

Pisa University  
  
  
TASK 1  
LARGE-SCALE AND MULTI-STRUCTURED DATABASES

**tutorial: JPA with Hibernate**  
academic year 2019-2020  
  
  
  
  
stefano petrocchi, Andrea Tubak, Francesco Ronchieri, Alsessandro Madonna

Summary

[Associations 3](#_Toc23508817)

[JPA Associations 3](#_Toc23508818)

[Associations Types 3](#_Toc23508819)

[Fetching Types 3](#_Toc23508820)

[One-to-One Association 4](#_Toc23508821)

# 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 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 have been 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 films 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. }
13. // Other getter and setter
15. @OneToOne(cascade = CascadeType.ALL, mappedBy = "idCinema", fetch = FetchType.EAGER)
16. **public** Address getAddress() {
17. **return** address;
18. }
19. **public** **void** setAddress(Address address) {
20. **this**.address = address;
21. }
22. }

**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 getter and setter
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 “on-update” and “on-delete” actions on the association object 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);