Бесплатная электронная книга

учусь

jpa

Free unaffiliated eBook created from
Stack Overflow contributors.

#jpa

Около

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

глава 1: Начало работы с jpa

замечания

JPA - это Java Persistence API, спецификация, обрабатывающая сопоставление объектов Java и их отношений с реляционной базой данных. Это называется объектно-реляционным картографом (ORM). Это альтернатива (или дополнение) к более низкоуровневому JDBC . Это наиболее полезно при реализации подхода, ориентированного на Java, и когда необходимо сохранить сложные графические объекты.

JPA сама по себе не является реализацией. Для этого вам понадобится постоянный провайдер (см. Примеры). Текущими реализациями последнего стандарта JPA 2.1 являются EclipseLink (также эталонная реализация для JPA 2.1, что означает «доказательство того, что спецификация может быть реализована»); Hibernate и DataNucleus .

Метаданные

Отображение между объектами Java и таблицами базы данных определяется с помощью **метаданных сохранения** . Поставщик JPA будет использовать информацию метаданных настойчивости для выполнения правильных операций с базой данных. JPA обычно определяет метаданные через аннотации в классе Java.

Архитектура объектно-реляционной сущности

Архитектура сущности состоит из:

- юридические лица
- единицы персистентности
- контексты персистентности
- заводы-застройщики
- руководители подразделений

Версии

Версия	Группа экспертов	Релиз
1,0	JSR-220	2006-11-06
2,0	JSR-317	2009-12-10
2,1	JSR-338	2013-05-22

Examples

Установка или настройка

Требования к классу

EclipseLink

Должны быть включены API Eclipselink и JPA. Примеры зависимостей 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>
```

зимовать

Требуется спящий стержень. Пример зависимости 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

datanucleus-core, datanucleus-api-jpa и datanucleus-rdbms (при использовании хранилищ данных РСУБД). Пример зависимости 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>
```

Детали конфигурации

JPA требует использования файла *persistence.xml* , расположенного в `META-INF` из корня CLASSPATH. Этот файл содержит определение доступных единиц выживаемости, из которых может работать JPA.

JPA дополнительно позволяет использовать файл конфигурации сопоставления *orm.xml* , также помещенный в `META-INF` . Этот файл сопоставления используется для настройки того, как классы сопоставляются с хранилищем данных, и является альтернативой / дополнением к использованию аннотаций Java в самих классах JPA.

Минимальный пример persistence.xml

Hibernate (и встроенная H2 DB)

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

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

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

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

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

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

Eclipselink (и встроенная H2 DB)

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

<persistence-unit name="persistenceUnit">
    <provider>org.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 (и встроенная H2 DB)

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

<persistence-unit name="persistenceUnit">
    <provider>org.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>
```

Привет, мир

Давайте посмотрим на весь базовый компонент для создания простого Hallo World.

1. Определите, какую реализацию JPA 2.1 мы будем использовать
2. Постройте соединение с базой данных, создавая `persistence-unit`
3. Осуществляет объекты
4. Осуществляет DAO (объект доступа к данным) для управления объектами
5. Проверить приложение

Библиотеки

Используя maven, нам нужны эти зависимости:

```
<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>
```

Единица сохранения

В папке ресурсов нам нужно создать файл с именем `persistence.xml` . Самый простой способ определить это так:

```
<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>
```

Внедрить сущность

Я создаю класс `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;
    }
}
```

Внедрение 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` - **СИНГЛТОН:**

```
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;
    }
}
```

Проверить приложение

пакет it.hello.jpa.test;

открытый класс 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));
}
```

}

Выход будет:

> Запуск it.hello.jpa.test.TestJpa Hibernate: drop table BIKER if exists Hibernate: drop sequence if exists hibernate_sequence Hibernate: создать последовательность hibernate_sequence начать с 1 приращения на 1 Спящий режим: создать таблицу BIKER (id bigint not null, battleName varchar (255)), beard boolean, дата рождения, bikerName varchar (255), registrationDate time, primary key (id)) Hibernate: изменить таблицу BIKER добавить ограничение UK_a64ce28nywyk8wqrvfkkuapli unique (battleName) Спящий режим: включить в BIKER (battleName, борода, день рождения, bikerName, registrationDate , id) значения (?,?,?,?,?,?) mar 01, 2017 11:00:02 PM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation INFO: HHH000204: Обработка PersistenceUnitInfo [имя: hello- jpa-pu ...] Результаты:
>
> Пробеги: 1, Сбой: 0, Ошибки: 0, Пропущено: 0

Прочитайте Начало работы с jpa онлайн: https://riptutorial.com/ru/jpa/topic/2125/начало-работы-с-jpa

глава 2: Множество карт

параметры

колонка	колонка
@TableGenerator	Использует стратегию генератора таблиц для автоматического создания идентификатора
@GeneratedValue	Указывает, что значение, применяемое к полям, является сгенерированным значением
@Я бы	Аннотирует поле как идентификатор
@ManyToOne	Определяет отношения между многими сотрудниками и отделом. Эта аннотация отмечена со многих сторон. т.е. несколько сотрудников принадлежат к одному отделу. Таким образом, отдел аннотируется с @ManyToOne в сущности Employee.
@JoinColumn	Задает столбец таблицы базы данных, в котором хранится внешний ключ для связанного объекта

Examples

Сотрудник отдела Многопользовательские отношения

Сотрудник

```
@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
}
```

Департамент

```
@Entity
public class Department {

    @Id
    private int iddepartment;
    private String name;

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

Класс тестирования

```
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();

    }

}
```

Прочитайте Множество карт онлайн: https://riptutorial.com/ru/jpa/topic/6531/множество-карт

глава 3: Основное картографирование

параметры

аннотирование	подробности
`@Id`	Пометяет поле / столбец как *ключ* объекта
`@Basic`	Помечает поле для сопоставления как *базового* типа. Это применимо к примитивным типам и их оберткам, `String` , `Date` и `Calendar` . Аннотации фактически необязательны, если параметры не заданы, но хороший стиль будет диктовать ваши намерения в явном виде.
`@Transient`	Поля, отмеченные как переходные, не рассматриваются для сохранения, как и `transient` ключевое слово для сериализации.

замечания

Всегда должен быть конструктор по умолчанию, то есть без параметров. В базовом примере не было указателя конструктора, поэтому Java добавил один; но если вы добавите конструктор с аргументами, обязательно добавьте конструктор без параметров.

Examples

Очень простой объект

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

    @Basic
    String note;

    @Basic
    int count;
}
```

Getters, сеттеры и т. Д. Опущены для краткости, но в любом случае они не нужны для JPA.

Этот Java-класс будет отображаться в следующей таблице (в зависимости от вашей базы данных, приведенной в одном возможном сопоставлении Postgres):

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

Поставщики JPA могут использоваться для генерации DDL и, скорее всего, производят DDL, отличный от показанного здесь, но пока типы совместимы, это не вызовет проблем во время выполнения. Лучше не полагаться на автоматическое генерирование DDL.

Опускание поля из отображения

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

    @Basic
    String note;

    @Transient
    String parsedNote;

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

Если вашему классу нужны поля, которые не должны быть записаны в базу данных, отметьте их как `@Transient` . После чтения из базы данных поле будет равно `null` .

Время и дата сопоставления

Время и дата встречаются на разных языках Java: теперь историческая `Date` и `Calendar` , а также более поздние `LocalDate` и `LocalDateTime` . `Timestamp` , `Instant` , `ZonedLocalDateTime` и типы времени Joda. На стороне базы данных у нас есть `time` , `date` и `timestamp` (время и дата), возможно с часовым поясом или без него.

Дата и время до Java 8

Сопоставление по *умолчанию* для типов pre-Java-8 `java.util.Date` , `java.util.Calendar` и `java.sql.Timestamp` - это `timestamp` в SQL; для `java.sql.Date` это `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;
}
```

Это будет идеально соответствовать следующей таблице:

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

Возможно, это не намерение. Например, часто Java `Date` или `Calendar` используется для представления только даты (для даты рождения). Чтобы изменить отображение по умолчанию или просто сделать отображение явным, вы можете использовать аннотацию `@Temporal` .

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

    @Temporal(TemporalType.TIME)
    private Date date;

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

Эквивалентная таблица SQL:

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

Примечание 1: Тип, заданный с помощью `@Temporal` влияет на генерацию DDL; но вы также можете иметь тип карты `date` типа `Date` только с аннотацией `@Basic` .

Примечание 2: `Calendar` не может сохраняться только `time` .

Дата и время с Java 8

JPA 2.1 *не* определяет поддержку типов `java.time` предоставляемых на Java 8. Большинство реализаций JPA 2.1 предлагают поддержку для этих типов, хотя это, строго говоря, расширения поставщиков.

Для DataNucleus эти типы просто работают из коробки и предлагают широкий диапазон возможностей отображения, связанных с аннотацией `@Temporal` .

Для Hibernate, если использовать Hibernate 5.2+, они должны работать из коробки, просто используя аннотацию `@Basic` . Если вы используете Hibernate 5.0-5.1, вам нужно добавить зависимость `org.hibernate:hibernate-java8` . Представленные отображения

- `LocalDate` на `date`
- `Instant` , `LocalDateTime` и `ZonedDateTime` для `timestamp`

Альтернативой, альтернативной поставщикам, также будет определение JPA 2.1 `AttributeConverter` для любого типа Java Java `java.time` который должен быть сохранен.

Объект с управляемым идентификатором Id

Здесь у нас есть класс и мы хотим, чтобы в поле идентификатора (`userUid`) было создано его значение через SEQUENCE в базе данных. Предполагается, что эта SEQUENCE называется `USER_UID_SEQ` и может быть создана администратором баз данных или может быть создана поставщиком 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;
}
```

Прочитайте Основное картографирование онлайн: https://riptutorial.com/ru/jpa/topic/3691/основное-картографирование

глава 4: От многих до многих

Вступление

Отображение `ManyToMany` описывает взаимосвязь между объектами, которые могут быть связаны с несколькими экземплярами друг друга, и определяется аннотацией `@ManyToMany` .

В отличие от `@OneToMany` где может использоваться столбец внешнего ключа в таблице объекта, `ManyToMany` требует таблицу соединений, которая сопоставляет сущности друг другу.

параметры

аннотирование	Цель
`@TableGenerator`	Определяет генератор первичных ключей, на который может ссылаться по имени, если для аннотации `GeneratedValue` указан элемент генератора
`@GeneratedValue`	Обеспечивает спецификацию стратегий генерации для значений первичных ключей. Он может применяться к свойству первичного ключа или поле объекта или сопоставленному суперклассу в сочетании с аннотацией идентификатора.
`@ManyToMany`	Определяет отношения между Employee и Project, так что многие сотрудники могут работать с несколькими проектами.
`mappedBy="projects"`	Определяет двунаправленную связь между Employee и Project
`@JoinColumn`	Указывает имя столбца, которое будет ссылаться на Объект, который будет считаться владельцем ассоциации
`@JoinTable`	Задает таблицу в базе данных, в которой сотрудник связывается с проектами с использованием внешних ключей

замечания

- @TableGenerator и @GeneratedValue используются для автоматического создания идентификатора с использованием генератора jpa-таблицы.
- Аннотации @ManyToMany определяют отношения между Employee и объектами Project.
- @JoinTable указывает имя таблицы для использования в качестве таблицы join jpa

для многих сопоставлений между Employee и Project с использованием name = "employee_project". Это делается потому, что нет способа определить право собственности на сопоставление jpa many to many, поскольку таблицы базы данных не содержат внешних ключей для ссылки на другую таблицу.

- @JoinColumn указывает имя столбца, которое будет ссылаться на Entity, которое будет считаться владельцем ассоциации, а @inverseJoinColumn указывает имя обратной стороны отношения. (Вы можете выбрать любую сторону, которая будет считаться владельцем. Просто убедитесь, что эти стороны находятся в отношениях). В нашем случае мы выбрали Employee в качестве владельца, поэтому @JoinColumn ссылается на столбец idemployee в таблице соединений employee_project, а @ InverseJoinColumn ссылается на idproject, который является обратной стороной jpa для многих сопоставлений.
- @ManyToMany аннотация в объекте Project показывает обратную связь, поэтому она использует mappedBy = projects для ссылки на поле в сущности Employee.

Полный пример можно отнести сюда

Examples

Сотрудник для сопоставления многих проектов

Сущность сотрудника.

```
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;
    }

}
```

Объект проекта:

```
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;
    }

}
```

Тестовый код

/ * Создать EntityManagerFactory * / EntityManagerFactory emf = Стойкость . 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();
```

Как обрабатывать составной ключ без встраиваемой аннотации

Если у вас есть

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


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

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

В приведенном выше сценарии таблица `rightrole` имеет составной ключ и для доступа к нему в JPA пользователю необходимо создать объект с `Embeddable` аннотацией.

Как это:

Объект для таблицы rightrole:

```
    @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;

   .....

}
```

Встраиваемая аннотация отлично подходит для одного объекта, но она будет выдавать проблему при вставке массовых записей.

Проблема заключается в том, когда пользователь хочет создать новую `role` с `rights` тогда первый пользователь должен `store(persist) role` объект, а затем пользователь должен выполнить `flush` для получения вновь созданного `id` для роли. тогда и тогда пользователь может поместить его в объект объекта `rightrole` .

Чтобы решить этот вопрос, пользователь может писать объект несколько иначе.

Объект для таблицы ролей:

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

Аннотации @JoinTable позаботятся о вставке в таблицу `rightrole` даже без сущности (пока в этой таблице есть только столбцы id роли и права).

Затем пользователь может просто:

```
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);
```

Вы должны написать @ManyToMany (cascade = {CascadeType.PERSIST}) в inverseJoinColumns, иначе ваши родительские данные будут удалены, если ребенок будет удален.

Прочитайте От многих до многих онлайн: https://riptutorial.com/ru/jpa/topic/6532/от-многих-до-многих

глава 5: От одного до большого

параметры

аннотирование	Цель
@TableGenerator	Задает имя генератора и имя таблицы, где можно найти генератор
@GeneratedValue	Определяет стратегию генерации и ссылается на имя генератора
@ManyToOne	Определяет отношения между несколькими сотрудниками и отделом
@OneToMany (mappedBy = "отдел")	создает двунаправленную связь между Employee and Department, просто обращаясь к аннотации @ManyToOne в сущности Employee

Examples

От одного до большого

Отображение от одного до многих, как правило, представляет собой просто двунаправленную связь отображения Many to One. Мы возьмем тот же пример, что и для сопоставления многого к одному.

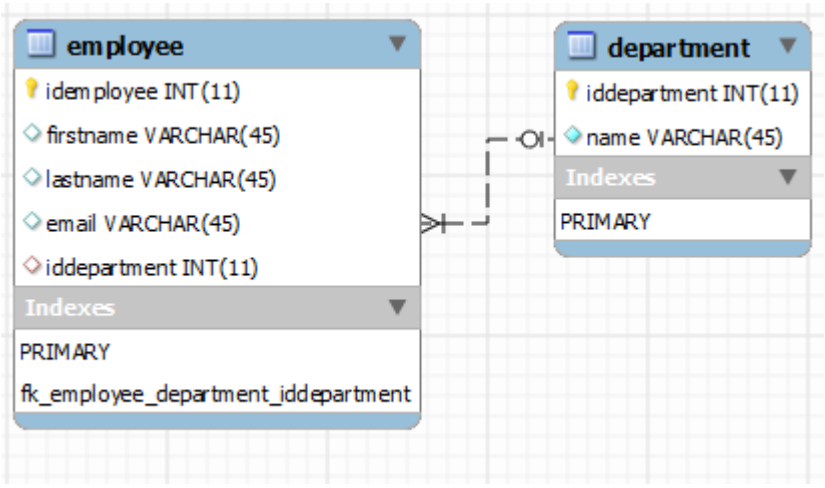
Employee.java

```
@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
}
```

Это соотношение представлено в базе данных, как показано ниже.



Есть два момента, чтобы помнить о jpa one to many mapping:

- Многосторонняя сторона - это собственная сторона отношений. Столбец определен с этой стороны.
- Отображение от одного к другому - это обратная сторона стороны, поэтому элемент mappedBy должен использоваться на обратной стороне.

Полный пример можно назвать здесь

Прочитайте От одного до большого онлайн: https://riptutorial.com/ru/jpa/topic/6529/от-одного-до-большого

глава 6: От одного к одному

параметры

аннотирование	Цель
@TableGenerator	Задает имя генератора и имя таблицы, где можно найти генератор
@GeneratedValue	Определяет стратегию генерации и ссылается на имя генератора
@Один к одному	Определяет отношения один к одному между сотрудником и рабочим столом, здесь Employee является владельцем отношения
mappedBy	Этот элемент предоставляется на обратной стороне отношения. Это позволяет использовать двунаправленную связь

Examples

Отношение «один к одному» между сотрудником и столом

Рассмотрим взаимно однозначные отношения между сотрудником и рабочим столом.

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

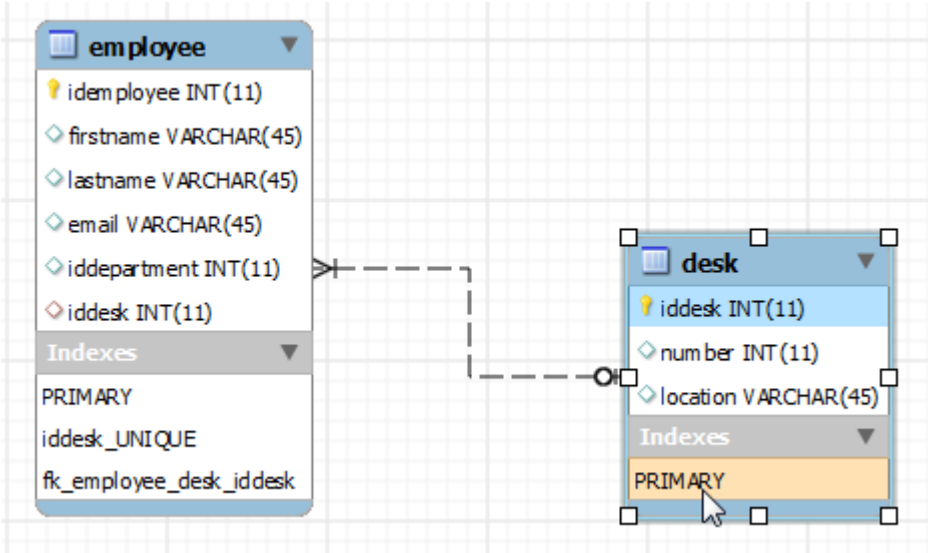
```
@Entity
public class Desk {
```

```
    @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
}
```

Тестовый код

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

Диаграмма базы данных изображена ниже.



- Аннотации **@JoinColumn** переходят к отображению объекта, который

сопоставляется с таблицей, содержащей объединение colulmn. Владелец отношений. В нашем случае таблица Employee имеет столбец join, поэтому **@JoinColumn** находится в поле «Рабочий стол» объекта Employee.

- Элемент **mappedBy** должен быть указан в ассоциации **@OneToOne** в объекте, обратном стороне отношения. т.е. объект, который не предоставляет столбец соединения по аспекту базы данных. В нашем случае Desk является обратным объектом.

Полный пример можно найти здесь

Прочитайте От одного к одному онлайн: https://riptutorial.com/ru/jpa/topic/6474/от-одного-к-одному

глава 7: Отношения между субъектами

замечания

Отношения между сущностями

Внешний ключ может быть одним или несколькими столбцами, которые ссылаются на уникальный ключ, обычно первичный ключ, в другой таблице.

Внешний ключ и основной родительский ключ, на которые он ссылается, должны иметь одинаковое количество и тип полей.

Внешние ключи представляют **отношения** из столбца или столбцов в одной таблице столбцу или столбцам в другой таблице.

Examples

Множественность сущностных отношений

Множественность сущностных отношений

Множества имеют следующие типы:

- **Индивидуально** : каждый экземпляр объекта связан с одним экземпляром другого объекта.
- **Один-ко-многим** : экземпляр объекта может быть связан с несколькими экземплярами других объектов.
- **«Много-к-одному»** : несколько экземпляров объекта могут быть связаны с одним экземпляром другого объекта.
- **Многие-ко-многим** : экземпляры объектов могут быть связаны с несколькими экземплярами друг друга.

Индивидуальное сопоставление

Индивидуальное сопоставление определяет однозначную связь с другой сущностью, которая имеет множественность один-к-одному. Это сопоставление отношений использует аннотацию `@OneToOne` для соответствующего постоянного свойства или поля.

Пример: объекты `Vehicle` *and* `ParkingPlace` *.*

Сопоставление «один ко многим»

Экземпляр объекта может быть связан с несколькими экземплярами других объектов.

Соотношения «один ко многим» используют аннотацию `@OneToMany` в соответствующем постоянном свойстве или поле.

Элемент `mappedBy` необходим для ссылки на атрибут, аннотированный ManyToOne в соответствующем объекте:

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

Ассоциация «один ко многим» должна отображать коллекцию объектов.

Сопоставление «много-к-одному»

`@ManyToOne` сопоставление определяется путем аннотации атрибута в исходном объекте (атрибуте, который относится к целевому объекту) с аннотацией `@ManyToOne` .

`@JoinColumn(name="FK_name")` описывают ключ для взаимодействия.

Отображение многих ко многим

Экземпляры сущности могут быть связаны с несколькими экземплярами друг друга.

`@ManyToMany` Many-to-many используют аннотацию `@ManyToMany` в соответствующем постоянном свойстве или поле.

Мы должны использовать третью таблицу для сопоставления двух типов сущностей (join table).

@JoinTable Пример аннотации

При сопоставлении отношений «многие ко многим» в JPA конфигурация таблицы, используемой для соединения внешних ключей, может быть предоставлена с `@JoinTable` аннотации `@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;
   [...]
}
```

В этом примере, который состоит из EntityA , имеющие много-ко-многим отношения к EntityB, осуществлянного `entitiesB` области, мы используем @JoinTable аннотации , чтобы указать , что имя таблицы для присоединения таблицы `table_join_A_B` , состоящее из колонн `id_A` и `id_B` , внешние ключи, соответственно ссылающиеся на `id` столбца в таблице EntityA и в таблице EntityB; `(id_A,id_B)` будет составным первичным ключом для таблицы `table_join_A_B` .

Прочитайте Отношения между субъектами онлайн: https://riptutorial.com/ru/jpa/topic/6305/отношения-между-субъектами

глава 8: Стратегия наследования

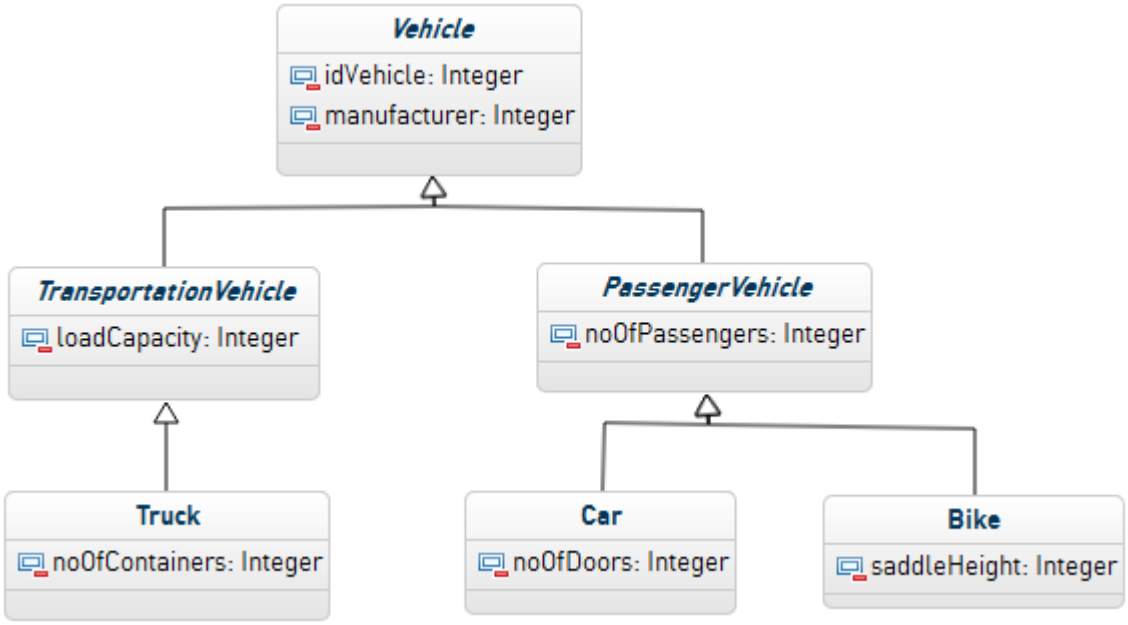
параметры

аннотирование	Цель
@Inheritance	Указывает тип используемой стратегии наследования
@DiscriminatorColumn	Задает столбец в базе данных, который будет использоваться для идентификации разных объектов на основе определенного идентификатора, назначенного каждому объекту
@MappedSuperClass	сопоставленные суперклассы не являются постоянными и используются только для хранения состояния для своих подклассов. Обычно абстрактные классы Java помечены знаком @MapperSuperClass

Examples

Стратегия наследования

Пример диаграммы классов, на основе которой мы увидим реализацию 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
}
```

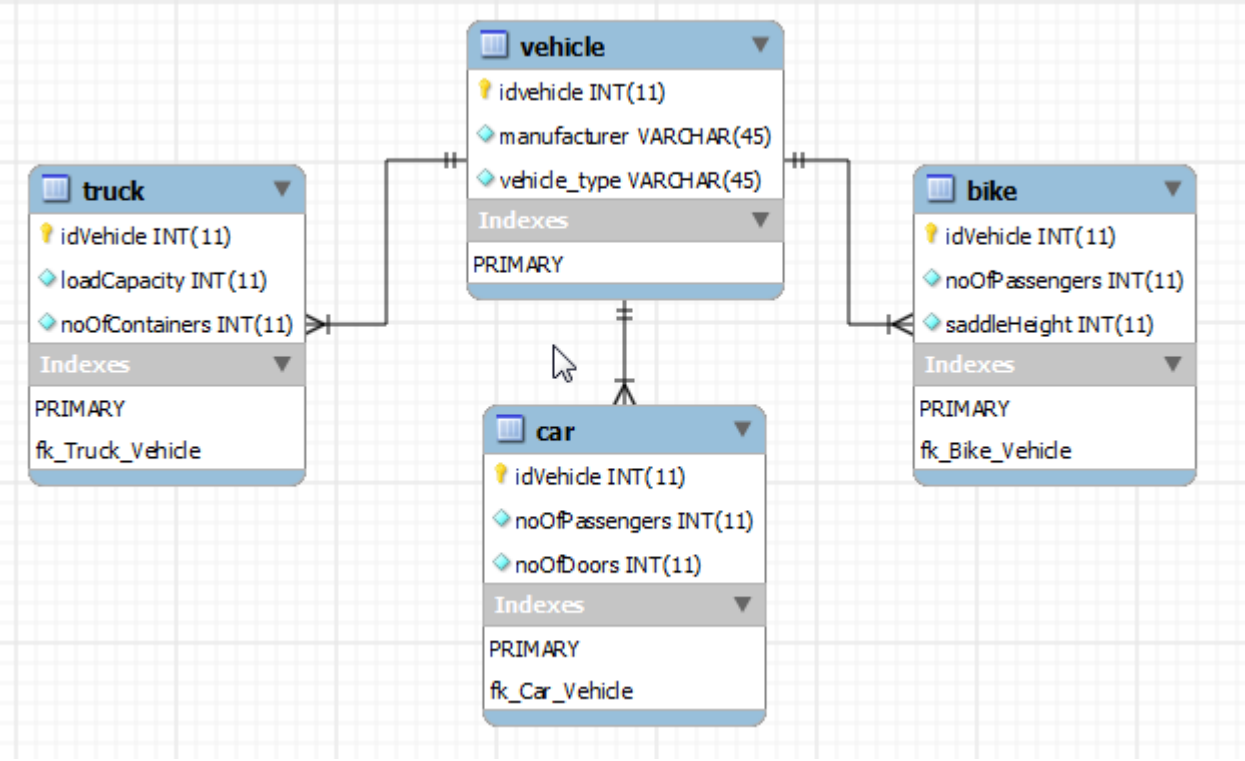
Bike.java

```
@Entity
public class Bike extends PassengerVehicle {

    private int saddleHeight;

    // getters and setters

}
```

Тестовый код

```
/* 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();
```

Диаграмма базы данных будет выглядеть следующим образом.



Преимущество объединенной стратегии наследования заключается в том, что он не тратит пространство базы данных, как в стратегии с одной таблицей. С другой стороны, из-за множественных объединений, задействованных для каждой вставки и извлечения, производительность становится проблемой, когда иерархии наследования становятся широкими и глубокими.

Полный пример с объяснением можно прочитать здесь

Прочитайте Стратегия наследования онлайн: https://riptutorial.com/ru/jpa/topic/6473/стратегия-наследования

глава 9: Стратегия наследования на одном столе

параметры

аннотирование	Цель
@Inheritance	Указывает тип используемой стратегии наследования
@DiscriminatorColumn	Задает столбец в базе данных, который будет использоваться для идентификации разных объектов на основе определенного идентификатора, назначенного каждому объекту
@MappedSuperClass	сопоставленные суперклассы не являются постоянными и используются только для хранения состояния для своих подклассов. Обычно абстрактные классы Java помечены знаком @MapperSuperClass
@DiscriminatorValue	Значение, указанное в столбце, определяемом @ DiscriminatorColumn. Это значение помогает идентифицировать тип объекта

замечания

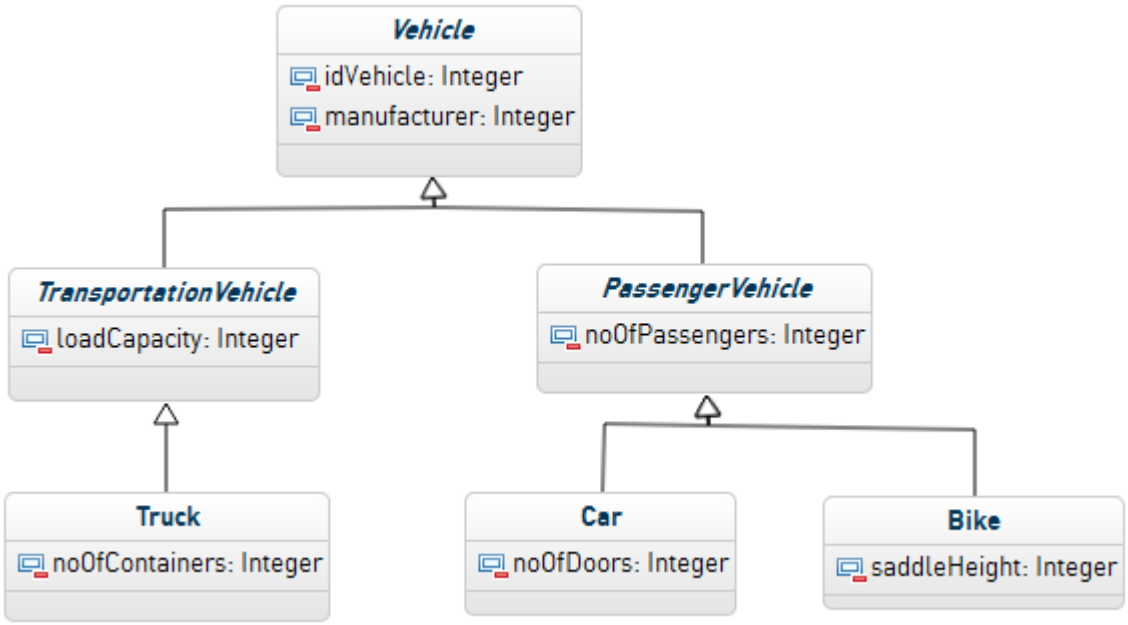
Преимущество стратегии одиночной таблицы заключается в том, что она не требует сложных объединений для извлечения и вставки сущностей, но, с другой стороны, она уничтожает пространство базы данных, так как многие столбцы должны быть обнуляемыми и для них нет данных.

Полный пример и статья можно найти здесь

Examples

Стратегия наследования с одиночной таблицей

Простой пример иерархии транспортных средств можно использовать для стратегии однонаправленного наследования.



Аннотация Класс автомобиля:

```
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;
    }

}
```

Транспортируемый пакет Vehicle.java 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;
    }

}
```

Тестовый код:

```
/* 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();
```

Прочитайте Стратегия наследования на одном столе онлайн:
https://riptutorial.com/ru/jpa/topic/6277/стратегия-наследования-на-одном-столе

глава 10: Таблица на конкретную стратегию наследования класса

замечания

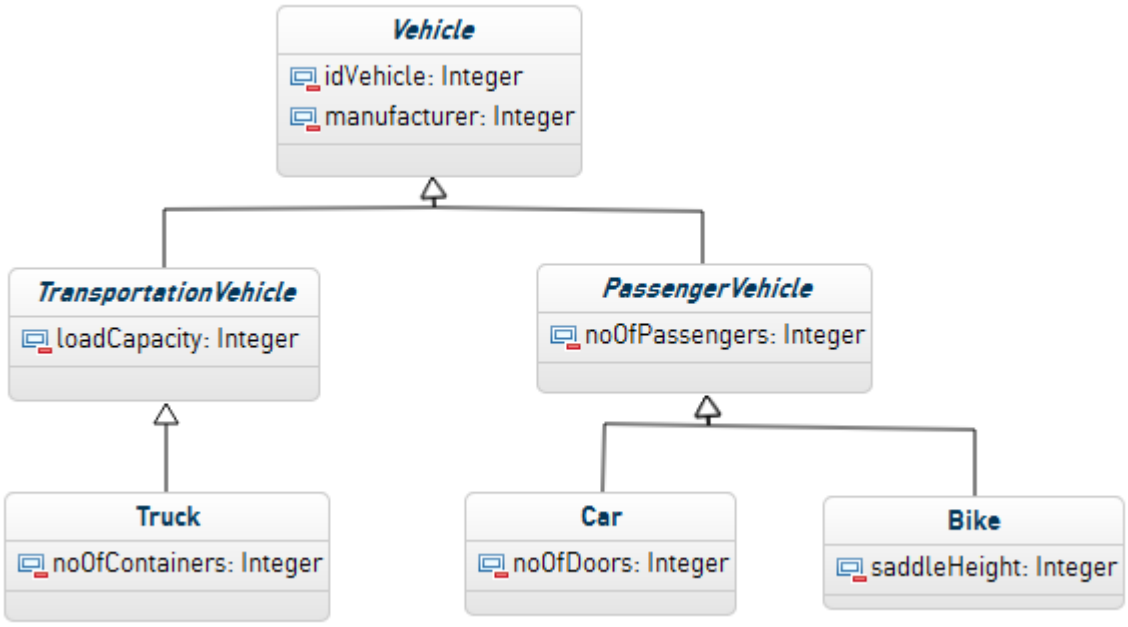
- Транспортное средство, транспортный транспорт и пассажирский автомобиль являются абстрактными классами, и у них не будет отдельной таблицы в базе данных.
- Грузовик, автомобиль и велосипед - это конкретные классы, поэтому они будут сопоставлены с соответствующими таблицами. Эти таблицы должны включать все поля для классов, аннотированных с помощью @MappedSuperClass, потому что у них нет соответствующих таблиц в базе данных.
- Таким образом, стол Truck будет иметь столбцы для хранения полей, унаследованных от TransportVehicle и Vehicle.
- Аналогичным образом, Car and Bike будет иметь столбцы для хранения полей, унаследованных от PassengerVehicle и Vehicle.

Полный пример можно найти здесь

Examples

Таблица на конкретную стратегию наследования класса

Мы возьмем пример иерархии транспортных средств, как показано ниже.



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

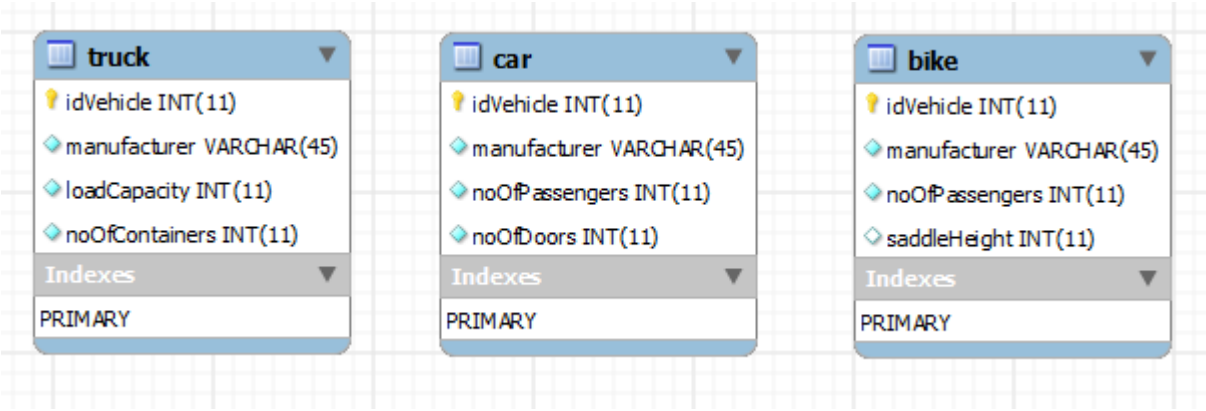
```
package com.thejavageek.jpa.entities;

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;
    }

}
```

Представление базы данных будет следующим:



Прочитайте Таблица на конкретную стратегию наследования класса онлайн: https://riptutorial.com/ru/jpa/topic/6255/таблица-на-конкретную-стратегию-наследования-класса

кредиты

S. No	Главы	Contributors
1	Начало работы с jpa	Billy Frost, Community, DimaSan, , Manuel Spigolon, Michael Piefel, Neil Stockton, ppeterka
2	Множество карт	Prasad Kharkar
3	Основное картографирование	Jeffrey Brett Coleman, Michael Piefel, Neil Stockton, Pete
4	От многих до многих	Prasad Kharkar, Ronak Patel, Vetle
5	От одного до большого	Prasad Kharkar
6	От одного к одному	Prasad Kharkar
7	Отношения между субъектами	DimaSan,
8	Стратегия наследования	bw_üezi, Prasad Kharkar
9	Стратегия наследования на одном столе	Prasad Kharkar
10	Таблица на конкретную стратегию наследования класса	Prasad Kharkar