PISA UNIVERSITY

TASK 1
LARGE-SCALE AND MULTI-STRUCTURED DATABASES

TUTORIAL: JPA WITH HIBERNATE AND MYSQL

ACADEMIC YEAR 2019-2020

STEFANO PETROCCHI, ANDREA TUBAK, FRANCESCO RONCHIERI, ALESSANDRO MADONNA

SUMMARY

SETUP

DEPENDENCIES

In order to use JPA with Hibernate and MySQL the *POM* file has to be edited to add the right dependencies:

```xml
<project … >
    …
    <dependencies>
        <dependency>
            <groupId>org.hibernate.javax.persistence</groupId>
            <artifactId>hibernate-jpa-2.1-api</artifactId>
            <version>1.0.0.Final</version>
            <type>jar</type>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>4.3.1.Final</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.17</version>
            <type>jar</type>
        </dependency>
    </dependencies>
    …
    <name>PisaFlix</name>
</project>
```

- `hibernate-jpa-2.1-api` contains a clean-room definition of JPA APIs intended for use in developing Hibernate JPA implementation.
- `hibernate-entitymanager` contains the Hibernate O/RM implementation of the JPA specification.
- `mysql-connector-java` contains the JDBC driver for MySQL.

PERSISTENCE UNIT

A persistence unit defines a set of all entity classes that are managed by *EntityManager* instances in an application. This set of entity classes represents the data contained within a single data store.

Persistence units are defined by the `persistence.xml` configuration file. The following is an example `persistence.xml` file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence … >
  <persistence-unit name="PisaFlix" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>com.lsmsdbgroup.pisaflix.Entities.Projection</class>
    <class>com.lsmsdbgroup.pisaflix.Entities.Cinema</class>
    <class>com.lsmsdbgroup.pisaflix.Entities.Film</class>
    <class>com.lsmsdbgroup.pisaflix.Entities.User</class>
    <class>com.lsmsdbgroup.pisaflix.Entities.Comment</class>
    <properties>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:mysql://localhost:3306/PisaFlix?serverTimezone=UTC"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password" value="root"/>
    </properties>
  </persistence-unit>
</persistence>
```

- `<provider>` tag has to be set with `org.hibernate.ejb.HibernatePersistence` in order to use Hibernate framework.
- `<class>` is used to specify the classes in the project corresponding in entities in the database.
- `<properties>` is used to specify the database properties like the URL, the driver used, user and password.

ENTITIES

A **JPA entity** class is a POJO (Plain Old Java Object) class, i.e. an ordinary Java class that is marked (annotated) as having the ability to represent objects in the database.

This is an example of an entity class:

```java
package com.lsmsdbgroup.pisaflix.Entities;

import java.io.Serializable;
import java.util.*;
import javax.persistence.*;

@Entity
@Table(name = "User")
public class User implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "idUser", nullable = false, unique = false)
    private Long idUser;

    // …

    @Column(name = "lastName", length = 50, nullable = false, unique = false)
    private String lastName;

    public User() {
    }

    // …

    public Integer getIdUser() {
        return idUser;
    }

    public void setIdUser(Integer idUser) {
        this.idUser = idUser;
    }

    // Other fields, getters and setters…

}
```

- `@Entity` declares the class as an entity.
- `@Id` declares the identifier property of this entity.
- `@GeneratedValue` specifies how to generate identifier.
- `@Table` is set at the class level; it allows to define the table, catalog, and schema names for the entity mapping.

- `@Column` annotation is used to mention the details of the column, i.e. name, length and other properties. Without it, the name of the field will be used as the name of the column.

ENTITY MANAGER

EntityManager is a part of the Java Persistence API, it implements the programming interfaces and lifecycle rules defined by the JPA specification.
Moreover, *Persistence* context can be accessed by using the APIs in *EntityManager*.

In order to use it has to be created an ***EntityManagerFactory*** then the *EntityManager* from the factory:

```java
package com.lsmsdbgroup.pisaflix;

import com.lsmsdbgroup.pisaflix.Entities.*;
import java.util.*;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class DBManager {

    private static EntityManagerFactory factory;
    private static EntityManager entityManager;

    public static void setup() {
       factory = Persistence.createEntityManagerFactory("PisaFlix");
    }

    public static void exit() {
       factory.close();
    }
}
```

In the above code the *createEntityManagerFactory ()* creates a persistence unit by providing the same unique name which is provided for persistence-unit in `persistence.xml` file.

The entitymanagerfactory object will create the *EntityManger* instance by using ***createEntityManager ()*** method.

The *EntityManger* object creates ***EntityTransaction*** instance for transaction management. By using *EntityManger* object, we can persist entities into database.

It is important to close EntityManager and EntityManagerFactory at the end of a transaction and closing the application.

TRANSACTIONS

CREATE

Example of a *create* transaction using JPA and Hibernate:

```java
public static void create(String username, /*…*/, int privilegeLevel) {

    User user = new User();

    user.setUsername(username);
    // …
    user.setPrivilegeLevel(privilegeLevel);

    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        entityManager.persist(user);
        entityManager.getTransaction().commit();
    } catch (Exception ex) {
        System.out.println("A problem occurred in creating the user!");
    } finally {
        entityManager.close();
    }

}
```

Before starting a transaction an *EntityMangager* has to be crated using the EntityManagerFactory and the transaction can be started by using `entityManager.getTransaction().begin()`.

In this case `entityManager.persist(user)` is used to indicate that Hibernate is going to attach the *User* entity to the currently running Persistence Context.

`entityManager.getTransaction().commit()` is used to commit the current resource transaction, writing any unflushed changes to the database.

The `entityManager.close()` method closes the *EntityMangage* releasing its persistence context and other resources.

READ

Example of a *read* transaction using JPA and Hibernate:

```java
public static User read(int userId) {

    User user = null;

    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        user = entityManager.find(User.class, userId);
        if (user == null) {
            System.out.println("User not found!");
        }
    } catch (Exception ex) {
        System.out.println("A problem occurred in retriving a user!");
    } finally {
        entityManager.close();
    }

    return user;

}
```

`entityManager.find(User.class, userId)` find by primary key. Search for an entity of the specified class and primary key. If the entity instance is contained in the persistence context, it is returned from there.

UPDATE

Example of an *update* transaction using JPA and Hibernate:

```java
public static void update(int idUser, /*…*/, int privilegeLevel) {

    User user = new User(idUser);

    user.setUsername(username);
    // …
    user.setPrivilegeLevel(privilegeLevel);

    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        entityManager.merge(user);
        entityManager.getTransaction().commit();
    } catch (Exception ex) {
        System.out.println("A problem occurred in updating a user!");
    } finally {
        entityManager.close();
    }

}
```

The `entityManager.merge(user)` operation is used to *merge* the changes made to a detached object into the persistence context. *merge* does not directly update the object into the database, it merges the changes into the persistence context, then `entityManager.getTransaction().commit()` is used to commit the changes to the database.

DELETE

Example of a *delete* transaction using JPA and Hibernate:

```java
public static void delete(int userId) {

    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        User reference = entityManager.getReference(User.class, userId);
        entityManager.remove(reference);
        entityManager.getTransaction().commit();
    } catch (Exception ex) {
        ex.printStackTrace();
        System.out.println("A problem occurred in deleting a User!");
    } finally {
        entityManager.close();
    }

}
```

The `getReference(User.class, userId)` method works like the `find` method except that if the entity object is not already managed by the *EntityMangage* a *hollow* object might be returned (`null` is never returned). A hollow object is initialized with the valid primary key, but all its other persistent fields are uninitialized. The method is useful when a reference to an entity object is required but not its content.

The `entityManager.remove(reference)` operation is used to *delete* an object from the database. *remove* does not directly delete the object from the database, it marks the object to be deleted in the persistence context, then `entityManager.getTransaction().commit()` is used to commit the changes to the database.

QUERY

The `EntityManager.createQuery()` and `EntityManager.createNamedQuery()` methods are used to query the datastore by using Java Persistence query language queries.

The `EntityManager.createQuery()` method is used to create ***dynamic queries***, which are queries defined directly within an application's business logic:

```java
public static Set<User> getByName(String name) {

    Set<User> users = null;

    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        users = entityManager.createQuery("SELECT u FROM User u WHERE u.name = '"
                                           + name + "'").getResultList();
        if (users == null) {
            System.out.println("No one with that name!");
        }
    } catch (Exception ex) {
        System.out.println("A problem occurred in retrieving the users!");
    } finally {
        entityManager.close();
    }

    return users;

}
```

The `EntityManager.createNamedQuery()` method is used to create ***static queries***, or queries that are defined in metadata by using the `javax.persistence.NamedQuery` annotation. The `name` element of `@NamedQuery` specifies the name of the query that will be used with the method and the `query` element is the query:

```java
@NamedQuery(
    name="selectUsersByName",
    query="SELECT u FROM user u WHERE u.name = :userName"
)
```

Example of a *static query*:

```java
public static Set<User> getByName(String name) {

    Set<User> users = null;

    // …

        entityManager.getTransaction().begin();
        users = entityManager.createNamedQuery("selectUsersByName").setParameter("username
", name).getResultList();

    // …

    return users;
}
```

ASSOCIATIONS

JPA ASSOCIATIONS

ASSOCIATIONS TYPES

Hibernate is a persistence framework for ***plain old java object*** (an ordinary java object), therefore association types that can be used are mainly those monodirectional, typical of java. Those associations are typically expressed with references or collections in function of the multiplicity of the associations. JPA specific provide four types of association.

ONE-TO-ONE

One instance of an entity is associated with only one instance of another entity.
They are expressed with the annotation `javax.persistence.OneToOne` in correspondence of the persistence object's property or field.
Exemple: a cinema can have only one address.

ONE-TO-MANY

One instance of an entity is associated with multiple instances of another entity.
They are expressed with the annotation `javax.persistence.OneToMany` in correspondence of the persistence object's property or field.
Exemple: a cinema can have multiple film projections.

MANY-TO-ONE

Many instances of an entity are associated with only one instance of another entity.
Is the opposite of one-to-many association. They are expressed with the annotation `javax.persistence.ManyToOne` in correspondence of the persistence object's property or field.
Exemple: multiple comments can be written by one user.

MANY-TO-MANY

Many instances of an entity are associated with multiple instances of another entity.
They are expressed with the annotation `javax.persistence.ManyToMany` in correspondence of the persistence object's property or field.
Exemple: multiple cinemas can be the favourites of multiple users.

FETCHING TYPES

Every association can use different fetching types to retrieve the information from the database. The two possible fetch types are ***eager*** and ***lazy***.

EAGER

The association objects must be retrieved immediately with the object that uses them.

LAZY

If an object has a set of association objects, using a lazy-load, they are not retrieved with the object. They will be loaded when requested to do so. Lazy-loading can help improve the performance significantly since often you won't need all the association objects and so they will not be loaded. Beware that Hibernate will not actually load all objects when accessing the collection. Instead, it will load each object individually. When iterating over the set, this causes a query for every object.

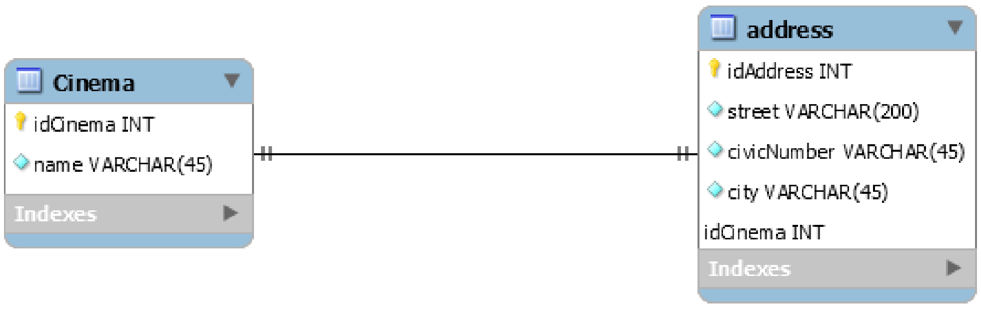
CASCADE ACTIONS

JPA does offer possibility to *cascade* operations to associated entities. Logic is in JPA and does not utilize database cascades. There is no JPA standard compliant way to do cascades with database cascades. The cascade types are:

- **Merge:** *merge()* operations cascade to related entities.
- **Persist:** *save()* or *persist()* operations cascade to related entities.
- **Refresh:** *refresh()* operations cascade to related entities.
- **Remove:** *remove()* operations cascade to related entities.
- **All:** is shorthand for all of the above cascade operations.

ONE-TO-ONE ASSOCIATION

Considering the association existing between a cinema and his address, represented by the model in the image below, where a cinema can have only one address:



The SQL script that creates the two tables is:

```sql
CREATE TABLE IF NOT EXISTS `PisaFlix`.`cinema` (
  `idCinema` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`idCinema`))
ENGINE = InnoDB;

CREATE TABLE IF NOT EXISTS `PisaFlix`.`address` (
  `idAddress` INT NOT NULL AUTO_INCREMENT,
  `street` VARCHAR(200) NOT NULL,
  `civicNumber` VARCHAR(45) NOT NULL,
  `city` VARCHAR(45) NOT NULL,
  `idCinema` INT UNSIGNED NOT NULL,
  PRIMARY KEY (`idAddress`, `idCinema`),
  INDEX `fk_address_cinema1_idx` (`idCinema` ASC),
  CONSTRAINT `fk_address_Cinema1`
    FOREIGN KEY (`idCinema`)
    REFERENCES `PisaFlix`.`Cinema` (`idCinema`)
    ON DELETE CASCADE
    ON UPDATE CASCADE)
ENGINE = InnoDB;
```

Using the annotations, the mapping between class *Cinema* and *Address* is:

```java
@Entity
@Table( name = "CINEMA" )
public class Cinema {
    private int idCinema;
    private String name;
    private Address address;

    @Id
    public String getIdCinema() {
        return idCinema;
    }

    // Other fields, getters and setters…

    @OneToOne(cascade = CascadeType.ALL, mappedBy = "idCinema", fetch = FetchType.EAGER)
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
}
```

Using the annotations, the mapping between class *Address* and *Cinema* is:

```java
@Entity
@Table( name = "ADDRESS" )
public class Address{
    private Integer idAddress;
    private String street;
    private String civicNumber;
    private String city;
    private Cinema cinema;

    @Id
    public Integer getIdAddress() {
        return idAddress;
    }

    // Other fields, getters and setters…

    @OneToOne( fetch = FetchType.EAGER )
    @JoinColumn( name ="idCinema" )
    public Cinema getCinema() {
        return cinema;
    }
}
```

The fetching type can be set by using `fetch = FetchType.EAGER` (or lazy) after the `@OneToOne` tag.
The annotation `@JoinColumn` is used to indicate the column in ***address*** table that contain the reference value of ***cinema*** table.
An optional possibility is to map the inverse association between ***cinema*** and ***address*** by indicating the property of *Address* class that map the association, using `mappedBy = "idCinema"` after the `@OneToOne` tag.
To indicate the cascade actions on the related entity is necessary to set `cascade = CascadeType.ALL` (or other actions) after `@OneToOne` tag.

Retrieve a cinema and is address can be done using:

```java
Cinema cinema = entityManager.find(Cinema.class, idCinema);
System.out.println( cinema.toString() + ", " + cinema.getAddress().toString());
```

Retrieve an address and the cinema associated to it can be done in the same way:

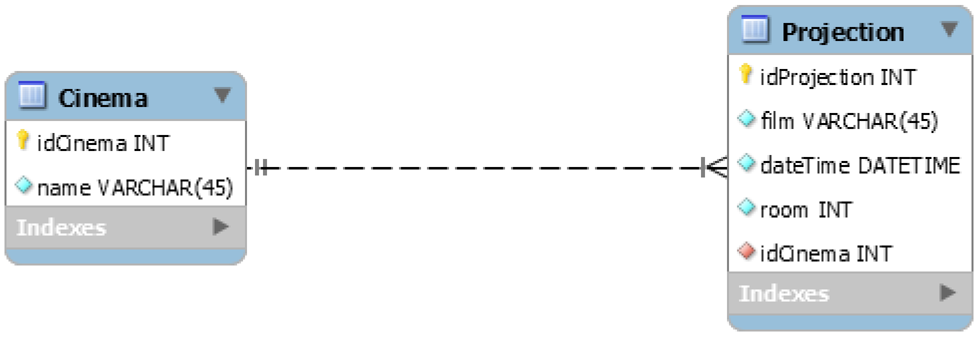
```
Address address = entityManager.find(Address.class, idAddress);
System.out.println( address.getCinema().toString() + ", " + address.toString());
```

Create/Modify the address can be done using:

```
Address = new Address(/*values*/);
cinema.setAddress(newAddress);
entityManager.merge(Cinema);
```

ONE-TO-MANY AND MANY-TO-ONE ASSOCIATIONS

Considering the association existing between a cinema and film projections, represented by the model in the image below, where a cinema can have many projections:



The SQL script that creates the two tables is:

```
CREATE TABLE IF NOT EXISTS `PisaFlix`.`Cinema` (
  `idCinema` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`idCinema`))
ENGINE = InnoDB;

CREATE TABLE IF NOT EXISTS `PisaFlix`.`Projection` (
  `idProjection` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `film` VARCHAR(45) NOT NULL,
  `dateTime` DATETIME NOT NULL,
  `room` INT UNSIGNED NOT NULL,
  `idCinema` INT UNSIGNED NOT NULL,
  INDEX `fk_Projection_Cinema1_idx` (`idCinema` ASC),
  PRIMARY KEY (`idProjection`),
  CONSTRAINT `fk_Projection_Cinema1`
    FOREIGN KEY (`idCinema`)
    REFERENCES `PisaFlix`.`Cinema` (`idCinema`)
    ON DELETE CASCADE
    ON UPDATE CASCADE)
ENGINE = InnoDB;
```

Using the annotations, the mapping between class *Cinema* and *Projection* is:

```
@Entity
@Table( name = "CINEMA" )
public class Cinema {
    private int idCinema;
    private String name;
    private Collection<Projection> projections;

    @Id
    public String getIdCinema() {
        return idCinema;
    }

    // Other fields, getters and setters…

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "idCinema", fetch = FetchType.EAGER)
    public Set<Projection> getProjections() {
        return projections;
    }
    public void setProjections(Set<Projection> Projections) {
        this.projections = projections;
    }
}
```

Using the annotations, the mapping between class *Projection* and *Cinema* is:

```
@Entity
@Table( name = "PROJECTION" )
public class Projection {
    private int idProjection;
    // …
    private Cinema cinema;

    @Id
    public String getIdProjection() {
        return idProjection;
    }

    // Other fields, getters and setters…

    @JoinColumn(name = "idCinema")
    @ManyToOne(fetch = FetchType.EAGER)
    public Cinema getCinema() {
        return cinema;
    }
    public void setCinema(Cinema cinema) {
        this.cinema = cinema;
    }
}
```

The annotation `@JoinColumn` is used to indicate the column in ***projection*** table that contain the reference value of ***cinema*** table.
Mapping the inverse association between ***cinema*** and ***address*** can be done by indicating the property of *Projection* class that map the association, using `mappedBy = "idCinema"` after the `@OneToMany` tag.

Retrieve the list of all projection associated to a cinema can be done using:

```
Cinema cinema = entityManager.find( Cinema.class, idCinema);
for ( Projection projection : cinema.getProjections() ) {
    System.out.println( projection.toString() );
}
```

Create/Modify the projections related to a cinema can be done using:

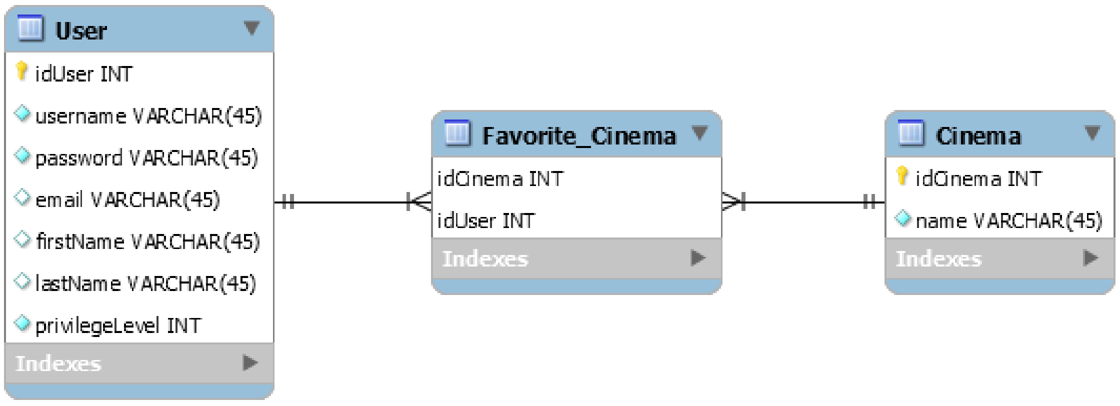
```
cinema.setProjections(newProjectionsSet);
entityManager.merge(Cinema);
```

Create/Modify the cinema related to a projection can be done using:

```
projection.setCinema(newCinema);
entityManager.merge(Projection);
```

MANY-TO-MANY ASSOCIATION

Considering the association existing between users and favourites cinemas, represented by the model in the image below, where multiple users can have many favourites cinemas:



The SQL script that creates the tables is:

```sql
CREATE TABLE IF NOT EXISTS `PisaFlix`.`User` (
  `idUser` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(45) NOT NULL,
  `password` VARCHAR(45) NOT NULL,
  `email` VARCHAR(45) NULL DEFAULT NULL,
  `firstName` VARCHAR(45) NULL DEFAULT NULL,
  `lastName` VARCHAR(45) NULL DEFAULT NULL,
  `privilegeLevel` INT UNSIGNED NOT NULL,
  PRIMARY KEY (`idUser`),
  UNIQUE INDEX `username_UNIQUE` (`username` ASC),
  UNIQUE INDEX `idUser_UNIQUE` (`idUser` ASC))
ENGINE = InnoDB;

CREATE TABLE IF NOT EXISTS `PisaFlix`.`Cinema` (
  `idCinema` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`idCinema`))
ENGINE = InnoDB;

CREATE TABLE IF NOT EXISTS `PisaFlix`.`Favorite_Cinema` (
  `idCinema` INT UNSIGNED NOT NULL,
  `idUser` INT UNSIGNED NOT NULL,
  PRIMARY KEY (`idCinema`, `idUser`),
  INDEX `fk_Favorite_cinema_User1_idx` (`idUser` ASC),
  INDEX `fk_Favorite_cinema_Cinema1_idx` (`idCinema` ASC),
  CONSTRAINT `fk_Favorite_Cinema_Cinema1`
    FOREIGN KEY (`idCinema`)
    REFERENCES `PisaFlix`.`Cinema` (`idCinema`)
    ON DELETE NO ACTION
    ON UPDATE CASCADE,
  CONSTRAINT `fk_Favorite_Cinema_User_User1`
    FOREIGN KEY (`idUser`)
    REFERENCES `PisaFlix`.`User` (`idUser`)
    ON DELETE NO ACTION
    ON UPDATE CASCADE)
ENGINE = InnoDB;
```

Using the annotations, the mapping between class *User* and *Cinema* is:

```java
@Entity
@Table( name = "USER" )
public class User {
    private int idUser;
    // …
    private Set<Cinema> favoriteCinemas;

    @Id
    public String getIdUser() {
        return idUser;
    }

    // Other fields, getters and setters…

    @ManyToMany(mappedBy = "userSet", fetch = FetchType.EAGER, cascade = {CascadeType.MERGE, CascadeType.PERSIST})
    public Set<Cinema> getFavoriteCinemas () {
        return favoriteCinemas;
    }
    public void setFavoriteCinemas(Set<Cinema> favoriteCinemas) {
        this.favoriteCinemas = favoriteCinemas;
    }
}
```

Using the annotations, the mapping between class *Cinema* and *User* is:

```java
@Entity
@Table( name = "CINEMA" )
public class Cinema {
    private int idCinema;
    private String name;
    private Set<User> users;

    @Id
    public String getIdCinema() {
        return idCinema;
    }

    // Other fields, getters and setters…

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "Favorite_Cinema", joinColumns = {
    @JoinColumn(name = "idCinema", referencedColumnName = "idCinema")}, inverseJoinColumns =
    @JoinColumn(name = "idUser", referencedColumnName = "idUser")})
    public Set<User> getUsers() {
        return Users;
    }
}
```

The ***favorite_cinema*** table has not a model in the project but is simply referenced by the annotation `@JoinTable` that is used to indicate the mapping between ***user*** and ***cinema*** entities.

Retrieve the list of all favourite cinemas of a user can be done using:

```java
User user = entityManager.find( User.class, idUser);
for ( Cinema cinema : cinema.getFavoriteCinemas() ) {
    System.out.println( cinema.toString() );
}
```

Create/Modify user's favourite cinemas:

```java
user.setFavoriteCinemas (newCinemaSet);
entityManager.merge(User);
```