EBook Gratis

APRENDIZAJE

jpa

Free unaffiliated eBook created from
Stack Overflow contributors.

#jpa

Tabla de contenido

Acerca de

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

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

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

Capítulo 1: Empezando con jpa

Observaciones

JPA es la API de persistencia de Java, una especificación que maneja la asignación de objetos Java y sus relaciones con una base de datos relacional. Esto se llama un mapeador objeto-relacional (ORM). Es una alternativa para (o complementar) el JDBC de más bajo nivel. Es más útil cuando se persigue un enfoque orientado a Java y cuando deben persistir gráficos de objetos complejos.

JPA en sí mismo no es una implementación. Necesitará un proveedor de persistencia para eso (ver ejemplos). Las implementaciones actuales del último estándar JPA 2.1 son EclipseLink (también la implementación de referencia para JPA 2.1, que significa "prueba de que la especificación puede implementarse"); Hibernate , y DataNucleus .

Metadatos

La asignación entre los objetos de Java y las tablas de la base de datos se define a través de **metadatos de persistencia** . El proveedor de JPA utilizará la información de metadatos de persistencia para realizar las operaciones de base de datos correctas. JPA normalmente define los metadatos a través de anotaciones en la clase Java.

Arquitectura de entidad relacional de objetos

La arquitectura de la entidad se compone de:

- entidades
- unidades de persistencia
- contextos de persistencia
- gestor de entidades fábricas
- administradores de entidades

Versiones

Versión	Grupo de expertos	Lanzamiento
1.0	JSR-220	2006-11-06
2.0	JSR-317	2009-12-10
2.1	JSR-338	2013-05-22

Examples

Requisitos de classpath

Eclipselink

Es necesario incluir Eclipselink y JPA API. Ejemplo de dependencias de Maven:

```
<dependencies>
  <dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>eclipselink</artifactId>
    <version>2.6.3</version>
  </dependency>
  <dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>javax.persistence</artifactId>
    <version>2.1.1</version>
  </dependency>
  <!-- ... -->
</dependencies>
```

Hibernar

Hibernate-core es obligatorio. Ejemplo de dependencia de Maven:

```
<dependencies>
  <dependency>
    <!-- requires Java8! -->
    <!-- as of 5.2, hibernate-entitymanager is merged into hibernate-core -->
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.1.Final</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.1-api</artifactId>
    <version>1.0.0</version>
  </dependency>
  <!-- ... -->
</dependencies>
```

DataNucleus

Se requieren datanucleus-core, datanucleus-api-jpa y datanucleus-rdbms (cuando se usan almacenes de datos RDBMS). Ejemplo de dependencia de Maven:

```
<dependencies>
  <dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-core</artifactId>
    <version>5.0.0-release</version>
  </dependency>
```

```
  <dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-api-jpa</artifactId>
    <version>5.0.0-release</version>
  </dependency>
  <dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-rdbms</artifactId>
    <version>5.0.0-release</version>
  </dependency>
  <dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>javax.persistence</artifactId>
    <version>2.1.2</version>
  </dependency>
  <!-- ... -->
</dependencies>
```

Detalles de configuración

JPA requiere el uso de un archivo *persistence.xml* , ubicado bajo `META-INF` desde la raíz de CLASSPATH. Este archivo contiene una definición de las unidades de persistencia disponibles desde las cuales JPA puede operar.

JPA además permite el uso de un archivo de configuración de mapeo *orm.xml* , también ubicado en `META-INF` . Este archivo de asignación se usa para configurar cómo se asignan las clases al almacén de datos, y es una alternativa / complemento al uso de las anotaciones de Java en las propias clases de entidad JPA.

Ejemplo de persistencia mínima.xml

Hibernar (y incrustado H2 DB)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                             http://java.sun.com/xml/ns/persistence/persistence_2_1.xsd"
         version="2.1">

<persistence-unit name="persistenceUnit">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <class>my.application.entities.MyEntity</class>

    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.h2.Driver" />
        <property name="javax.persistence.jdbc.url" value="jdbc:h2:data/myDB.db" />
        <property name="javax.persistence.jdbc.user" value="sa" />

        <!-- DDL change options -->
        <property name="javax.persistence.schema-generation.database.action" value="drop-and-
create"/>
```

```
        <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
        <property name="hibernate.flushMode" value="FLUSH_AUTO" />
    </properties>
</persistence-unit>
</persistence>
```

Eclipselink (y H2 DB incrustada)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                             http://java.sun.com/xml/ns/persistence/persistence_2_1.xsd"
         version="2.1">

<persistence-unit name="persistenceUnit">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

    <class>my.application.entities.MyEntity</class>

    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
        <property name="javax.persistence.jdbc.url" value="jdbc:h2:data/myDB.db"/>
        <property name="javax.persistence.jdbc.user" value="sa"/>

        <!-- Schema generation : drop and create tables -->
        <property name="javax.persistence.schema-generation.database.action" value="drop-and-
create-tables" />
    </properties>
</persistence-unit>

</persistence>
```

DataNucleus (y DB H2 incrustado)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                             http://java.sun.com/xml/ns/persistence/persistence_2_1.xsd"
         version="2.1">

<persistence-unit name="persistenceUnit">
    <provider>org.datanucleus.api.jpa.PersistenceProviderImpl</provider>

    <class>my.application.entities.MyEntity</class>

    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
        <property name="javax.persistence.jdbc.url" value="jdbc:h2:data/myDB.db"/>
        <property name="javax.persistence.jdbc.user" value="sa"/>

        <!-- Schema generation : drop and create tables -->
        <property name="javax.persistence.schema-generation.database.action" value="drop-and-
create-tables" />
    </properties>
```

```
</persistence-unit>

</persistence>
```

Hola Mundo

Veamos todos los componentes básicos para crear un simple Hallo World.

1. Definir qué implementación de JPA 2.1 usaremos
2. Construye la conexión a la base de datos creando la `persistence-unit`
3. Implementa las entidades.
4. Implementa DAO (objeto de acceso a datos) para manipular las entidades.
5. Probar la aplicación

Bibliotecas

Usando maven, necesitamos estas dependencias:

```
<dependencies>

    <!-- JPA is a spec, I'll use the implementation with HIBERNATE -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.2.6.Final</version>
    </dependency>

    <!-- JDBC Driver, use in memory DB -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>1.4.193</version>
    </dependency>

</dependencies>
```

Unidad de Persistencia

En la carpeta de recursos necesitamos crear un archivo llamado `persistence.xml` . La forma más fácil de definirlo es así:

```
<persistence-unit name="hello-jpa-pu" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <properties>
        <!-- ~ = relative to current user home directory -->
        <property name="javax.persistence.jdbc.url" value="jdbc:h2:./test.db"/>
        <property name="javax.persistence.jdbc.user" value=""/>
        <property name="javax.persistence.jdbc.password" value=""/>
        <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
        <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
```

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

        <!-- This create automatically the DDL of the database's table -->
        <property name="hibernate.hbm2ddl.auto" value="create-drop"/>

    </properties>
</persistence-unit>
```

Implementar una entidad

Creo un `Biker` clase:

```
package it.hello.jpa.entities;


import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;
import java.util.List;

@Entity
@Table(name = "BIKER")
public class Biker implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

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

    @Column(unique = true, updatable = false)
    private String battleName;

    private Boolean beard;

    @Temporal(TemporalType.DATE)
    private Date birthday;

    @Temporal(TemporalType.TIME)
    private Date registrationDate;

    @Transient // --> this annotiation make the field transient only for JPA
    private String criminalRecord;

    public Long getId() {
        return this.id;
    }

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

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public String getBattleName() {
        return battleName;
    }

    public void setBattleName(String battleName) {
        this.battleName = battleName;
    }

    public Boolean getBeard() {
        return this.beard;
    }

    public void setBeard(Boolean beard) {
        this.beard = beard;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public Date getRegistrationDate() {
        return registrationDate;
    }

    public void setRegistrationDate(Date registrationDate) {
        this.registrationDate = registrationDate;
    }

    public String getCriminalRecord() {
        return criminalRecord;
    }

    public void setCriminalRecord(String criminalRecord) {
        this.criminalRecord = criminalRecord;
    }
}
```

Implementar un DAO

```
package it.hello.jpa.business;

import it.hello.jpa.entities.Biker;

import javax.persistence.EntityManager;
import java.util.List;

public class MotorcycleRally {

    public Biker saveBiker(Biker biker) {
        EntityManager em = EntityManagerUtil.getEntityManager();
        em.getTransaction().begin();
```

```
        em.persist(biker);
        em.getTransaction().commit();
        return biker;
    }

}
```

`EntityManagerUtil` es un singleton:

```
package it.hello.jpa.utils;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class EntityManagerUtil {

    // USE THE SAME NAME IN persistence.xml!
    public static final String PERSISTENCE_UNIT_NAME = "hello-jpa-pu";

    private static EntityManager entityManager;

    private EntityManagerUtil() {
    }

    public static EntityManager getEntityManager() {
        if (entityManager == null) {
            // the same in persistence.xml
            EntityManagerFactory emFactory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);


            return emFactory.createEntityManager();
        }
        return entityManager;
    }
}
```

Probar la aplicación

paquete it.hello.jpa.test;

clase pública TestJpa {

```
@Test
public void insertBiker() {
    MotorcycleRally crud = new MotorcycleRally();

    Biker biker = new Biker();
    biker.setName("Manuel");
    biker.setBeard(false);

    biker = crud.saveBiker(biker);

    Assert.assertEquals(biker.getId(), Long.valueOf(1L));
}
```

}

La salida será:

> Ejecutando it.hello.jpa.test.TestJpa Hibernate: descarte la tabla BIKER si existe Hibernate: suelte la secuencia si existe hibernate_sequence Hibernate: cree la secuencia hibernate_sequence comience con 1 incremento por 1 Hibernate: cree la tabla BIKER (id bigint not null, battleName varchar (255)), barba booleana, fecha de cumpleaños, nombre del motorista varchar (255), hora de registro de fecha, clave principal (id) Hibernar: modificar tabla BIKER agregar restricción UK_a64ce28nywyk8wqrvfkkuapli unique (battleName), hibernar: insertar en BIKER (battleName, beard, cumpleaños, bikerName, registro , id) valores (?,?,?,?,?,?) mar 01, 2017 11:00:02 PM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation INFO: HHH000204: Processing PersistenceUnitInfo [nombre: hello- Resultados de jpa-pu ...
>
> Las pruebas ejecutan: 1, fallas: 0, errores: 0, omitidos: 0

Lea Empezando con jpa en línea: https://riptutorial.com/es/jpa/topic/2125/empezando-con-jpa

Capítulo 2: Estrategia de herencia de mesa única

Parámetros

Anotación	Propósito
@Herencia	Especifica el tipo de estrategia de herencia utilizada.
@DiscriminatorColumn	Especifica una columna en la base de datos que se utilizará para identificar diferentes entidades en función de cierta ID asignada a cada entidad
@MappedSuperClass	las súper clases asignadas no son persistentes y solo se utilizan para mantener el estado de sus subclases. Generalmente las clases abstractas de java están marcadas con @MapperSuperClass
@DiscriminatorValue	Un valor especificado en la columna definida por @DiscriminatorColumn. Este valor ayuda a identificar el tipo de entidad.

Observaciones

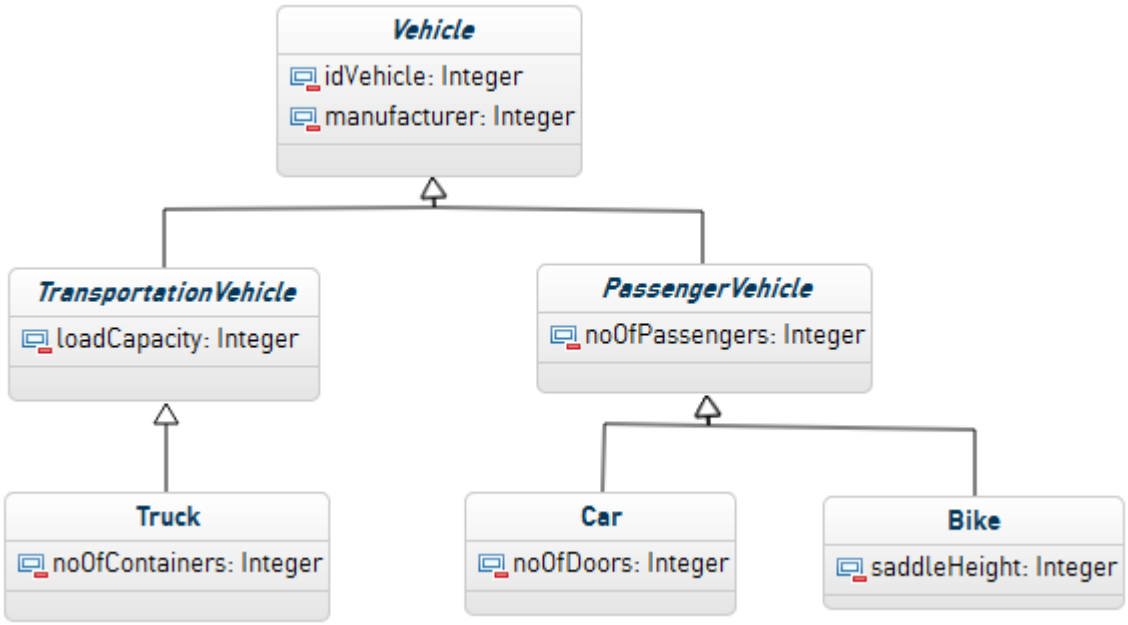
La ventaja de la estrategia de una sola tabla es que no requiere uniones complejas para la recuperación e inserción de entidades, pero, por otro lado, desperdicia espacio en la base de datos, ya que muchas columnas deben ser anulables y no hay datos para ellas.

El ejemplo completo y el artículo se pueden encontrar aquí.

Examples

Estrategia de herencia de una sola mesa.

Se puede tomar un ejemplo simple de jerarquía de vehículos para la estrategia de herencia de una sola tabla.



Clase de vehículo abstracto:

```
package com.thejavageek.jpa.entities;

import javax.persistence.DiscriminatorColumn;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;
import javax.persistence.TableGenerator;

@Entity
@Table(name = "VEHICLE")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "VEHICLE_TYPE")
public abstract class Vehicle {

    @TableGenerator(name = "VEHICLE_GEN", table = "ID_GEN", pkColumnName = "GEN_NAME",
valueColumnName = "GEN_VAL", allocationSize = 1)
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "VEHICLE_GEN")
    private int idVehicle;
    private String manufacturer;

    public int getIdVehicle() {
        return idVehicle;
    }

    public void setIdVehicle(int idVehicle) {
        this.idVehicle = idVehicle;
    }

    public String getManufacturer() {
        return manufacturer;
    }
```

```
    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }

}
```

TransportableVehicle.java package com.thejavageek.jpa.entities;

importar javax.persistence.MappedSuperclass;

```
@MappedSuperclass
public abstract class TransportationVehicle extends Vehicle {

    private int loadCapacity;

    public int getLoadCapacity() {
        return loadCapacity;
    }

    public void setLoadCapacity(int loadCapacity) {
        this.loadCapacity = loadCapacity;
    }

}
```

PasajeroVehicle.java

```
package com.thejavageek.jpa.entities;

import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class PassengerVehicle extends Vehicle {

    private int noOfpassengers;

    public int getNoOfpassengers() {
        return noOfpassengers;
    }

    public void setNoOfpassengers(int noOfpassengers) {
        this.noOfpassengers = noOfpassengers;
    }

}
```

Truck.java

```
package com.thejavageek.jpa.entities;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue(value = "Truck")
public class Truck extends TransportationVehicle{
```

```
    private int noOfContainers;

    public int getNoOfContainers() {
        return noOfContainers;
    }

    public void setNoOfContainers(int noOfContainers) {
        this.noOfContainers = noOfContainers;
    }

}
```

Bike.java

```
package com.thejavageek.jpa.entities;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue(value = "Bike")
public class Bike extends PassengerVehicle {

    private int saddleHeight;

    public int getSaddleHeight() {
        return saddleHeight;
    }

    public void setSaddleHeight(int saddleHeight) {
        this.saddleHeight = saddleHeight;
    }

}
```

Car.java

```
package com.thejavageek.jpa.entities;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue(value = "Car")
public class Car extends PassengerVehicle {

    private int noOfDoors;

    public int getNoOfDoors() {
        return noOfDoors;
    }

    public void setNoOfDoors(int noOfDoors) {
        this.noOfDoors = noOfDoors;
    }

}
```

Código de prueba:

```
/* Create EntityManagerFactory */
EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("AdvancedMapping");

/* Create EntityManager */
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
transaction.begin();

Bike cbr1000rr = new Bike();
cbr1000rr.setManufacturer("honda");
cbr1000rr.setNoOfpassengers(1);
cbr1000rr.setSaddleHeight(30);
em.persist(cbr1000rr);

Car avantador = new Car();
avantador.setManufacturer("lamborghini");
avantador.setNoOfDoors(2);
avantador.setNoOfpassengers(2);
em.persist(avantador);

Truck truck = new Truck();
truck.setLoadCapacity(100);
truck.setManufacturer("mercedes");
truck.setNoOfContainers(2);
em.persist(truck);

transaction.commit();
```

Lea Estrategia de herencia de mesa única en línea:
https://riptutorial.com/es/jpa/topic/6277/estrategia-de-herencia-de-mesa-unica

Capítulo 3: Estrategia de herencia unida

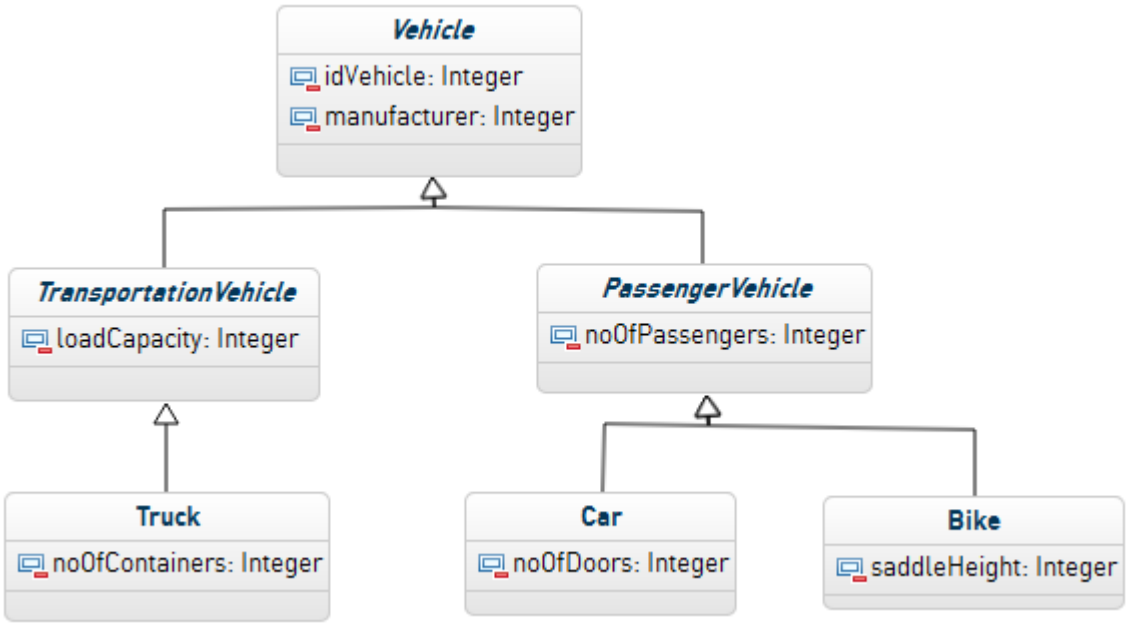
Parámetros

Anotación	Propósito
@Herencia	Especifica el tipo de estrategia de herencia utilizada.
@DiscriminatorColumn	Especifica una columna en la base de datos que se utilizará para identificar diferentes entidades en función de cierta ID asignada a cada entidad
@MappedSuperClass	las súper clases asignadas no son persistentes y solo se utilizan para mantener el estado de sus subclases. Generalmente las clases abstractas de java están marcadas con @MapperSuperClass

Examples

Estrategia de herencia unida.

Un diagrama de clase de muestra basado en el cual veremos la implementación de JPA.




```
@Entity
@Table(name = "VEHICLE")
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "VEHICLE_TYPE")
public abstract class Vehicle {
```

```
    @TableGenerator(name = "VEHICLE_GEN", table = "ID_GEN", pkColumnName = "GEN_NAME",
valueColumnName = "GEN_VAL", allocationSize = 1)
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "VEHICLE_GEN")
    private int idVehicle;
    private String manufacturer;

    // getters and setters
}
```

TransporteVehicle.java

```
@MappedSuperclass
public abstract class TransportationVehicle extends Vehicle {

    private int loadCapacity;

    // getters and setters
}
```

Truck.java

```
@Entity
public class Truck extends TransportationVehicle {

    private int noOfContainers;

    // getters and setters

}
```

PasajeroVehicle.java

```
@MappedSuperclass
public abstract class PassengerVehicle extends Vehicle {

    private int noOfpassengers;

    // getters and setters
}
```

Car.java

```
@Entity
public class Car extends PassengerVehicle {

    private int noOfDoors;

    // getters and setters
}
```

Bike.java

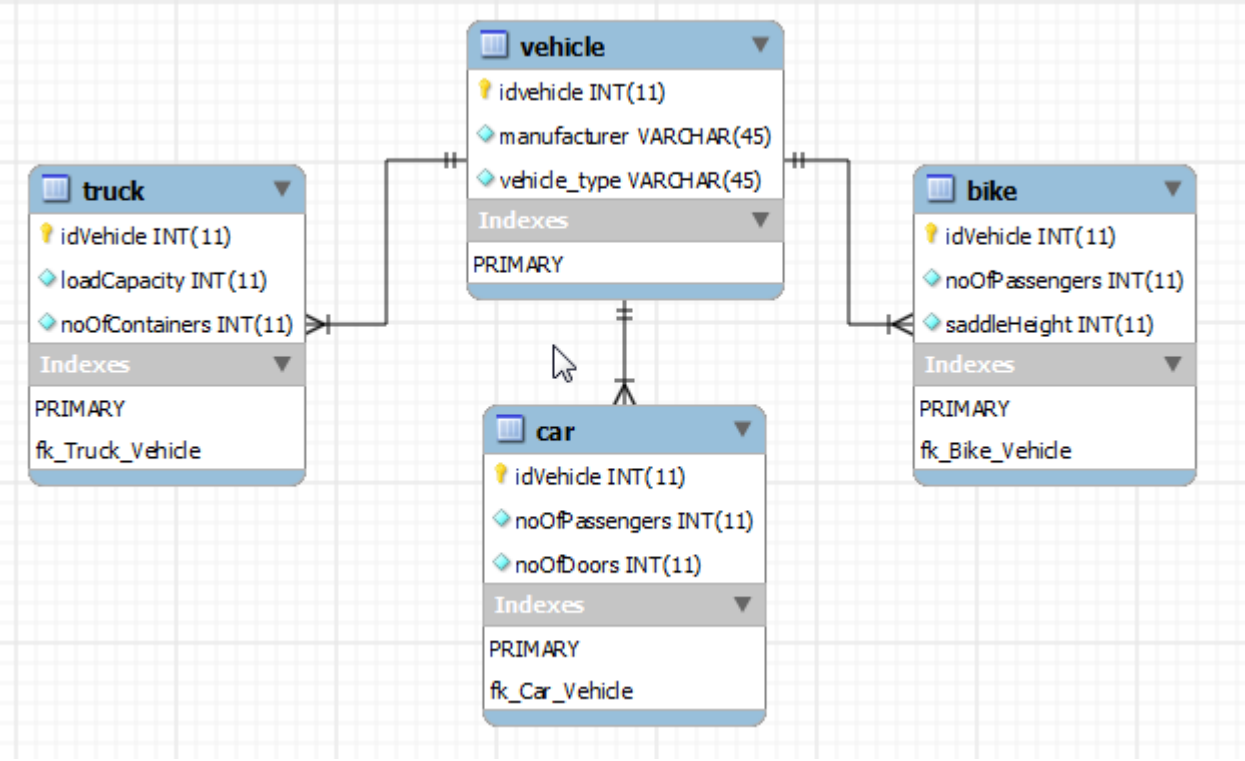
```
@Entity
public class Bike extends PassengerVehicle {
```

```
    private int saddleHeight;

    // getters and setters

}
```

Código de prueba

```
/* Create EntityManagerFactory */
EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("AdvancedMapping");

/* Create EntityManager */
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();

transaction.begin();

Bike cbr1000rr = new Bike();
cbr1000rr.setManufacturer("honda");
cbr1000rr.setNoOfpassengers(1);
cbr1000rr.setSaddleHeight(30);
em.persist(cbr1000rr);

Car aventador = new Car();
aventador.setManufacturer("lamborghini");
aventador.setNoOfDoors(2);
aventador.setNoOfpassengers(2);
em.persist(aventador);

Truck truck = new Truck();
truck.setLoadCapacity(1000);
truck.setManufacturer("volvo");
truck.setNoOfContainers(2);
em.persist(truck);

transaction.commit();
```

El diagrama de la base de datos sería el siguiente.



La ventaja de la estrategia de herencia unida es que no desperdicia espacio de base de datos como en la estrategia de tabla única. Por otro lado, debido a las múltiples uniones involucradas para cada inserción y recuperación, el rendimiento se convierte en un problema cuando las jerarquías de herencia se vuelven amplias y profundas.

El ejemplo completo con la explicación se puede leer aquí

Lea Estrategia de herencia unida en línea: https://riptutorial.com/es/jpa/topic/6473/estrategia-de-herencia-unida

Capítulo 4: Mapeo basico

Parámetros

Anotación	Detalles
`@Id`	Marca el campo / columna como la *clave* de la entidad.
`@Basic`	Marks campo solicitado para mapear como un tipo *básico* . Esto es aplicable a los tipos primitivos y sus envoltorios, `String` , `Date` y `Calendar` . La anotación es en realidad opcional si no se dan parámetros, pero un buen estilo dictaría que sus intenciones sean explícitas.
`@Transient`	Los campos marcados como transitorios no se consideran para la persistencia, al igual que la palabra clave `transient` para la serialización.

Observaciones

Siempre debe haber un constructor predeterminado, es decir, el que no tiene parámetros. En el ejemplo básico, no se especificó ningún constructor, por lo que Java agregó uno; pero si agrega un constructor con argumentos, asegúrese de agregar también el constructor sin parámetros.

Examples

Una entidad muy simple

```
@Entity
class Note {
    @Id
    Integer id;

    @Basic
    String note;

    @Basic
    int count;
}
```

Los buscadores, los colocadores, etc., se omiten por brevedad, pero de todos modos no son necesarios para la APP.

Esta clase de Java se asignaría a la siguiente tabla (dependiendo de su base de datos, aquí en una posible asignación de Postgres):

```
CREATE TABLE Note (
  id integer NOT NULL,
```

```
  note text,
  count integer NOT NULL
)
```

Los proveedores de JPA se pueden usar para generar el DDL, y es probable que produzcan un DDL diferente al que se muestra aquí, pero mientras los tipos sean compatibles, esto no causará problemas en el tiempo de ejecución. Es mejor no confiar en la autogeneración de DDL.

Omitiendo el campo de la cartografía.

```
@Entity
class Note {
    @Id
    Integer id;

    @Basic
    String note;

    @Transient
    String parsedNote;

    String readParsedNote() {
        if (parsedNote == null) { /* initialize from note */ }
        return parsedNote;
    }
}
```

Si su clase necesita campos que no deben escribirse en la base de datos, `@Transient` como `@Transient` . Después de leer de la base de datos, el campo será `null` .

Asignación de hora y fecha

La hora y la fecha vienen en varios tipos diferentes en Java: La `Date` y el `Calendar` ahora históricos, y el `LocalDate` y `LocalDateTime` más recientes. Y `Timestamp` , `Instant` , `ZonedLocalDateTime` y los tipos de tiempo Joda. En el lado de la base de datos, tenemos `time` , `date` y `timestamp` (tanto hora como fecha), posiblemente con o sin zona horaria.

Fecha y hora antes de Java 8

La asignación *predeterminada* para los tipos pre-Java-8 `java.util.Date` , `java.util.Calendar` y `java.sql.Timestamp` es la `timestamp` de `timestamp` en SQL; para `java.sql.Date` es la `date` .

```
@Entity
class Times {
    @Id
    private Integer id;

    @Basic
    private Timestamp timestamp;

    @Basic
```

```
    private java.sql.Date sqldate;

    @Basic
    private java.util.Date utildate;

    @Basic
    private Calendar calendar;
}
```

Esto se asignará perfectamente a la siguiente tabla:

```
CREATE TABLE times (
    id integer not null,
    timestamp timestamp,
    sqldate date,
    utildate timestamp,
    calendar timestamp
)
```

Esta puede no ser la intención. Por ejemplo, a menudo se usa una `Date` o `Calendar` Java para representar la fecha solamente (para la fecha de nacimiento). Para cambiar la asignación predeterminada, o simplemente para hacer explícita la asignación, puede usar la anotación `@Temporal` .

```
@Entity
class Times {
    @Id
    private Integer id;

    @Temporal(TemporalType.TIME)
    private Date date;

    @Temporal(TemporalType.DATE)
    private Calendar calendar;
}
```

La tabla SQL equivalente es:

```
CREATE TABLE times (
    id integer not null,
    date time,
    calendar date
)
```

Nota 1: El tipo especificado con `@Temporal` influye en la generación de DDL; pero también puede tener una columna de tipo `date` map to `Date` con solo la anotación `@Basic` .

Nota 2: El `Calendar` no puede persistir solo el `time` .

Fecha y hora con Java 8

JPA 2.1 *no* define el soporte para los tipos `java.time` proporcionados en Java 8. Sin embargo, la

mayoría de las implementaciones de JPA 2.1 ofrecen soporte para estos tipos, aunque estas son, estrictamente hablando, extensiones de proveedores.

Para DataNucleus, estos tipos simplemente funcionan fuera de la caja y ofrecen una amplia gama de posibilidades de mapeo, junto con la anotación `@Temporal` .

Para Hibernate, si usa Hibernate 5.2+, deberían trabajar fuera de la caja, simplemente usando la anotación `@Basic` . Si usa Hibernate 5.0-5.1, debe agregar la dependencia `org.hibernate:hibernate-java8` . Las asignaciones proporcionadas son

- `LocalDate` hasta la `date`
- `Instant` , `LocalDateTime` y `ZonedDateTime` para `timestamp` de `timestamp`

Una alternativa neutral para el proveedor también sería definir un `AttributeConverter` JPA 2.1 para cualquier tipo de Java 8 `java.time` que deba persistir.

Entidad con id de secuencia gestionada

Aquí tenemos una clase y queremos que el campo de identidad (`userUid`) tenga su valor generado a través de una SECUENCIA en la base de datos. Se supone que esta SECUENCIA se denomina `USER_UID_SEQ` , y puede ser creada por un DBA, o puede ser creada por el proveedor de JPA.

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

    @Id
    @SequenceGenerator(name="USER_UID_GENERATOR", sequenceName="USER_UID_SEQ")
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="USER_UID_GENERATOR")
    private Long userUid;

    @Basic
    private String userName;
}
```

Lea Mapeo basico en línea: https://riptutorial.com/es/jpa/topic/3691/mapeo-basico

Capítulo 5: Mapeo uno a uno

Parámetros

Anotación	Propósito
@TableGenerator	Especifica el nombre del generador y el nombre de la tabla donde se puede encontrar el generador
@GeneratedValue	Especifica la estrategia de generación y se refiere al nombre del generador.
@Doce y cincuenta y nueve de la noche	Especifica la relación uno a uno entre el empleado y el escritorio, aquí el empleado es el propietario de la relación
mappedBy	Este elemento se proporciona en el reverso de la relación. Esto permite la relación bidireccional.

Examples

Relación uno a uno entre empleado y escritorio.

Considere la posibilidad de una relación bidireccional entre el empleado y el escritorio.

Empleado.java

```
@Entity
public class Employee {

    @TableGenerator(name = "employee_gen", table = "id_gen", pkColumnName = "gen_name",
valueColumnName = "gen_val", allocationSize = 100)
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "employee_gen")
    private int idemployee;
    private String firstname;
    private String lastname;
    private String email;

    @OneToOne
    @JoinColumn(name = "iddesk")
    private Desk desk;

    // getters and setters
}
```

Escritorio.java

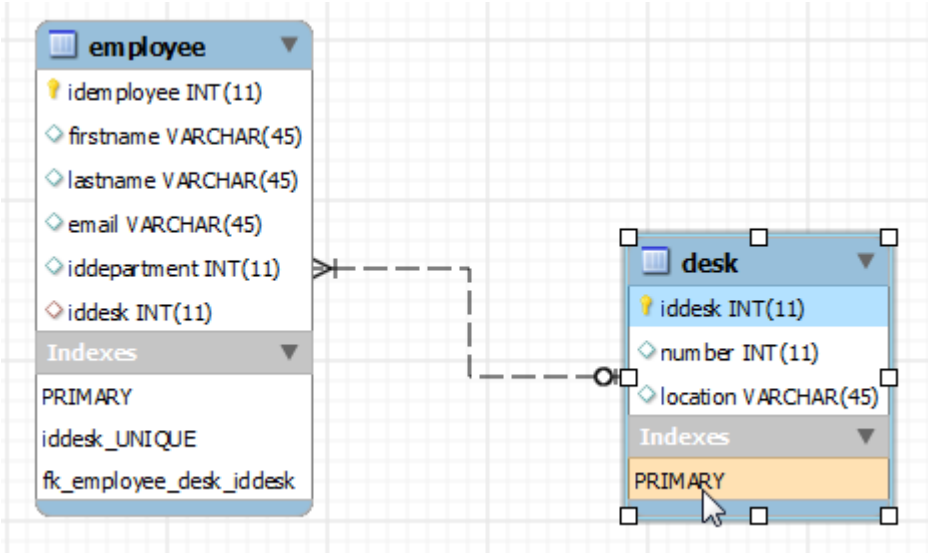
```
@Entity
public class Desk {
```

```
    @TableGenerator(table = "id_gen", name = "desk_gen", pkColumnName = "gen_name",
valueColumnName = "gen_value", allocationSize = 1)
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "desk_gen")
    private int iddesk;
    private int number;
    private String location;
    @OneToOne(mappedBy = "desk")
    private Employee employee;

    // getters and setters
}
```

Código de prueba

```
/* Create EntityManagerFactory */
    EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("JPAExamples");

    /* Create EntityManager */
    EntityManager em = emf.createEntityManager();

    Employee employee;

    employee = new Employee();
    employee.setFirstname("pranil");
    employee.setLastname("gilda");
    employee.setEmail("sdfsdf");

    Desk desk = em.find(Desk.class, 1); // retrieves desk from database
    employee.setDesk(desk);

    em.persist(employee);

    desk = em.find(Desk.class, 1); // retrieves desk from database
    desk.setEmployee(employee);
    System.out.println(desk.getEmployee());
```

El diagrama de la base de datos se muestra a continuación.



- La anotación **@JoinColumn** continúa en el mapeo de la entidad que se asigna a la tabla

que contiene la combinación colulmn.The propietario de la relación. En nuestro caso, la tabla Empleado tiene la columna de unión, por lo que **@JoinColumn** está en el campo Escritorio de la entidad Empleado.

- El elemento **mappedBy** debe especificarse en la asociación **@OneToOne** en la entidad que invierte el lado de la relación. es decir, la entidad que no proporciona una columna de unión en el aspecto de la base de datos. En nuestro caso, Desk es la entidad inversa.

El ejemplo completo se puede encontrar aquí

Lea Mapeo uno a uno en línea: https://riptutorial.com/es/jpa/topic/6474/mapeo-uno-a-uno

Capítulo 6: Muchos a muchos mapeos

Introducción

Un mapeo `ManyToMany` describe una relación entre entidades donde ambas pueden estar relacionadas con más de una instancia de la otra y se define mediante la anotación `@ManyToMany` .

A diferencia de `@OneToMany` donde se puede `@OneToMany` una columna de clave externa en la tabla de la entidad, `ManyToMany` requiere una tabla de unión, que asigna las entidades entre sí.

Parámetros

Anotación	Propósito
`@TableGenerator`	Define un generador de clave principal al que se puede hacer referencia por nombre cuando se especifica un elemento generador para la anotación `GeneratedValue`
`@GeneratedValue`	Proporciona la especificación de estrategias de generación para los valores de las claves primarias. Puede aplicarse a una propiedad o campo de clave principal de una entidad o una superclase asignada junto con la anotación de Id.
`@ManyToMany`	Especifica la relación entre el empleado y las entidades del proyecto, de modo que muchos empleados puedan trabajar en múltiples proyectos.
`mappedBy="projects"`	Define una relación bidireccional entre empleado y proyecto.
`@JoinColumn`	Especifica el nombre de la columna que se referirá a la Entidad para ser considerado como propietario de la asociación
`@JoinTable`	Especifica la tabla en la base de datos que mantendrá a los empleados para proyectar relaciones usando claves externas

Observaciones

- @TableGenerator y @GeneratedValue se utilizan para la creación automática de ID utilizando el generador de tablas jpa.
- La anotación @ManyToMany especifica la relación entre las entidades Empleado y Proyecto.
- @JoinTable especifica el nombre de la tabla para usar como unir la tabla jpa muchos a muchos mapeo entre el empleado y el proyecto usando name = "employee_project". Esto se hace porque no hay manera de determinar la propiedad de un jpa muchos a muchos

mapeos, ya que las tablas de la base de datos no contienen claves externas para hacer referencia a otra tabla.

- @JoinColumn especifica el nombre de la columna que se referirá a la Entidad para que se considere como propietario de la asociación, mientras que @inverseJoinColumn especifica el nombre del lado inverso de la relación. (Puedes elegir cualquier lado para ser considerado como propietario. Solo asegúrate de que esos lados en la relación). En nuestro caso, hemos elegido Empleado como propietario, por lo que @JoinColumn se refiere a la columna idemployee en join table employee_project y @InverseJoinColumn se refiere a idproject que es el lado inverso de jpa muchos a muchos mapeos.
- La anotación @ManyToMany en la entidad del proyecto muestra una relación inversa, por lo tanto, utiliza los proyectos mappedBy = para referirse al campo en la entidad del empleado.

El ejemplo completo puede ser referido aquí

Examples

Empleado para proyectar muchos a muchos mapeo

Entidad empleada.

```
package com.thejavageek.jpa.entities;

import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.TableGenerator;

@Entity
public class Employee {

    @TableGenerator(name = "employee_gen", table = "id_gen", pkColumnName = "gen_name",
valueColumnName = "gen_val", allocationSize = 100)
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "employee_gen")
    private int idemployee;
    private String name;

    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(name = "employee_project", joinColumns = @JoinColumn(name = "idemployee"),
inverseJoinColumns = @JoinColumn(name = "idproject"))
    private List<Project> projects;

    public int getIdemployee() {
        return idemployee;
    }

    public void setIdemployee(int idemployee) {
        this.idemployee = idemployee;
```

```
    }

    public String getName() {
        return name;
    }

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

    public List<Project> getProjects() {
        return projects;
    }

    public void setProjects(List<Project> projects) {
        this.projects = projects;
    }

}
```

Entidad del proyecto:

```
package com.thejavageek.jpa.entities;

import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.TableGenerator;

@Entity
public class Project {

    @TableGenerator(name = "project_gen", allocationSize = 1, pkColumnName = "gen_name",
valueColumnName = "gen_val", table = "id_gen")
    @Id
    @GeneratedValue(generator = "project_gen", strategy = GenerationType.TABLE)
    private int idproject;
    private String name;

    @ManyToMany(mappedBy = "projects", cascade = CascadeType.PERSIST)
    private List<Employee> employees;

    public int getIdproject() {
        return idproject;
    }

    public void setIdproject(int idproject) {
        this.idproject = idproject;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

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

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

}
```

Código de prueba

/ * Crear EntityManagerFactory * / EntityManagerFactory emf = Persistence .createEntityManagerFactory ("JPAExamples");

```
    /* Create EntityManager */
    EntityManager em = emf.createEntityManager();

    EntityTransaction transaction = em.getTransaction();

    transaction.begin();

    Employee prasad = new Employee();
    prasad.setName("prasad kharkar");

    Employee harish = new Employee();
    harish.setName("Harish taware");

    Project ceg = new Project();
    ceg.setName("CEG");

    Project gtt = new Project();
    gtt.setName("GTT");

    List<Project> projects = new ArrayList<Project>();
    projects.add(ceg);
    projects.add(gtt);

    List<Employee> employees = new ArrayList<Employee>();
    employees.add(prasad);
    employees.add(harish);

    ceg.setEmployees(employees);
    gtt.setEmployees(employees);

    prasad.setProjects(projects);
    harish.setProjects(projects);

    em.persist(prasad);

    transaction.commit();
```

Cómo manejar claves compuestas sin anotación incrustable

Si usted tiene

```
Role:
+----------------------------+
| roleId | name | discription |
+----------------------------+


Rights:
+----------------------------+
| rightId | name | discription|
+----------------------------+

rightrole
+------------------+
| roleId | rightId |
+------------------+
```

En el escenario `rightrole` tabla de `rightrole` tiene una clave compuesta y, para acceder a ella, el usuario de JPA debe crear una entidad con anotación `Embeddable` .

Me gusta esto:

Entidad para la mesa de control es:

```
    @Entity
    @Table(name = "rightrole")
    public class RightRole extends BaseEntity<RightRolePK> {

        private static final long serialVersionUID = 1L;

        @EmbeddedId
        protected RightRolePK rightRolePK;


        @JoinColumn(name = "roleID", referencedColumnName = "roleID", insertable = false,
updatable = false)
        @ManyToOne(fetch = FetchType.LAZY)
        private Role role;

        @JoinColumn(name = "rightID", referencedColumnName = "rightID", insertable = false,
updatable = false)
        @ManyToOne(fetch = FetchType.LAZY)
        private Right right;

        ......
     }


    @Embeddable
    public class RightRolePK implements Serializable {
    private static final long serialVersionUID = 1L;

      @Basic(optional = false)
      @NotNull
      @Column(nullable = false)
      private long roleID;

      @Basic(optional = false)
      @NotNull
      @Column(nullable = false)
```

```
     private long rightID;

   .....

}
```

La anotación incorporable está bien para un solo objeto, pero dará un problema al insertar registros masivos.

El problema es que cada vez que el usuario desea crear una nueva `role` con `rights` , el primer usuario debe `store(persist)` objeto de la `role` y luego el usuario debe `flush` para obtener la `id` recién generada para la función. entonces y luego el usuario puede ponerlo en el objeto de la entidad de `rightrole` de la `rightrole`

Para resolver este usuario puede escribir entidad de forma ligeramente diferente.

La entidad para la tabla de roles es:

```
@Entity
@Table(name = "role")
public class Role {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @NotNull
    @Column(nullable = false)
    private Long roleID;


    @OneToMany(cascade = CascadeType.ALL, mappedBy = "role", fetch = FetchType.LAZY)
    private List<RightRole> rightRoleList;

    @ManyToMany(cascade = {CascadeType.PERSIST})
    @JoinTable(name = "rightrole",
            joinColumns = {
                @JoinColumn(name = "roleID", referencedColumnName = "ROLE_ID")},
            inverseJoinColumns = {
                @JoinColumn(name = "rightID", referencedColumnName = "RIGHT_ID")})
    private List<Right> rightList;
.......
}
```

La anotación @JoinTable se encargará de insertar en la tabla de `rightrole` incluso sin una entidad (siempre que esa tabla tenga solo las columnas de ID de rol y derecha).

El usuario puede entonces simplemente:

```
Role role = new  Role();
List<Right> rightList = new ArrayList<>();
Right right1 = new Right();
Right right2 = new Right();
rightList.add(right1);
rightList.add(right2);
```

```
role.setRightList(rightList);
```

Debe escribir @ManyToMany (cascade = {CascadeType.PERSIST}) en inverseJoinColumns, de lo contrario, sus datos principales se eliminarán si se elimina el elemento secundario.

Lea Muchos a muchos mapeos en línea: https://riptutorial.com/es/jpa/topic/6532/muchos-a-muchos-mapeos

Capítulo 7: Muchos a uno mapeo

Parámetros

Columna	Columna
@TableGenerator	Utiliza la estrategia del generador de tablas para la creación automática de id.
@GeneratedValue	Especifica que el valor aplicado a los campos es un valor generado.
@Carné de identidad	Anota el campo como identificador
@ManyToOne	Especifica la relación muchos a uno entre el empleado y el departamento. Esta anotación está marcada en muchos lados. Es decir, múltiples empleados pertenecen a un solo departamento. Así que el Departamento está anotado con @ManyToOne en la entidad Empleado.
@JoinColumn	Especifica la columna de la tabla de la base de datos que almacena la clave externa para la entidad relacionada

Examples

Relación empleado-departamento ManyToOne

Entidad empleada

```
@Entity
public class Employee {

    @TableGenerator(name = "employee_gen", table = "id_gen", pkColumnName = "gen_name",
valueColumnName = "gen_val", allocationSize = 1)
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "employee_gen")
    private int idemployee;
    private String firstname;
    private String lastname;
    private String email;

    @ManyToOne
    @JoinColumn(name = "iddepartment")
    private Department department;

    // getters and setters
    // toString implementation
}
```

Departamento de entidad

```
@Entity
public class Department {

    @Id
    private int iddepartment;
    private String name;

    // getters, setters and toString()
}
```

Clase de prueba

```
public class Test {

    public static void main(String[] args) {

        EntityManagerFactory emf = Persistence
                .createEntityManagerFactory("JPAExamples");
        EntityManager em = emf.createEntityManager();
        EntityTransaction txn = em.getTransaction();

        Employee employee = new Employee();
        employee.setEmail("someMail@gmail.com");
        employee.setFirstname("Prasad");
        employee.setLastname("kharkar");

        txn.begin();
        Department department = em.find(Department.class, 1);//returns the department named
vert
        System.out.println(department);
        txn.commit();

        employee.setDepartment(department);

        txn.begin();
        em.persist(employee);
        txn.commit();

    }

}
```

Lea Muchos a uno mapeo en línea: https://riptutorial.com/es/jpa/topic/6531/muchos-a-uno-mapeo

Capítulo 8: Relación uno a muchos

Parámetros

Anotación	Propósito
@TableGenerator	Especifica el nombre del generador y el nombre de la tabla donde se puede encontrar el generador
@GeneratedValue	Especifica la estrategia de generación y se refiere al nombre del generador.
@ManyToOne	Especifica la relación de muchos a uno entre el empleado y el departamento
@OneToMany (mappedBy = "departamento")	crea una relación bidireccional entre el empleado y el departamento simplemente refiriéndose a la anotación @ManyToOne en la entidad del empleado

Examples

Uno a muchos relación

La asignación de uno a muchos es generalmente una relación bidireccional de la asignación de muchos a uno. Tomaremos el mismo ejemplo que tomamos para la asignación de Muchos a uno.

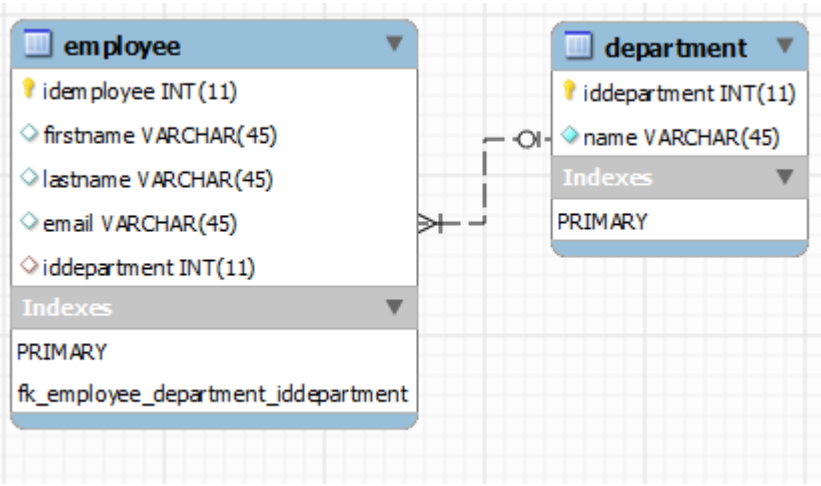
Empleado.java

```
@Entity
public class Employee {

    @TableGenerator(name = "employee_gen", table = "id_gen", pkColumnName = "gen_name",
valueColumnName = "gen_val", allocationSize = 100)
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "employee_gen")
    private int idemployee;
    private String firstname;
    private String lastname;
    private String email;

    @ManyToOne
    @JoinColumn(name = "iddepartment")
    private Department department;

    // getters and setters
}
```

Departamento.java

```
@Entity
public class Department {

    @TableGenerator(table = "id_gen", pkColumnName = "gen_name", valueColumnName = "gen_val",
name = "department_gen", allocationSize = 1)
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "department_gen")
    private int iddepartment;
    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees;

    // getters and setters
}
```

Esta relación se representa en la base de datos de la siguiente manera.



Hay dos puntos para recordar acerca de jpa uno a muchos mapas:

- El muchos a un lado es el lado propietario de la relación. La columna se define en ese lado.
- El mapeo uno a muchos es el lado inverso, por lo que el elemento mappedBy debe usarse en el lado inverso.

El ejemplo completo puede ser referido aquí

Lea Relación uno a muchos en línea: https://riptutorial.com/es/jpa/topic/6529/relacion-uno-a-muchos

Capítulo 9: Relaciones entre entidades

Observaciones

Conceptos básicos de relaciones entre entidades

Una **clave externa** puede ser una o más columnas que hacen referencia a una clave única, generalmente la clave principal, en otra tabla.

Una clave externa y la clave principal primaria a la que hace referencia deben tener el mismo número y tipo de campos.

Las claves externas representan **relaciones** de una columna o columnas en una tabla a una columna o columnas en otra tabla.

Examples

Multiplicidad en las relaciones de la entidad

Multiplicidad en las relaciones de la entidad

Las multiplicidades son de los siguientes tipos:

- **Uno a uno** : cada instancia de entidad está relacionada con una instancia única de otra entidad.
- **De uno a muchos** : una instancia de entidad puede estar relacionada con múltiples instancias de las otras entidades.
- **Muchos a uno** : varias instancias de una entidad pueden relacionarse con una sola instancia de la otra entidad.
- **Muchos a muchos** : las instancias de la entidad pueden relacionarse con múltiples instancias entre sí.

Mapeo uno a uno

El mapeo uno a uno define una asociación de un solo valor con otra entidad que tiene una multiplicidad uno a uno. Esta asignación de relaciones utiliza la anotación `@OneToOne` en la propiedad o campo persistente correspondiente.

Ejemplo: Entidades de `Vehicle` *y* `ParkingPlace` *.*

Mapeo uno a muchos

Una instancia de entidad puede estar relacionada con múltiples instancias de las otras entidades.

Las relaciones de uno a muchos usan la anotación `@OneToMany` en la propiedad o campo persistente correspondiente.

El elemento `mappedBy` es necesario para referirse al atributo anotado por ManyToOne en la entidad correspondiente:

```
 @OneToMany(mappedBy="attribute")
```

Una asociación de uno a muchos necesita mapear la colección de entidades.

Mapeo de muchos a uno

Una asignación muchos a uno se define al anotar el atributo en la entidad de origen (el atributo que se refiere a la entidad objetivo) con la anotación `@ManyToOne` .

Una `@JoinColumn(name="FK_name")` clave de una relación.

Mapeo de muchos a muchos

Las instancias de entidad pueden estar relacionadas con múltiples instancias entre sí.

Las relaciones muchos a muchos usan la anotación `@ManyToMany` en la propiedad o campo persistente correspondiente.

Debemos usar una tercera tabla para asociar los dos tipos de entidad (unir tabla).

Ejemplo de anotación de @JoinTable

Cuando se mapean relaciones de muchos a muchos en JPA, la configuración de la tabla utilizada para la unión de claves foráneas se puede proporcionar mediante la anotación `@JoinTable` :

```
@Entity
public class EntityA {
   @Id
   @Column(name="id")
   private long id;
   [...]
   @ManyToMany
   @JoinTable(name="table_join_A_B",
              joinColumns=@JoinColumn(name="id_A"), referencedColumnName="id"
              inverseJoinColumns=@JoinColumn(name="id_B", referencedColumnName="id"))
   private List<EntityB> entitiesB;
   [...]
}

@Entity
public class EntityB {
   @Id
   @Column(name="id")
   private long id;
   [...]
}
```

En este ejemplo, que consiste en que EntityA tiene una relación de muchos a muchos con EntityB, realizada por el campo `entitiesB` , usamos la anotación @JoinTable para especificar que el nombre de la tabla para la tabla de unión es `table_join_A_B` , compuesto por las columnas `id_A` y `id_B` , claves externas que hacen referencia respectivamente a la columna `id` en la tabla de EntityA y en la tabla de EntityB; `(id_A,id_B)` será una clave primaria compuesta para `table_join_A_B` table.

Lea Relaciones entre entidades en línea: https://riptutorial.com/es/jpa/topic/6305/relaciones-entre-entidades

Capítulo 10: Tabla por estrategia de herencia de clase concreta.

Observaciones

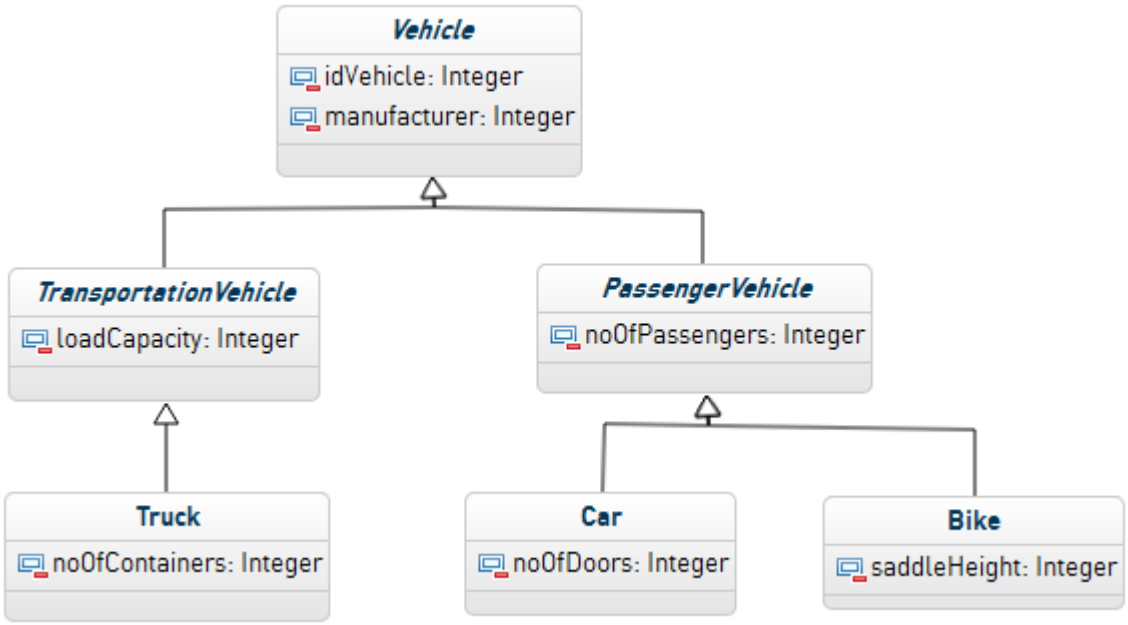
- Vehicle, TransportationVehicle y PassengerVehicle son clases abstractas y no tendrán una tabla separada en la base de datos.
- Camión, Coche y Bicicleta son clases concretas, por lo que se asignarán a las tablas correspondientes. Estas tablas deben incluir todos los campos para las clases anotadas con @MappedSuperClass porque no tienen tablas correspondientes en la base de datos.
- Por lo tanto, la tabla de camiones tendrá columnas para almacenar campos heredados de TransportationVehicle y Vehicle.
- De manera similar, Car and Bike tendrá columnas para almacenar campos heredados de PassengerVehicle y Vehicle.

El ejemplo completo se puede encontrar aquí

Examples

Tabla por estrategia de herencia de clase concreta.

Tomaremos el ejemplo de jerarquía de vehículos como se muestra a continuación.



Vehicle.java

```
package com.thejavageek.jpa.entities;

import javax.persistence.Entity;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.TableGenerator;

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Vehicle {

    @TableGenerator(name = "VEHICLE_GEN", table = "ID_GEN", pkColumnName = "GEN_NAME",
valueColumnName = "GEN_VAL", allocationSize = 1)
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "VEHICLE_GEN")
    private int idVehicle;
    private String manufacturer;

    public int getIdVehicle() {
        return idVehicle;
    }

    public void setIdVehicle(int idVehicle) {
        this.idVehicle = idVehicle;
    }

    public String getManufacturer() {
        return manufacturer;
    }

    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }

}
```

TransporteVehilcle.java

```
package com.thejavageek.jpa.entities;

import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class TransportationVehicle extends Vehicle {

    private int loadCapacity;

    public int getLoadCapacity() {
        return loadCapacity;
    }

    public void setLoadCapacity(int loadCapacity) {
        this.loadCapacity = loadCapacity;
    }

}
```

Truck.java

```
package com.thejavageek.jpa.entities;

import javax.persistence.Entity;

@Entity
public class Truck extends TransportationVehicle {

    private int noOfContainers;

    public int getNoOfContainers() {
        return noOfContainers;
    }

    public void setNoOfContainers(int noOfContainers) {
        this.noOfContainers = noOfContainers;
    }

}
```

PasajeroVehicle.java

```
package com.thejavageek.jpa.entities;

import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class PassengerVehicle extends Vehicle {

    private int noOfpassengers;

    public int getNoOfpassengers() {
        return noOfpassengers;
    }

    public void setNoOfpassengers(int noOfpassengers) {
        this.noOfpassengers = noOfpassengers;
    }

}
```

Car.java

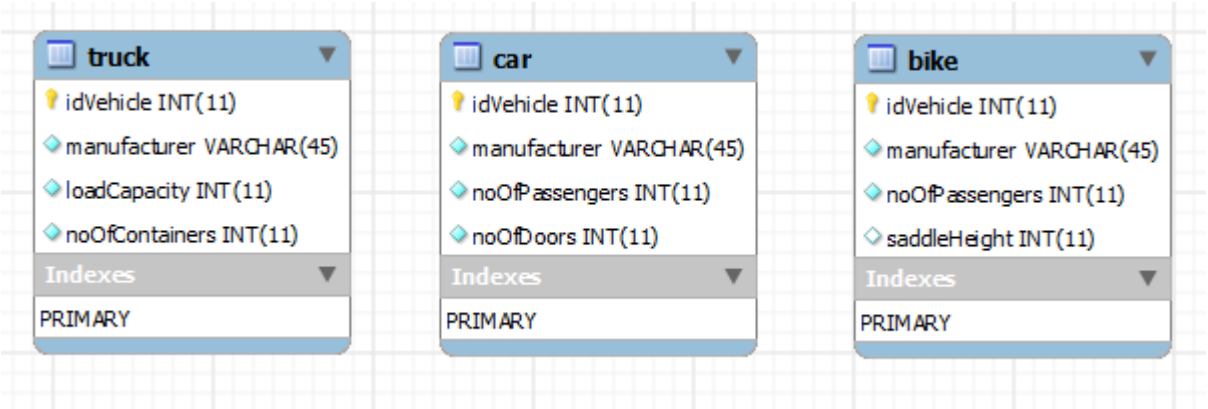
```
package com.thejavageek.jpa.entities;

import javax.persistence.Entity;

@Entity
public class Car extends PassengerVehicle {

    private int noOfDoors;

    public int getNoOfDoors() {
        return noOfDoors;
    }

    public void setNoOfDoors(int noOfDoors) {
        this.noOfDoors = noOfDoors;
    }
```

```
}
```

Bike.java

```
package com.thejavageek.jpa.entities;

import javax.persistence.Entity;

@Entity
public class Bike extends PassengerVehicle {

    private int saddleHeight;

    public int getSaddleHeight() {
        return saddleHeight;
    }

    public void setSaddleHeight(int saddleHeight) {
        this.saddleHeight = saddleHeight;
    }

}
```

La representación de la base de datos será la siguiente



Lea Tabla por estrategia de herencia de clase concreta. en línea: https://riptutorial.com/es/jpa/topic/6255/tabla-por-estrategia-de-herencia-de-clase-concreta-

Creditos

S. No	Capítulos	Contributors
1	Empezando con jpa	Billy Frost, Community, DimaSan, , Manuel Spigolon, Michael Piefel, Neil Stockton, ppeterka
2	Estrategia de herencia de mesa única	Prasad Kharkar
3	Estrategia de herencia unida	bw_üezi, Prasad Kharkar
4	Mapeo basico	Jeffrey Brett Coleman, Michael Piefel, Neil Stockton, Pete
5	Mapeo uno a uno	Prasad Kharkar
6	Muchos a muchos mapeos	Prasad Kharkar, Ronak Patel, Vetle
7	Muchos a uno mapeo	Prasad Kharkar
8	Relación uno a muchos	Prasad Kharkar
9	Relaciones entre entidades	DimaSan,
10	Tabla por estrategia de herencia de clase concreta.	Prasad Kharkar