

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, Alsessandro Madonna

Summary

[Setup 3](#_Toc23781942)

[Dependencies 3](#_Toc23781943)

[Persistence Unit 3](#_Toc23781944)

[Entities 4](#_Toc23781945)

[Entity Manager 5](#_Toc23781946)

[Transactions 6](#_Toc23781947)

[Create 6](#_Toc23781948)

[Read 6](#_Toc23781949)

[Update 7](#_Toc23781950)

[Delete 7](#_Toc23781951)

[Query 8](#_Toc23781952)

[Associations 9](#_Toc23781953)

[JPA Associations 9](#_Toc23781954)

[Associations Types 9](#_Toc23781955)

[Fetching Types 9](#_Toc23781956)

[Cascade Actions 10](#_Toc23781957)

[One-to-One Association 10](#_Toc23781958)

[One-to-Many and Many-to-One Associations 12](#_Toc23781959)

[Many-to-Many Association 14](#_Toc23781960)

# Setup

## Dependencies

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

1. **<project** … **>**
2. **…**
3. **<dependencies>**
4. **<dependency>**
5. **<groupId>**org.hibernate.javax.persistence**</groupId>**
6. **<artifactId>**hibernate-jpa-2.1-api**</artifactId>**
7. **<version>**1.0.0.Final**</version>**
8. **<type>**jar**</type>**
9. **</dependency>**
10. **<dependency>**
11. **<groupId>**org.hibernate**</groupId>**
12. **<artifactId>**hibernate-entitymanager**</artifactId>**
13. **<version>**4.3.1.Final**</version>**
14. **</dependency>**
15. **<dependency>**
16. **<groupId>**mysql**</groupId>**
17. **<artifactId>**mysql-connector-java**</artifactId>**
18. **<version>**8.0.17**</version>**
19. **<type>**jar**</type>**
20. **</dependency>**
21. **</dependencies>**
22. **…**
23. **<name>**PisaFlix**</name>**
24. **</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:

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<persistence** … **>**
3. **<persistence-unit** name="PisaFlix" transaction-type="RESOURCE\_LOCAL"**>**
4. **<provider>**org.hibernate.ejb.HibernatePersistence**</provider>**
5. **<class>**com.lsmsdbgroup.pisaflix.Entities.Projection**</class>**
6. **<class>**com.lsmsdbgroup.pisaflix.Entities.Cinema**</class>**
7. **<class>**com.lsmsdbgroup.pisaflix.Entities.Film**</class>**
8. **<class>**com.lsmsdbgroup.pisaflix.Entities.User**</class>**
9. **<class>**com.lsmsdbgroup.pisaflix.Entities.Comment**</class>**
10. **<properties>**
11. **<property** name="javax.persistence.jdbc.url"
12. value="jdbc:mysql://localhost:3306/PisaFlix?serverTimezone=UTC"**/>**
13. **<property** name="javax.persistence.jdbc.user" value="root"**/>**
14. **<property** name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"**/>**
15. **<property** name="javax.persistence.jdbc.password" value="root"**/>**
16. **</properties>**
17. **</persistence-unit>**
18. **</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:

1. **package** com.lsmsdbgroup.pisaflix.Entities;
2. **import** java.io.Serializable;
3. **import** java.util.\*;
4. **import** javax.persistence.\*;
6. @Entity
7. @Table(name = "User")
8. **public** **class** User **implements** Serializable {
10. @Id
11. @GeneratedValue(strategy = GenerationType.AUTO)
12. @Column(name = "idUser", nullable = **false**, unique = **false**)
13. **private** Long idUser;
15. // …
17. @Column(name = "lastName", length = 50, nullable = **false**, unique = **false**)
18. **private** String lastName;
20. **public** User() {
21. }
23. // …
25. **public** Integer getIdUser() {
26. **return** idUser;
27. }
29. **public** **void** setIdUser(Integer idUser) {
30. **this**.idUser = idUser;
31. }
33. // Other fields, getters and setters…
35. }

* @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 ithas to be created an ***EntityManagerFactory*** then the *EntityManager* from the factory:

1. **package** com.lsmsdbgroup.pisaflix;
3. **import** com.lsmsdbgroup.pisaflix.Entities.\*;
4. **import** java.util.\*;
5. **import** javax.persistence.EntityManager;
6. **import** javax.persistence.EntityManagerFactory;
7. **import** javax.persistence.Persistence;
9. **public** **class** DBManager {
11. **private** **static** EntityManagerFactory factory;
12. **private** **static** EntityManager entityManager;
14. **public** **static** **void** setup() {
15. factory = Persistence.createEntityManagerFactory("PisaFlix");
16. }
18. **public** **static** **void** exit() {
19. factory.close();
20. }
21. }

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:

1. **public** **static** **void** create(String username, /\*…\*/, **int** privilegeLevel) {
3. User user = **new** User();
4. user.setUsername(username);
5. // …
6. user.setPrivilegeLevel(privilegeLevel);
8. **try** {
9. entityManager = factory.createEntityManager();
10. entityManager.getTransaction().begin();
11. entityManager.persist(user);
12. entityManager.getTransaction().commit();
13. } **catch** (Exception ex) {
14. System.out.println("A problem occurred in creating the user!");
15. } **finally** {
16. entityManager.close();
17. }
18. }

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:

1. **public** **static** User read(**int** userId) {
2. User user = **null**;
3. **try** {
4. entityManager = factory.createEntityManager();
5. entityManager.getTransaction().begin();
6. user = entityManager.find(User.**class**, userId);
7. **if** (user == **null**) {
8. System.out.println("User not found!");
9. }
10. } **catch** (Exception ex) {
11. System.out.println("A problem occurred in retriving a user!");
12. } **finally** {
13. entityManager.close();
14. }
15. **return** user;
16. }

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:

1. **public** **static** **void** update(int idUser, /\*…\*/, **int** privilegeLevel) {
3. User user = **new** User(idUser);
4. user.setUsername(username);
5. // …
6. user.setPrivilegeLevel(privilegeLevel);
8. **try** {
9. entityManager = factory.createEntityManager();
10. entityManager.getTransaction().begin();
11. entityManager.merge(user);
12. entityManager.getTransaction().commit();
13. } **catch** (Exception ex) {
14. System.out.println("A problem occurred in updating a user!");
15. } **finally** {
16. entityManager.close();
17. }
18. }

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:

1. **public** **static** **void** delete(**int** userId) {
2. **try** {
3. entityManager = factory.createEntityManager();
4. entityManager.getTransaction().begin();
5. User reference = entityManager.getReference(User.**class**, userId);
6. entityManager.remove(reference);
7. entityManager.getTransaction().commit();
8. } **catch** (Exception ex) {
9. ex.printStackTrace();
10. System.out.println("A problem occurred in deleting a User!");
11. } **finally** {
12. entityManager.close();
13. }
14. }

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:

1. **public** **static** Collection<User> getByName(String name) {
3. Collection<User> users = **null**;
4. **try** {
5. entityManager = factory.createEntityManager();
6. entityManager.getTransaction().begin();
7. users = entityManager.createQuery("SELECT u FROM User u WHERE u.name = '"
8. + name + "'").getResultList();
9. **if** (users == **null**) {
10. System.out.println("No one with that name!");
11. }
12. } **catch** (Exception ex) {
13. System.out.println("A problem occurred in retrieving the users!");
14. } **finally** {
15. entityManager.close();
16. }
17. **return** users;
18. }

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:

1. @NamedQuery(
2. name="selectUsersByName",
3. query="SELECT u FROM user u WHERE u.name = :userName"
4. )

Example of a *static query*:

1. **public** **static** Collection<User> getByName(String name) {
3. Collection<User> users = **null**;
4. // …
5. entityManager.getTransaction().begin();
6. users = entityManager.createNamedQuery("selectUsersByName").setParameter(“username”, name).getResultList();
7. // …
8. **return** users;
9. }

# 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 collection 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 collection, this causes a query for every object.

### Cascade Actions

JPA does offer possibility to [*cascade*](http://docs.oracle.com/javaee/6/api/javax/persistence/CascadeType.html) 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:

Immagine che contiene screenshot

Descrizione generata automaticamente

**The SQL script that creates the two tables is:**

1. **CREATE** **TABLE** IF NOT EXISTS `PisaFlix`.`cinema` (
2. `idCinema` **INT** UNSIGNED NOT NULL AUTO\_INCREMENT,
3. `**name**` **VARCHAR**(45) NOT NULL,
4. **PRIMARY** **KEY** (`idCinema`))
5. ENGINE = InnoDB;
6. **CREATE** **TABLE** IF NOT EXISTS `PisaFlix`.`address` (
7. `idAddress` **INT** NOT NULL AUTO\_INCREMENT,
8. `street` **VARCHAR**(200) NOT NULL,
9. `civicNumber` **VARCHAR**(45) NOT NULL,
10. `city` **VARCHAR**(45) NOT NULL,
11. `idCinema` **INT** UNSIGNED NOT NULL,
12. **PRIMARY** **KEY** (`idAddress`, `idCinema`),
13. **INDEX** `fk\_address\_cinema1\_idx` (`idCinema` **ASC**),
14. **CONSTRAINT** `fk\_address\_Cinema1`
15. **FOREIGN** **KEY** (`idCinema`)
16. **REFERENCES** `PisaFlix`.`Cinema` (`idCinema`)
17. **ON** **DELETE** **CASCADE**
18. **ON** **UPDATE** **CASCADE**)
19. ENGINE = InnoDB;

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

1. @Entity
2. @Table( name = "CINEMA" )
3. **public** **class** Cinema {
4. **private** **int** idCinema;
5. **private** String name;
6. **private** Address address;
8. @Id
9. **public** String getIdCinema() {
10. **return** idCinema;
11. }
12. // Other fields, getters and setters…
14. @OneToOne(cascade = CascadeType.ALL, mappedBy = "idCinema", fetch = FetchType.EAGER)
15. **public** Address getAddress() {
16. **return** address;
17. }
18. **public** **void** setAddress(Address address) {
19. **this**.address = address;
20. }
21. }

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

1. @Entity
2. @Table( name = "ADDRESS" )
3. **public** **class** Address{
4. **private** Integer idAddress;
5. **private** String street;
6. **private** String civicNumber;
7. **private** String city;
8. **private** Cinema cinema;
10. @Id
11. **public** Integer getIdAddress() {
12. **return** idAddress;
13. }
15. // Other fields, getters and setters…
17. @OneToOne( fetch = FetchType.EAGER )
18. @JoinColumn( name ="idCinema" )
19. **public** Cinema getCinema() {
20. **return** cinema;
21. }
22. }

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:**

1. Cinema cinema = entityManager.find(Cinema.**class**, idCinema);
2. System.out.println( cinema.toString() + ", " + cinema.getAddress().toString());

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

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

**Create/Modify the address can be done using:**

1. Address = **new** Address(/\*values\*/);
2. cinema.setAddress(newAddress);
3. 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:

Immagine che contiene screenshot

Descrizione generata automaticamente

**The SQL script that creates the two tables is:**

1. **CREATE** **TABLE** IF NOT EXISTS `PisaFlix`.`Cinema` (
2. `idCinema` **INT** UNSIGNED NOT NULL AUTO\_INCREMENT,
3. `**name**` **VARCHAR**(45) NOT NULL,
4. **PRIMARY** **KEY** (`idCinema`))
5. ENGINE = InnoDB;
7. **CREATE** **TABLE** IF NOT EXISTS `PisaFlix`.`Projection` (
8. `idProjection` **INT** UNSIGNED NOT NULL AUTO\_INCREMENT,
9. `film` **VARCHAR**(45) NOT NULL,
10. `dateTime` DATETIME NOT NULL,
11. `room` **INT** UNSIGNED NOT NULL,
12. `idCinema` **INT** UNSIGNED NOT NULL,
13. **INDEX** `fk\_Projection\_Cinema1\_idx` (`idCinema` **ASC**),
14. **PRIMARY** **KEY** (`idProjection`),
15. **CONSTRAINT** `fk\_Projection\_Cinema1`
16. **FOREIGN** **KEY** (`idCinema`)
17. **REFERENCES** `PisaFlix`.`Cinema` (`idCinema`)
18. **ON** **DELETE** **CASCADE**
19. **ON** **UPDATE** **CASCADE**)
20. ENGINE = InnoDB;

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

1. @Entity
2. @Table( name = "CINEMA" )
3. **public** **class** Cinema {
4. **private** **int** idCinema;
5. **private** String name;
6. **private** Collection<Projection> projections;
8. @Id
9. **public** String getIdCinema() {
10. **return** idCinema;
11. }
13. // Other fields, getters and setters…
15. @OneToMany(cascade = CascadeType.ALL, mappedBy = "idCinema", fetch = FetchType.EAGER)
16. **public** Collection<Projection> getProjections() {
17. **return** projections;
18. }
19. **public** **void** setProjections(Collection<Projection> Projections) {
20. **this**.projections = projections;
21. }
22. }

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

1. @Entity
2. @Table( name = "PROJECTION" )
3. **public** **class** Projection {
4. **private** **int** idProjection;
5. // …
6. **private** Cinema cinema;
8. @Id
9. **public** String getIdProjection() {
10. **return** idProjection;
11. }
13. // Other fields, getters and setters…
15. @JoinColumn(name = "idCinema")
16. @ManyToOne(fetch = FetchType.EAGER)
17. **public** Cinema getCinema() {
18. **return** cinema;
19. }
20. **public** **void** setCinema(Cinema cinema) {
21. **this**.cinema = cinema;
22. }
23. }

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:**

1. Cinema cinema = entityManager.find( Cinema.**class**, idCinema);
2. **for** ( Projection projection : cinema.getProjections() ) {
3. System.out.println( projection.toString() );
4. }

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

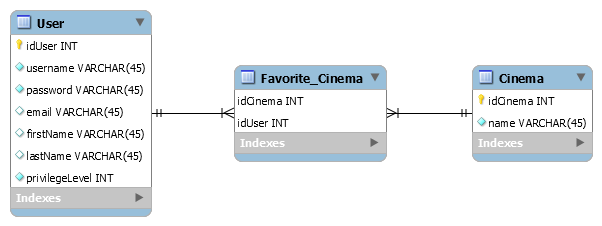
1. cinema.setProjections(newProjectionsCollection);
2. entityManager.merge(Cinema);

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

1. projection.setCinema(newCinema);
2. 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:**

1. **CREATE** **TABLE** IF NOT EXISTS `PisaFlix`.`User` (
2. `idUser` **INT** UNSIGNED NOT NULL AUTO\_INCREMENT,
3. `username` **VARCHAR**(45) NOT NULL,
4. `**password**` **VARCHAR**(45) NOT NULL,
5. `email` **VARCHAR**(45) NULL **DEFAULT** NULL,
6. `firstName` **VARCHAR**(45) NULL **DEFAULT** NULL,
7. `lastName` **VARCHAR**(45) NULL **DEFAULT** NULL,
8. `privilegeLevel` **INT** UNSIGNED NOT NULL,
9. **PRIMARY** **KEY** (`idUser`),
10. **UNIQUE** **INDEX** `username\_UNIQUE` (`username` **ASC**),
11. **UNIQUE** **INDEX** `idUser\_UNIQUE` (`idUser` **ASC**))
12. ENGINE = InnoDB;
14. **CREATE** **TABLE** IF NOT EXISTS `PisaFlix`.`Cinema` (
15. `idCinema` **INT** UNSIGNED NOT NULL AUTO\_INCREMENT,
16. `**name**` **VARCHAR**(45) NOT NULL,
17. **PRIMARY** **KEY** (`idCinema`))
18. ENGINE = InnoDB;
20. **CREATE** **TABLE** IF NOT EXISTS `PisaFlix`.`Favorite\_Cinema` (
21. `idCinema` **INT** UNSIGNED NOT NULL,
22. `idUser` **INT** UNSIGNED NOT NULL,
23. **PRIMARY** **KEY** (`idCinema`, `idUser`),
24. **INDEX** `fk\_Favorite\_cinema\_User1\_idx` (`idUser` **ASC**),
25. **INDEX** `fk\_Favorite\_cinema\_Cinema1\_idx` (`idCinema` **ASC**),
26. **CONSTRAINT** `fk\_Favorite\_Cinema\_Cinema1`
27. **FOREIGN** **KEY** (`idCinema`)
28. **REFERENCES** `PisaFlix`.`Cinema` (`idCinema`)
29. **ON** **DELETE** **NO** **ACTION**
30. **ON** **UPDATE** **CASCADE**,
31. **CONSTRAINT** `fk\_Favorite\_Cinema\_User\_User1`
32. **FOREIGN** **KEY** (`idUser`)
33. **REFERENCES** `PisaFlix`.`User` (`idUser`)
34. **ON** **DELETE** **NO** **ACTION**
35. **ON** **UPDATE** **CASCADE**)
36. ENGINE = InnoDB;

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

1. @Entity
2. @Table( name = "USER" )
3. **public** **class** User {
4. **private** **int** idUser;
5. // …
6. **private** Collection<Cinema> favoriteCinemas;
8. @Id
9. **public** String getIdUser() {
10. **return** idUser;
11. }
13. // Other fields, getters and setters…
15. @ManyToMany(mappedBy = "userCollection", fetch = FetchType.EAGER, cascade = {CascadeType.MERGE, CascadeType.PERSIST})
16. **public** Collection<Cinema> getFavoriteCinemas () {
17. **return** favoriteCinemas;
18. }
19. **public** **void** setFavoriteCinemas(Collection<Cinema> favoriteCinemas) {
20. **this**.favoriteCinemas = favoriteCinemas;
21. }
22. }

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

1. @Entity
2. @Table( name = "CINEMA" )
3. **public** **class** Cinema {
4. **private** **int** idCinema;
5. **private** String name;
6. **private** Collection<User> users;
8. @Id
9. **public** String getIdCinema() {
10. **return** idCinema;
11. }
13. // Other fields, getters and setters…
15. @ManyToMany(fetch = FetchType.EAGER)
16. @JoinTable(name = "Favorite\_Cinema", joinColumns = {
17. @JoinColumn(name = "idCinema", referencedColumnName = "idCinema")}, inverseJoinColumns =
18. @JoinColumn(name = "idUser", referencedColumnName = "idUser")})
19. **public** Collection<User> getUsers() {
20. **return** Users;
21. }
22. }

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:**

1. User user = entityManager.find( User.**class**, idUser);
2. **for** ( Cinema cinema : cinema.getFavoriteCinemas() ) {
3. System.out.println( cinema.toString() );
4. }

**Create/Modify user’s favourite cinemas:**

1. user.setFavoriteCinemas (newCinemaCollection);
2. entityManager.merge(User);