EBook Gratuito

APPRENDIMENTO

jpa

Free unaffiliated eBook created from
Stack Overflow contributors.

#jpa

Sommario

Di

Capitolo 1: Iniziare con jpa

Osservazioni

JPA è Java Persistence API, una specifica che gestisce la mappatura degli oggetti Java e le loro relazioni con un database relazionale. Questo è chiamato mappatore di oggetti relazionale (ORM). È un'alternativa per (o integrare a) il JDBC di livello più basso. È molto utile quando si persegue un approccio orientato a Java e quando i grafici di oggetti complessi devono essere mantenuti.

L'APP di per sé non è un'implementazione. Avrai bisogno di un fornitore di persistenza per questo (vedi esempi). Le attuali implementazioni dell'ultimo standard JPA 2.1 sono EclipseLink (anche l'implementazione di riferimento per JPA 2.1, che significa "prova che le specifiche possono essere implementate"); Hibernate e DataNucleus .

Metadati

La mappatura tra oggetti Java e tabelle di database viene definita tramite **metadati di persistenza** . Il provider JPA utilizzerà le informazioni sui metadati di persistenza per eseguire le operazioni corrette del database. In genere, JPA definisce i metadati tramite annotazioni nella classe Java.

Object-Relational Entity Architecture

L'architettura dell'entità è composta da:

- entità
- unità di persistenza
- contesti di persistenza
- fabbriche di gestori di entità
- gestori di entità

Versioni

Versione	Gruppo di esperti	pubblicazione
1.0	JSR-220	2006-11-06
2.0	JSR-317	2009-12-10
2.1	JSR-338	2013/05/22

Examples

Requisiti del percorso di classe

EclipseLink

È necessario includere l'API Eclipse e JPA. Esempi di dipendenze 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>
```

ibernare

Hibernate-core è richiesto. Esempio di dipendenza da 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

sono richiesti datanucleus-core, datanucleus-api-jpa e datanucleus-rdbms (quando si utilizzano i datastore RDBMS). Esempio di dipendenza da 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>
```

Dettagli di configurazione

JPA richiede l'uso di un file *persistence.xml* , che si trova sotto `META-INF` dalla radice di CLASSPATH. Questo file contiene una definizione delle unità di persistenza disponibili da cui JPA può operare.

JPA consente inoltre l'uso di un file di configurazione di mappatura *orm.xml* , anch'esso collocato sotto `META-INF` . Questo file di mappatura viene utilizzato per configurare il modo in cui le classi sono mappate all'archivio dati ed è un'alternativa / supplemento all'uso delle annotazioni Java nelle stesse classi di entità JPA.

Minimal persistence.xml esempio

Ibernazione (e DB H2 incorporato)

```
<?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 (e H2 DB incorporato)

```
<?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 (e DB H2 incorporato)

```
<?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>
```

Ciao mondo

Vediamo tutti i componenti di base per creare un semplice Hallo World.

1. Definire quale implementazione di JPA 2.1 useremo
2. Costruisci la connessione al database creando l' `persistence-unit`
3. Implementa le entità
4. Implementa DAO (oggetto di accesso ai dati) per manipolare le entità
5. Testare l'applicazione

biblioteche

Usando Maven, abbiamo bisogno di queste dipendenze:

```
<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>
```

Unità di Persistenza

Nella cartella delle risorse è necessario creare un file chiamato `persistence.xml` . Il modo più semplice per definirlo è come questo:

```
<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>
```

Implementare un'entità

Creo un `Biker` classe:

```
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;
    }
}
```

Implementa 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` è 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;
    }
}
```

Testare l'applicazione

pacchetto it.hello.jpa.test;

classe pubblica 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));
}
```

}

L'output sarà:

> Running it.hello.jpa.test.TestJpa Hibernate: drop table BIKER se esiste Hibernate: drop sequence se esiste hibernate_sequence Hibernate: crea sequenza hibernate_sequence inizia con 1 incrementa di 1 Hibernate: crea table BIKER (id bigint non nullo, battleName varchar (255), barba boolean, data compleanno, bikerName varchar (255), registrationDate time, chiave primaria (id)) Ibernazione: alter table BIKER aggiungi vincolo UK_a64ce28nywyk8wqrvfkkuapli unique (battleName) Hibernate: inserisci in BIKER (battleName, beard, birthday, bikerName, registrationDate , id) valori (?,?,?,?,?,?) mar 01, 2017 11:00:02 PM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation INFO: HHH000204: Elaborazione PersistenceUnitInfo [nome: ciao- jpa-pu ...] Risultati:
>
> Esecuzione test: 1, Errori: 0, Errori: 0, Salto: 0

Leggi Iniziare con jpa online: https://riptutorial.com/it/jpa/topic/2125/iniziare-con-jpa

Capitolo 2: Mappatura di base

Parametri

Annotazione	Dettagli
`@Id`	Segna il campo / colonna come *chiave* dell'entità
`@Basic`	Segna il campo richiesto per mapparlo come tipo *base* . Questo è applicabile ai tipi primitivi e ai loro wrapper, `String` , `Date` e `Calendar` . L'annotazione è effettivamente facoltativa se non vengono forniti parametri, ma un buon stile imporrebbe di rendere esplicite le tue intenzioni.
`@Transient`	I campi contrassegnati come transitori non sono considerati per la persistenza, proprio come la parola chiave `transient` per la serializzazione.

Osservazioni

C'è sempre bisogno di essere un costruttore predefinito, cioè quello senza parametri. Nell'esempio di base, non è stato specificato alcun costruttore, quindi Java ne ha aggiunto uno; ma se aggiungi un costruttore con argomenti, assicurati di aggiungere anche il costruttore senza parametri.

Examples

Un'entità molto semplice

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

    @Basic
    String note;

    @Basic
    int count;
}
```

Getters, setter, ecc. Sono ammessi per brevità, ma non sono comunque necessari per JPA.

Questa classe Java si assocerebbe alla seguente tabella (a seconda del database, qui indicato in una possibile mappatura di Postgres):

```
CREATE TABLE Note (
  id integer NOT NULL,
  note text,
  count integer NOT NULL
```

```
)
```

I provider JPA possono essere utilizzati per generare il DDL e probabilmente genereranno DDL diverso da quello mostrato qui, ma finché i tipi sono compatibili, ciò non causerà problemi in fase di runtime. È meglio non fare affidamento sull'auto-generazione di DDL.

Omettendo il campo dalla mappatura

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

    @Basic
    String note;

    @Transient
    String parsedNote;

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

Se la tua classe ha bisogno di campi che non dovrebbero essere scritti nel database, contrassegnali come `@Transient` . Dopo aver letto dal database, il campo sarà `null` .

Mappatura di ora e data

Ora e data sono disponibili in diversi tipi in Java: la `Date` e il `Calendar` ormai storici e i più recenti `LocalDate` e `LocalDateTime` . E `Timestamp` , `Instant` , `ZonedLocalDateTime` e i tipi Joda-time. Sul lato del database, abbiamo `time` , `date` e `timestamp` (sia per ora che per data), possibilmente con o senza fuso orario.

Data e ora prima di Java 8

Il mapping *predefinito* per i tipi pre-Java-8 `java.util.Date` , `java.util.Calendar` e `java.sql.Timestamp` è il `timestamp` in SQL; per `java.sql.Date` è 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;
}
```

Questo si mapperà perfettamente alla seguente tabella:

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

Questo potrebbe non essere l'intenzione. Ad esempio, spesso viene utilizzata una `Date` o un `Calendar` Java per rappresentare solo la data (per la data di nascita). Per modificare la mappatura predefinita o solo per rendere esplicita la mappatura, è possibile utilizzare l'annotazione `@Temporal` .

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

    @Temporal(TemporalType.TIME)
    private Date date;

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

La tabella SQL equivalente è:

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

Nota 1: il tipo specificato con `@Temporal` influenza la generazione DDL; ma puoi anche avere una colonna di tipo `date` map to `Date` con solo l'annotazione `@Basic` .

Nota 2: il `Calendar` non può mantenere solo il `time` .

Data e ora con Java 8

JPA 2.1 *non* definisce il supporto per i tipi `java.time` forniti in Java 8. La maggior parte delle implementazioni di JPA 2.1 offre tuttavia supporto per questi tipi, sebbene si tratti di estensioni dei produttori in senso stretto.

Per DataNucleus, questi tipi funzionano appena fuori dagli schemi e offrono una vasta gamma di possibilità di mappatura, in `@Temporal` con l'annotazione `@Temporal` .

Per Hibernate, se usi Hibernate 5.2+, dovrebbero funzionare immediatamente, usando solo l'annotazione `@Basic` . Se si utilizza Hibernate 5.0-5.1 è necessario aggiungere la dipendenza `org.hibernate:hibernate-java8` . Le mappature fornite sono

- `LocalDate` fino ad `date`
- `Instant` , `LocalDateTime` e `ZonedDateTime` su `timestamp`

Un'alternativa indipendente dal fornitore sarebbe anche quella di definire un `AttributeConverter` JPA 2.1 per qualsiasi tipo `java.time` Java 8 che è necessario mantenere.

Entità con ID gestito in sequenza

Qui abbiamo una classe e vogliamo che il campo identità (`userUid`) abbia il suo valore generato tramite un SEQUENCE nel database. Si presume che questa SEQUENZA sia denominata `USER_UID_SEQ` e può essere creata da un DBA o creata dal provider 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;
}
```

Leggi Mappatura di base online: https://riptutorial.com/it/jpa/topic/3691/mappatura-di-base

Capitolo 3: Mappatura One to One

Parametri

Annotazione	Scopo
@TableGenerator	Specifica il nome del generatore e il nome della tabella in cui è possibile trovare il generatore
@GeneratedValue	Specifica la strategia di generazione e si riferisce al nome del generatore
@Uno a uno	Specifica la relazione uno a uno tra impiegato e desk, qui il Dipendente è il proprietario della relazione
mappedBy	Questo elemento è fornito sul retro della relazione. Questo abilita la relazione bidirezionale

Examples

Relazione One to One tra impiegato e desk

Considera una relazione bidirezionale tra dipendente e scrivania.

Employee.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
}
```

Desk.java

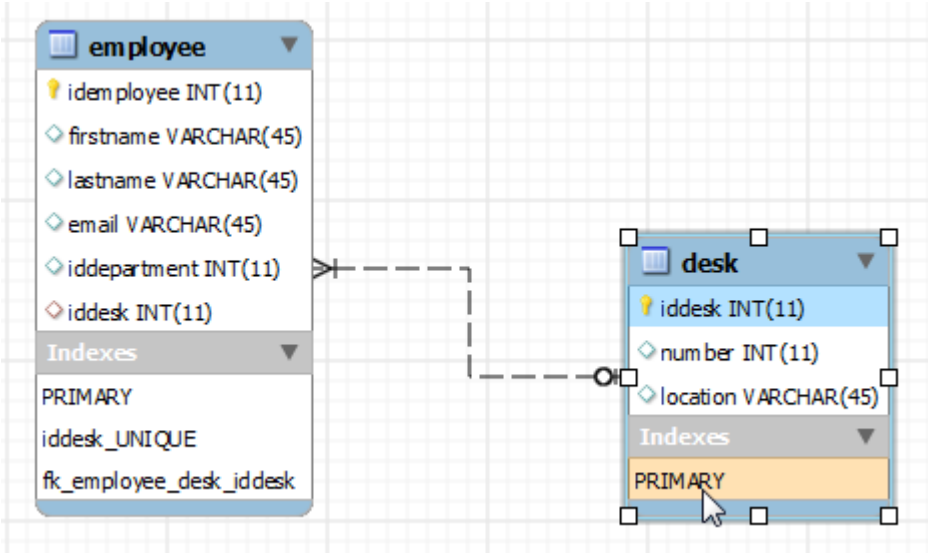
```
@Entity
public class Desk {
```

```
    @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
}
```

Codice di prova

```
/* 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());
```

Il diagramma del database è illustrato come di seguito.



- L'annotazione **@JoinColumn** continua sulla mappatura dell'entità che è mappata alla

tabella che contiene il join colulmn. Il proprietario della relazione. Nel nostro caso, la tabella Employee ha la colonna join in modo che **@JoinColumn** sia nel campo Desk dell'entità Employee.

- L'elemento **mappedBy** deve essere specificato nell'associazione **@OneToOne** nell'entità sul lato opposto della relazione. cioè L'entità che non fornisce la colonna di join sull'aspetto del database. Nel nostro caso, Desk è l'entità inversa.

L'esempio completo può essere trovato qui

Leggi Mappatura One to One online: https://riptutorial.com/it/jpa/topic/6474/mappatura-one-to-one

Capitolo 4: Mapping da molti a molti

introduzione

Una mappatura `ManyToMany` descrive una relazione tra entità in cui entrambe possono essere correlate a più istanze l'una dell'altra ed è definita `@ManyToMany` .

A differenza di `@OneToMany` cui è possibile utilizzare una colonna di chiave esterna nella tabella dell'entità, `ManyToMany` richiede una tabella di join, che associa le entità l'una all'altra.

Parametri

Annotazione	Scopo
`@TableGenerator`	Definisce un generatore di chiavi primarie a cui è possibile fare riferimento per nome quando viene specificato un elemento generatore per l'annotazione `GeneratedValue`
`@GeneratedValue`	Fornisce la specifica delle strategie di generazione per i valori delle chiavi primarie. Può essere applicato a una proprietà o campo chiave primaria di un'entità o superclasse mappata insieme all'annotazione Id.
`@ManyToMany`	Specifica la relazione tra Employee e entità del progetto in modo tale che molti dipendenti possano lavorare su più progetti.
`mappedBy="projects"`	Definisce una relazione bidirezionale tra Impiegato e Progetto
`@JoinColumn`	Specifica il nome della colonna che farà riferimento all'entità da considerare come proprietario dell'associazione
`@JoinTable`	Specifica la tabella nel database che manterrà dipendente per proiettare le relazioni utilizzando chiavi esterne

Osservazioni

- @TableGenerator e @GeneratedValue vengono utilizzati per la creazione automatica dell'ID utilizzando il generatore di tabelle jpa.
- @ManyToMany annotation specifica la relazione tra Employee e Project entities.
- @JoinTable specifica il nome della tabella da utilizzare come tabella di join jpa da molti a molti tra Employee e Project usando name = "employee_project". Ciò avviene perché non è possibile determinare la proprietà di un mapping jpa molti a molti poiché le tabelle del database non contengono chiavi esterne per fare riferimento ad altre tabelle.
- @JoinColumn specifica il nome della colonna che farà riferimento all'entità da considerare come proprietario dell'associazione mentre @inverseJoinColumn specifica il nome del lato

inverso della relazione. (Puoi scegliere qualsiasi lato per essere considerato come proprietario. Assicurati solo che quei lati siano in relazione). Nel nostro caso abbiamo scelto Employee come proprietario in modo che @JoinColumn faccia riferimento alla colonna idemployee nella tabella join employee_project e @InverseJoinColumn fa riferimento a idproject che è il lato inverso di jpa molti a molti mapping.

- @ManyToMany annotation nell'entità Project mostra una relazione inversa, quindi utilizza mappedBy = projects per fare riferimento al campo nell'entità Employee.

L'esempio completo può essere riferito qui

Examples

Dipendente per la mappatura da molti a molti

Entità dipendente.

```
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;
    }

}
```

Entità del progetto:

```
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;
    }

}
```

Codice di prova

/ * Crea EntityManagerFactory * / EntityManagerFactory emf = Persistenza .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();
```

Come gestire la chiave composta senza annotazione incorporabile

Se hai

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

```
+----------------------------+


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

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

Nella tabella `rightrole` sopra lo scenario è presente una chiave composta e per accedervi nell'utente JPA è necessario creare un'entità con annotazione `Embeddable` .

Come questo:

L'entità per la tabella del rightrole è:

```
    @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;

   .....
```

```
}
```

L'annotazione incorporabile va bene per un singolo oggetto ma causerà un problema durante l'inserimento di record di massa.

Il problema è che ogni volta che l'utente vuole creare un nuovo `role` con `rights` primo utente deve `store(persist)` oggetto `role` e quindi l'utente deve eseguire lo `flush` per ottenere l' `id` appena generato per il ruolo. allora e poi l'utente può inserirlo `rightrole` dell'entità `rightrole` .

Per risolvere questo utente si può scrivere un'entità leggermente diversa.

L'entità per la tabella dei ruoli è:

```
@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;
.......
}
```

L'annotazione @JoinTable si prenderà cura dell'inserimento nella tabella `rightrole` anche senza un'entità (purché tale tabella abbia solo le colonne id del ruolo e destra).

L'utente può quindi semplicemente:

```
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);
```

Devi scrivere @ManyToMany (cascade = {CascadeType.PERSIST}) in inverseJoinColumns

altrimenti i tuoi dati parent verranno cancellati se il figlio viene cancellato.

Leggi Mapping da molti a molti online: https://riptutorial.com/it/jpa/topic/6532/mapping-da-molti-a-molti

Capitolo 5: Mapping molti a uno

Parametri

Colonna	Colonna
@TableGenerator	Utilizza la strategia del generatore di tabelle per la creazione automatica dell'ID
@GeneratedValue	Specifica che il valore applicato ai campi è un valore generato
@id	Annota il campo come identificatore
@ManyToOne	Specifica la relazione Molti a uno tra Dipendente e Reparto. Questa annotazione è contrassegnata da più parti. vale a dire che più dipendenti appartengono a un singolo dipartimento. Quindi Department è annotato con @ManyToOne nell'entità Employee.
@JoinColumn	Specifica la colonna della tabella del database che memorizza la chiave esterna per l'entità correlata

Examples

Dipendente del dipartimento ManyToOne

Entità dei dipendenti

```
@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
}
```

Dipartimento Entità

```
@Entity
public class Department {

    @Id
    private int iddepartment;
    private String name;

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

Classe di prova

```
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();

    }

}
```

Leggi Mapping molti a uno online: https://riptutorial.com/it/jpa/topic/6531/mapping-molti-a-uno

Capitolo 6: Relazione da uno a molti

Parametri

Annotazione	Scopo
@TableGenerator	Specifica il nome del generatore e il nome della tabella in cui è possibile trovare il generatore
@GeneratedValue	Specifica la strategia di generazione e si riferisce al nome del generatore
@ManyToOne	Specifica una relazione molti a uno tra Dipendente e Reparto
@OneToMany (mappedBy = "reparto")	crea una relazione bidirezionale tra Dipendente e Dipartimento semplicemente facendo riferimento all'annotazione @ManyToOne nell'entità Dipendente

Examples

Relazione One To Many

La mappatura da uno a molti è generalmente semplicemente una relazione bidirezionale di mappatura Molti a Uno. Prendiamo lo stesso esempio che abbiamo utilizzato per la mappatura da molti a uno.

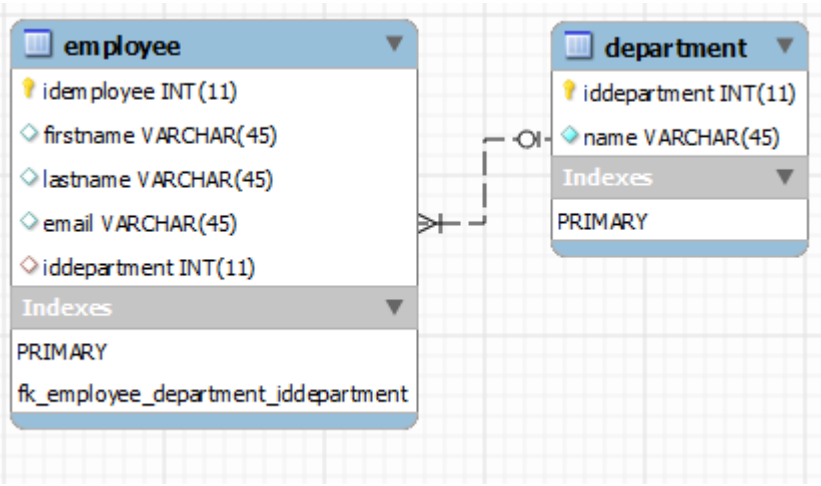
Employee.java

```
@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
}
```

Department.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
}
```

Questa relazione è rappresentata nel database come di seguito.



Ci sono due punti da ricordare sulla mappatura da jpa one a many:

- Da molti a parte c'è il lato proprietario della relazione. La colonna è definita su quel lato.
- La mappatura da uno a molti è il lato inverso, quindi l'elemento mappedBy deve essere usato sul lato inverso.

L'esempio completo può essere riferito qui

Leggi Relazione da uno a molti online: https://riptutorial.com/it/jpa/topic/6529/relazione-da-uno-a-molti

Capitolo 7: Relazioni tra entità

Osservazioni

Relazioni tra le nozioni di base delle entità

Una **chiave esterna** può essere una o più colonne che fanno riferimento a una chiave univoca, in genere la chiave primaria, in un'altra tabella.

Una chiave esterna e la chiave genitore primaria a cui fa riferimento devono avere lo stesso numero e tipo di campi.

Le chiavi esterne rappresentano le **relazioni** da una colonna o colonne in una tabella a una colonna o colonne in un'altra tabella.

Examples

Molteplicità nelle relazioni tra entità

Molteplicità nelle relazioni tra entità

Le molteplicità sono dei seguenti tipi:

- **Uno a uno** : ogni istanza di entità è correlata a una singola istanza di un'altra entità.
- **Uno a molti** : un'istanza di entità può essere correlata a più istanze delle altre entità.
- **Molte-a-uno** : più istanze di un'entità possono essere correlate a una singola istanza dell'altra entità.
- **Molti-a-molti** : le istanze di entità possono essere correlate a più istanze l'una dell'altra.

Mappatura One-to-One

Il mapping uno-a-uno definisce un'associazione a valore singolo con un'altra entità con molteplicità uno-a-uno. Questo mapping delle relazioni utilizza l'annotazione `@OneToOne` nella proprietà o campo persistente corrispondente.

Esempio: entità `Vehicle` *e* `ParkingPlace` *.*

Mappatura uno a molti

Un'istanza di entità può essere correlata a più istanze delle altre entità.

Le relazioni uno-a-molti utilizzano l'annotazione `@OneToMany` sulla proprietà o campo persistente corrispondente.

L'elemento `mappedBy` è necessario per fare riferimento all'attributo annotato da ManyToOne nell'entità corrispondente:

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

Un'associazione uno-a-molti deve mappare la raccolta di entità.

Mapping molti-a-uno

Un mapping molti-a-uno viene definito annotando l'attributo nell'entità di origine (l'attributo che si riferisce all'entità di destinazione) con l'annotazione `@ManyToOne` .

`@JoinColumn(name="FK_name")` una chiave di una relazione.

Mapping molti-a-molti

Le istanze di entità possono essere correlate a più istanze l'una dell'altra.

Le relazioni many-to-many utilizzano l'annotazione `@ManyToMany` sulla proprietà o campo persistente corrispondente.

Dobbiamo utilizzare una terza tabella per associare i due tipi di entità (tabella join).

@JoinTable Esempio di annotazione

Quando si mappano le relazioni molti-a-molti in JPA, è possibile fornire la configurazione per la tabella utilizzata per l'unione di chiavi esterne utilizzando l'annotazione `@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;
   [...]
}
```

In questo esempio, che consiste EntityA avente molti-a-molti relazione EntityB, realizzata dal `entitiesB`

campo, si usa l'annotazione @JoinTable per specificare che il nome di tabella per la tabella join è `table_join_A_B` , composta dalle colonne `id_A` e `id_B` , chiavi esterne che fanno riferimento rispettivamente colonna `id` nella tabella di EntityA e nella tabella di EntityB; `(id_A,id_B)` sarà una chiave primaria composta per la tabella `table_join_A_B` .

Leggi Relazioni tra entità online: https://riptutorial.com/it/jpa/topic/6305/relazioni-tra-entita

Capitolo 8: Strategia di ereditarietà a tabella singola

Parametri

Annotazione	Scopo
@Eredità	Specifica il tipo di strategia di ereditarietà utilizzata
@DiscriminatorColumn	Specifica una colonna nel database che verrà utilizzata per identificare entità diverse in base a determinati ID assegnati a ciascuna entità
@MappedSuperClass	le super classi mappate non sono persistenti e vengono utilizzate solo per mantenere lo stato per le sue sottoclassi. Classi java astratte in generale sono contrassegnate con @MapperSuperClass
@DiscriminatorValue	Un valore specificato nella colonna definita da @DiscriminatorColumn. Questo valore aiuta a identificare il tipo di entità

Osservazioni

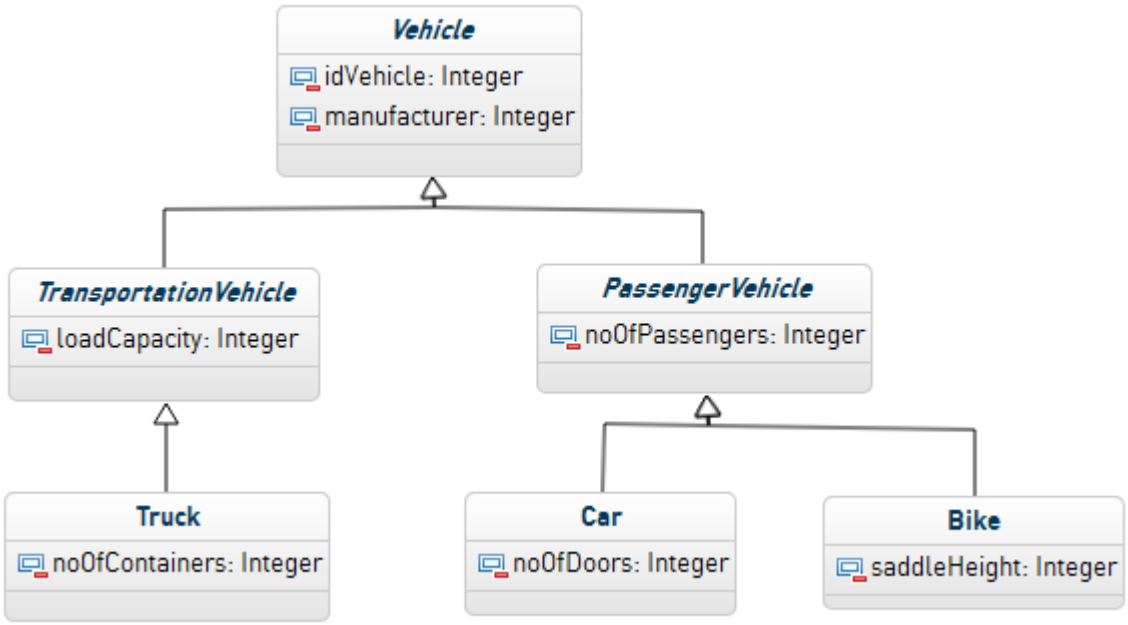
Il vantaggio della strategia a tabella singola è che non richiede join complessi per il recupero e l'inserimento di entità, ma d'altra parte spreca spazio nel database in quanto molte colonne devono essere annullabili e non ci sono dati per esse.

L'esempio completo e l'articolo possono essere trovati qui

Examples

Strategia di ereditarietà a una tabella

Un semplice esempio di gerarchia del veicolo può essere preso per la strategia di ereditarietà di una tabella singola.



Classe veicolo astratta:

```
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;

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;
    }

}
```

PassengerVehicle.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;
    }

}
```

Codice di prova:

```
/* 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();
```

Leggi Strategia di ereditarietà a tabella singola online:
https://riptutorial.com/it/jpa/topic/6277/strategia-di-ereditarieta-a-tabella-singola

Capitolo 9: Strategia di ereditarietà unita

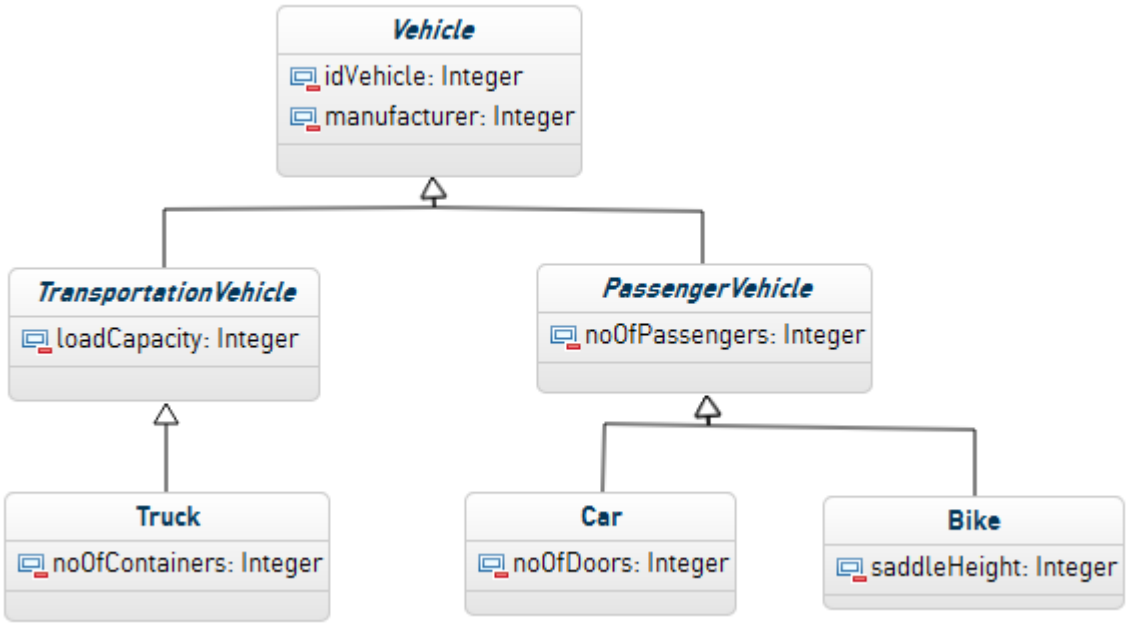
Parametri

Annotazione	Scopo
@Eredità	Specifica il tipo di strategia di ereditarietà utilizzata
@DiscriminatorColumn	Specifica una colonna nel database che verrà utilizzata per identificare entità diverse in base a determinati ID assegnati a ciascuna entità
@MappedSuperClass	le super classi mappate non sono persistenti e vengono utilizzate solo per mantenere lo stato per le sue sottoclassi. Classi java astratte in generale sono contrassegnate con @MapperSuperClass

Examples

Strategia di ereditarietà unita

Un diagramma di classe di esempio in base al quale vedremo l'implementazione di 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
}
```

TransportationVehicle.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

}
```

PassengerVehicle.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

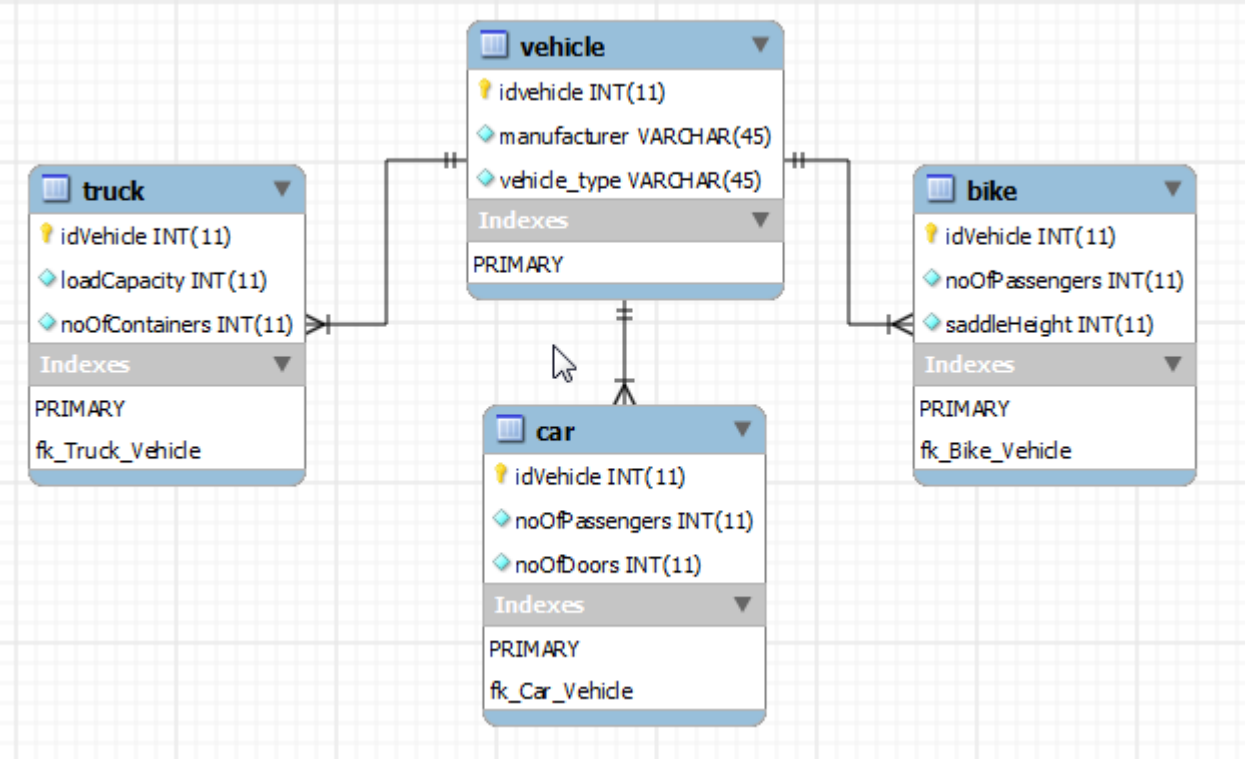
```
@Entity
public class Bike extends PassengerVehicle {
```

```
    private int saddleHeight;

    // getters and setters

}
```

Codice di prova

```
/* 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();
```

Lo schema del database sarà come di seguito.



Il vantaggio della strategia di ereditarietà unita è che non spreca lo spazio del database come nella strategia a tabella singola. D'altro canto, a causa di più join coinvolti per ogni inserimento e recupero, le prestazioni diventano un problema quando le gerarchie di ereditarietà diventano ampie e profonde.

L'esempio completo con spiegazione può essere letto qui

Leggi Strategia di ereditarietà unita online: https://riptutorial.com/it/jpa/topic/6473/strategia-di-ereditarieta-unita

Capitolo 10: Tabella per strategia di ereditarietà della classe concreta

Osservazioni

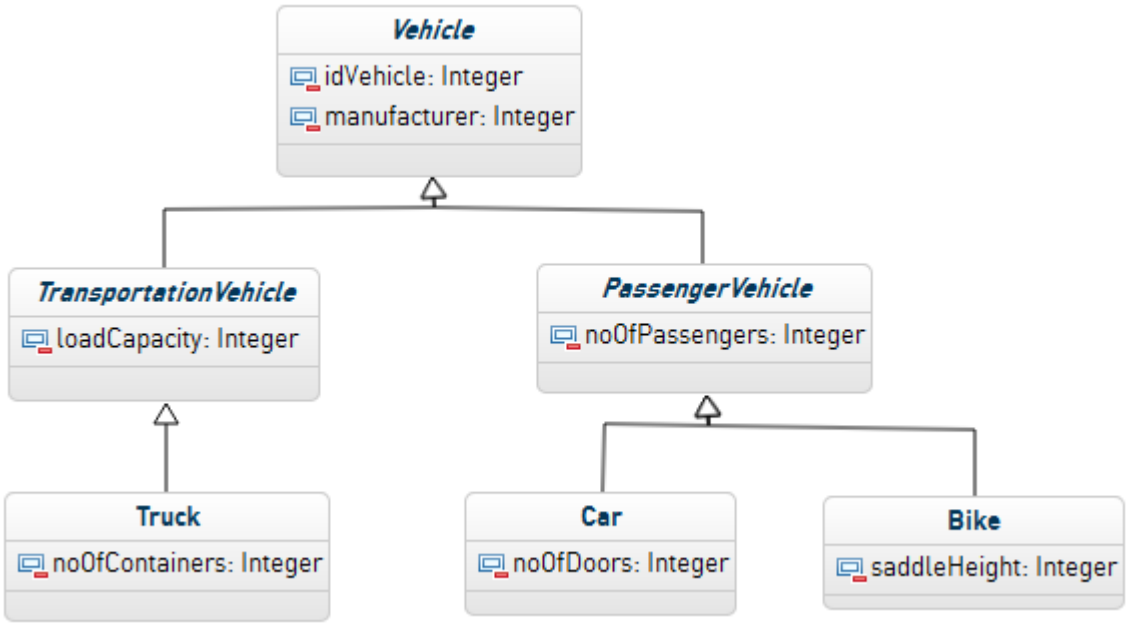
- Vehicle, TransportationVehicle and PassengerVehicle sono classi astratte e non avranno una tabella separata nel database.
- Truck, Car and Bike sono classi concrete in modo che vengano mappate alle tabelle corrispondenti. Queste tabelle dovrebbero includere tutti i campi per le classi annotate con @MappedSuperClass perché non hanno tabelle corrispondenti nel database.
- Quindi, la tabella Truck avrà colonne per memorizzare i campi ereditati da TransportationVehicle e Vehicle.
- Allo stesso modo, Car and Bike avrà colonne per memorizzare i campi ereditati da Passenger Veicolo e Veicolo.

L'esempio completo può essere trovato qui

Examples

Tabella per strategia di ereditarietà della classe concreta

Prenderemo esempio di gerarchia del veicolo come illustrato di seguito.



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;
    }

}
```

TransportationVehilcle.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;
    }

}
```

PassengerVehicle.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

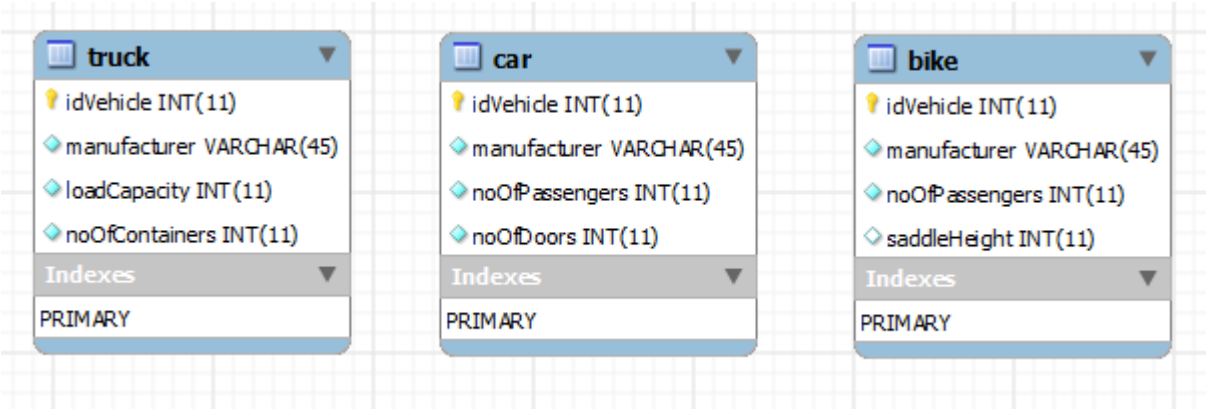
```
package com.thejavageek.jpa.entities;

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 rappresentazione del database sarà la seguente



Leggi Tabella per strategia di ereditarietà della classe concreta online: https://riptutorial.com/it/jpa/topic/6255/tabella-per-strategia-di-ereditarieta-della-classe-concreta

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con jpa	Billy Frost, Community, DimaSan, , Manuel Spigolon, Michael Piefel, Neil Stockton, ppeterka
2	Mappatura di base	Jeffrey Brett Coleman, Michael Piefel, Neil Stockton, Pete
3	Mappatura One to One	Prasad Kharkar
4	Mapping da molti a molti	Prasad Kharkar, Ronak Patel, Vetle
5	Mapping molti a uno	Prasad Kharkar
6	Relazione da uno a molti	Prasad Kharkar
7	Relazioni tra entità	DimaSan,
8	Strategia di ereditarietà a tabella singola	Prasad Kharkar
9	Strategia di ereditarietà unita	bw_üezi, Prasad Kharkar
10	Tabella per strategia di ereditarietà della classe concreta	Prasad Kharkar