eBook Gratuit

APPRENEZ jpa

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#jpa

Table des matières

À propos 1

Chapitre 1: Commencer avec jpa 2

Remarques 2

Métadonnées 2

Architecture d'entité objet-relationnelle 2

Versions 2

Examples 2

Installation ou configuration 3

Exigences de Classpath 3

Eclipselink 3

Hiberner 3

DataNucleus 3

Détails de la configuration 4

Exemple minimal de persistence.xml 4

Hibernate (et H2 DB intégrée) 4

Eclipselink (et H2 DB intégrée) 5

DataNucleus (et la base de données H2 intégrée) 5

Bonjour le monde 6

Bibliothèques 6

Unité de persistance 6

Implémenter une entité 7

Implémenter un DAO 8

Tester l'application 9

Chapitre 2: Cartographie de base 11

Paramètres 11

Remarques 11

Examples 11

Une entité très simple 11

Omettre le champ de la cartographie 12

Mappage de l'heure et de la date 12

Date et heure avant Java 8 12
Date et heure avec Java 8 13
Entité avec ID géré par séquence 14
Chapitre 3: Cartographie individuelle 15
Paramètres 15
Examples 15
One To One relation entre employé et bureau 15
Chapitre 4: Mappage de plusieurs à un 18
Paramètres 18
Examples 18
Relation entre l'employé et le département ManyToOne 18
Chapitre 5: Plusieurs à plusieurs cartographie 20
Introduction 20
Paramètres 20
Remarques 20
Examples 21
Employé à projeter de nombreuses cartes à plusieurs 21
Comment gérer la clé composée sans annotation intégrable 23
Chapitre 6: Relation un à plusieurs 27
Paramètres 27
Examples 27
Relation One to Many 27
Chapitre 7: Relations entre entités 29
Remarques 29
Relations entre les entités 29
Examples 29
Multiplicité dans les relations d'entités 29
Multiplicité dans les relations d'entités 29
Cartographie individuelle 29
Cartographie un-à-plusieurs 29
Cartographie multi-un 30

Cartographie de plusieurs à plusieurs 30
Exemple d'annotation @JoinTable 30
Chapitre 8: Stratégie d'héritage à table unique 32
Paramètres 32
Remarques 32
Examples 32
Stratégie d'héritage à table unique 32
Chapitre 9: Stratégie d'héritage joint 37
Paramètres 37
Examples 37
Stratégie d'héritage jointe 37
Chapitre 10: Tableau par stratégie d'héritage de classe concrète 41
Remarques 41
Examples 41
Tableau par stratégie d'héritage de classe concrète 41
Crédits 45

À propos

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

Chapitre 1: Commencer avec jpa

Remarques

JPA est l'API Java Persistence, une spécification gérant le mappage des objets Java et leurs relations avec une base de données relationnelle. C'est ce qu'on appelle un mappeur objet-relationnel (ORM). C'est une alternative pour (ou compléter) les JDBC de plus bas niveau. Il est très utile pour une approche orientée Java et lorsque des graphes d'objet complexes doivent être conservés.

JPA en soi n'est pas une implémentation. Vous aurez besoin d'un fournisseur de persistance pour cela (voir les exemples). Les implémentations actuelles de la dernière norme JPA 2.1 sont EclipseLink (également l'implémentation de référence de JPA 2.1, ce qui signifie "la preuve que la spécification peut être implémentée"); Hibernate et DataNucleus .

Métadonnées

Le mappage entre les objets Java et les tables de base de données est défini via **des métadonnées de persistance** . Le fournisseur JPA utilisera les informations de métadonnées de persistance pour effectuer les opérations de base de données correctes. JPA définit généralement les métadonnées via des annotations dans la classe Java.

Architecture d'entité objet-relationnelle

L'architecture d'entité est composée de:

- entités
- unités de persistance
- contextes de persistance
- usines de gestionnaire d'entités
- gestionnaires d'entités

Versions

Version	Groupe d'experts	Libération
1.0	JSR-220	2006-11-06
2.0	JSR-317	2009-12-10
2.1	JSR-338	2013-05-22

Examples

Exigences de Classpath

Eclipselink

Les API Eclipselink et JPA doivent être incluses. Exemple de dépendances 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>
```

Hiberner

Hibernate-core est requis. Exemple de dépendance 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 et datanucleus-rdbms (lors de l'utilisation de datastores RDBMS) sont requis. Exemple de dépendance 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>
```

Détails de la configuration

JPA nécessite l'utilisation d'un fichier *persistence.xml* , situé sous `META-INF` à la racine de CLASSPATH. Ce fichier contient une définition des unités de persistance disponibles à partir desquelles JPA peut fonctionner.

JPA permet en outre d'utiliser un fichier de configuration de mappage *orm.xml* , également placé sous `META-INF` . Ce fichier de mappage permet de configurer la manière dont les classes sont mappées à la banque de données et constitue une alternative / un complément à l'utilisation des annotations Java dans les classes d'entités JPA elles-mêmes.

Exemple minimal de persistence.xml

Hibernate (et H2 DB intégrée)

```
<?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 (et H2 DB intégrée)

```
<?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 (et la base de données H2 intégrée)

```
<?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>
```

Bonjour le monde

Voyons tout le composant de base pour créer un monde Hallo simple.

1. Définir quelle implémentation de JPA 2.1 nous utiliserons
2. Construire la connexion à la base de données en créant l' `persistence-unit`
3. Implémente les entités
4. Implémente DAO (objet d'accès aux données) pour manipuler les entités
5. Tester l'application

Bibliothèques

En utilisant maven, nous avons besoin de ces dépendances:

```
<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>
```

Unité de persistance

Dans le dossier des ressources, nous devons créer un fichier appelé `persistence.xml` . La façon la plus simple de le définir est la suivante:

```
<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>
```

Implémenter une entité

Je crée une classe `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;
    }
}
```

Implémenter un DAO

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

import it.hello.jpa.entities.Biker;

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

public class MotorcycleRally {

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

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

}
```

`EntityManagerUtil` est un singleton:

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

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

public class EntityManagerUtil {

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

    private static EntityManager entityManager;

    private EntityManagerUtil() {
    }

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


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

Tester l'application

paquet it.hello.jpa.test;

classe publique 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));
}
```

}

Le résultat sera:

> Lancer it.hello.jpa.test.TestJpa Hibernate: drop table BIKER si existe Hibernate: drop séquence if existe hibernate_sequence Hibernate: créer la séquence hibernate_sequence démarrer avec 1 incrément de 1), beard boolean, date anniversaire, bikerName varchar (255), heure de registrationDate, clé primaire (id)) Hibernate: alter table BIKER ajouter une contrainte UK_a64ce28nywyk8wqrvfkkuapli unique (battleName) , id) valeurs (?,?,?,?,?,?) mar. 01, 2017 11:00:02 org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation INFO: HHH000204: Traitement de PersistenceUnitInfo [name: hello- jpa-pu ...] Résultats:
>
> Tests réalisés: 1, échecs: 0, erreurs: 0, ignorés: 0

Lire Commencer avec jpa en ligne: https://riptutorial.com/fr/jpa/topic/2125/commencer-avec-jpa

Chapitre 2: Cartographie de base

Paramètres

Annotation	Détails
`@Id`	Champ / colonne de marques en tant que *clé* de l'entité
`@Basic`	Champ demandé à mapper en tant que type de *base* . Ceci est applicable aux types primitifs et à leurs wrappers, `String` , `Date` et `Calendar` . L'annotation est facultative si aucun paramètre n'est donné, mais un bon style dicterait pour que vos intentions soient explicites.
`@Transient`	Les champs marqués comme transitoires ne sont pas pris en compte pour la persistance, un peu comme le mot-clé `transient` pour la sérialisation.

Remarques

Il doit toujours y avoir un constructeur par défaut, c'est-à-dire sans constructeur. Dans l'exemple de base, aucun constructeur n'a été spécifié, alors Java en a ajouté un; mais si vous ajoutez un constructeur avec des arguments, veillez à ajouter également le constructeur sans paramètre.

Examples

Une entité très simple

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

    @Basic
    String note;

    @Basic
    int count;
}
```

Les Getters, les Setters, etc.

Cette classe Java correspondrait au tableau suivant (en fonction de votre base de données, donnée ici dans un possible mappage Postgres):

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

```
)
```

Les fournisseurs JPA peuvent être utilisés pour générer le DDL et produiront probablement du DDL différent de celui présenté ici, mais tant que les types sont compatibles, cela ne causera pas de problèmes lors de l'exécution. Il est préférable de ne pas compter sur l'auto-génération de DDL.

Omettre le champ de la cartographie

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

    @Basic
    String note;

    @Transient
    String parsedNote;

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

Si votre classe a besoin de champs qui ne doivent pas être écrits dans la base de données, marquez-les comme `@Transient` . Après avoir lu la base de données, le champ sera `null` .

Mappage de l'heure et de la date

L'heure et la date sont de différents types en Java: la `Date` et le `Calendar` désormais historiques, et les `LocalDate` plus récentes `LocalDate` et `LocalDateTime` . Et les types `Timestamp` , `Instant` , `ZonedLocalDateTime` et Joda-time. Du côté de la base de données, nous avons l' `time` , la `date` et l' `timestamp` (heure et date), éventuellement avec ou sans fuseau horaire.

Date et heure avant Java 8

Le mappage *par défaut* des types pré-Java- `java.util.Date` , `java.util.Calendar` et `java.sql.Timestamp` est `timestamp` in SQL; pour `java.sql.Date` c'est `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;
}
```

Cela correspondra parfaitement au tableau suivant:

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

Cela peut ne pas être l'intention. Par exemple, une `Date` ou un `Calendar` Java est souvent utilisé pour représenter la date uniquement (pour la date de naissance). Pour modifier le mappage par défaut ou simplement pour rendre le mappage explicite, vous pouvez utiliser l'annotation `@Temporal` .

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

    @Temporal(TemporalType.TIME)
    private Date date;

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

La table SQL équivalente est:

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

Note 1: Le type spécifié avec `@Temporal` influence la génération de DDL; mais vous pouvez aussi avoir une colum de type type `date` à `Date` avec juste l'annotation `@Basic` .

Note 2: Le `Calendar` ne peut pas persister uniquement l' `time` .

Date et heure avec Java 8

JPA 2.1 ne définit *pas la* prise en `java.time` types `java.time` fournis en Java 8. La majorité des implémentations JPA 2.1 prennent toutefois en charge ces types, bien qu'il s'agisse à proprement

parler d'extensions de fournisseurs.

Pour DataNucleus, ces types fonctionnent simplement et offrent un large éventail de possibilités de mappage, `@Temporal` annotation `@Temporal` .

Pour Hibernate, si vous utilisez Hibernate 5.2+, ils devraient fonctionner `@Basic` avec l'annotation `@Basic` . Si vous utilisez Hibernate 5.0-5.1, vous devez ajouter la dépendance `org.hibernate:hibernate-java8` . Les mappages fournis sont

- `LocalDate` à `date`
- `Instant` , `LocalDateTime` et `ZonedDateTime` à l' `timestamp`

Une alternative `java.time` fournisseur consisterait également à définir un JPA 2.1 `AttributeConverter` pour tout type Java 8 `java.time` devant être conservé.

Entité avec ID géré par séquence

Ici, nous avons une classe et nous voulons que le champ d'identité (`userUid`) soit généré via une SEQUENCE dans la base de données. Cette SEQUENCE est supposée s'appeler `USER_UID_SEQ` et peut être créée par un administrateur de base de données ou peut être créée par le fournisseur 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;
}
```

Lire Cartographie de base en ligne: https://riptutorial.com/fr/jpa/topic/3691/cartographie-de-base

Chapitre 3: Cartographie individuelle

Paramètres

Annotation	Objectif
@TableGenerator	Spécifie le nom du générateur et le nom de la table où le générateur peut être trouvé
@GeneratedValue	Spécifie la stratégie de génération et fait référence au nom du générateur
@Un par un	Spécifie une relation individuelle entre l'employé et le bureau, ici l'employé est le propriétaire de la relation
mappéPar	Cet élément est fourni au verso de la relation. Cela permet une relation bidirectionnelle

Examples

One To One relation entre employé et bureau

Considérez une relation bidirectionnelle entre l'employé et le bureau.

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

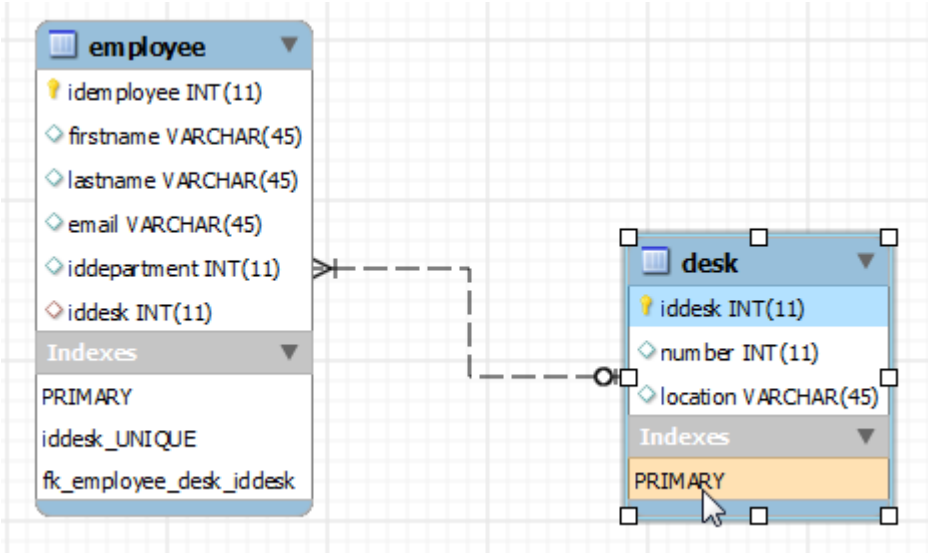
```
@Entity
public class Desk {
```

```
    @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
}
```

Code de test

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

Le schéma de base de données est représenté ci-dessous.



- L'annotation **@JoinColumn** passe au mappage de l'entité mappée sur la table contenant la

jointure colulmn.Le propriétaire de la relation. Dans notre cas, la table Employee a la colonne join pour que **@JoinColumn** soit sur le champ Desk de l'entité Employee.

- L'élément **mappedBy** doit être spécifié dans l'association **@OneToOne** dans l'entité à l'envers de la relation. C'est-à-dire l'entité qui ne fournit pas de colonne de jointure dans l'aspect base de données. Dans notre cas, Desk est l'entité inverse.

Un exemple complet peut être trouvé ici

Lire Cartographie individuelle en ligne: https://riptutorial.com/fr/jpa/topic/6474/cartographie-individuelle

Chapitre 4: Mappage de plusieurs à un

Paramètres

Colonne	Colonne
@TableGenerator	Utilise la stratégie du générateur de tables pour la création automatique d'identifiants
@GeneratedValue	Spécifie que la valeur appliquée aux champs est une valeur générée
@ Id	Annote le champ comme identifiant
@ManyToOne	Spécifie la relation entre un employé et un service. Cette annotation est marquée de nombreux côtés. C'est-à-dire que plusieurs employés appartiennent à un seul département. Ainsi, Département est annoté avec @ManyToOne dans l'entité Employé.
@JoinColumn	Spécifie la colonne de la table de base de données qui stocke la clé étrangère pour l'entité associée

Examples

Relation entre l'employé et le département ManyToOne

Entité d'employé

```
@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
}
```

Entité Départementale

```
@Entity
public class Department {

    @Id
    private int iddepartment;
    private String name;

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

Classe de test

```
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();

    }

}
```

Lire Mappage de plusieurs à un en ligne: https://riptutorial.com/fr/jpa/topic/6531/mappage-de-plusieurs-a-un

Chapitre 5: Plusieurs à plusieurs cartographie

Introduction

Un mappage `ManyToMany` décrit une relation entre des entités où les deux peuvent être associées à plusieurs instances l'une de l'autre et est définie par l'annotation `@ManyToMany` .

Contrairement à `@OneToMany` où une colonne de clé étrangère dans la table de l'entité peut être utilisée, `ManyToMany` nécessite une table de jointure, qui mappe les entités entre elles.

Paramètres

Annotation	Objectif
`@TableGenerator`	Définit un générateur de clé primaire pouvant être référencé par nom lorsqu'un élément generator est spécifié pour l'annotation `GeneratedValue`
`@GeneratedValue`	Fournit la spécification de stratégies de génération pour les valeurs des clés primaires. Il peut être appliqué à une propriété de clé primaire ou à un champ d'une entité ou à une super-classe mappée conjointement avec l'annotation Id.
`@ManyToMany`	Spécifie la relation entre les entités Employé et Projet, de sorte que de nombreux employés puissent travailler sur plusieurs projets.
`mappedBy="projects"`	Définit une relation bidirectionnelle entre Employee et Project
`@JoinColumn`	Spécifie le nom de la colonne qui fera référence à l'entité à considérer comme propriétaire de l'association
`@JoinTable`	Spécifie la table dans la base de données qui tiendra l'employé à projeter des relations à l'aide de clés étrangères

Remarques

- @TableGenerator et @GeneratedValue sont utilisés pour la création automatique d'ID à l'aide du générateur de table jpa.
- L'annotation @ManyToMany spécifie la relation entre les entités Employé et Projet.
- @JoinTable spécifie le nom de la table à utiliser en tant que table de jointure jpa à plusieurs mappages entre Employee et Project en utilisant name = “employee_project”. Cela est fait car il n'existe aucun moyen de déterminer la propriété d'un mappage jpa plusieurs à plusieurs, car les tables de base de données ne contiennent pas de clés étrangères pour

faire référence à une autre table.

- @JoinColumn spécifie le nom de la colonne qui fera référence à l'entité à considérer en tant que propriétaire de l'association, tandis que @inverseJoinColumn spécifie le nom du côté inverse de la relation. (Vous pouvez choisir n'importe quel côté pour être considéré comme propriétaire. Assurez-vous juste que ces côtés en relation). Dans notre cas, nous avons choisi Employee comme propriétaire, de sorte que @JoinColumn se réfère à la colonne idemployee dans la table de jointure employee_project et que @InverseJoinColumn fait référence à idproject qui est inversé par rapport au mappage plusieurs à plusieurs.
- L'annotation @ManyToMany dans l'entité de projet montre une relation inverse et utilise donc mappedBy = projects pour faire référence au champ dans l'entité Employee.

L'exemple complet peut être référencé ici

Examples

Employé à projeter de nombreuses cartes à plusieurs

Entité employé.

```
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;
    }

}
```

Entité de projet:

```
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;
    }

}
```

Code de test

/ * Créer 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();
```

Comment gérer la clé composée sans annotation intégrable

Si tu as

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


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

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

Dans le scénario ci-dessus, la table de `rightrole` a une clé composée et pour y accéder dans l'utilisateur JPA, il faut créer une entité avec une annotation `Embeddable` .

Comme ça:

Entité pour la table de la droite est:

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

        private static final long serialVersionUID = 1L;

        @EmbeddedId
        protected RightRolePK rightRolePK;


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

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

        ......
     }


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

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

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

```
     private long rightID;

   .....

}
```

L'annotation intégrable convient à un seul objet, mais cela pose problème lors de l'insertion d'enregistrements en masse.

Le problème est à chaque fois que l' utilisateur veut créer un nouveau `role` avec les `rights` alors premier utilisateur doivent `store(persist) role` objet et l' utilisateur doit faire `flush` pour obtenir nouvellement généré `id` pour le rôle. Ensuite, l'utilisateur peut le placer dans l'objet de l'entité de `rightrole` .

Pour résoudre cet utilisateur peut écrire entité de manière légèrement différente.

L'entité pour la table de rôles est:

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

    private static final long serialVersionUID = 1L;

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


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

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

L'annotation @JoinTable se chargera d'insérer dans la table de `rightrole` même sans entité (tant que cette table n'a que les colonnes id de rôle et de droite).

L'utilisateur peut alors simplement:

```
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);
```

Vous devez écrire @ManyToMany (cascade = {CascadeType.PERSIST}) dans inverseJoinColumns, sinon vos données parentes seront supprimées si l'enfant est supprimé.

Lire Plusieurs à plusieurs cartographie en ligne: https://riptutorial.com/fr/jpa/topic/6532/plusieurs-a-plusieurs-cartographie

Chapitre 6: Relation un à plusieurs

Paramètres

Annotation	Objectif
@TableGenerator	Spécifie le nom du générateur et le nom de la table où le générateur peut être trouvé
@GeneratedValue	Spécifie la stratégie de génération et fait référence au nom du générateur
@ManyToOne	Spécifie plusieurs à une relation entre l'employé et le service
@OneToMany (mappedBy = "department")	crée une relation bidirectionnelle entre l'employé et le service en se référant simplement à l'annotation @ManyToOne dans l'entité Employé

Examples

Relation One to Many

Le mappage un à plusieurs est généralement une relation bidirectionnelle du mappage Many to One. Nous allons prendre le même exemple que nous avons pris pour Many à une cartographie.

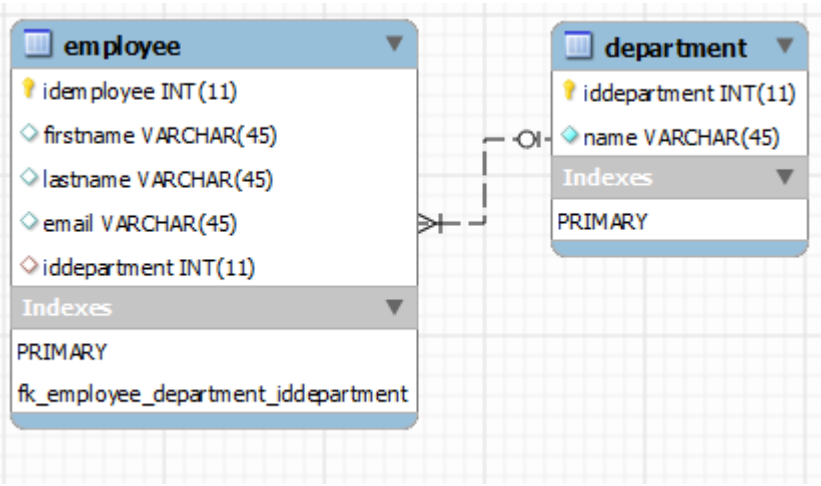
Employee.java

```
@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
}
```

Département.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
}
```

Cette relation est représentée dans la base de données comme ci-dessous.



Il y a deux points à retenir à propos de jpa one to many mapping:

- Le plus grand côté est le côté propriétaire de la relation. La colonne est définie de ce côté.
- Le mappage un à plusieurs est le côté côté inverse, donc l'élément mappedBy doit être utilisé du côté inverse.

L'exemple complet peut être référencé ici

Lire Relation un à plusieurs en ligne: https://riptutorial.com/fr/jpa/topic/6529/relation-un-a-plusieurs

Chapitre 7: Relations entre entités

Remarques

Relations entre les entités

Une **clé étrangère** peut être une ou plusieurs colonnes faisant référence à une clé unique, généralement la clé primaire, dans une autre table.

Une clé étrangère et la clé parent principale référencée doivent avoir le même nombre et le même type de champs.

Les clés étrangères représentent les **relations** entre une ou plusieurs colonnes d'une table et une ou plusieurs colonnes d'une autre table.

Examples

Multiplicité dans les relations d'entités

Multiplicité dans les relations d'entités

Les multiplicités sont des types suivants:

- **One-to-one** : chaque instance d'entité est liée à une instance unique d'une autre entité.
- **One-to-many** : une instance d'entité peut être associée à plusieurs instances des autres entités.
- **Plusieurs à un** : plusieurs instances d'une entité peuvent être associées à une seule instance de l'autre entité.
- **Plusieurs à plusieurs** : les instances d'entité peuvent être associées à plusieurs instances l'une de l'autre.

Cartographie individuelle

Le mappage un à un définit une association à une seule valeur pour une autre entité à multiplicité un à un. Ce mappage de relation utilise l'annotation `@OneToOne` sur la propriété ou le champ persistant correspondant.

Exemple: entités `Vehicle` *et* `ParkingPlace` *.*

Cartographie un-à-plusieurs

Une instance d'entité peut être liée à plusieurs instances des autres entités.

Les relations un-à-plusieurs utilisent l'annotation `@OneToMany` sur la propriété ou le champ persistant

correspondant.

L'élément `mappedBy` est nécessaire pour faire référence à l'attribut annoté par ManyToOne dans l'entité correspondante:

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

Une association un-à-plusieurs doit mapper la collection d'entités.

Cartographie multi-un

Un mappage plusieurs à un est défini en annotant l'attribut dans l'entité source (l'attribut faisant référence à l'entité cible) avec l'annotation `@ManyToOne` .

Une `@JoinColumn(name="FK_name")` décrit une clé étrangère d'une relation.

Cartographie de plusieurs à plusieurs

Les instances d'entité peuvent être associées à plusieurs instances l'une de l'autre.

Les relations plusieurs à plusieurs utilisent l'annotation `@ManyToMany` sur la propriété ou le champ persistant correspondant.

Nous devons utiliser une troisième table pour associer les deux types d'entité (table de jointure).

Exemple d'annotation @JoinTable

Lors du mappage de relations plusieurs à plusieurs dans JPA, la configuration de la table utilisée pour la jointure de clés étrangères peut être fournie à l'aide de l'annotation `@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;
   [...]
}
```

Dans cet exemple, qui consiste en l' Entité ayant un grand nombre à plusieurs rapport à l' Entité, réalisé par le `entitiesB` domaine, nous utilisons l'annotation @JoinTable pour spécifier que le nom de la table pour la table de jointure est `table_join_A_B` , composée par les colonnes `id_A` et `id_B` , les clés étrangères référençant respectivement l' `id` colonne dans la table EntityA et dans la table EntityB; `(id_A,id_B)` sera une clé primaire composite pour la table `table_join_A_B` .

Lire Relations entre entités en ligne: https://riptutorial.com/fr/jpa/topic/6305/relations-entre-entites

Chapitre 8: Stratégie d'héritage à table unique

Paramètres

Annotation	Objectif
@Héritage	Spécifie le type de stratégie d'héritage utilisé
@DiscriminatorColumn	Spécifie une colonne dans la base de données qui sera utilisée pour identifier différentes entités en fonction de certains identifiants attribués à chaque entité
@MappedSuperClass	Les super-classes mappées ne sont pas persistantes et ne sont utilisées que pour maintenir l'état de ses sous-classes. Les classes Java généralement abstraites sont marquées avec @MapperSuperClass
@DiscriminatorValue	Une valeur spécifiée dans la colonne définie par @DiscriminatorColumn. Cette valeur permet d'identifier le type d'entité

Remarques

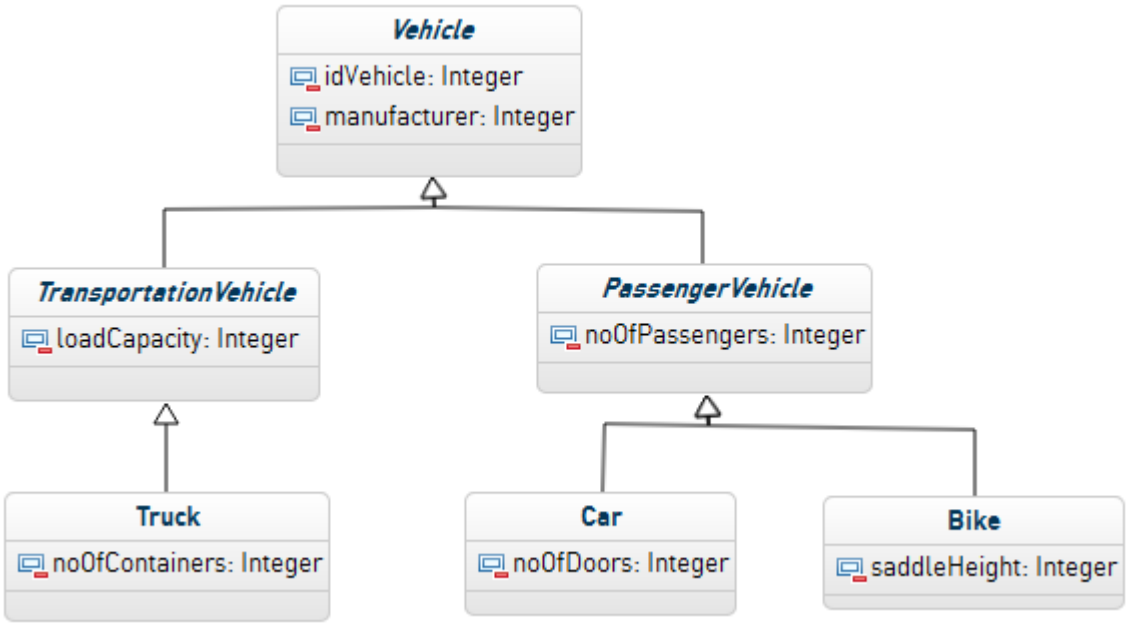
L'avantage de la stratégie à table unique est qu'elle ne nécessite pas de jointures complexes pour la récupération et l'insertion d'entités, mais elle gaspille de l'espace de base de données car de nombreuses colonnes doivent être nullables et ne contiennent aucune donnée.

L'exemple complet et l'article peuvent être trouvés ici

Examples

Stratégie d'héritage à table unique

Un exemple simple de hiérarchie de véhicule peut être pris pour la stratégie d'héritage à table unique.



Classe de véhicule abstrait:

```
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;
    }

}
```

Package **transportableVehicle.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;
    }

}
```

Code de test:

```
/* 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();
```

Lire Stratégie d'héritage à table unique en ligne: https://riptutorial.com/fr/jpa/topic/6277/strategie-d-heritage-a-table-unique

Chapitre 9: Stratégie d'héritage joint

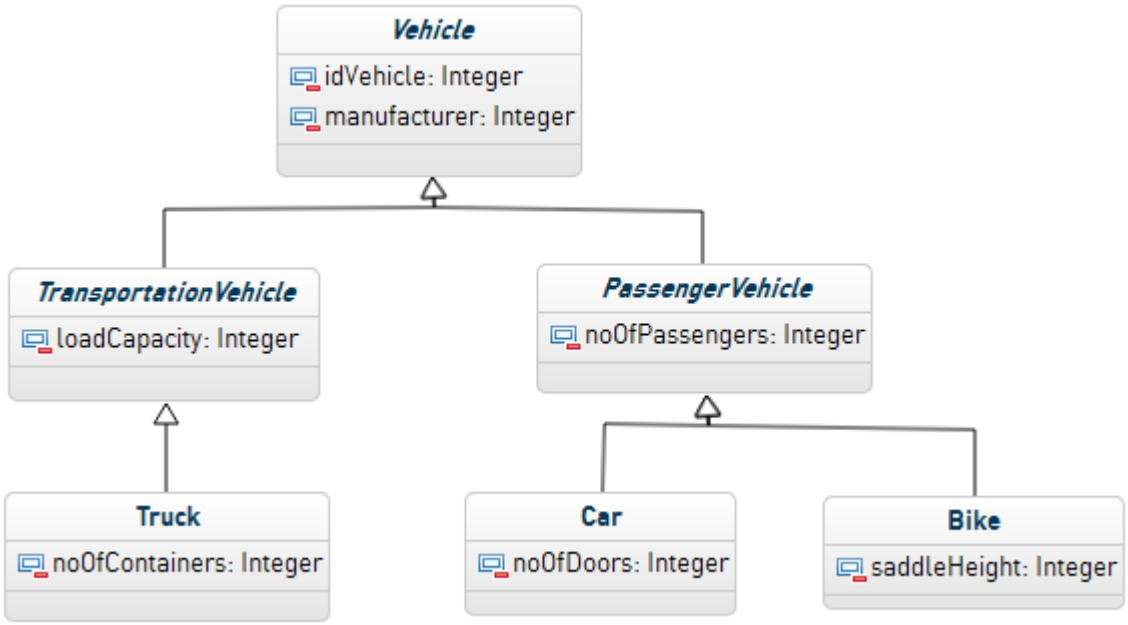
Paramètres

Annotation	Objectif
@Héritage	Spécifie le type de stratégie d'héritage utilisé
@DiscriminatorColumn	Spécifie une colonne dans la base de données qui sera utilisée pour identifier différentes entités en fonction de certains identifiants attribués à chaque entité
@MappedSuperClass	Les super-classes mappées ne sont pas persistantes et ne sont utilisées que pour maintenir l'état de ses sous-classes. Les classes Java généralement abstraites sont marquées avec @MapperSuperClass

Examples

Stratégie d'héritage jointe

Un exemple de diagramme de classes sur lequel nous verrons l'implémentation de JPA.




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

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

    // getters and setters
}
```

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

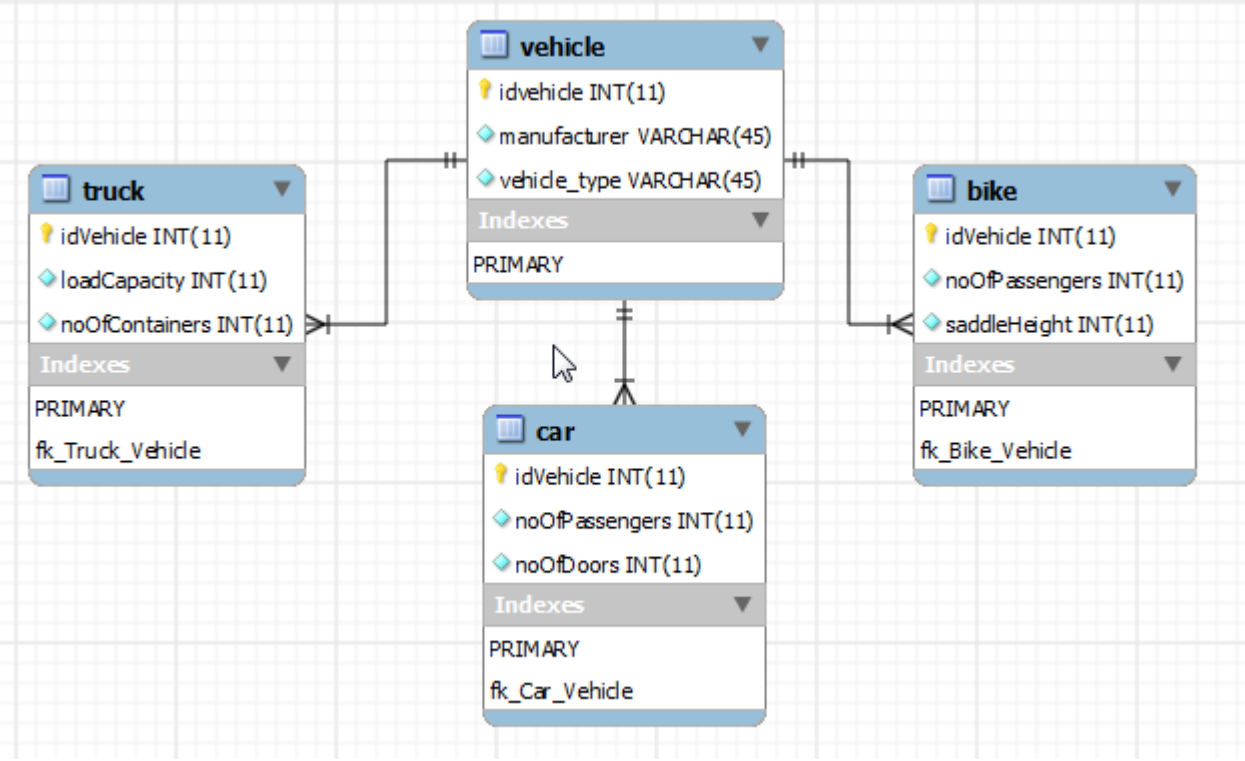
```
@Entity
public class Bike extends PassengerVehicle {
```

```
    private int saddleHeight;

    // getters and setters

}
```

Code de test

```
/* 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();
```

Le schéma de base de données serait comme ci-dessous.



L'avantage de la stratégie d'héritage jointe est qu'elle ne gaspille pas l'espace de la base de données comme dans une stratégie à table unique. D'autre part, en raison des multiples jointures impliquées pour chaque insertion et récupération, les performances deviennent un problème lorsque les hiérarchies d'héritage deviennent larges et profondes.

Un exemple complet avec explication peut être lu ici

Lire Stratégie d'héritage joint en ligne: https://riptutorial.com/fr/jpa/topic/6473/strategie-d-heritage-joint

Chapitre 10: Tableau par stratégie d'héritage de classe concrète

Remarques

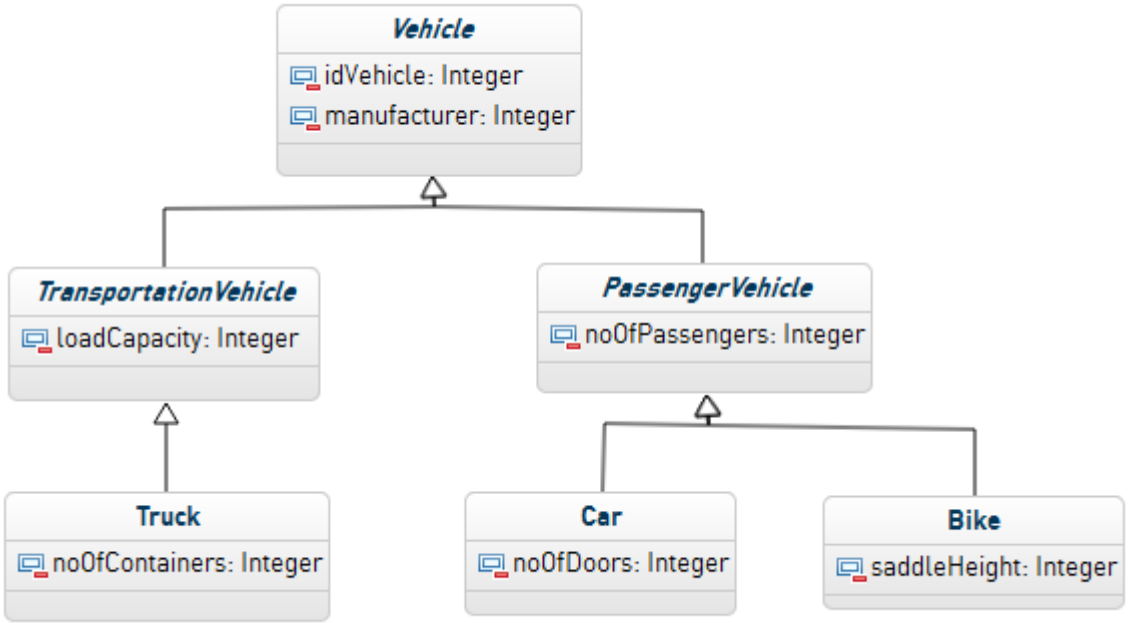
- Vehicle, TransportationVehicle et PassengerVehicle sont des classes abstraites et elles n'auront pas de table séparée dans la base de données.
- Camion, Voiture et Vélo sont des classes concrètes, elles seront donc associées aux tables correspondantes. Ces tables doivent inclure tous les champs des classes annotées avec @MappedSuperClass car elles ne disposent pas de tables correspondantes dans la base de données.
- Ainsi, la table Truck aura des colonnes pour stocker les champs hérités de TransportationVehicle et Vehicle.
- De même, Car and Bike aura des colonnes pour stocker les champs hérités de PassengerVehicle et Vehicle.

L'exemple complet peut être trouvé ici

Examples

Tableau par stratégie d'héritage de classe concrète

Nous prendrons l'exemple de la hiérarchie des véhicules tel que décrit ci-dessous.



Véhicule.java

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

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

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

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

    public int getIdVehicle() {
        return idVehicle;
    }

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

    public String getManufacturer() {
        return manufacturer;
    }

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

}
```

TransportationVehilcle.java

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

import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class TransportationVehicle extends Vehicle {

    private int loadCapacity;

    public int getLoadCapacity() {
        return loadCapacity;
    }

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

}
```

Truck.java

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

import javax.persistence.Entity;

@Entity
public class Truck extends TransportationVehicle {

    private int noOfContainers;

    public int getNoOfContainers() {
        return noOfContainers;
    }

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

}
```

PassengerVehicle.java

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

import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class PassengerVehicle extends Vehicle {

    private int noOfpassengers;

    public int getNoOfpassengers() {
        return noOfpassengers;
    }

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

}
```

Car.java

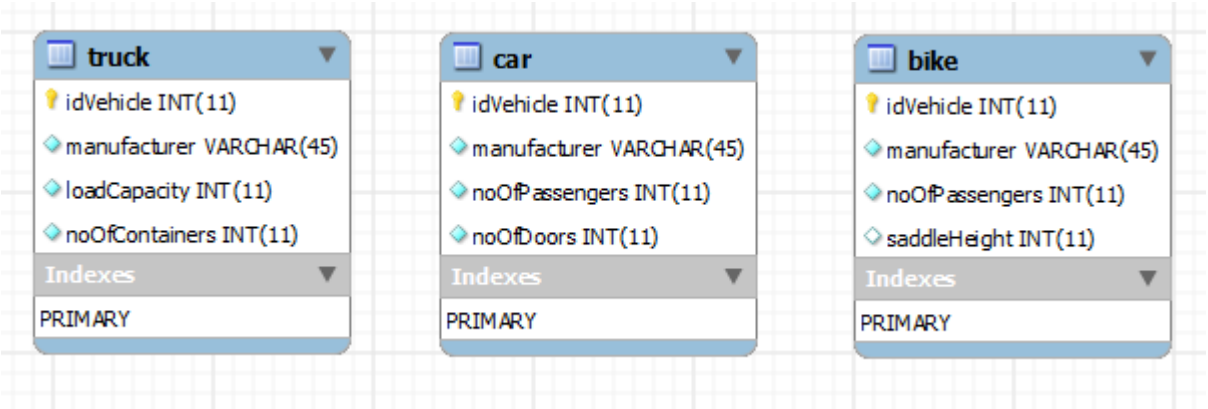
```
package com.thejavageek.jpa.entities;

import javax.persistence.Entity;

@Entity
public class Car extends PassengerVehicle {

    private int noOfDoors;

    public int getNoOfDoors() {
        return noOfDoors;
    }

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

```
}
```

Bike.java

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

import javax.persistence.Entity;

@Entity
public class Bike extends PassengerVehicle {

    private int saddleHeight;

    public int getSaddleHeight() {
        return saddleHeight;
    }

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

}
```

La représentation de la base de données sera comme ci-dessous



Lire Tableau par stratégie d'héritage de classe concrète en ligne:
https://riptutorial.com/fr/jpa/topic/6255/tableau-par-strategie-d-heritage-de-classe-concrete

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec jpa	Billy Frost, Community, DimaSan, , Manuel Spigolon, Michael Piefel, Neil Stockton, ppeterka
2	Cartographie de base	Jeffrey Brett Coleman, Michael Piefel, Neil Stockton, Pete
3	Cartographie individuelle	Prasad Kharkar
4	Mappage de plusieurs à un	Prasad Kharkar
5	Plusieurs à plusieurs cartographie	Prasad Kharkar, Ronak Patel, Vetle
6	Relation un à plusieurs	Prasad Kharkar
7	Relations entre entités	DimaSan,
8	Stratégie d'héritage à table unique	Prasad Kharkar
9	Stratégie d'héritage joint	bw_üezi, Prasad Kharkar
10	Tableau par stratégie d'héritage de classe concrète	Prasad Kharkar