FREE eBook

LEARNING

jpa

Free unaffiliated eBook created from
Stack Overflow contributors.

#jpa

Table of Contents

About

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

Chapter 1: Getting started with jpa

Remarks

JPA is the Java Persistence API, a specification handling the mapping of Java objects and their relationships to a relational database. This is called an object-relational mapper (ORM). It is an alternative for (or supplement to) the more low-level JDBC. It is most useful when pursuing a Java-oriented approach and when complex object graphs need to be persisted.

JPA in itself is not an implementation. You will need a persistence provider for that (see examples). Current implementations of the latest JPA 2.1 standard are EclipseLink (also the reference implementation for JPA 2.1, which means "proof that the spec can be implemented"); Hibernate, and DataNucleus.

Metadata

The mapping between Java objects and database tables is defined via **persistence metadata**. The JPA provider will use the persistence metadata information to perform the correct database operations. JPA typically defines the metadata via annotations in the Java class.

Object-Relational Entity Architecture

The entity architecture is composed of:

- entities
- persistence units
- persistence contexts
- entity manager factories
- entity managers

Versions

Version	Expert Group	Release
1.0	JSR-220	2006-11-06
2.0	JSR-317	2009-12-10
2.1	JSR-338	2013-05-22

Examples

Installation or Setup

Classpath requirements

Eclipselink

The Eclipselink and JPA API need to be included. Example Maven dependencies:

```
<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>
```

Hibernate

Hibernate-core is required. Example Maven dependency:

```
<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 and datanucleus-rdbms (when using RDBMS datastores) are required. Example Maven dependency:

```
<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>
```

Configuration Details

JPA requires the use of a file *persistence.xml*, located under `META-INF` from the root of the CLASSPATH. This file contains a definition of the available persistence units from which JPA can operate.

JPA additionally allows use of a mapping configuration file *orm.xml*, also placed under `META-INF`. This mapping file is used to configure how classes are mapped to the datastore, and is an alternative/supplement to use of Java annotations in the JPA entity classes themselves.

Minimal persistence.xml example

Hibernate (and embedded 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 (and embedded 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 (and embedded 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>
```

Hello World

Let's see all the basic component for create a simple Hallo World.

1. Define which implementation of JPA 2.1 we will use
2. Build the connection to database creating the `persistence-unit`
3. Implements the entities
4. Implements DAO (Data access object) for manipulate the entities
5. Test the application

Libraries

Using maven, we need this dependancies:

```
<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>
```

Persistence Unit

In the resources folder we need to create a file called `persistence.xml`. The easiest way for define it is like this:

```
<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>
```

Implement an Entity

I create a class `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;
    }
}
```

Implement a 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`

is a 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;
    }
}
```

Test the application

package it.hello.jpa.test;

public class 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));
}
```

}

The output will be:

Running it.hello.jpa.test.TestJpa Hibernate: drop table BIKER if exists Hibernate: drop

sequence if exists hibernate_sequence Hibernate: create sequence hibernate_sequence start with 1 increment by 1 Hibernate: create table BIKER (id bigint not null, battleName varchar(255), beard boolean, birthday date, bikerName varchar(255), registrationDate time, primary key (id)) Hibernate: alter table BIKER add constraint UK_a64ce28nywyk8wqrvfkkuapli unique (battleName) Hibernate: insert into BIKER (battleName, beard, birthday, bikerName, registrationDate, id) values (?, ?, ?, ?, ?, ?) mar 01, 2017 11:00:02 PM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation INFO: HHH000204: Processing PersistenceUnitInfo [name: hello-jpa-pu ...] Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

Read Getting started with jpa online: https://riptutorial.com/jpa/topic/2125/getting-started-with-jpa

Chapter 2: Basic mapping

Parameters

Annotation	Details
`@Id`	Marks field/column as the *key* of the entity
`@Basic`	Marks requested field to mapped as a *basic* type. This is applicable to primitive types and their wrappers, `String`, `Date` and `Calendar`. The annotation is actually optional if no parameters are given, but good style would dictate to make your intentions explicit.
`@Transient`	Fields marked as transient are not considered for persistence, much like the `transient` keyword for serialization.

Remarks

There always needs to be a default constructor, that is, the parameterless one. In the basic example, there was no constructor specified, so Java added one; but if you add a constructor with arguments, be sure to add the parameterless constructor, too.

Examples

A very simple entity

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

    @Basic
    String note;

    @Basic
    int count;
}
```

Getters, setters etc. are ommitted for brevity, but they are not needed for JPA anyway.

This Java class would map to the following table (depending on your database, here given in one possible Postgres mapping):

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

```
)
```

JPA providers may be used to generate the DDL, and will likely produce DDL different from the one shown here, but as long as the types are compatible, this will not cause problems at runtime. It is best not to rely on auto-generation of DDL.

Omitting field from the mapping

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

    @Basic
    String note;

    @Transient
    String parsedNote;

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

If your class needs fields that should not be written to the database, mark them as `@Transient`. After reading from the database, the field will be `null`.

Mapping time and date

Time and date come in a number of different types in Java: The now historic `Date` and `Calendar`, and the more recent `LocalDate` and `LocalDateTime`. And `Timestamp`, `Instant`, `ZonedLocalDateTime` and the Joda-time types. On the database side, we have `time`, `date` and `timestamp` (both time and date), possibly with or without time zone.

Date and time before Java 8

The *default* mapping for the pre-Java-8 types `java.util.Date`, `java.util.Calendar` and `java.sql.Timestamp` is `timestamp` in SQL; for `java.sql.Date` it is `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;
}
```

This will map perfectly to the following table:

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

This may not be the intention. For instance, often a Java `Date` or `Calendar` is used to represent the date only (for date of birth). To change the default mapping, or just to make the mapping explicit, you can use the `@Temporal` annotation.

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

    @Temporal(TemporalType.TIME)
    private Date date;

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

The equivalent SQL table is:

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

Note 1: The type specified with `@Temporal` influences DDL generation; but you can also have a colum of type `date` map to `Date` with just the `@Basic` annotation.

Note 2: `Calendar` cannot persist `time` only.

Date and time with Java 8

JPA 2.1 does *not* define support for `java.time` types provided in Java 8. The majority of JPA 2.1 implementations offer support for these types however, though these are strictly speaking vendor extensions.

For DataNucleus, these types just work out of the box, and offers a wide range of mapping possibilities, coupling in with the `@Temporal` annotation.

For Hibernate, if using Hibernate 5.2+ they should work out of the box, just using the `@Basic` annotation. If using Hibernate 5.0-5.1 you need to add the dependency `org.hibernate:hibernate-java8`. The mappings provided are

- `LocalDate` to `date`
- `Instant`, `LocalDateTime` and `ZonedDateTime` to `timestamp`

A vendor-neutral alternative would also be to define a JPA 2.1 `AttributeConverter` for any Java 8 `java.time` type that is required to be persisted.

Entity with sequence managed Id

Here we have a class and we want the identity field (`userUid`) to have its value generated via a SEQUENCE in the database. This SEQUENCE is assumed to be called `USER_UID_SEQ`, and can be created by a DBA, or can be created by the JPA provider.

```
@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;
}
```

Read Basic mapping online: https://riptutorial.com/jpa/topic/3691/basic-mapping

Chapter 3: Joined Inheritance strategy

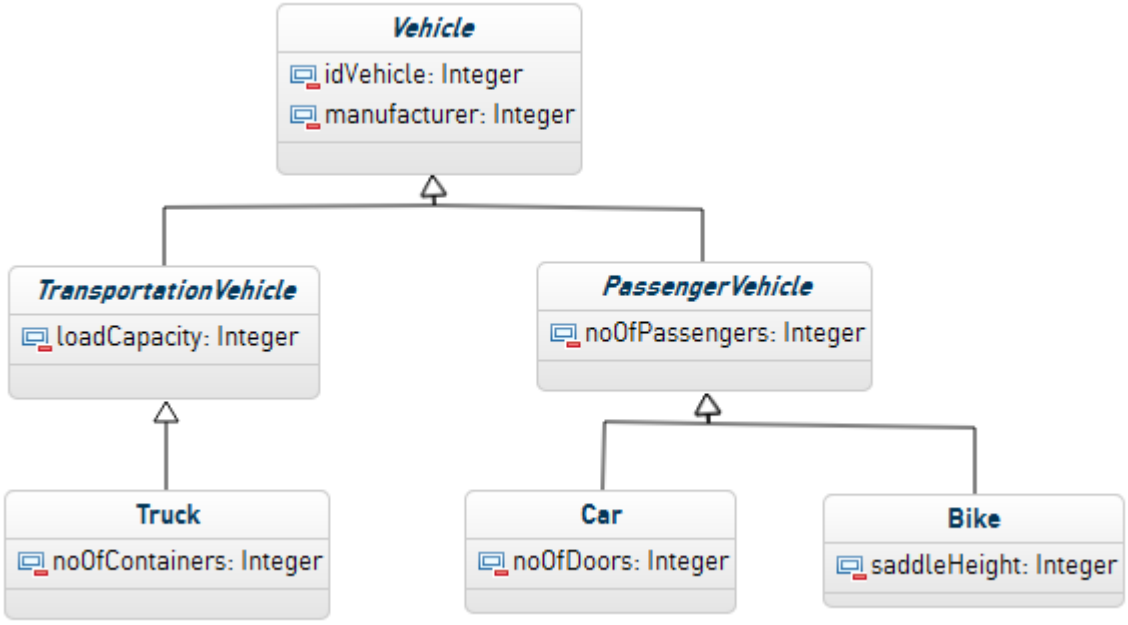
Parameters

Annotation	Purpose
@Inheritance	Specifies type of inheritance strategy used
@DiscriminatorColumn	Specifies a column in database which will be used to identify different entities based on certain ID assigned to each entity
@MappedSuperClass	mapped super classes are not persistent and only used to hold state for its subclasses. Generally abstract java classes are marked with @MapperSuperClass

Examples

Joined inheritance strategy

A Sample class diagram based on which we will see JPA implementation.




```
@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

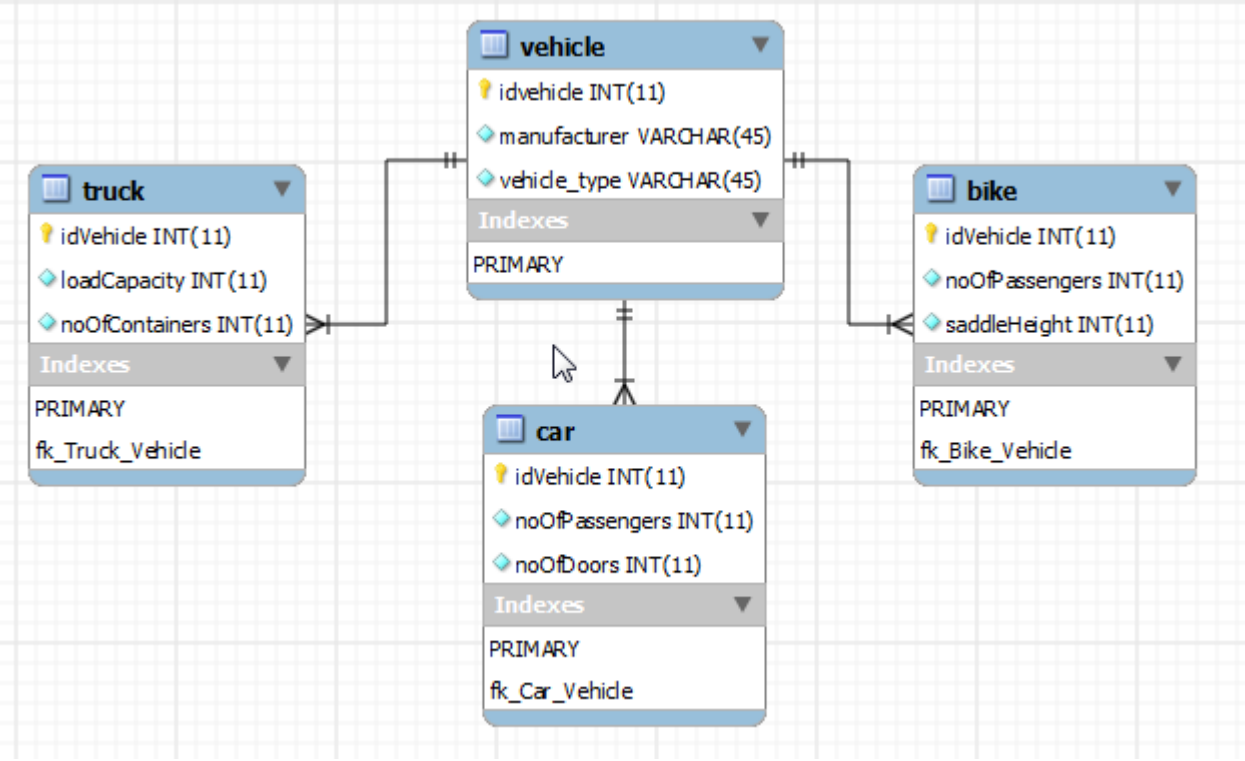
```
@Entity
public class Bike extends PassengerVehicle {

    private int saddleHeight;
```

```
    // getters and setters

}
```

Test Code

```
/* 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();
```

Database diagram would be as below.



The advantage of joined inheritance strategy is that it does not waste database space as in single table strategy. On the other hand, because of multiple joins involved for every insertion and retrieval, performance becomes an issue when inheritance hierarchies become wide and deep.

Full example with explanation can be read here

Read Joined Inheritance strategy online: https://riptutorial.com/jpa/topic/6473/joined-inheritance-strategy

Chapter 4: Many to Many Mapping

Introduction

A `ManyToMany` mapping describes a relationship between to entities where both can be related to more than one instance of each other, and is defined by the `@ManyToMany` annotation.

Unlike `@OneToMany` where a foreign key column in the table of the entity can be used, `ManyToMany` requires a join table, which maps the entities to each other.

Parameters

Annotation	Purpose
`@TableGenerator`	Defines a primary key generator that may be referenced by name when a generator element is specified for the `GeneratedValue` annotation
`@GeneratedValue`	Provides for the specification of generation strategies for the values of primary keys. It may be applied to a primary key property or field of an entity or mapped superclass in conjunction with the Id annotation.
`@ManyToMany`	Specifies relationship between Employee and Project entities such that many employees can work on multiple projects.
`mappedBy="projects"`	Defines a bidirectional relationship between Employee and Project
`@JoinColumn`	Specifies the name of column that will refer to the Entity to be considered as owner of the association
`@JoinTable`	Specifies the table in database which will hold employee to project relationships using foreign keys

Remarks

- @TableGenerator and @GeneratedValue are used for automatic ID creation using jpa table generator.
- @ManyToMany annotation specifies the relationship between Employee and Project entities.
- @JoinTable specifies the name of the table to use as join table jpa many to many mapping between Employee and Project using name = “employee_project”. This is done because there is no way to determine the ownership of a jpa many to many mapping as the database tables do not contain foreign keys to reference to other table.
- @JoinColumn specifies the name of column that will refer to the Entity to be considered as owner of the association while @inverseJoinColumn specifies the name of inverse side of

relationship. (You can choose any side to be considered as owner. Just make sure those sides in relationship). In our case we have chosen Employee as the owner so @JoinColumn refers to idemployee column in join table employee_project and @InverseJoinColumn refers to idproject which is inverse side of jpa many to many mapping.
- @ManyToMany annotation in Project entity shows inverse relationship hence it uses mappedBy=projects to refer to the field in Employee entity.

Full example can be referred here

Examples

Employee to Project Many to Many mapping

Employee entity.

```
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;
    }

}
```

Project Entity:

```
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;
    }

}
```

Test Code

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

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

    EntityTransaction transaction = em.getTransaction();

    transaction.begin();

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

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

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

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

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

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

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

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

    em.persist(prasad);

    transaction.commit();
```

How to handle compound key without Embeddable annotation

If You have

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

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

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

In above scenario `rightrole` table has compound key and to access it in JPA user have to create entity with `Embeddable` annotation.

Like this:

Entity for rightrole table is:

```
    @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;

   .....
```

```
}
```

Embeddable annotation is fine for single object but it will give an issue while inserting bulk records.

Problem is whenever user want to create new `role` with `rights` then first user have to `store(persist) role` object and then user have to do `flush` to get newly generated `id` for role. then and then user can put it in `rightrole` entity's object.

To solve this user can write entity slightly different way.

Entity for role table is:

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

The @JoinTable annotation will take care of inserting in the `rightrole` table even without an entity (as long as that table have only the id columns of role and right).

User can then simply:

```
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);
```

You have to write @ManyToMany(cascade = {CascadeType.PERSIST}) in inverseJoinColumns otherwise your parent data will get deleted if child get deleted.

Read Many to Many Mapping online: https://riptutorial.com/jpa/topic/6532/many-to-many-mapping

Chapter 5: Many To One Mapping

Parameters

Column	Column
@TableGenerator	Uses table generator strategy for automatic id creation
@GeneratedValue	Specifies that the value applied to fields is a generated value
@Id	Annotates the field as identifier
@ManyToOne	Specifies Many to One relationship between Employee and Department. This annotation is marked on many side. i.e. Multiple employees belong to a single department. So Department is annotated with @ManyToOne in Employee entity.
@JoinColumn	Specifies database table column which stores foreign key for related entity

Examples

Employee to Department ManyToOne relationship

Employee Entity

```
@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
}
```

Department Entity

```
@Entity
```

```
public class Department {

    @Id
    private int iddepartment;
    private String name;

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

Test class

```
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();

    }

}
```

Read Many To One Mapping online: https://riptutorial.com/jpa/topic/6531/many-to-one-mapping

Chapter 6: One to Many relationship

Parameters

Annotation	Purpose
@TableGenerator	Specifies generator name and table name where generator can be found
@GeneratedValue	Specifies generation strategy and refers to name of generator
@ManyToOne	Specifies many to one relationship between Employee and Department
@OneToMany(mappedBy="department")	creates bi-directional relationship between Employee and Department by simply referring to @ManyToOne annotation in Employee entity

Examples

One To Many relationship

One to Many mapping is generally simply a bidirectional relationship of Many to One mapping. We will take same example that we took for Many to one mapping.

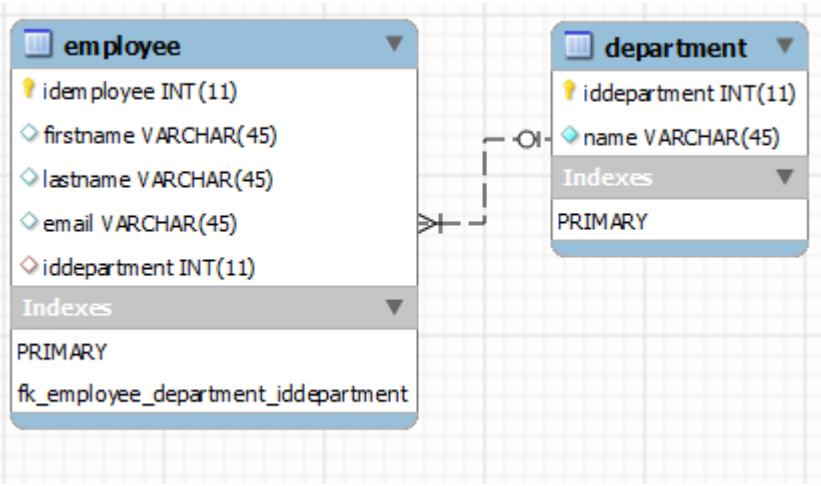
Employee.java

```
@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
}
```

This relationship is represented in database as below.



There are two points to remember about jpa one to many mapping:

- The many to one side is the owning side of relationship. The column is defined on that side.
- The one to many mapping is the inverse side side so the mappedBy element must be used on the inverse side.

Complete example can be referred here

Read One to Many relationship online: https://riptutorial.com/jpa/topic/6529/one-to-many-relationship

Chapter 7: One to One mapping

Parameters

Annotation	Purpose
@TableGenerator	Specifies generator name and table name where generator can be found
@GeneratedValue	Specifies generation strategy and refers to name of generator
@OneToOne	Specifies one to one relationship between employee and desk, here Employee is owner of relation
mappedBy	This element is provided on reverse side of relation. This enables bidirectional relationship

Examples

One To One relation between employee and desk

Consider a one to one bidirectional relationship between employee and desk.

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

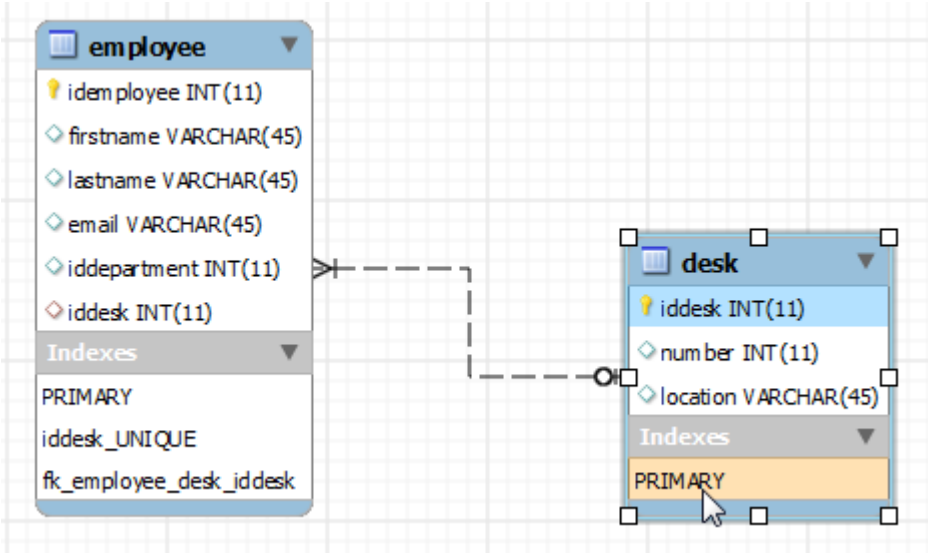
```
@Entity
public class Desk {
```

```
    @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
}
```

Test Code

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

Database diagram is depicted as below.



- The **@JoinColumn** annotation goes on mapping of the entity that is mapped to the table

containing the join colulmn.The owner of relationship. In our case, Employee table has the join column so @**JoinColumn** is on Desk field of Employee entity.

- The **mappedBy** element should be specified in the @**OneToOne** association in the entity that reverse side of relationship. i.e. The entity which does not provide join column on database aspect. In our case, Desk is the inverse entity.

Complete example can be found here

Read One to One mapping online: https://riptutorial.com/jpa/topic/6474/one-to-one-mapping

Chapter 8: Relations between entities

Remarks

Relations Between Entities Basics

A **foreign key** can be one or more columns that reference a unique key, usually the primary key, in another table.

A foreign key and the primary parent key it references must have the same number and type of fields.

Foreign keys represents **relationships** from a column or columns in one table to a column or columns in another table.

Examples

Multiplicity in Entity Relationships

Multiplicity in Entity Relationships

Multiplicities are of the following types:

- **One-to-one**: Each entity instance is related to a single instance of another entity.
- **One-to-many**: An entity instance can be related to multiple instances of the other entities.
- **Many-to-one**: multiple instances of an entity can be related to a single instance of the other entity.
- **Many-to-many**: The entity instances can be related to multiple instances of each other.

One-to-One Mapping

One-to-one mapping defines a single-valued association to another entity that has one-to-one multiplicity. This relationship mapping use the `@OneToOne` annotation on the corresponding persistent property or field.

Example: `Vehicle` *and* `ParkingPlace` *entities.*

One-to-Many Mapping

An entity instance can be related to multiple instances of the other entities.

One-to-many relationships use the `@OneToMany` annotation on the corresponding persistent property or field.

The `mappedBy` element is needed to refer to the attribute annotated by ManyToOne in the corresponding entity:

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

A one-to-many association needs to map the collection of entities.

Many-to-One Mapping

A many-to-one mapping is defined by annotating the attribute in the source entity (the attribute that refers to the target entity) with the `@ManyToOne` annotation.

A `@JoinColumn(name="FK_name")` annotation discribes a foreing key of a relationship.

Many-to-Many Mapping

The entity instances can be related to multiple instances of each other.

Many-to-many relationships use the `@ManyToMany` annotation on the corresponding persistent property or field.

We must use a third table to associate the two entity types (join table).

@JoinTable Annotation Example

When mapping many-to-many relationships in JPA, configuration for the table used for the joining foreign-keys can be provided using the `@JoinTable` annotation:

```
@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 this example, which consists of EntityA having a many-to-many relation to EntityB, realized by

the `entitiesB` field, we use the @JoinTable annotation to specify that the table name for the join table is `table_join_A_B`, composed by the columns `id_A` and `id_B`, foreign keys respectively referencing column `id` in EntityA's table and in EntityB's table; `(id_A,id_B)` will be a composite primary-key for `table_join_A_B` table.

Read Relations between entities online: https://riptutorial.com/jpa/topic/6305/relations-between-entities

Chapter 9: Single Table Inheritance Strategy

Parameters

Annotation	Purpose
@Inheritance	Specifies type of inheritance strategy used
@DiscriminatorColumn	Specifies a column in database which will be used to identify different entities based on certain ID assigned to each entity
@MappedSuperClass	mapped super classes are not persistent and only used to hold state for its subclasses. Generally abstract java classes are marked with @MapperSuperClass
@DiscriminatorValue	A value specified in column defined by @DiscriminatorColumn. This value helps identify the type of entity

Remarks

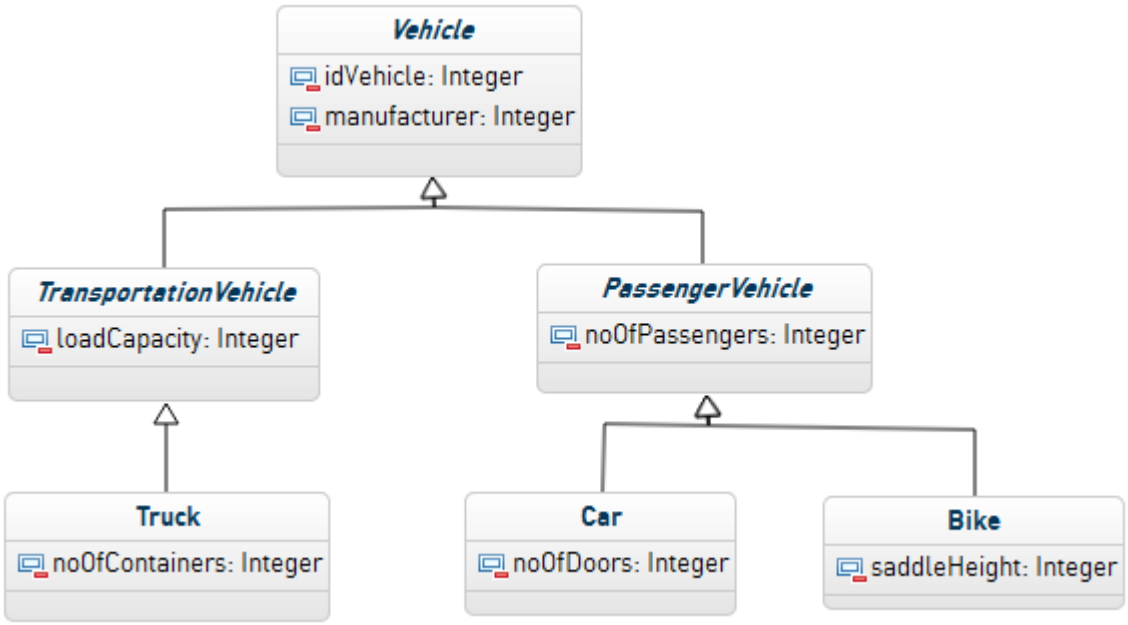
The advantage of single table strategy is it does not require complex joins for retrieval and insertion of entities, but on the other hand it wastes database space as many columns need to be nullable and there isn't any data for them.

Complete example and article can be found here

Examples

Single table inheritance strategy

A simple example of Vehicle hierarchy can be taken for single table inheritance strategy.



Abstract Vehicle class:

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

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

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

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

    public int getIdVehicle() {
        return idVehicle;
    }

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

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

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

}
```

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

import javax.persistence.MappedSuperclass;

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

    private int loadCapacity;

    public int getLoadCapacity() {
        return loadCapacity;
    }

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

}
```

PassengerVehicle.java

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

import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class PassengerVehicle extends Vehicle {

    private int noOfpassengers;

    public int getNoOfpassengers() {
        return noOfpassengers;
    }

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

}
```

Truck.java

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

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

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

```
    private int noOfContainers;

    public int getNoOfContainers() {
        return noOfContainers;
    }

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

}
```

Bike.java

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

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

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

    private int saddleHeight;

    public int getSaddleHeight() {
        return saddleHeight;
    }

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

}
```

Car.java

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

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

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

    private int noOfDoors;

    public int getNoOfDoors() {
        return noOfDoors;
    }

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

}
```

Test Code:

```
/* 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();
```

Read Single Table Inheritance Strategy online: https://riptutorial.com/jpa/topic/6277/single-table-inheritance-strategy

Chapter 10: Table per concrete class inheritance strategy

Remarks

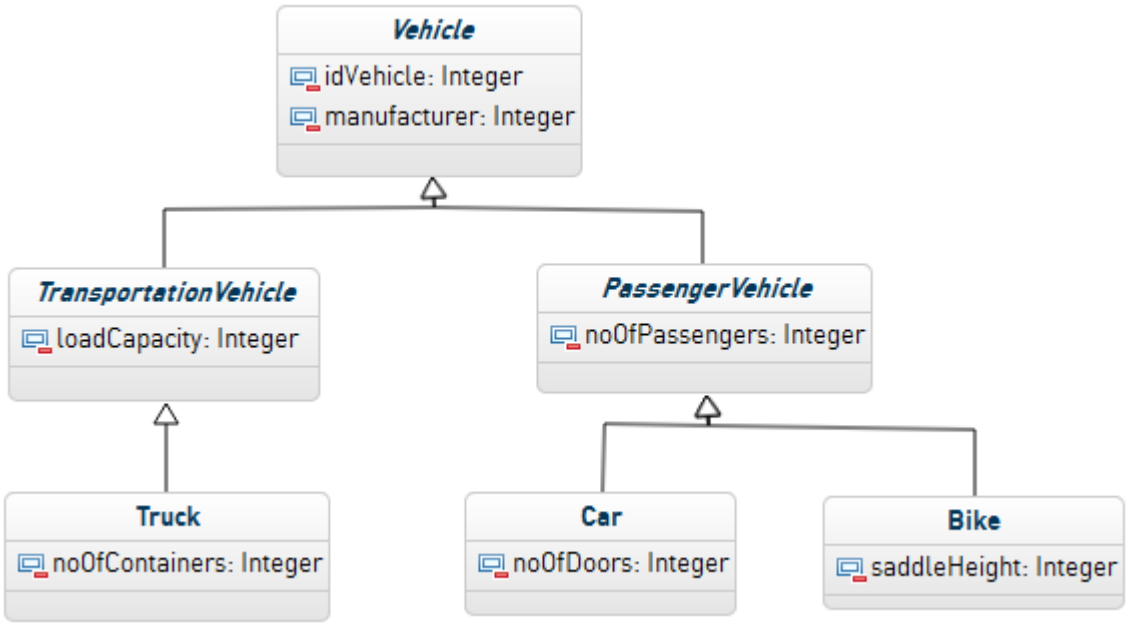
- Vehicle, TransportationVehicle and PassengerVehicle are abstract classes and they will not have separate table in database.
- Truck, Car and Bike are concrete classes so they will be mapped to corresponding tables. These tables should include all the fields for classes annotated with @MappedSuperClass because they don't have corresponding tables in database.
- So, Truck table will have columns to store fields inherited from TransportationVehicle and Vehicle.
- Similarly, Car and Bike will have columns to store fields inherited from PassengerVehicle and Vehicle.

Full example can be found here

Examples

Table per concrete class inheritance strategy

We will take vehicle hierarchy example as depicted below.



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

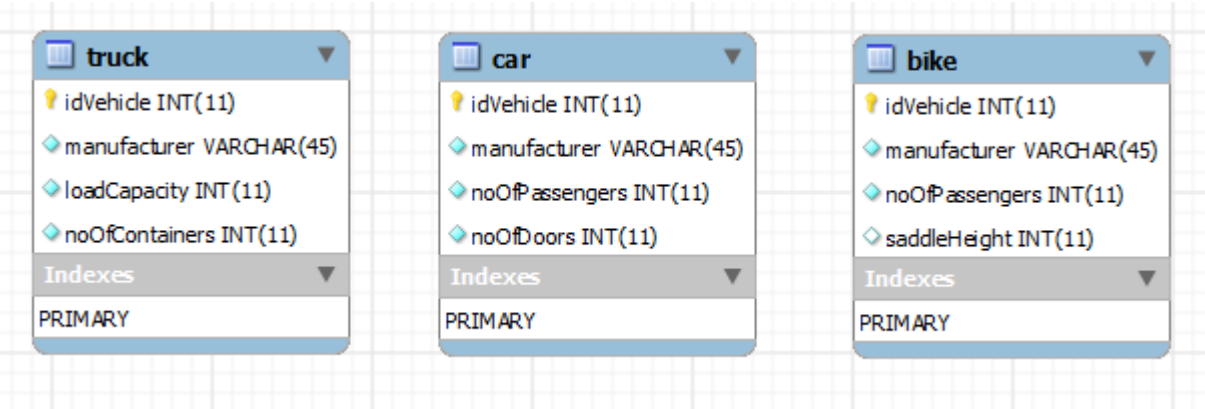
```
package com.thejavageek.jpa.entities;

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;
    }

}
```

Database representation will be as below



Read Table per concrete class inheritance strategy online:
https://riptutorial.com/jpa/topic/6255/table-per-concrete-class-inheritance-strategy

Credits

S. No	Chapters	Contributors
1	Getting started with jpa	Billy Frost, Community, DimaSan, , Manuel Spigolon, Michael Piefel, Neil Stockton, ppeterka
2	Basic mapping	Jeffrey Brett Coleman, Michael Piefel, Neil Stockton, Pete
3	Joined Inheritance strategy	bw_üezi, Prasad Kharkar
4	Many to Many Mapping	Prasad Kharkar, Ronak Patel, Vetle
5	Many To One Mapping	Prasad Kharkar
6	One to Many relationship	Prasad Kharkar
7	One to One mapping	Prasad Kharkar
8	Relations between entities	DimaSan,
9	Single Table Inheritance Strategy	Prasad Kharkar
10	Table per concrete class inheritance strategy	Prasad Kharkar