Kostenloses eBook

LERNEN

jpa

Free unaffiliated eBook created from
Stack Overflow contributors.

#jpa

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit jpa

Bemerkungen

JPA ist die Java Persistence API, eine Spezifikation, die die Zuordnung von Java-Objekten und deren Beziehungen zu einer relationalen Datenbank behandelt. Dies wird als objektrelationaler Mapper (ORM) bezeichnet. Es ist eine Alternative für (oder eine Ergänzung) zu JDBC mit niedrigerem Level. Dies ist besonders nützlich, wenn Sie einen Java-orientierten Ansatz verfolgen und wenn komplexe Objektdiagramme persistent sein müssen.

JPA an sich ist keine Implementierung. Sie benötigen dafür einen Persistenzanbieter (siehe Beispiele). Aktuelle Implementierungen des neuesten JPA 2.1-Standards sind EclipseLink (auch die Referenzimplementierung für JPA 2.1, was "den Nachweis der Implementierung der Spezifikation" bedeutet). Ruhezustand und DataNucleus .

Metadaten

Die Zuordnung zwischen Java-Objekten und Datenbanktabellen wird über **Persistenz-Metadaten** definiert. Der JPA-Anbieter verwendet die Persistenz-Metadaten-Informationen, um die korrekten Datenbankvorgänge auszuführen. JPA definiert die Metadaten normalerweise über Anmerkungen in der Java-Klasse.

Objektbezogene Entity-Architektur

Die Entitätsarchitektur besteht aus:

- Entitäten
- Persistenzeinheiten
- Persistenzkontexte
- Entitätenmanagerfabriken
- Entity-Manager

Versionen

Ausführung	Expertengruppe	Veröffentlichung
1,0	JSR-220	2006-11-06
2,0	JSR-317	2009-12-10
2.1	JSR-338	2013-05-22

Examples

Klassenpfadanforderungen

Eclipselink

Die Eclipselink- und JPA-API muss enthalten sein. Beispiel Maven-Abhängigkeiten:

```
<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>
```

Überwintern

Hibernate-Core ist erforderlich. Beispiel für eine Maven-Abhängigkeit:

```
<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

Datanucleus-Core, Datanucleus-Api-Jpa und Datanucleus-Rdbms (bei Verwendung von RDBMS-Datastores) sind erforderlich. Beispiel für eine Maven-Abhängigkeit:

```
<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>
```

Konfigurationsdetails

JPA erfordert die Verwendung einer Datei *persistence.xml* , die sich unter `META-INF` im Stammverzeichnis des CLASSPATH befindet. Diese Datei enthält eine Definition der verfügbaren Persistenz-Einheiten, von denen aus JPA arbeiten kann.

JPA erlaubt außerdem die Verwendung einer Mapping-Konfigurationsdatei *orm.xml* , die ebenfalls unter `META-INF` *abgelegt* ist. Diese Zuordnungsdatei wird verwendet, um zu konfigurieren, wie Klassen dem Datastore zugeordnet werden. Sie ist eine Alternative / Ergänzung zur Verwendung von Java-Annotationen in den JPA-Entitätsklassen.

Beispiel für minimale Persistenz.xml

Ruhezustand (und eingebettete H2-Datenbank)

```
<?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 (und eingebettete H2-Datenbank)

```
<?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 (und eingebettete H2-Datenbank)

```
<?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>
```

Hallo Welt

Lassen Sie uns alle grundlegenden Komponenten für die Erstellung einer einfachen Hallo-Welt sehen.

1. Definieren Sie, welche Implementierung von JPA 2.1 verwendet wird
2. Stellen Sie die Verbindung zur Datenbank her und erstellen Sie die `persistence-unit`
3. Implementiert die Entitäten
4. Implementiert ein DAO (Datenzugriffsobjekt) zum Bearbeiten der Entitäten
5. Testen Sie die Anwendung

Bibliotheken

Mit Maven brauchen wir diese Abhängigkeiten:

```
<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>
```

Persistenzeinheit

Im Ressourcenordner müssen wir eine Datei mit dem Namen `persistence.xml` erstellen. Der einfachste Weg zum Definieren ist wie folgt:

```
<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>
```

Implementiere eine Entität

Ich erstelle eine Klasse `Biker` :

```
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;
    }
}
```

Implementieren Sie ein 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` ist ein 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;
    }
}
```

Testen Sie die Anwendung

Paket it.hello.jpa.test;

öffentliche Klasse 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));
}
```

}

Die Ausgabe wird sein:

> It.hello.jpa.test.TestJpa ausführen Ruhezustand: Drop-Tabelle BIKER, falls vorhanden Ruhezustand: Drop-Sequenz, wenn vorhanden Hibernate_sequence Hibernate: Sequenz Hibernate_sequence beginnen mit 1 Inkrement um 1 Hibernate: Tabelle BIKER erstellen (id bigint nicht null, battleName varar (255)), Bart Boolean, Geburtstagsdatum, BikerName Varchar (255), RegistrationDatum-Zeit, Primärschlüssel (ID)) Ruhezustand: Änderungstabelle BIKER Add-Einschränkung , id) Werte (?,?,?,?,?,?) 01. März 2017 23:00:02 PM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation INFO: HHH000204: Verarbeitung von PersistenceUnitInfo [Name: hello- jpa-pu ...] Ergebnisse:
>
> Tests: 1, Fehler: 0, Fehler: 0, Übersprungen: 0

Erste Schritte mit jpa online lesen: https://riptutorial.com/de/jpa/topic/2125/erste-schritte-mit-jpa

Kapitel 2: Beziehungen zwischen Entitäten

Bemerkungen

Beziehungen zwischen Entitäten Grundlagen

Ein **Fremdschlüssel** kann eine oder mehrere Spalten sein, die auf einen eindeutigen Schlüssel, normalerweise den Primärschlüssel, in einer anderen Tabelle verweisen.

Ein Fremdschlüssel und der übergeordnete übergeordnete Schlüssel, auf den er verweist, müssen die gleiche Anzahl und Art von Feldern haben.

Fremdschlüssel stehen für **Beziehungen** von einer oder mehreren Spalten in einer Tabelle zu einer oder mehreren Spalten in einer anderen Tabelle.

Examples

Vielfalt in Entitätsbeziehungen

Vielfalt in Entitätsbeziehungen

Multiplizitäten sind von der folgenden Art:

- **Eins-zu-Eins** : Jede Entitätsinstanz bezieht sich auf eine einzelne Instanz einer anderen Entität.
- **One-to-Many** : Eine Entitätsinstanz kann mit mehreren Instanzen der anderen Entitäten verknüpft werden.
- **Many-to-one** : Mehrere Instanzen einer Entität können mit einer einzelnen Instanz der anderen Entität verknüpft werden.
- **Many-to-many** : Die Entitätsinstanzen können mit mehreren Instanzen voneinander verbunden sein.

Eins-zu-Eins-Mapping

Eins-zu-Eins-Zuordnung definiert eine einwertige Zuordnung zu einer anderen Entität mit Eins-zu-Eins-Multiplizität. Diese Beziehungszuordnung verwendet die Annotation `@OneToOne` für die entsprechende persistente Eigenschaft oder das entsprechende Feld.

Beispiel: `Vehicle` *und* `ParkingPlace` *Entitäten.*

One-to-Many-Zuordnung

Eine Entitätsinstanz kann mit mehreren Instanzen der anderen Entitäten verknüpft werden.

One-to- `@OneToMany` -Beziehungen verwenden die `@OneToMany` Annotation für die entsprechende persistente Eigenschaft oder das entsprechende Feld.

Das Element `mappedBy` wird benötigt, um auf das von ManyToOne in der entsprechenden Entität annotierte Attribut zu verweisen:

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

Eine Eins-zu-Viele-Zuordnung muss die Sammlung von Entitäten zuordnen.

Many-to-One-Zuordnung

Eine Many-to-One-Zuordnung wird definiert, indem das Attribut in der Quellenentität (das Attribut, das sich auf die `@ManyToOne` bezieht) mit der Annotation `@ManyToOne` .

Eine `@JoinColumn(name="FK_name")` beschreibt einen vorläufigen Schlüssel einer Beziehung.

Viele-zu-Viele-Mapping

Die Entitätsinstanzen können mit mehreren Instanzen voneinander verbunden sein.

Viele-zu-viele-Beziehungen verwenden die Annotation `@ManyToMany` für die entsprechende persistente Eigenschaft oder das entsprechende Feld.

Wir müssen eine dritte Tabelle verwenden, um die beiden Entitätstypen zu verknüpfen (Join-Tabelle).

@JoinTable-Anmerkungsbeispiel

Beim `@JoinTable` to- `@JoinTable` -Beziehungen in JPA kann die Konfiguration für die Tabelle, die für die verbundenen Fremdschlüssel verwendet wird, mithilfe der Annotation `@JoinTable` bereitgestellt werden:

```
@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 diesem Beispiel, das aus EntityA besteht, das eine EntityB-Beziehung zwischen EntityB und EntityB aufweist, die durch das `entitiesB` Feld realisiert wird, verwenden wir die Annotation @JoinTable, um anzugeben, dass der Tabellenname für die Join-Tabelle `table_join_A_B` und aus den Spalten `id_A` und besteht `id_B` : Fremdschlüssel, die auf die Spalten- `id` in der EntityA-Tabelle und in der EntityB-Tabelle verweisen. `(id_A,id_B)` ist ein zusammengesetzter Primärschlüssel für `table_join_A_B` Tabelle `table_join_A_B` .

Beziehungen zwischen Entitäten online lesen:
https://riptutorial.com/de/jpa/topic/6305/beziehungen-zwischen-entitaten

Kapitel 3: Eine zu viele Beziehung

Parameter

Anmerkung	Zweck
@TableGenerator	Gibt den Namen des Generators und den Tabellennamen an, wo der Generator gefunden werden kann
@GeneratedValue	Gibt die Erzeugungsstrategie an und bezieht sich auf den Namen des Generators
@ManyToOne	Gibt eine viele-zu-eins-Beziehung zwischen Mitarbeiter und Abteilung an
@OneToMany (mappedBy = "Abteilung")	erstellt eine bidirektionale Beziehung zwischen Mitarbeiter und Abteilung, indem einfach auf die @ManyToOne-Anmerkung in der Entität Employee Bezug genommen wird

Examples

Eine zu viele Beziehung

Eins-zu-viele-Zuordnungen ist im Allgemeinen einfach eine bidirektionale Beziehung zwischen Viel-zu-Eins-Zuordnung. Wir werden dasselbe Beispiel nehmen, das wir für viele zu einer Abbildung genommen haben.

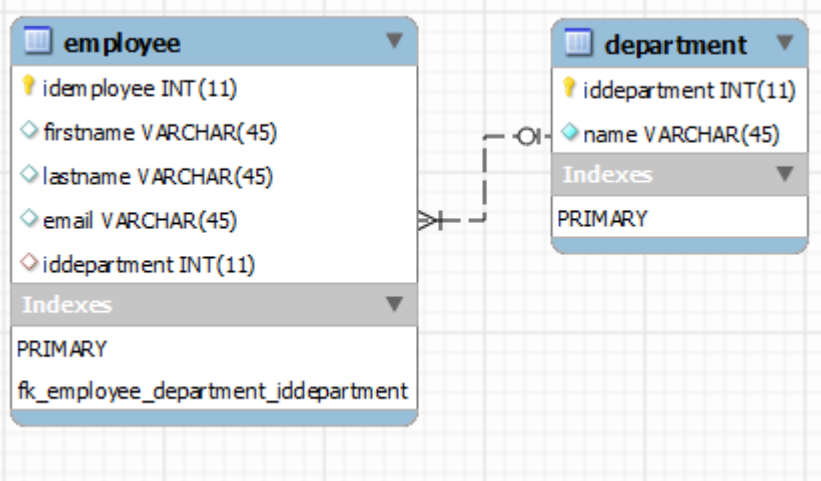
Mitarbeiter.java

```
@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
}
```

Abteilung.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
}
```

Diese Beziehung wird in der Datenbank wie folgt dargestellt.



Es gibt zwei Punkte, die Sie bei jpa eins zu vielen Mapping berücksichtigen sollten:

- Das Viele auf einer Seite ist die besitzende Seite der Beziehung. Die Spalte ist auf dieser Seite definiert.
- Die Eins-zu-Viele-Zuordnung ist die inverse Seite, daher muss das Element mappedBy auf der inversen Seite verwendet werden.

Das vollständige Beispiel kann hier referenziert werden

Eine zu viele Beziehung online lesen: https://riptutorial.com/de/jpa/topic/6529/eine-zu-viele-beziehung

Kapitel 4: Eins zu Eins Mapping

Parameter

Anmerkung	Zweck
@TableGenerator	Gibt den Namen des Generators und den Tabellennamen an, wo der Generator gefunden werden kann
@GeneratedValue	Gibt die Erzeugungsstrategie an und bezieht sich auf den Namen des Generators
@Eins zu eins	Gibt eine Eins-zu-Eins-Beziehung zwischen Mitarbeiter und Schreibtisch an. Hier ist der Mitarbeiter Inhaber der Beziehung
mappedBy	Dieses Element befindet sich auf der Rückseite der Relation. Dies ermöglicht eine bidirektionale Beziehung

Examples

Eins-zu-Eins-Beziehung zwischen Mitarbeiter und Schreibtisch

Betrachten Sie eine bidirektionale Eins-zu-Eins-Beziehung zwischen Mitarbeiter und Schreibtisch.

Mitarbeiter.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

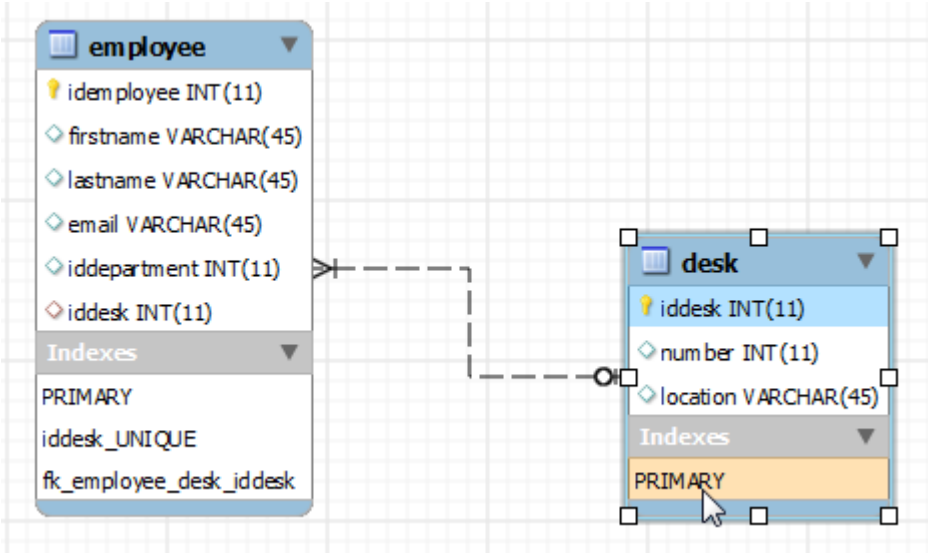
```
@Entity
public class Desk {
```

```
    @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
}
```

Testcode

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

Das Datenbankdiagramm ist wie folgt dargestellt.



- Die Annotation **@JoinColumn nimmt** die Zuordnung der Entität vor, die der Tabelle

zugeordnet ist, die die Join-Spalte enthält. Der Eigentümer der Beziehung. In unserem Fall enthält die Employee-Tabelle die Join-Spalte. **@JoinColumn** befindet sich im Desk-Feld der Employee-Entität.

- Das Element **mappedBy** sollte in der **@OneToOne-** Zuordnung in der Entität angegeben werden, die die Beziehungsseite **kehrt** . dh Die Entität, die keine Join-Spalte für Datenbankaspekte bereitstellt. In unserem Fall ist Desk die inverse Entität.

Das vollständige Beispiel finden Sie hier

Eins zu Eins Mapping online lesen: https://riptutorial.com/de/jpa/topic/6474/eins-zu-eins-mapping

Kapitel 5: Grundlegende Zuordnung

Parameter

Anmerkung	Einzelheiten
`@Id`	Markiert Feld / Spalte als *Schlüssel* der Entität
`@Basic`	Marks angeforderte Feld als *Basistyp* zugeordnet. Dies gilt für primitive Typen und deren Wrapper, `String` , `Date` und `Calendar` . Die Anmerkung ist eigentlich optional, wenn keine Parameter angegeben werden. Ein guter Stil würde jedoch dazu führen, dass Ihre Absichten explizit gemacht werden.
`@Transient`	Felder, die als transient markiert sind, werden nicht als persistent betrachtet, ähnlich wie das `transient` Schlüsselwort für die Serialisierung.

Bemerkungen

Es muss immer einen Standardkonstruktor geben, dh den parameterlosen. Im grundlegenden Beispiel wurde kein Konstruktor angegeben. Java fügte einen hinzu. Wenn Sie jedoch einen Konstruktor mit Argumenten hinzufügen, müssen Sie auch den parameterlosen Konstruktor hinzufügen.

Examples

Eine sehr einfache Entität

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

    @Basic
    String note;

    @Basic
    int count;
}
```

Getter, Setter usw. sind aus Gründen der Kürze nicht vorgesehen, werden aber für JPA ohnehin nicht benötigt.

Diese Java-Klasse würde der folgenden Tabelle zugeordnet werden (abhängig von Ihrer Datenbank, hier in einer möglichen Postgres-Zuordnung):

```
CREATE TABLE Note (
```

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

JPA-Provider können zum Generieren der DDL verwendet werden und erzeugen wahrscheinlich andere DDL als die hier gezeigte. Solange die Typen jedoch kompatibel sind, führt dies zur Laufzeit nicht zu Problemen. Es ist am besten, sich nicht auf die automatische Generierung von DDL zu verlassen.

Feld aus der Zuordnung auslassen

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

    @Basic
    String note;

    @Transient
    String parsedNote;

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

Wenn Ihre Klasse Felder benötigt, die nicht in die Datenbank geschrieben werden sollen, markieren Sie sie als `@Transient` . Nach dem Lesen aus der Datenbank ist das Feld `null` .

Zuordnungszeit und Datum

Zeit und Datum gibt es in Java in einer Reihe verschiedener Typen: Das jetzt historische `Date` und der aktuelle `Calendar` sowie das aktuellere `LocalDate` und `LocalDateTime` . Und `Timestamp` , `Instant` , `ZonedLocalDateTime` und die Joda- `ZonedLocalDateTime` . Auf der Datenbankseite haben wir `time` , `date` und `timestamp` (sowohl Zeit als auch Datum), möglicherweise mit oder ohne Zeitzone.

Datum und Uhrzeit vor Java 8

Die *Standardzuordnung* für die Pre-Java-8-Typen `java.util.Date` , `java.util.Calendar` und `java.sql.Timestamp` ist der `timestamp` in SQL. für `java.sql.Date` es `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;
}
```

Dies wird der folgenden Tabelle perfekt zugeordnet:

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

Dies kann nicht die Absicht sein. Beispielsweise wird ein Java- `Date` oder `Calendar` häufig nur zur Darstellung des Datums (für das Geburtsdatum) verwendet. Um die Standardzuordnung zu ändern oder um die Zuordnung explizit zu machen, können Sie die Annotation `@Temporal` .

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

    @Temporal(TemporalType.TIME)
    private Date date;

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

Die äquivalente SQL-Tabelle ist:

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

Hinweis 1: Der mit `@Temporal` angegebene `@Temporal` beeinflusst die DDL-Generierung. Sie können aber auch einen colum vom Typ haben `date` Karte `Date` nur mit der `@Basic` Anmerkung.

Hinweis 2: Der `Calendar` kann nicht nur die `time` beibehalten.

Datum und Uhrzeit mit Java 8

In JPA 2.1 ist *keine* Unterstützung für `java.time` Typen definiert, die in Java 8 bereitgestellt

werden. Die meisten Implementierungen von JPA 2.1 bieten jedoch Unterstützung für diese Typen. Allerdings handelt es sich hierbei streng um Herstellererweiterungen.

Für DataNucleus sind diese Typen einfach `@Temporal` und bieten eine Vielzahl von Mapping-Möglichkeiten, die mit der `@Temporal` Annotation `@Temporal` .

Wenn Sie Hibernate 5.2 oder höher verwenden, sollten Sie sofort mit der `@Basic` Annotation arbeiten. Wenn Sie Hibernate 5.0-5.1 verwenden, müssen Sie die Abhängigkeit `org.hibernate:hibernate-java8` hinzufügen `org.hibernate:hibernate-java8` . Die bereitgestellten Zuordnungen sind

- `LocalDate` bis `date`
- `Instant` , `LocalDateTime` und `ZonedDateTime` zum `timestamp`

Eine herstellerneutrale Alternative wäre auch die Definition eines JPA 2.1 `AttributeConverter` für jeden Java 8-Typ von `java.time` , der persistent sein muss.

Entität mit sequenzverwalteter ID

Hier haben wir eine Klasse und wir möchten, dass das Identitätsfeld (`userUid`) über eine SEQUENCE in der Datenbank generiert wird. Es wird angenommen, dass diese SEQUENCE als `USER_UID_SEQ` wird und von einem DBA oder vom JPA-Provider erstellt werden kann.

```
@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;
}
```

Grundlegende Zuordnung online lesen: https://riptutorial.com/de/jpa/topic/3691/grundlegende-zuordnung

Kapitel 6: Strategie für die Vererbung einzelner Tabellen

Parameter

Anmerkung	Zweck
@Erbe	Gibt den Typ der verwendeten Vererbungsstrategie an
@DiscriminatorColumn	Gibt eine Spalte in der Datenbank an, die zur Identifizierung verschiedener Entitäten verwendet wird, basierend auf einer bestimmten Entität, die jeder Entität zugeordnet ist
@MappedSuperClass	Zugeordnete Superklassen sind nicht persistent und werden nur verwendet, um den Status für ihre Unterklassen aufrechtzuerhalten. Im Allgemeinen sind abstrakte Java-Klassen mit @MapperSuperClass gekennzeichnet
@DiscriminatorValue	Ein in einer von @DiscriminatorColumn definierten Spalte angegebener Wert. Dieser Wert hilft beim Identifizieren des Entitätstyps

Bemerkungen

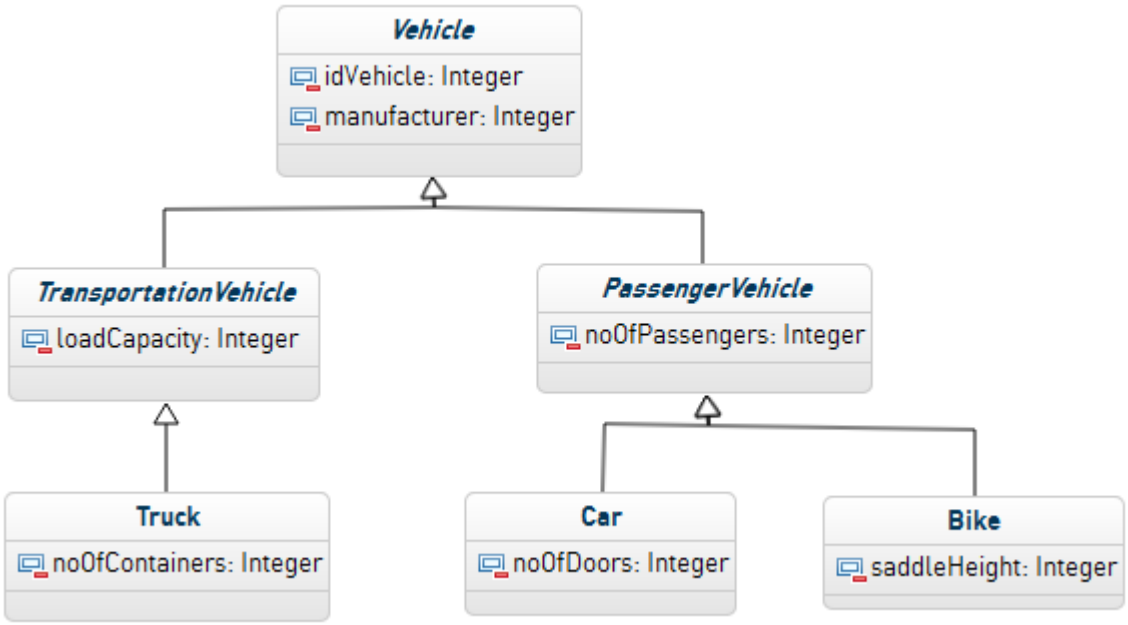
Der Vorteil der Single-Table-Strategie besteht darin, dass zum Abrufen und Einfügen von Entitäten keine komplexen Joins erforderlich sind. Andererseits wird Datenbankplatz verschwendet, da viele Spalten nullfähig sein müssen und keine Daten vorhanden sind.

Vollständiges Beispiel und Artikel finden Sie hier

Examples

Strategie zur Vererbung einzelner Tabellen

Ein einfaches Beispiel für die Fahrzeughierarchie kann für die Vererbungsstrategie für einzelne Tabellen verwendet werden.



Zusammenfassung Fahrzeugklasse:

```
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 Paket 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;
    }

}
```

Testcode:

```
/* 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();
```

Strategie für die Vererbung einzelner Tabellen online lesen:
https://riptutorial.com/de/jpa/topic/6277/strategie-fur-die-vererbung-einzelner-tabellen

Kapitel 7: Tabelle pro konkreter Klassenvererbungsstrategie

Bemerkungen

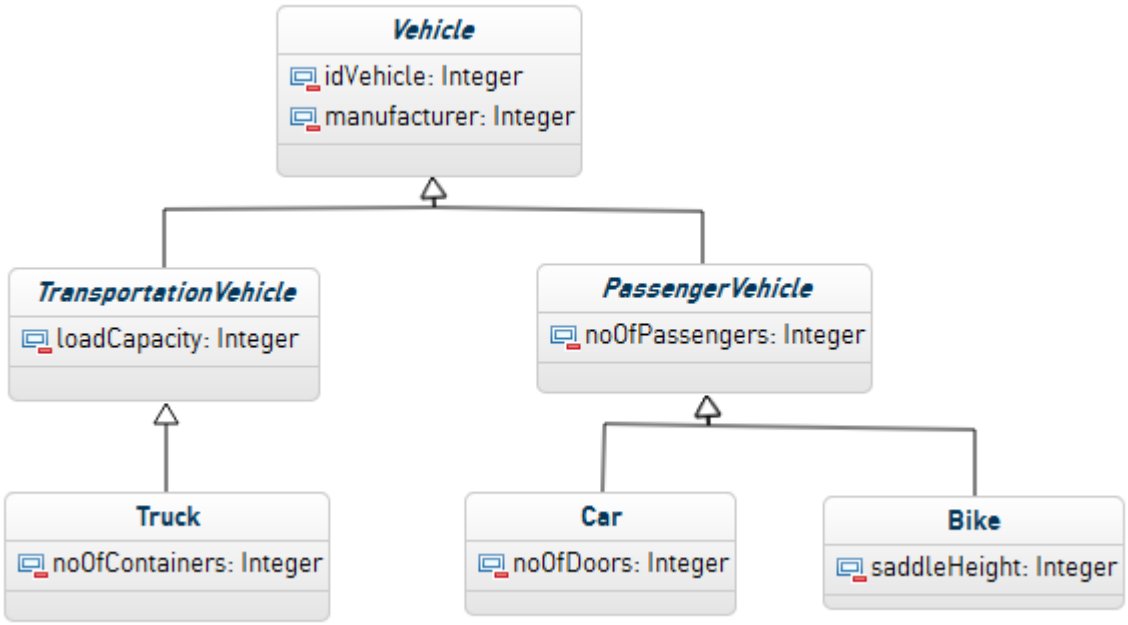
- Fahrzeug, Transportfahrzeug und Passagierfahrzeug sind abstrakte Klassen und haben keine separate Tabelle in der Datenbank.
- Truck, Car und Bike sind konkrete Klassen, daher werden sie entsprechenden Tabellen zugeordnet. Diese Tabellen sollten alle Felder für mit @MappedSuperClass annotierte Klassen enthalten, da sie keine entsprechenden Tabellen in der Datenbank haben.
- Die Truck-Tabelle enthält Spalten, um Felder zu speichern, die von TransportationVehicle und Vehicle geerbt wurden.
- Auf ähnliche Weise verfügen Auto und Fahrrad über Spalten, um Felder zu speichern, die von Passagierfahrzeug und Fahrzeug geerbt wurden.

Ein vollständiges Beispiel finden Sie hier

Examples

Tabelle pro konkreter Klassenvererbungsstrategie

Wir werden ein Beispiel für die Fahrzeughierarchie nehmen, wie unten dargestellt.



Fahrzeug.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;
    }

}
```

Die Datenbankdarstellung wird wie folgt sein

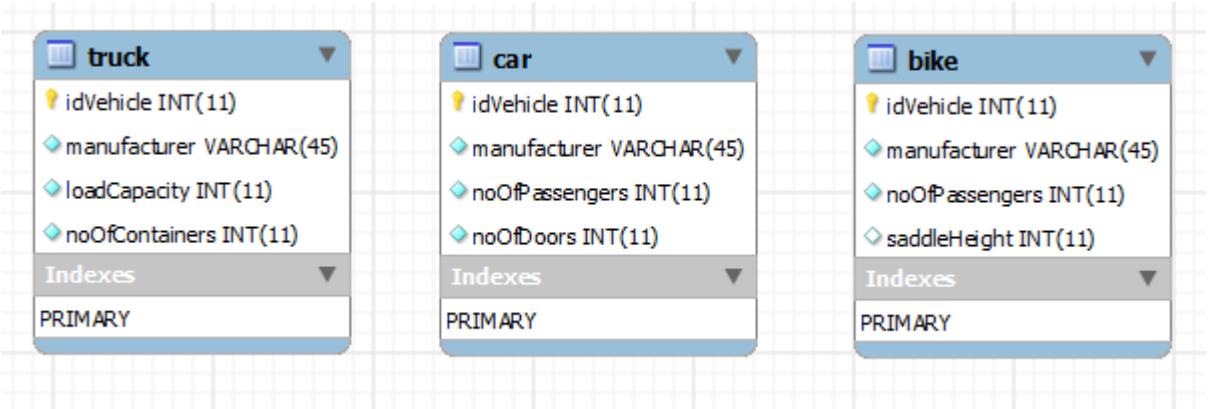


Tabelle pro konkreter Klassenvererbungsstrategie online lesen:
https://riptutorial.com/de/jpa/topic/6255/tabelle-pro-konkreter-klassenvererbungsstrategie

Kapitel 8: Verbundene Vererbungsstrategie

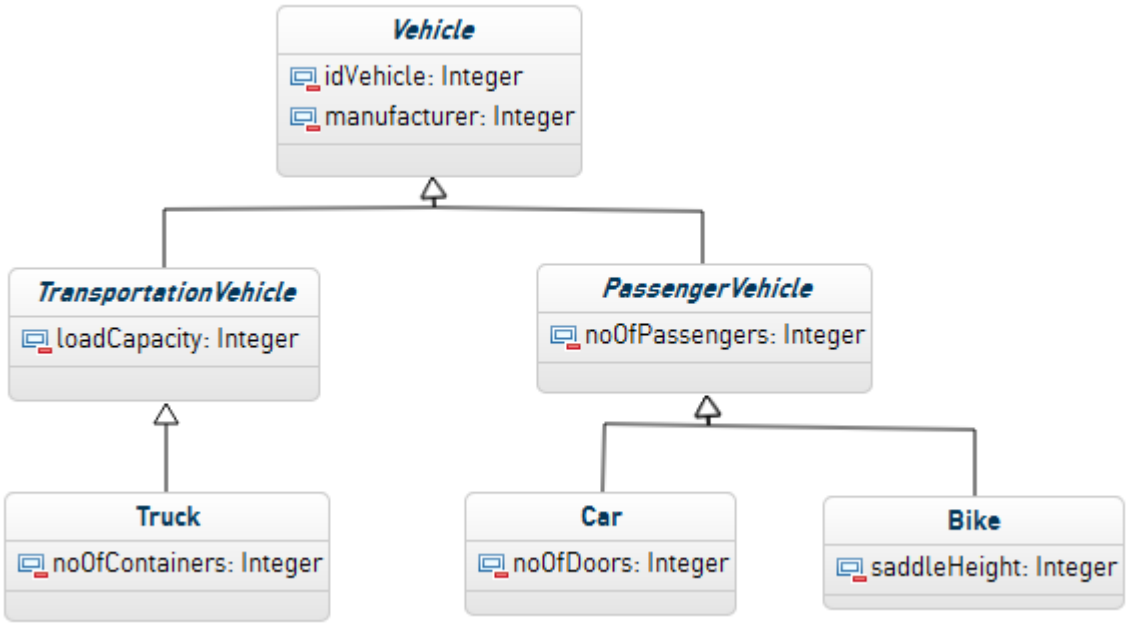
Parameter

Anmerkung	Zweck
@Erbe	Gibt den Typ der verwendeten Vererbungsstrategie an
@DiscriminatorColumn	Gibt eine Spalte in der Datenbank an, die zur Identifizierung verschiedener Entitäten verwendet wird, basierend auf einer bestimmten Entität, die jeder Entität zugeordnet ist
@MappedSuperClass	Zugeordnete Superklassen sind nicht persistent und werden nur verwendet, um den Status für ihre Unterklassen aufrechtzuerhalten. Im Allgemeinen sind abstrakte Java-Klassen mit @MapperSuperClass gekennzeichnet

Examples

Verbundene Vererbungsstrategie

Ein Beispielklassendiagramm, auf dessen Grundlage die JPA-Implementierung angezeigt wird.




```
@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

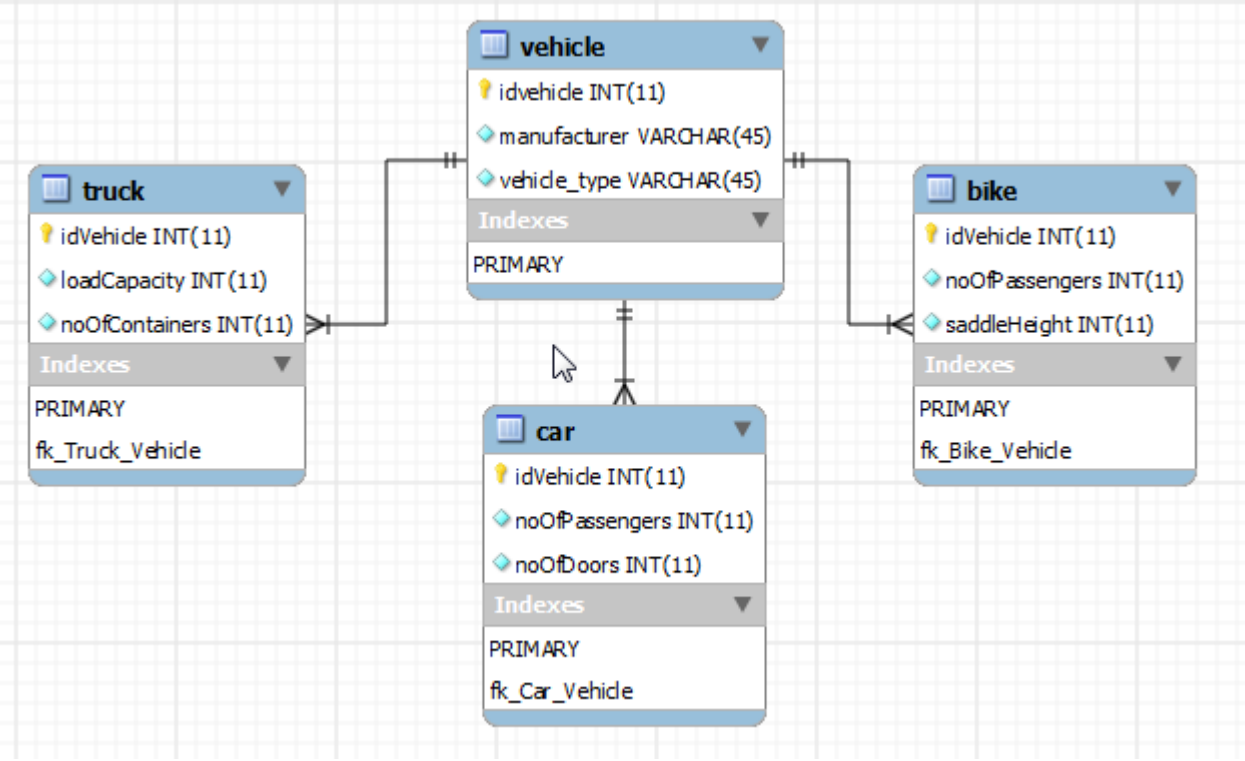
```
@Entity
public class Bike extends PassengerVehicle {
```

```
    private int saddleHeight;

    // getters and setters

}
```

Testcode

```
/* 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();
```

Datenbankdiagramm wäre wie folgt.



Der Vorteil der Strategie der verknüpften Vererbung besteht darin, dass sie keinen Datenbankspeicherplatz verschwendet wie bei einer Einzeltabellenstrategie. Auf der anderen Seite wird die Leistung aufgrund mehrerer Verknüpfungen für jedes Einfügen und Abrufen zu einem Problem, wenn Vererbungshierarchien groß und tief werden.

Vollständiges Beispiel mit Erklärung kann hier gelesen werden

Verbundene Vererbungsstrategie online lesen:
https://riptutorial.com/de/jpa/topic/6473/verbundene-vererbungsstrategie

Kapitel 9: Viele zu einer Zuordnung

Parameter

Säule	Säule
@TableGenerator	Verwendet eine Tabellengeneratorstrategie für die automatische ID-Erstellung
@GeneratedValue	Gibt an, dass der auf Felder angewendete Wert ein generierter Wert ist
@Ich würde	Annotiert das Feld als Bezeichner
@ManyToOne	Gibt eine Viele-zu-Eins-Beziehung zwischen Mitarbeiter und Abteilung an. Diese Anmerkung ist auf vielen Seiten markiert. dh mehrere Mitarbeiter gehören zu einer Abteilung. Die Abteilung wird daher mit @ManyToOne in der Entität Employee kommentiert.
@JoinColumn	Gibt die Datenbanktabellenspalte an, in der der Fremdschlüssel für die zugehörige Entität gespeichert wird

Examples

Beziehung zwischen Mitarbeiter und Abteilung ManyToOne

Mitarbeiterentität

```
@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
}
```

Abteilungsentität

```
@Entity
public class Department {

    @Id
    private int iddepartment;
    private String name;

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

Testklasse

```
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();

    }

}
```

Viele zu einer Zuordnung online lesen: https://riptutorial.com/de/jpa/topic/6531/viele-zu-einer-zuordnung

Kapitel 10: Viele zu viele Karten

Einführung

Ein `ManyToMany` Mapping beschreibt eine Beziehung zwischen Entitäten, bei denen beide mehr als eine Instanz voneinander betreffen können, und wird durch die Annotation `@ManyToMany` definiert.

Im Gegensatz zu `@OneToMany` wo eine Fremdschlüsselspalte in der Entitätstabelle verwendet werden kann, benötigt `ManyToMany` eine Join-Tabelle, die die Entitäten einander zuordnet.

Parameter

Anmerkung	Zweck
`@TableGenerator`	Definiert einen Primärschlüsselgenerator, auf den anhand des Namens verwiesen werden kann, wenn ein Generatorelement für die `GeneratedValue` Annotation angegeben wird
`@GeneratedValue`	Ermöglicht die Angabe von Generierungsstrategien für die Werte von Primärschlüsseln. Sie kann in Verbindung mit der Id-Annotation auf eine Primärschlüsseleigenschaft oder ein Feld einer Entität oder einer zugeordneten Superklasse angewendet werden.
`@ManyToMany`	Gibt die Beziehung zwischen Employee- und Project-Entitäten an, sodass viele Mitarbeiter an mehreren Projekten arbeiten können.
`mappedBy="projects"`	Definiert eine bidirektionale Beziehung zwischen Mitarbeiter und Projekt
`@JoinColumn`	Gibt den Namen der Spalte an, die sich auf die Entität bezieht, die als Eigentümer der Zuordnung betrachtet wird
`@JoinTable`	Gibt die Tabelle in der Datenbank an, in der Mitarbeiter mithilfe von Fremdschlüsseln projektiert werden

Bemerkungen

- @TableGenerator und @GeneratedValue werden für die automatische ID-Erstellung mit dem jpa-Tabellengenerator verwendet.
- Die Annotation @ManyToMany gibt die Beziehung zwischen Entitäten Employee und Project an.
- @JoinTable gibt den Namen der Tabelle an, die als Join-Tabelle verwendet werden soll. Viele Zuordnungen zwischen Employee und Project mit name = "employee_project". Dies geschieht, da es keine Möglichkeit gibt, den Besitz einer Jpa-Many-To-Mapping-Zuordnung zu bestimmen, da die Datenbanktabellen keine Fremdschlüssel enthalten, die auf andere

Tabellen verweisen.

- @JoinColumn gibt den Namen der Spalte an, die sich auf die Entität bezieht, die als Eigentümer der Zuordnung betrachtet wird, während @inverseJoinColumn den Namen der inversen Seite der Beziehung angibt. (Sie können eine beliebige Seite als Eigentümer auswählen. Stellen Sie nur sicher, dass diese Seiten in Beziehung stehen). In unserem Fall haben wir Employee als Eigentümer ausgewählt, sodass @JoinColumn auf die idemployee-Spalte in der join-Tabelle employee_project und @InverseJoinColumn auf idproject verweist, das eine inverse Seite von jpa viele in viele darstellt.
- Die @ManyToMany-Annotation in der Projektentität zeigt eine umgekehrte Beziehung. Daher verwendet sie mappedBy = -Projekte, um auf das Feld in der Mitarbeiterentität zu verweisen.

Ein vollständiges Beispiel kann hier herangezogen werden

Examples

Mitarbeiter projizieren viele zu viele Mapping

Mitarbeiter-Entität

```
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;
    }

}
```

Projekteinheit:

```
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;
    }

}
```

Testcode

/ * EntityManagerFactory erstellen * / EntityManagerFactory emf = Persistenz .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();
```

Umgang mit zusammengesetztem Schlüssel ohne eingebettete Annotation

Wenn Sie haben

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


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

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

In obigem Szenario hat die `rightrole` Tabelle einen zusammengesetzten Schlüssel und für den Zugriff auf JPA muss der Benutzer eine Entität mit `Embeddable` Annotation erstellen.

So was:

Entität für Rightrole-Tabelle ist:

```
    @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;

   .....

}
```

Die eingebettete Annotation ist für einzelne Objekte geeignet, führt jedoch beim Einfügen von Massendatensätzen zu einem Problem.

Das Problem ist , wenn Benutzer neue erstellen möchten `role` mit `rights` dann erste Nutzer müssen `store(persist) role` Objekt und dann Benutzer zu tun haben , `flush` neu erzeugten zu bekommen `id` für Rolle. dann kann der Benutzer es in das Objekt der `rightrole` Instanz `rightrole` .

Um dieses Problem zu lösen, kann der Benutzer etwas anders schreiben.

Entität für Rollentabelle ist:

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

Die Annotation @JoinTable sorgt für das Einfügen in die `rightrole` Tabelle auch ohne Entität (sofern diese Tabelle nur die id-Spalten der Rolle und des Rechtes hat).

Der Benutzer kann dann einfach:

```
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);
```

Sie müssen @ManyToMany (cascade = {CascadeType.PERSIST}) in inverseJoinColumns schreiben. Andernfalls werden Ihre übergeordneten Daten gelöscht, wenn das untergeordnete Element gelöscht wird.

Viele zu viele Karten online lesen: https://riptutorial.com/de/jpa/topic/6532/viele-zu-viele-karten

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit jpa	Billy Frost, Community, DimaSan, , Manuel Spigolon, Michael Piefel, Neil Stockton, ppeterka
2	Beziehungen zwischen Entitäten	DimaSan,
3	Eine zu viele Beziehung	Prasad Kharkar
4	Eins zu Eins Mapping	Prasad Kharkar
5	Grundlegende Zuordnung	Jeffrey Brett Coleman, Michael Piefel, Neil Stockton, Pete
6	Strategie für die Vererbung einzelner Tabellen	Prasad Kharkar
7	Tabelle pro konkreter Klassenvererbungsstrategie	Prasad Kharkar
8	Verbundene Vererbungsstrategie	bw_üezi, Prasad Kharkar
9	Viele zu einer Zuordnung	Prasad Kharkar
10	Viele zu viele Karten	Prasad Kharkar, Ronak Patel, Vetle