La siguiente imagen muestra cómo se puede mapear esta entidad:



Para mapear este campo adicional, impleméntelo como el siguiente código:

```
importar java.util.List;

importar javax.persistence. *;

@Entidad
clase pública Persona {

        @Identificación
        @GeneratedValue
        int privado identificación;

        privado Nombre de cadena;

        @Uno a muchos (mappedBy = "persona" )
        privado Lista de perros <PersonDog>;

        // obtener y configurar
}
```

```
importar java.util.List;

importar javax.persistence. *;

@Entidad
clase pública Perro {

        @Identificación
        @GeneratedValue
        int privado identificación;
```

```
        privado Nombre de cadena;

        @Uno a muchos (mappedBy = "perro" )
        privado Enumerar <PersonDog> personas;

        // obtener y configurar
}
```

El código anterior usa la relación @OneToMany con la opción mappedBy. Tenga en cuenta que no existe una relación @ManyToMany entre las entidades, pero hay una entidad PersonDog que une a ambas entidades.

A continuación se muestra el código PersonDog:

```
importar java.util.Date;

importar javax.persistence. *;

@Entidad
@IdClass (PersonDogId. clase )
clase pública PersonDog {

        @Identificación
        @ManyToOne
        @UnirseColumna (nombre = "person_id" )
        privado Persona persona;

        @Identificación
        @ManyToOne
        @UnirseColumna (nombre = "dog_id" )
        privado Perro perro;

        @Temporal (TemporalType.DATE)
        privado Fecha de adopción Fecha;

        // obtener y configurar
}
```

En el código anterior, puede ver las relaciones entre PersonDog, Dog y Person, y un atributo adicional para almacenar la fecha de adopción. Hay una clase de identificación para almacenar los identificadores de relación denominada "PersonDogId":

```
importar java.io.Serializable;

clase pública PersonDogId implementos Serializable {

        privado estático final largo serialVersionUID = 1L;

        int privado persona;
        int privado perro;

        público int getPerson () {
                regreso persona;
        }
```

```
    vacío público setPerson ( En t persona) {
        esta . person = persona;
    }

    público int getDog () {
        regreso perro;
    }

    vacío público setDog ( En t perro) {
        esta . perro = perro;
    }

    @Anular
    público int código hash() {
        regreso persona + perro; }


    @Anular
    booleano público es igual a (Objeto obj) {
        si ( obj en vez de PersonDogId) {
            PersonDogId personDogId = (PersonDogId) obj;
            regreso personDogId.dog == perro && personDogId.person == persona;
        }

        falso retorno ;
    }
}
```

Una cosa a tener en cuenta aquí es que los atributos de persona y perro deben tener el mismo nombre: "persona" y "perro" aquí entre las entidades PersonDogId y PersonDog. Así es como funciona JPA. Puede encontrar más información sobre claves complejas en secciones anteriores de este documento.

¿Cómo funciona la funcionalidad Cascade? ¿Cómo debería un desarrollador utilizar OrphanRemoval? Manejo de org.hibernate.TransientObjectException

Es muy común que dos o más entidades reciban actualizaciones en la misma transacción. Al editar los datos de una persona, por ejemplo, podríamos cambiar su nombre, dirección, edad, color del coche, etc. Estos cambios deberían activar actualizaciones en tres entidades diferentes: Persona, Coche y Dirección.

Las actualizaciones anteriores se pueden realizar como se muestra en el fragmento de código a continuación:

```
car.setColor (Color.RED);
car.setOwner (nuevaPersona);
car.setSoundSystem (newSound);
```

Si se ejecutara el siguiente código, se lanzaría la excepción org.hibernate.TransientObjectException:

```
EntityManager entityManager = // obtener un administrador de entidad válido

Coche coche = nuevo Auto(); car.setName ( "Trueno
Negro" );

Dirección dirección = nuevo Dirección();
address.setName ( "Calle un" );

entityManager.getTransaction (). begin ();

Persona person = entityManager.find (Person. clase , 33 ); person.setCar (coche);

person.setAddress (dirección);

entityManager.getTransaction (). commit ();
entityManager.close ();
```

Con la implementación de EclipseLink JPA, se dispararía el siguiente mensaje: " *Causado por: java.lang.IllegalStateException: durante la sincronización se encontró un nuevo objeto a través de una relación que no estaba marcada como cascada PERSIST ".*

¿Qué significa que una entidad es transitoria? ¿O de qué se trata una relación no marcada con persistencia en cascada?

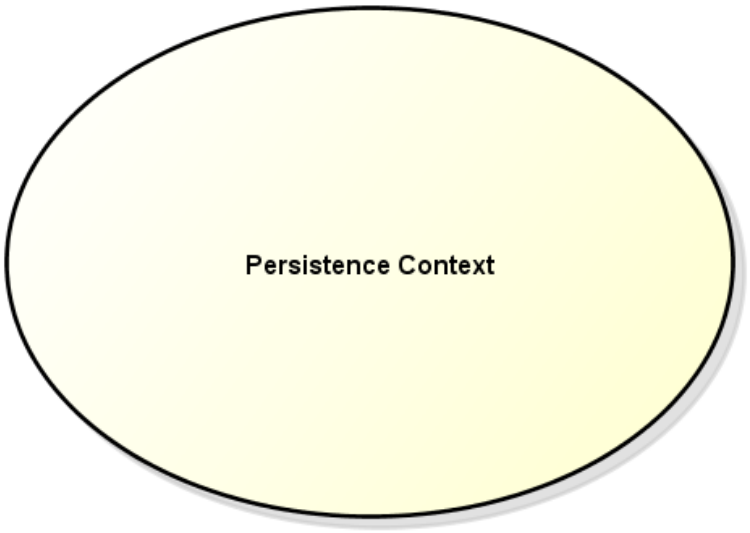
JPA funciona como un rastreador para cada entidad que participa en una transacción. Una entidad que participa en una transacción es una entidad que será creada, actualizada, eliminada. JPA necesita saber de dónde vino esa entidad y hacia dónde se dirige. Cuando se abre una transacción, todas las entidades traídas de la base de datos se “adjuntan”. Con “adjunto” queremos decir que la entidad está dentro de una transacción, siendo monitoreada por JPA. Una entidad permanece adjunta hasta que se cierra la transacción (las reglas para las aplicaciones JSE o las transacciones sin EJB Stateful Session Beans con Persistence Scope Extended son diferentes); para ser adjuntada, una entidad debe provenir de una base de datos (mediante consulta, método de búsqueda del administrador de entidad…), o recibir algún contacto dentro de la transacción (fusionar, actualizar).

Observe el código a continuación:

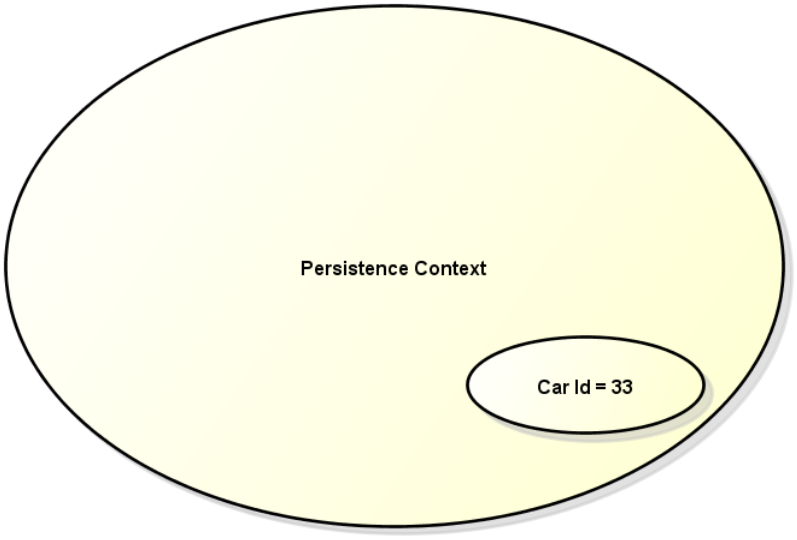
```
entityManager.getTransaction (). begin ();
Coche myCar = entityManager.find (Coche. clase , 33 ); myCar.setColor
(Color.RED);
entityManager. getTransaction (). commit ();
```

Se abre la transacción, se realiza una actualización en la entidad y se compromete la transacción. No era necesario realizar una actualización explícita de la entidad, ya que la transacción que confirma que todas las actualizaciones realizadas a la entidad se mantendrán en la base de datos. Las actualizaciones realizadas en la entidad Car se conservaron en la base de datos porque la entidad está adjunta, cualquier actualización de una entidad adjunta se mantendrá en la base de datos después de que la transacción se confirme () o se realice una llamada de descarga.

Como se muestra en el fragmento de código anterior, la entidad "myCar" se trajo de la base de datos dentro de una transacción, por lo que JPA tendrá la entidad "myCar" adjunta a ese contexto de persistencia. Es posible definir un contexto de persistencia como un lugar donde JPA colocará todas las entidades adjuntas a esa transacción, o como una gran bolsa. Las imágenes a continuación muestran este concepto:



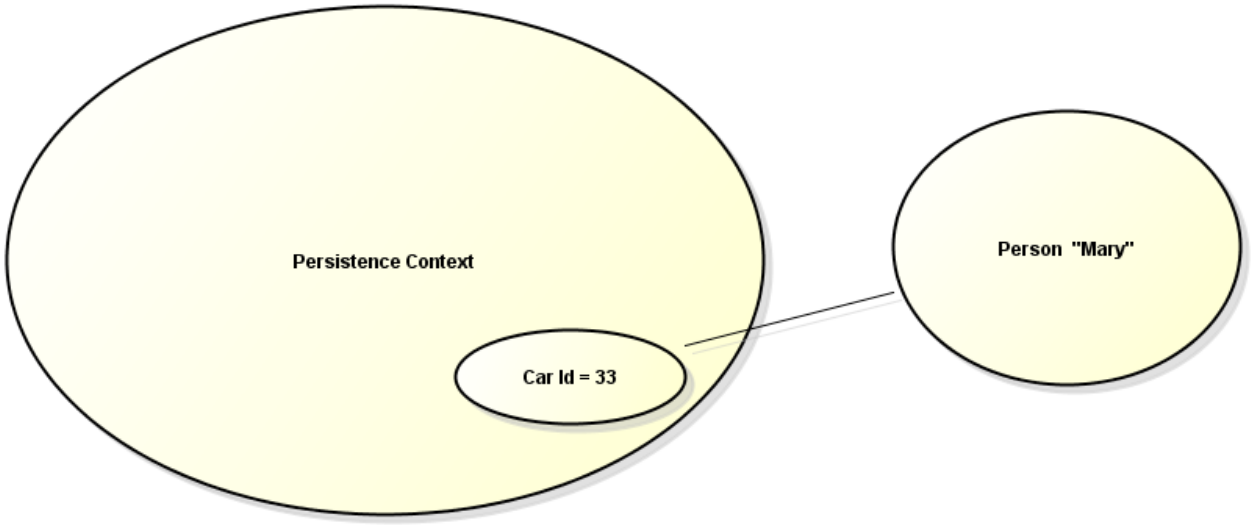
Car myCar = entityManager.find(Car.class, 33);



Observe en la imagen de arriba que la entidad Car se agrega al contexto de persistencia solo después de que se realiza una consulta de base de datos para recuperarla de la base de datos. Cada actualización en la entidad Car será monitoreada por JPA. Una vez finalizada la transacción (o invocado el comando flush), JPA mantendrá estos cambios en la base de datos.

El problema antes mencionado ocurre cuando actualizamos una relación entre dos entidades. Verifique el código y la imagen a continuación:

```
entityManager.getTransaction (). begin ();
Persona nuevaPersona = nuevo Persona();
newPerson.setName ( "María" );
Coche myCar = entityManager.find (Coche. clase , 33 ); myCar.setOwner
(nuevaPersona);
entityManager. getTransaction (). commit ();
```



La entidad Coche establece una relación con la entidad Persona. El problema es que la entidad Persona está fuera del contexto de persistencia, observe que la entidad persona no se trajo de la base de datos ni se adjuntó a la transacción. Tras la confirmación, JPA no puede reconocer que Person es una entidad nueva, que no existe en la base de datos. Incluso si la entidad Person existiera en la base de datos, dado que proviene de fuera de una transacción (por ejemplo, un JSF ManagedBean, Struts Action), se considerará inmanejable en este contexto de persistencia. Un objeto fuera del contexto de persistencia se conoce como "separado".

Esta situación de entidad separada puede ocurrir en cualquier operación de JPA: INSERTAR, ACTUALIZAR, ELIMINAR ...

Para ayudar en estas situaciones, JPA creó la opción Cascade. Esta opción se puede definir en las anotaciones: @OneToOne, @OneToMany y @ManyToMany. La enumeración javax.persistence.CascadeType tiene todas las opciones de Cascade disponibles.

Compruebe las opciones de cascada a continuación:

- CascadeType.DETACH
- CascadeType.MERGE
- CascadeType.PERSIST
- CascadeType.REFRESH
- CascadeType.REMOVE
- CascadeType.ALL

Cascade aplica el comportamiento para repetir la acción definida en la relación. Vea el código a continuación:

```
importar javax.persistence. *;

@Entidad
clase pública Auto {

	@Identificación
	@GeneratedValue
	int privado identificación;

	privado Nombre de cadena;

	@Doce y cincuenta y nueve de la noche (cascada = CascadeType.PERSIST)
	privado Persona persona;

	// obtener y configurar
}
```

En el código de persistencia anterior se define que la relación @OneToOne Person tendrá la acción Cascade.PERSIST ejecutada cada vez que se ejecute el comando entityManager.persist (car); por lo tanto, para cada acción persistente invocada en una entidad Coche, JPA también invocará persistir en la relación Persona.

La ventaja de Cascade es que la propagación de una acción es automática, solo es necesario configurarla en una relación.

Compruebe a continuación las opciones de Cascade:

Escribe	Acción	Desencadenado por
CascadeType.DETACH	Cuando la entidad se elimina del Contexto de persistencia (hará que la entidad se separe) esta acción se reflejará en el relación.	Contexto de persistencia terminado o por comando: entityManager.detach (), entityManager.clear ().
CascadeType.MERGE	Cuando una entidad tiene alguna	Cuando la entidad se actualiza y

	Los datos actualizados de esta acción se verán reflejados en la relación.	la transacción finaliza o por comando: entityManager.merge ().
CascadeType.PERSIST	Cuando una nueva entidad persiste en la base de datos, esta acción será reflejado en el relación.	Cuando finaliza una transacción o por comando: entityManager.persis t ().
CascadeType.REFRESH	Cuando una entidad tiene sus datos sincronizados con la base de datos esta acción se verá reflejada en la relación.	Por comando: entityManager.refresh ().
CascadeType.REMOVE	Cuando se elimina una entidad de la base de datos, esta acción se reflejará en la relación.	Por comando: entityManager.remove ().
CascadeType.ALL	Cuando la JPA o un comando invoca cualquiera de las acciones anteriores, esta acción se reflejará en la relación.	Mediante cualquier comando o acción descrita anteriormente.

Una vez que se define la cascada, el siguiente código debería ejecutarse sin errores:

```
entityManager.getTransaction (). begin ();
Persona nuevaPersona = nuevo Persona();
newPerson.setName ( "María" );
Coche myCar = entityManager.find (Coche. clase , 33 ); myCar.setOwner
(nuevaPersona);
entityManager. getTransaction (). commit ();
```

Observaciones sobre Cascade:

- Un desarrollador debe tener cuidado al usar CascadeType.ALL en una relación. Cuando se elimina la entidad, su relación también se eliminará. En el código de muestra anterior, si el tipo de cascada en la entidad Coche se estableció en la opción TODOS, cuando una entidad Coche se eliminó de la base de datos, la entidad Persona también se eliminaría.

- CascadeType.ALL (o cascadas individuales) pueden causar un bajo rendimiento en cada acción desencadenada en la entidad. Si, por ejemplo, una entidad tiene muchas listas de entidades referenciadas, una acción merge () invocada en la entidad específica podría hacer que todas las listas también se fusionen.
- car.setOwner (personFromDB) => si la entidad "personFromDB" existe en la base de datos pero está separada

para el contexto de persistencia, Cascade no ayudará. Cuando el comando

Se ejecuta entityManager.persist (car) JPA ejecutará el comando persist para cada relación definida con CascadeType.PERSIST (por ejemplo, entityManager.persist (persona)). Si la entidad Person ya existe en la base de datos, JPA intentará insertar el mismo registro nuevamente y se lanzará un mensaje de error. En este caso solo se pueden usar entidades "adjuntas", la mejor manera de hacerlo es usando el getReference () método que presentamos aquí con más detalle .

Para que JPA active la cascada siempre es necesario ejecutar la acción en la entidad que tiene definida la opción cascada. Verifique el código a continuación:

```
importar javax.persistence. *;

@Entidad
clase pública Auto {

        @Identificación
        @GeneratedValue
        int privado identificación;

        privado Nombre de cadena;

        @Doce y cincuenta y nueve de la noche (cascada = CascadeType.PERSIST)
        privado Persona persona;

        // obtener y configurar
}
```

```
importar javax.persistence. *;

@Entidad
clase pública Persona {

        @Identificación
        int privado identificación;

        privado Nombre de cadena;

        @Doce y cincuenta y nueve de la noche (mappedBy = "persona" )
        privado Coche coche;

        // obtener y configurar
}
```

La forma correcta de activar la cascada en las entidades anteriores es:

```
entityManager.persist (coche);
```

JPA buscará dentro de la entidad Car si hay una opción en cascada que deba activarse. Si la persistencia se ejecutó como a continuación, se lanzaría un mensaje de error transitorio:

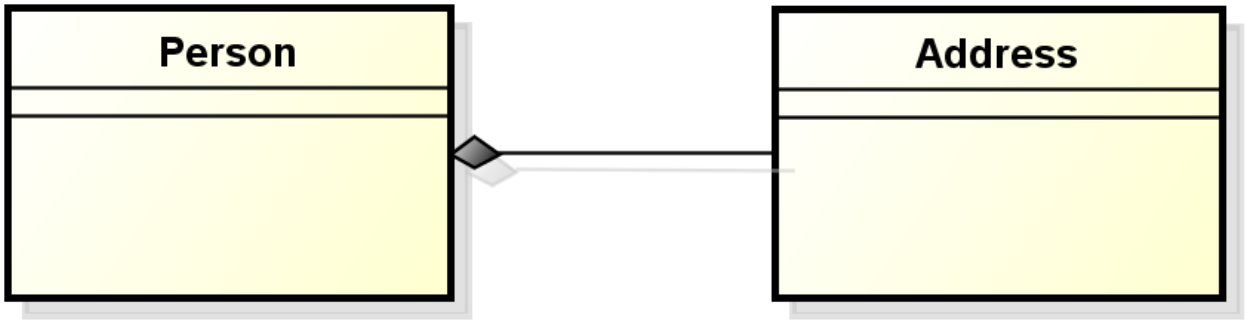
```
entityManager.persist (persona);
```

Recuerde: la cascada solo será disparada por JPA cuando la acción se ejecute en la entidad que tiene la Cascada configurada, en el ejemplo de arriba la clase Car definió la Cascada a Persona; solo la clase Coche activará la cascada.

Huérfano

La opción OrphanRemoval funciona casi como CascadeType.REMOVE. El OrphanRemoval generalmente se aplica en los casos en que una entidad simplemente existe dentro de otra entidad.

Imagine una situación en la que una entidad Dirección solo existirá dentro de una entidad Persona:



Si se conserva una nueva entidad Persona en la base de datos, también se creará una entidad Dirección. Conceptualmente, la entidad Dirección se crea solo cuando se crea una nueva entidad Persona y cuando se elimina la entidad Persona, la entidad Dirección también se elimina, al igual que al usar la opción CascadeType.REMOVE.

OrphanRemoval tiene casi la misma funcionalidad que CascadeType.REMOVE, pero conceptualmente debe aplicarse a nivel de composición de clase.

Mira el código a continuación:

```
importar javax.persistence. *;

@Entidad
clase pública Dirección {
```

```
        @Identificación
        @GeneratedValue
        int privado identificación;

        privado Nombre de cadena;

        // obtener y configurar
}
```

```
importar javax.persistence. *;

@Entidad
clase pública Persona {

        @Identificación
        int privado identificación;

        privado Nombre de cadena;

        @Uno a muchos (orphanRemoval = cierto )
        privado Enumere la dirección <Address>;

        // obtener y configurar
}
```

Imagine que una entidad de Dirección específica está asignada a una entidad de Persona específica y solo esa entidad de Persona tiene acceso a ella.

Como se indicó anteriormente, OrphanRemoval es *casi* como CascadeType.REMOVE. La diferencia es que la entidad dependiente se eliminará de la base de datos si el atributo se establece en nulo como se muestra en el fragmento de código a continuación:

```
person.setAddress ( nulo );
```

Cuando la entidad Persona se actualice en la base de datos, la entidad Dirección se eliminará de la base de datos. La person.setAddress (null) haría que la entidad Address fuera huérfana.

La opción OrphanRemoval solo está disponible en las anotaciones: @OneToOne y @OneToMany.

Cómo eliminar una entidad con relaciones. Aclare qué relaciones están generando la excepción

Cuando se elimina una entidad de la base de datos, puede aparecer un error de restricción, algo como: " *java.sql.SQLIntegrityConstraintViolationException ".* Este error puede ocurrir si intentamos eliminar una entidad Person que está relacionada con una entidad Car y CascadeType.REMOVE no está definido para la asociación específica (ver el último capítulo sobre la definición de Cascade).

Para realizar la eliminación de la entidad Person podríamos hacer:

- CascadeType.REMOVE => Una vez que se elimine la entidad, su entidad secundaria (Coche) también se eliminará. Consulte la sección anterior para ver cómo hacerlo.
- OrphanRemoval => Se puede aplicar pero tiene un comportamiento diferente al de CascadeType.REMOVE. Consulte el último capítulo para ver cómo hacerlo.
- Establezca la relación en nula antes de excluirla:

```
person.setCar ( nulo );
entityManager.remove (persona);
```

Desafortunadamente, localizar la entidad real que es la causa de la excepción específica no es nada sencillo. Lo que se puede hacer es capturar el nombre de la clase que se muestra en el mensaje de error de la excepción y continuar desde allí. Esta solución no es precisa porque el mensaje de error es una cadena con un texto largo que contiene el nombre de la tabla de la base de datos. Sin embargo, este mensaje puede cambiar según cada implementación de JPA, controlador JDBC, idioma de base de datos local, etc.

Creación de una EntityManagerFactory por aplicación

Cuando se controlan las transacciones de la base de datos mediante programación, se suele utilizar un EntityManagerFactory por aplicación. Este es el enfoque óptimo ya que cargar un EntityManagerFactory tiene un alto costo de rendimiento; JPA analizará la base de datos, validará entidades y realizará varias otras tareas al crear una nueva EntityManagerFactory. Por lo tanto, no es viable crear una nueva EntityManagerFactory por transacción.

A continuación se muestra un código que se puede utilizar:

```
importar javax.persistence. *;

clase abstracta pública ConnectionFactory {
	privado ConnectionFactory () {
	}

	estática privada EntityManagerFactory entityManagerFactory;

	público estático EntityManager getEntityManager () {
		si ( entityManagerFactory == nulo ) {
			entityManagerFactory = Persistence.createEntityManagerFactory ( "MyPersistenceUnit" );
		}

		regreso entityManagerFactory.createEntityManager ();
	}
}
```

Comprender cómo funciona la opción Lazy / Eager

Una entidad puede tener un atributo con un tamaño enorme (por ejemplo, un archivo grande) o una gran lista de entidades, por ejemplo, una persona puede tener varios coches o una imagen con 150 MB de tamaño.

Verifique el código a continuación:

```
importar javax.persistence. *;

@Entidad
clase pública Auto {

	@Identificación
	@GeneratedValue
	int privado identificación;

	privado Nombre de cadena;

	@ManyToOne
	privado Persona persona;

	// obtener y configurar
}
```

```
importar java.util.List;

importar javax.persistence. *;

@Entidad
clase pública Persona {

	@Identificación
	int privado identificación;

	privado Nombre de cadena;

	@Uno a muchos (mappedBy = "persona" , buscar = FetchType.LAZY)
	privado Lista de coches <Coche>;

	@Lob
	@Básico (buscar = FetchType.LAZY)
	privado Byte [] hugePicture;

	// obtener y configurar
}
```

Acerca del código anterior:

- Observe que a la colección se le asigna una opción FetchType.LAZY.
- Observe que a la imagen Byte [] hugePicture también se le asigna una opción FetchType.LAZY.

Cuando un campo está marcado con una opción de tipo de búsqueda diferida, se da a entender que JPA no debe cargar el atributo específico "naturalmente". Cuando se utiliza el comando entityManager.find (Person.class, person_id) para recuperar a la persona específica de la base de datos, la lista de "coches" y el campo "hugePicture" no se cargarán. La cantidad de datos devueltos desde la base de datos será menor. La ventaja de este enfoque es que la consulta tiene menos impacto en el rendimiento y el tráfico de datos a través de la red es menor.

Se puede acceder a estos datos cuando se invocan los métodos de obtención relevantes de la entidad Person. Cuando se ejecuta el comando person.getHugePicture (), se lanzará una nueva consulta contra la base de datos que busca esta información.

Cada atributo simple tendrá la opción FetchType.EAGER habilitada por defecto. Para marcar un atributo simple como LAZY obtenido, simplemente agregue la anotación @Basic con la opción Lazy seleccionada como arriba.

La relación entre entidades también tiene un comportamiento predeterminado:

- Las relaciones que terminan con "One" serán buscadas con MUCHO MOMENTO: @OneToOne y @ManyToOne.
- Las relaciones que terminan con “Many” serán buscadas con Pereza: @OneToMany y @ManyToMany.

Para cambiar el comportamiento predeterminado, debe hacer lo que se demostró anteriormente.

Cuando usamos el comportamiento LAZY, puede ocurrir la excepción "Lazy Initialization Exception". Este error ocurre cuando se accede al atributo LAZY sin ninguna conexión abierta. Consulte esta publicación para ver cómo resolver esto.
problema: Cuatro soluciones para LazyInitializationException .

El problema con el tipo de búsqueda EAGER en todas las listas / atributos es que la consulta de la base de datos puede aumentar mucho. Por ejemplo, si cada entidad Person tiene una lista de miles de registros de acciones, ¡imagine el costo de traer cientos o incluso miles de entidades Person de la base de datos! Un desarrollador debe tener mucho cuidado al elegir qué tipo de estrategia de búsqueda se utilizará para cada atributo / relación.

Manejo del error "no se pueden recuperar varias bolsas simultáneamente"

Este error ocurre cuando JPA recupera una entidad de la base de datos y esta entidad tiene más de una lista de campos marcados para buscar con EAGERly.

Verifique el código a continuación:

```
importar java.util.List;
```

```
importar javax.persistence. *;

@Entidad
clase pública Persona {

	@Identificación
	int privado identificación;

	privado Nombre de cadena;

	@Uno a muchos (mappedBy = "persona" , buscar = FetchType.EAGER, cascade = CascadeType.ALL)
	privado Lista de coches <Coche>;

	@Uno a muchos (buscar = FetchType.EAGER)
	privado Lista de perros <Perro>;

	// obtener y configurar
}
```

Cuando se activa una consulta para obtener la entidad Persona anterior, se muestra el siguiente mensaje de error: "javax.persistence.PersistenceException: org.hibernate.HibernateException: no se pueden recuperar simultáneamente varias bolsas", una lista se puede conocer como bolsa.

Este error ocurre porque Hibernate (el farmework de implementación de JPA específico usado para este ejemplo) intenta traer la misma cantidad de resultado para cada lista. Si el SQL generado devuelve 2 líneas para la lista de entidades "perro" y una para la lista de entidades "coche", Hibernate repetirá la consulta "coche" para igualar el resultado. Verifique las imágenes a continuación para comprender qué está sucediendo cuando se ejecuta el comando entityManager.find (Person.class, person_id):

Tabela DOG

id	name	age
1	Red	2
2	Black	3

Tabela CAR

id	name	color
1	Thunder	Green

entityManager.find(Person.class, 33)

person.id	dog.id	dog.name	dog.age	car.id	car.name	car.color
33	1	Red	2	1	Thunder	Green
33	2	Black	3	1	Thunder	Green

El problema surge cuando aparecen resultados repetidos y rompen el resultado correcto de la consulta. Los glóbulos rojos en la imagen de arriba son los resultados repetidos.

Hay cuatro soluciones para esta situación:

- Para usar java.util.Set en lugar de otros tipos de colección => con este sencillo cambio se puede evitar el error.

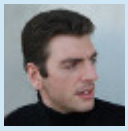
- Usar EclipseLink => es una solución radical, pero solo para los usuarios de JPA que usan solo anotaciones de JPA, este cambio tendrá un impacto mínimo.
- Para usar FetchType.LAZY en lugar de EAGER => esta solución es una solución temporal, porque si una consulta obtiene datos de dos colecciones, este error puede ocurrir nuevamente. Por ejemplo, la siguiente consulta podría desencadenar el error: "seleccione p de Persona p unirse a buscar p.dogs d unirse a buscar p.cars c". Además, cuando se utiliza este enfoque, LazyInitializationException puede ocurrir un error.

- Para usar @LazyCollection o @IndexColumn de la implementación de Hibernate en la colección => es una muy buena idea entender cómo funciona @IndexColumn y sus efectos cuando se usa, su comportamiento cambiará dependiendo de qué lado de la relación se agregue (la explicación de esta anotación está fuera del alcance de este mini libro).

SOBRE EL AUTOR

Hebert Coelho es un desarrollador senior de Java, con 4 certificaciones y un libro publicado sobre JSF (solo en portugués). Fundador del blog uaiHebert.com visitado desde más de 170 países diferentes.

ACERCA DEL EDITOR



Byron Kiourtzoglou es un ingeniero de software maestro que trabaja en los dominios de TI y telecomunicaciones. Es desarrollador de aplicaciones en una amplia variedad de aplicaciones / servicios. Actualmente se desempeña como líder del equipo y arquitecto técnico para una plataforma de integración y creación de servicios patentada para las industrias de TI y telecomunicaciones, además de una solución interna de análisis de big data en tiempo real. Siempre le fascinan SOA, los servicios de middleware y el desarrollo móvil. Byron es cofundador y editor ejecutivo de Java Code Geeks.