2025 — 2026

1TI – 1ACS



UCLL HOGESCHOOL

Back-End Development

JPA — Inheritance, OneToMany & ManyToMany

N. Jacobs, J. Pieck, E. Steegmans, B. Van Impe, S. Van Peborgh, V. Witters

CONTENTS

- Inheritance
- OneToMany
- ManyToMany

Inheritance

OneToMany

ManyToMany

Inheritance

INHERITANCE

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "species",
      discriminatorType = DiscriminatorType.STRING)
public class Animal {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   // For JPA, not public on purpose
   protected Animal() {}

   // ...
}
```

```java
@Entity
@DiscriminatorValue("pony")
public class Pony extends Animal {

   private String name;

   private int age;

   // ...
}
```

- The superclass indicates how the classes are translated to rows
  - In this case the superclass and all subclasses are stored in one table
  - Other strategies exist
- The “species” column holds a value that allows JPA to translate the row to the right subclass

INHERITANCE

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "species", discriminatorType = DiscriminatorType.STRING)
public class Animal {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // ...
}
```

```java
@Entity
@DiscriminatorValue("chicken")
public class Chicken extends Animal {

    private boolean laysEggs;

    // ...
}
```

```java
@Entity
@DiscriminatorValue("pony")
public class Pony extends Animal {

    private String name;

    private int age;

    // ...
}
```

```sql
DROP TABLE IF EXISTS ANIMAL;

CREATE TABLE ANIMAL(
    ID BIGINT PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(255),
    AGE INT,
    LAYS_EGGS BOOLEAN,
    SPECIES VARCHAR(255)
);
```

- Our database schema should accomodate fields of the superclass and all subclasses

INHERITANCE

```
public interface AnimalRepository extends JpaRepository<Animal, Long> {}
```

```
public interface PonyRepository extends JpaRepository<Pony, Long> {

    Pony findPonyByName(String name);
}
```

- Repositories can be defined on the level of the superclass or a subclass
- The repositories *encapsulate* how these objects are stored in the database

Inheritance

OneToMany

ManyToMany

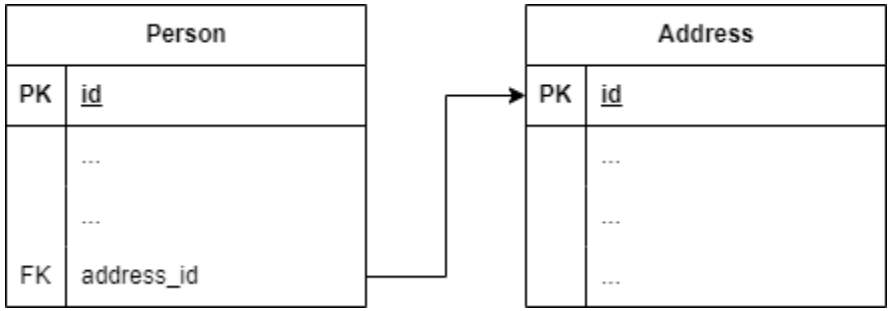
OneToMany

RELATIONSHIPS

- Mapping relationships for JPA is not easy
  - Relationships are handled differently in the database than in code
- Java
  - Relationships between classes are expressed through fields
  - A relationship can be unidirectional or bidirectional
- Relational Database
  - Relationships between tables are expressed through foreign keys
  - A relationship is always unidirectional

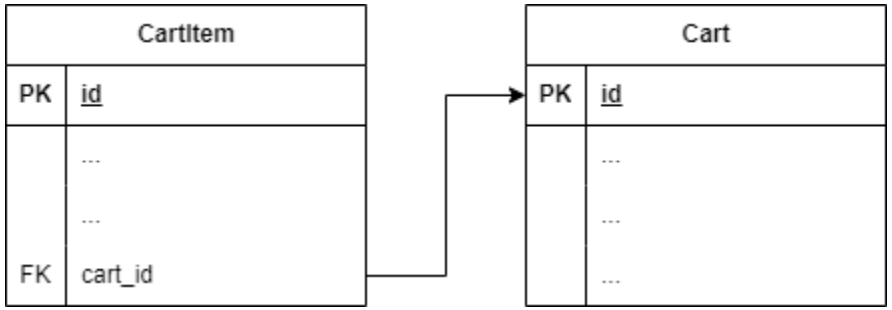
RELATIONSHIPS — DATABASE — ONETOONE

- One row from a table can be linked to one row from a different table
- The table holding the foreign key is called the “owner” of the relationship



RELATIONSHIPS — DATABASE — ONETOMANY

- One row from a table can be linked to multiple rows from a different table
- The table holding the foreign key is called the “owner” of the relationship
  - This is always the “many” side of a OneToMany relationship



RELATIONSHIPS — JAVA & DATABASE

- The complexity of relationships lies in the difference between the database and code
  - In a database a relationship is mostly *unidirectional*
    - Only one table holds the foreign key
    - ManyToMany is sort of an exception
  - In code a relationship can be *unidirectional* or *bidirectional*
    - Both objects can hold references to each other

- It is the job of JPA to translate from one "world" to the other
  - To do so, we need to configure the relationships correctly

UNIDIRECTIONAL ONETOONE — CONFIG

```java
@Entity
@Table(name = "person")
public class Person {

    @OneToOne
    @JoinColumn(name = "address_id")
    private Address address;
}
```

```java
@Entity
@Table(name = "address")
public class Address {

    // ...
}
```

- The field that maps to the column with the foreign key is marked with @JoinColumn
- The entity that holds the foreign key is called the “owning-side”

BIDIRECTIONAL ONETOONE — CONFIG

```java
@Entity
@Table(name = "person")
public class Person {

    @OneToOne
    @JoinColumn(name = "address_id")
    private Address address;
}
```

```java
@Entity
@Table(name = "address")
public class Address {

    @OneToOne(mappedBy = ”address")
    private Person person;
}
```

- The field that maps to the column with the foreign key is marked with @JoinColumn
- The entity that holds the foreign key is called the “owning-side”
- The “non-owning side” refers to the “owning side” via “mappedBy”

BIDIRECTIONAL ONETOMANY/MANYTOONE — CONFIG

```java
@Entity
@Table(name = "cart")
public class Cart {

    @OneToMany(mappedBy = "cart")
    private List<CartItem> items;
}
```

```java
@Entity
@Table(name = "cart_item")
public class CartItem {

    @ManyToOne
    @JoinColumn(name = "cart_id")
    private Cart cart;
}
```

- The reverse side of a OneToMany relationship is called a ManyToOne relationship
- The field that maps to the column with the foreign key is marked with @JoinColumn
- The entity that holds the foreign key is called the “owning-side”
- The “non-owning side” refers to the “owning side” via “mappedBy”

BIDIRECTIONAL ONETOMANY/MANYTOONE — JSON

```java
public class Cart {

    private String identifier;
    private List<CartItem> items;
}
```

```java
public class CartItem {

    private String name;
    private Cart cart;
}
```

```json
{
   "identifier": "cart-customer-0x1245",
   "items": [
      {
         "name": "cart-customer-0x1245",
         "cart": {
            "identifier": "Shampoo",
            "items": ...
         }
      }
   ]
}
```

- If bidirectionally connected objects are translated to JSON Jackson gets stuck in a loop

BIDIRECTIONAL ONETOMANY/MANYTOONE — JSON

```java
public class Cart {

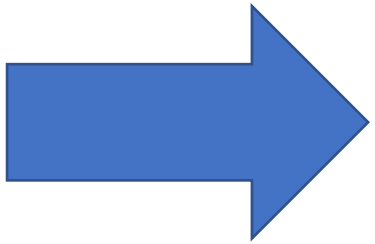
    private String identifier;

    @JsonManagedReference
    private Set<CartItem> items;
}
```

```java
public class CartItem {

    private String name;

    @JsonBackReference
    private Cart cart;
}
```



```json
{
    "identifier": "cart-customer-0x1245",
    "items": [
        {
            "name": "Shampoo"
        }
    ]
}
```

- This can be avoided by the use of @JsonManagedReference and @JsonBackReference
  - The side of the relationship annotated with @JsonManagedReference will be translated
  - The side of the relationship annotated with @JsonBackReference will be ignored

UNIDIRECTIONAL MANYTOONE — CONFIG

```java
@Entity
@Table(name = "cart")
public class Cart {

    // ...
}
```

```java
@Entity
@Table(name = "cart_item")
public class CartItem {

    @ManyToOne
    @JoinColumn(name = "cart_id")
    private Cart cart;
}
```

- The field that maps to the column with the foreign key is marked with @JoinColumn
- The entity that holds the foreign key is called the “owning-side”

UNIDIRECTIONAL ONETOMANY — CONFIG

```java
@Entity
@Table(name = "cart")
public class Cart {

    @OneToMany
    @JoinColumn(name = "cart_id",
       updateable = false, insertable = false)
    private List<CartItem> items;
}
```

```java
@Entity
@Table(name = "cart_item")
public class CartItem {

    // ...
}
```

- "Non-owning side" is annotated with @JoinColumn
- Avoid! Goes against the directionality of the database, very inefficient!

RELATIONSHIPS — HOW TO CODE

Code the correct configuration for the intended relationship in the entities.

- POST Requests, in the service:
    1. Get or create the necessary domain objects
        1. Via the requestbody or via the repositories
    2. Save the non-owning side via the repository (store result in variable!)
    3. Connect the participating domain objects
    4. Save the owning side via the repository

- DELETE requests, in the service:
    1. Get the necessary domain objects via the repositories
    2. Break the relationship between the participating domain objects
    3. Delete the requested entities via the repositories

- PUT requests, in the service:
    1. Make sure that relationships are taken into account when validating the new information

Inheritance

OneToMany

ManyToMany

ManyToMany

RELATIONSHIPS — DATABASE — MANYTOMANY

- One row from a table can be linked to multiple rows from a different table and vice versa
- No “owner” of the relationship, both sides are equal
- A join-table is necessary



BIDIRECTIONAL MANYTOMANY — CONFIG

```java
@Entity
@Table(name = "dog")
public class Dog {

    @ManyToMany
    @JoinTable(
        name = "dogtoy",
        joinColumns = @JoinColumn(name = "dog_id"),
        inverseJoinColumns = @JoinColumn(name = "toy_id")
    )
    private Set<Toy> favoriteToys;
}
```

```java
@Entity
@Table(name = "toy")
public class Toy {

    @ManyToMany(mappedBy = "favoriteToys")
    private Set<Dog> favoritedBy;
}
```

- Entity that has the @JoinTable annotation is called the “owning-side”
- Which entity is the “owning-side” is arbitrary
- @JoinTable annotation maps to the join-table in the database

BIDIRECTIONAL MANYTOMANY — JSON

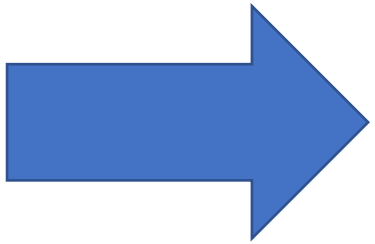
```
public class Dog {

    private String name;
    private Set<Toy> favoriteToys;
}

public class Toy {

    private String name;

    @JsonIgnore
    private Set<Dog> favoritedBy;
}
```



```
{
   "name":  "Max",
   "favoriteToys": [
      {
         "name":  "Ball"
      }
   ]
}
```

- @JsonIgnore can be used to completely exclude a property when translating to JSON
- Also usable for bidirectional @OneToOne relationships!
- More information on translating bidirectional relationships to JSON can be found here

UNIDIRECTIONAL MANYTOMANY — CONFIG

```java
@Entity
@Table(name = "dog")
public class Dog {

    @ManyToMany
    @JoinTable(
        name = "dogtoy",
        joinColumns = @JoinColumn(name = "dog_id"),
        inverseJoinColumns = @JoinColumn(name = "toy_id")
    )
    private Set<Toy> favoriteToys;
}
```

```java
@Entity
@Table(name = "toy")
public class Toy {

    // ...
}
```

- Entity that has the @JoinTable annotation is called the "owning-side"
- @JoinTable annotation maps to the join-table in the database
- Can be defined in the other direction as well

MANYTOMANY — DATABASE

```
-- ...

CREATE TABLE DOGTOY(
    DOG_ID BIGINT,
    TOY_ID BIGINT,
    PRIMARY KEY(DOG_ID, TOY_ID),
    FOREIGN KEY(DOG_ID) REFERENCES DOG
    FOREIGN KEY(TOY_ID) REFERENCES TOY
);

-- ...
```

- The database should also contain the join-table

RELATIONSHIPS — HOW TO CODE

Code the correct configuration for the intended relationship in the entities.

- POST Requests, in the service:
    1. Get or create the necessary domain objects
        1. Via the requestbody or via the repositories
    2. Save the non-owning side via the repository (store result in variable!)
    3. Connect the participating domain objects
    4. Save the owning side via the repository

- DELETE requests, in the service:
    1. Get the necessary domain objects via the repositories
    2. Break the relationship between the participating domain objects
    3. Delete the requested entities via the repositories

- PUT requests, in the service:
    1. Make sure that relationships are taken into account when validating the new information

EXERCISES @CLASS

- exercise1
- exercise2
- exercise3
- exercise4



EXERCISES @HOME

- exercise5



