Mapping d'une relation @ManyToOne

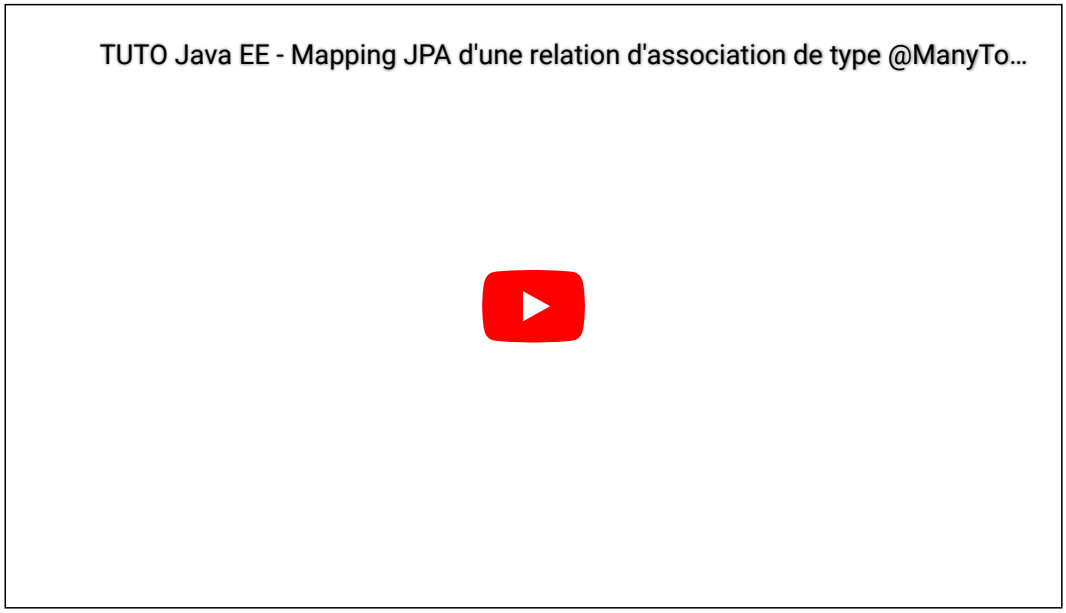
Mapping d'une relation @OneToOne

Mapping d'une relation @OneToMany

Accès rapide :

La vidéo

Cette vidéo vous présente les différentes techniques de mapping des relations de type @ManyToOne (avec ou sans table de jointures) proposées par JPA.

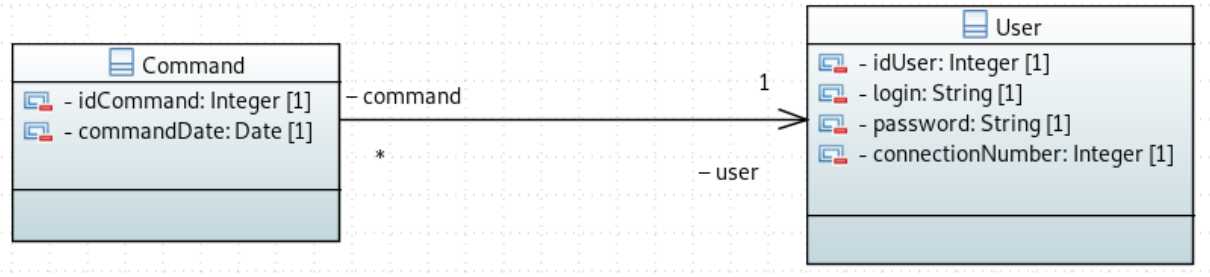


Mapping JPA d'une relation d'association de type @ManyToOne

Introduction à la relation d'association de type « Many-To-One »

Nous allons voir, au travers de ce document, comment réaliser des associations de type « Many-To-One » avec JPA et Hibernate. Une relation de type « Many-To-One » a du sens quand plusieurs éléments peuvent être associés à un même élément : par exemple, plusieurs commandes peuvent être associées à une même personne. Si une telle relation est mise en oeuvre, il vous sera possible, à partir de la commande, de retrouver l'utilisateur associé.

Le diagramme UML ci-dessous vous montre l'association que nous allons chercher à mettre en oeuvre. La classe `Command` représentera la commande et la classe `User` représentera la personne ayant passé la commande.



Mais attention : en base de données, vous avez deux manières de représenter une relation de type « Many-to-One ».

- Mapping d'une relation @ManyToOne sans table d'association : dans ce cas, la clé de jointure, permettant la mise en relation, sera portée par la table associée à la classe `Command`. Nous appellerons cette table `T_Commands`
- Mapping d'une relation @ManyToOne avec table d'association : dans ce cas, une troisième table (non associée à une entité JPA) servira à stocker les paires de clés de mise en relations.

Rappel : le terme de table d'association est aussi parfois appelé table de jointure. Il s'agit d'une table en base de données permettant d'associer deux enregistrements situés dans deux autres tables de la base de données en utilisant des « foreign keys ».

Note : dans ce chapitre, nous continuons à travailler en mode console (hors WAR Java/Jakarte EE) avec le SGBDr (Système de Gestion de Bases de données Relationnelles) MariaDB, comme proposé dans les exemples précédents. Je vous renvoie vers les tutoriels précédents pour de plus amples informations à ce sujet.

Mettons en oeuvre chacune de ces possibilités en commençant par la première.

Mapping d'une relation @ManyToOne sans table d'association.

Implémentation, en base de données.

Voici le code SQL permettant de créer une base de données (pour le SGBDr MariaDB) avec deux tables mises en association par une relation de type « Many-To-One ».

```
-- ------------------------------------------------------------------------
-- - Reconstruction de la base de données
-- ------------------------------------------------------------------------

DROP DATABASE IF EXISTS ManyToOne;
CREATE DATABASE ManyToOne;
USE ManyToOne;

-- ------------------------------------------------------------------------
-- - Construction de la table des utilisateurs                            -
-- ------------------------------------------------------------------------

CREATE TABLE T_Users (
    idUser              int         PRIMARY KEY AUTO_INCREMENT,
    login               varchar(20) NOT NULL,
    password            varchar(20) NOT NULL,
    connectionNumber    int         NOT NULL DEFAULT 0
);

INSERT INTO T_Users (login, password)
VALUES ( 'Anderson',    'Neo' ),
       ( 'Skywalker',   'Luke' ),
       ( 'Plissken',    'Snake' ),
       ( 'Ripley',      'Ellen' ),
       ( 'Bond',        'James' );

SELECT * FROM T_Users;

-- ------------------------------------------------------------------------
-- - Construction de la table des commandes                               --
-- ------------------------------------------------------------------------

CREATE TABLE T_Commands (
    idCommand       int         PRIMARY KEY AUTO_INCREMENT,
    idUser          int         NOT NULL REFERENCES T_Users(IdUser),
    commandDate     datetime    NOT NULL DEFAULT CURRENT_TIMESTAMP
```

```
);

INSERT INTO T_Commands (idUser)
VALUES (1), (2), (1);

SELECT * FROM T_Commands;
```

Le fichier Database.sql

Note : nous avons donc produit une base de données contenant cinq utilisateurs et trois commandes. Deux commandes sont associées à l'utilisateur `Anderson` et une autre est associée à l'utilisateur `Skywalker`.

Pour créer votre base de données, connectez-vous en mode ligne de commande à votre serveur puis exécutez l'ordre suivant :

```
MariaDB [(none)]> source Database.sql
Query OK, 2 rows affected (0.066 sec)

Query OK, 1 row affected (0.000 sec)

Database changed
Query OK, 0 rows affected (0.010 sec)

Query OK, 5 rows affected (0.002 sec)
Records: 5  Duplicates: 0  Warnings: 0

+--------+-----------+----------+------------------+
| idUser | login     | password | connectionNumber |
+--------+-----------+----------+------------------+
|      1 | Anderson  | Neo      |                0 |
|      2 | Skywalker | Luke     |                0 |
|      3 | Plissken  | Snake    |                0 |
|      4 | Ripley    | Ellen    |                0 |
|      5 | Bond      | James    |                0 |
+--------+-----------+----------+------------------+
5 rows in set (0.000 sec)

Query OK, 0 rows affected (0.013 sec)

Query OK, 3 rows affected (0.002 sec)
Records: 3  Duplicates: 0  Warnings: 0

+-----------+--------+---------------------+
| idCommand | idUser | commandDate         |
+-----------+--------+---------------------+
|         1 |      1 | 2020-01-21 11:42:01 |
|         2 |      2 | 2020-01-21 11:42:01 |
|         3 |      1 | 2020-01-21 11:42:01 |
+-----------+--------+---------------------+
3 rows in set (0.000 sec)

MariaDB [ManyToOne]>
```

Implémentation du mapping JPA.

Il nous faut maintenant définir les deux classes d'entités associées à nos tables et y ajouter les annotations JPA de mapping. En premier lieu, voici le code de la classe `User` : c'est la classe la plus simple à coder, car ce n'est pas elle qui porte la relation d'association.

```
 1 package fr.koor.webstore.business;
 2
 3 import javax.persistence.Entity;
 4 import javax.persistence.GeneratedValue;
 5 import javax.persistence.GenerationType;
 6 import javax.persistence.Id;
 7 import javax.persistence.Table;
 8
 9 @Entity @Table(name = "T_Users")
10 public class User {
11
12     @Id  @GeneratedValue( strategy=GenerationType.IDENTITY )
13     private int idUser;
14
15     private String login;
16
```

```
17      private String password;
18
19      private int connectionNumber;
20
21      public User() { }
22
23      public User( String login, String password, int connectionNumber ) {
24          super();
25          this.setLogin( login );
26          this.setPassword( password );
27          this.setConnectionNumber( connectionNumber );
28      }
29
30
31      public int getIdUser() {
32          return idUser;
33      }
34
35      public String getLogin() {
36          return login;
37      }
38
39      public void setLogin(String login) {
40          this.login = login;
41      }
42
43      public String getPassword() {
44          return password;
45      }
46
47      public void setPassword(String password) {
48          this.password = password;
49      }
50
51      public int getConnectionNumber() {
52          return connectionNumber;
53      }
54
55      public void setConnectionNumber(int connectionNumber) {
56          this.connectionNumber = connectionNumber;
57      }
58
59      public String toString() {
60          return this.idUser + ": " + this.login + "/" + this.password
61                  + " - " + this.connectionNumber + " connexion(s)";
62      }
63
64  }
```

La classe User et son mapping JPA

Note : comme les attributs `login`, `password` et `connectionNumber` ont les mêmes noms que leurs colonnes associées en base de données, il n'est pas nécessaire de les marquer via des annotations JPA.

Maintenant il nous faut définir la classe `Command` qui réalise la relation de type « Many-To-One » vers la classe `User`. Pour y mettre en place la relation, il va nous falloir utiliser l'annotation `@ManyToOne` comme le montre cet extrait de code.

```
1  @Entity @Table(name = "T_Commands")
2  public class Command {
3
4      // Autres attributs
5
6      @ManyToOne  @JoinColumn( name="idUser" )
7      private User user;
8
9      // Suite de la classe
10
11 }
```

Exemple d'une relation de type « Many-To-One »

Si vous le souhaitez, il est possible de cascader certaines actions initiées sur une instance de la classe `Command` à l'instance de type `User` qui lui est associée. Les actions pouvant être cascadées sont : `DETACH`, `MERGE`, `PERSIST` `REFRESH` et `REMOVE`. Par exemple, si l'on cascade le `REMOVE`, une suppression d'une instance de commande entraînera une suppression automatiquement de l'utilisateur associée. Nous reviendrons sur ces possibilités ultérieurement. En utilisant la constante `CascadeType.ALL` on demande à cascader toutes ces actions.

```java
@Entity @Table(name = "T_Commands")
public class Command {

    // Autres attributs

    @ManyToOne( cascade = CascadeType.ALL )
    @JoinColumn( name="idUser" )
    private User user;

    // Suite de la classe

}
```

Exemple d'une relation de type « Many-To-One » avec cascade des actions

Note : la syntaxe vous est ici présentée à titre d'information, mais je ne pense pas que dans notre cas concret le fait de cascader la suppression puisse présenter un quelconque intérêt. Bien au contraire, si on supprime une commande, on souhaite certainement conserver l'utilisateur qui possédera peut-être d'autres commandes. C'est plutôt dans le cas inverse que cela pourrait présenter un intérêt : si l'on supprime un utilisateur on cherchera peut-être à supprimer toutes ses commandes (nous explorerons cette possibilité lorsque nous étudierons la relation de type « One-To-Many »).

Il est aussi possible d'imposer la présence d'une instance de type `User` pour chaque commande en fixant l'attribut `nullable` à `false` sur l'annotation `@JoinColumn`.

```java
@Entity @Table(name = "T_Commands")
public class Command {

    // Autres attributs

    @ManyToOne
    @JoinColumn( name="idUser", nullable=false )
    private User user;

    // Suite de la classe

}
```

Exemple d'une relation de type « Many-To-One » imposant la présence d'un utilisateur

Voici le code complet de la classe `Command`.

```java
package fr.koor.webstore.business;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity  @Table(name="T_Commands")
public class Command {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int idCommand;

    @ManyToOne
    @JoinColumn(name="idUser", nullable=false)
    private User user;

    private Date commandDate;


    public Command() {}

    public Command( User user, Date commandDate ) {
        this.setUser( user );
        this.setCommandDate( commandDate );
    }

    public int getIdCommand() {
        return idCommand;
    }

    public User getUser() {
```

```
38         return user;
39     }
40
41     public void setUser(User user) {
42         this.user = user;
43     }
44
45     public Date getCommandDate() {
46         return commandDate;
47     }
48
49     public void setCommandDate(Date commandDate) {
50         this.commandDate = commandDate;
51     }
52
53     public String toString() {
54         StringBuilder builder = new StringBuilder();
55         builder.append( "Commande de >> " ).append( this.user )
56                .append( " - " ).append( this.commandDate ).append( "\n" );
57         return builder.toString();
58     }
59
60 }
```

La classe Command et son mapping JPA

Test de la relation d'association.

Pour configurer le projet JPA, il est nécessaire de fournir le fichier `META-INF/persistence.xml`. N'oubliez pas qu'il doit être accessible à partir de `CLASSPATH`. En voici sa définition.

```
1  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
2               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3               xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
4                     http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
5               version="2.0">
6
7      <persistence-unit name="WebStore">
8          <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
9
10         <class>fr.koor.webstore.business.User</class>
11         <class>fr.koor.webstore.business.Command</class>
12
13         <properties>
14             <property name="javax.persistence.jdbc.driver" value="org.mariadb.jdbc.Driver" />
15             <property name="javax.persistence.jdbc.url" value="jdbc:mariadb://localhost/ManyToOne" /
16             <property name="javax.persistence.jdbc.user" value="root" />
17             <property name="javax.persistence.jdbc.password" value="" />
18
19             <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
20             <property name="hibernate.format_sql" value="false" />
21         </properties>
22     </persistence-unit>
23
24 </persistence>
```

Le fichier de configuration JPA/Hibernate

Il vous faut, bien entendu un fichier de configuration pour Log4J (du moins, si vous souhaitez l'utiliser) : je vous renvoie à ce sujet sur un des chapitres précédents pour de plus amples informations.

Nous allons commencer par charger une instance de commande par sa clé primaire en invoquant la méthode `entityManager.find`. Nous descendons ensuite dans l'instance représentant l'utilisateur ayant passé la commande en invoquant la méthode `command.getUser()`. Normalement, JPA doit se débrouiller de charger les instances nécessaires de manière automatique (grâce au mapping). Voici l'exemple de code associé à ce scénario.

```
1  package fr.koor.webstore;
2
3  import javax.persistence.EntityManager;
4  import javax.persistence.EntityManagerFactory;
5  import javax.persistence.Persistence;
6
7  import fr.koor.webstore.business.Command;
8  import fr.koor.webstore.business.User;
9
10 public class Console {
11
```

```
    public static void main(String[] args) throws Exception {
        EntityManagerFactory entityManagerFactory = null;
        EntityManager entityManager = null;

        try {

            entityManagerFactory = Persistence.createEntityManagerFactory("WebStore");
            entityManager = entityManagerFactory.createEntityManager();

            Command command = entityManager.find( Command.class, 1 );
            System.out.println( command );

            User user = command.getUser();
            System.out.println( user );

        } finally {
            if ( entityManager != null ) entityManager.close();
            if ( entityManagerFactory != null ) entityManagerFactory.close();
        }
    }
}
```

Exemple de chargement d'une commande et de l'utilisateur associé.

Voici les résultats produits par cet exemple.

```
[main] WARN  | org.hibernate.orm.connections.pooling                          |
Commande de >> 1: Anderson/Neo - 0 connexion(s) - 2020-01-21 11:42:01.0

1: Anderson/Neo - 0 connexion(s)
```

Mapping d'une relation @ManyToOne avec table d'association.

Pour rappel, le terme de table d'association est aussi parfois appelé table de jointure. Il s'agit d'une table en base de données permettant d'associer deux enregistrements situés dans deux autres tables de la base de données en utilisant des « foreign keys » (des clés étrangères en français).

Parfois, des tables en base de données sont déjà existantes, mais aucune relation ne les relie. Vous pourriez simplement ajouter une clé de référence dans l'une des deux tables pour établir la relation. Mais si d'autres applications utilisent déjà ces tables, l'ajout d'une nouvelle colonne peut poser soucis. Dans ce cas, l'ajout d'une table d'association vous permet de gérer votre problème d'association sans impacter les applications existantes. Le contre coup étant qu'il faut réaliser plus de jointures pour extraire vos données (c'est un peu moins performant).

Implémentation, en base de données.

Voici le code SQL permettant de créer une base de données (pour le SGBDr MariaDB) avec nos deux tables `T_Users` et `T_Commands` ainsi qu'une nouvelle table d'association, appelée `T_Commands_Users_Associations` permettant de réaliser notre relation de type « Many-To-One ».

```
-- ------------------------------------------------------------------------
-- - Reconstruction de la base de données
-- ------------------------------------------------------------------------

DROP DATABASE IF EXISTS ManyToOne;
CREATE DATABASE ManyToOne;
USE ManyToOne;

-- ------------------------------------------------------------------------
-- - Construction de la table des utilisateurs                            -
-- ------------------------------------------------------------------------

CREATE TABLE T_Users (
    idUser              int         PRIMARY KEY AUTO_INCREMENT,
    login               varchar(20) NOT NULL,
    password            varchar(20) NOT NULL,
    connectionNumber    int         NOT NULL DEFAULT 0
);

INSERT INTO T_Users (login, password)
VALUES ( 'Anderson',    'Neo' ),
       ( 'Skywalker',   'Luke' ),
       ( 'Plissken',    'Snake' ),
       ( 'Ripley',      'Ellen' ),
```

```
       ( 'Bond',        'James' );

SELECT * FROM T_Users;

-- -----------------------------------------------------------------------------
-- - Construction de la table des commandes                                    -
-- -----------------------------------------------------------------------------

CREATE TABLE T_Commands (
    idCommand       int         PRIMARY KEY AUTO_INCREMENT,
    commandDate     datetime    NOT NULL DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO T_Commands () VALUES (), (), ();

SELECT * FROM T_Commands;

-- -----------------------------------------------------------------------------
-- - Construction de la table d'association T_Commands/T_Users                 -
-- -----------------------------------------------------------------------------

CREATE TABLE T_Commands_Users_Associations (
    IdCommand   int   NOT NULL REFERENCES T_Commands(IdCommand),
    IdUser      int   NOT NULL UNIQUE REFERENCES T_Users(IdUser)
);

INSERT INTO T_Commands_Users_Associations VALUES (1, 1), (2, 2), (3, 1);

SELECT * FROM T_Commands_Users_Associations;
```

Fichier Database.sql

Pour construire la base de données, veuillez de nouveau exécuter l'ordre `source Database.sql` dans la console de MariaDB. Voici les résultats produits.

```
MariaDB [(none)]> source Database.sql
Query OK, 3 rows affected (0.075 sec)

Query OK, 1 row affected (0.000 sec)

Database changed
Query OK, 0 rows affected (0.012 sec)

Query OK, 5 rows affected (0.002 sec)
Records: 5  Duplicates: 0  Warnings: 0

+--------+-----------+----------+------------------+
| idUser | login     | password | connectionNumber |
+--------+-----------+----------+------------------+
|      1 | Anderson  | Neo      |                0 |
|      2 | Skywalker | Luke     |                0 |
|      3 | Plissken  | Snake    |                0 |
|      4 | Ripley    | Ellen    |                0 |
|      5 | Bond      | James    |                0 |
+--------+-----------+----------+------------------+
5 rows in set (0.000 sec)

Query OK, 0 rows affected (0.014 sec)

Query OK, 3 rows affected (0.002 sec)
Records: 3  Duplicates: 0  Warnings: 0

+-----------+---------------------+
| idCommand | commandDate         |
+-----------+---------------------+
|         1 | 2020-01-21 12:35:06 |
|         2 | 2020-01-21 12:35:06 |
|         3 | 2020-01-21 12:35:06 |
+-----------+---------------------+
3 rows in set (0.000 sec)

Query OK, 0 rows affected (0.033 sec)

Query OK, 3 rows affected (0.006 sec)
Records: 3  Duplicates: 0  Warnings: 0

+-----------+--------+
| IdCommand | IdUser |
+-----------+--------+
|         1 |      1 |
|         2 |      2 |
```

```
|         3 |      1 |
+-----------+--------+
3 rows in set (0.000 sec)

MariaDB [ManyToOne]>
```

Implémentation du mapping JPA.

Rien ne change pour la classe `User`. Laissez-la telle qu'elle.

Pour réaliser une relation d'association de type « Many-To-One » avec table d'association, il faut encore utiliser l'annotation `@ManyToOne`, mais il va falloir la coupler à une annotation `@JoinTable` afin d'y spécifier les informations utiles à la jointure. Voici un extrait de code relatif à la définition d'une telle relation.

```
1  @Entity @Table(name = "T_Commands")
2  public class Command {
3
4      // Autres attributs
5
6      @ManyToOne
7      @JoinTable( name = "T_Commands_Users_Associations",
8                  joinColumns = @JoinColumn( name = "idCommand" ),
9                  inverseJoinColumns = @JoinColumn( name = "idUser" ) )
10     private User user;
11
12     // Suite de la classe
13
14 }
```

Mise en oeuvre d'une relation de type @ManyToOne avec table d'association

Voici le code complet de la classe `Command`.

```
1  package fr.koor.webstore.business;
2
3  import java.util.Date;
4
5  import javax.persistence.Entity;
6  import javax.persistence.GeneratedValue;
7  import javax.persistence.GenerationType;
8  import javax.persistence.Id;
9  import javax.persistence.JoinColumn;
10 import javax.persistence.JoinTable;
11 import javax.persistence.ManyToOne;
12 import javax.persistence.Table;
13
14 @Entity  @Table(name="T_Commands")
15 public class Command {
16
17     @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
18     private int idCommand;
19
20     @ManyToOne
21     @JoinTable( name = "T_Commands_Users_Associations",
22                 joinColumns = @JoinColumn( name = "idCommand" ),
23                 inverseJoinColumns = @JoinColumn( name = "idUser" ) )
24     private User user;
25
26     private Date commandDate;
27
28
29     public Command() {}
30
31     public Command( User user, Date commandDate ) {
32         this.setUser( user );
33         this.setCommandDate( commandDate );
34     }
35
36     public int getIdCommand() {
37         return idCommand;
38     }
39
40     public User getUser() {
41         return user;
42     }
43
44     public void setUser(User user) {
```

```
45          this.user = user;
46      }
47
48      public Date getCommandDate() {
49          return commandDate;
50      }
51
52      public void setCommandDate(Date commandDate) {
53          this.commandDate = commandDate;
54      }
55
56      public String toString() {
57          StringBuilder builder = new StringBuilder();
58          builder.append( "Commande de >> " ).append( this.user )
59                  .append( " - " ).append( this.commandDate ).append( "\n" );
60          return builder.toString();
61      }
62
63  }
```

La classe Command et son mapping JPA

Test de la relation d'association.

Normalement, vous pouvez récupérer le fichier de configuration JPA proposé plus haut dans ce document. Aucune modification ne devrait être nécessaire, sauf si vous avez changé le nom de la base de données.

Vous devriez aussi pouvoir reprendre les exemples d'utilisation de vos classes précédemment proposés et ils devraient correctement fonctionner. Pour rappel, voici un petit exemple d'utilisation.

```
 1 package fr.koor.webstore;
 2
 3 import javax.persistence.EntityManager;
 4 import javax.persistence.EntityManagerFactory;
 5 import javax.persistence.Persistence;
 6
 7 import fr.koor.webstore.business.Command;
 8 import fr.koor.webstore.business.User;
 9
10 public class Console {
11
12     public static void main(String[] args) throws Exception {
13         EntityManagerFactory entityManagerFactory = null;
14         EntityManager entityManager = null;
15         try {
16             entityManagerFactory = Persistence.createEntityManagerFactory("WebStore");
17             entityManager = entityManagerFactory.createEntityManager();
18
19             Command command = entityManager.find( Command.class, 1 );
20             System.out.println( command );
21
22             User user = command.getUser();
23             System.out.println( user );
24
25         } finally {
26             if ( entityManager != null ) entityManager.close();
27             if ( entityManagerFactory != null ) entityManagerFactory.close();
28         }
29     }
30 }
```

Utilisation d'une relation de type @ManyToOne avec table d'association

Les résultats affichés devraient rester inchangés.

```
[main] WARN   | org.hibernate.orm.connections.pooling                          |
Commande de >> 1: Anderson/Neo - 0 connexion(s) - 2020-01-21 12:39:55.0

1: Anderson/Neo - 0 connexion(s)
```

Les informations présentes dans ce site vous sont fournies dans le but de vous aider à acquérir les compétences nécessaires à l'utilisation des langages ou des technologies considérés. Infini Software ne pourra nullement être tenu responsable de l'utilisation des informations présentes dans ce site.

De plus, si vous remarquez des erreurs ou des oublis dans ce document, n'hésitez surtout pas à nous le signaler en envoyant un mail à l'adresse : dominique.liard@infini-software.com.

Les autres marques et les noms de produits cités dans ces documents sont la propriété de leurs éditeurs respectifs.