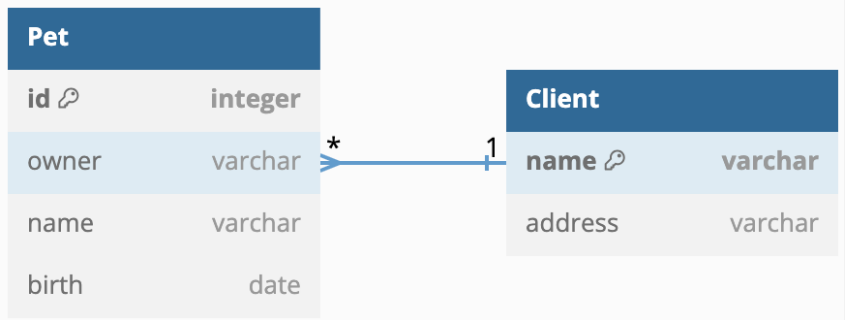
JSON / JPA

CIRCULAR REFERENCES

- **How to serialize bidirectional references ?**

```
{ "id" : 1473,
  "owner" : {
    "name" : "Jacques Chirac",
    "address" : "Champs Elysée",
    "pets" : [ {
      "id" : 1473,
      "owner" : {
        "name" : "Jacques Chirac",
        "address" : "Champs Elysée",
        "pets" : [ {
         "id": 1473,
         "owner" : {
```



SOLUTION 1 : UNIDIRECTIONAL REFERENCE

```java
@Entity
public class Client {
  @OneToMany(mappedBy = "owner")
  Set<Pet> pets;
  // ...

@Entity
private class Pet {
  @ManyToOne
  private Client owner;
  // ...
```

```
{
    "name" : "Jacques Chirac",
    "address" : "Champs Elysée”
}

{
  "id" : 1473,
  "owner" : {
        "name" : "Jacques Chirac",
        "address" : "Champs Elysée”
  }
  "name" : ”Medor",
  ”birth" : ”01/05/1980",
}
```

SOLUTION 2 : JSON MANAGED/BACK REFERENCES

```java
@Entity
public class Client {
  @OneToMany(mappedBy = "owner")
  @JsonManagedReference
  Set<Pet> pets;

@Entity
private class Pet {
  @ManyToOne
  @JsonBackReference
  private Client owner;
```

```
{
    "name" : "Jacques Chirac",
    "address" : "Champs Elysée”
    "pets” : [{
      "id" : 1473,
      "name" : ”Medor",
      "birth" : ”01/05/1980",
    }]
}
{
  "id" : 1473,
  "name" : ”Medor",
  "birth" : ”01/05/1980",
}
```

SOLUTION 3 : JSON IDENTITIES

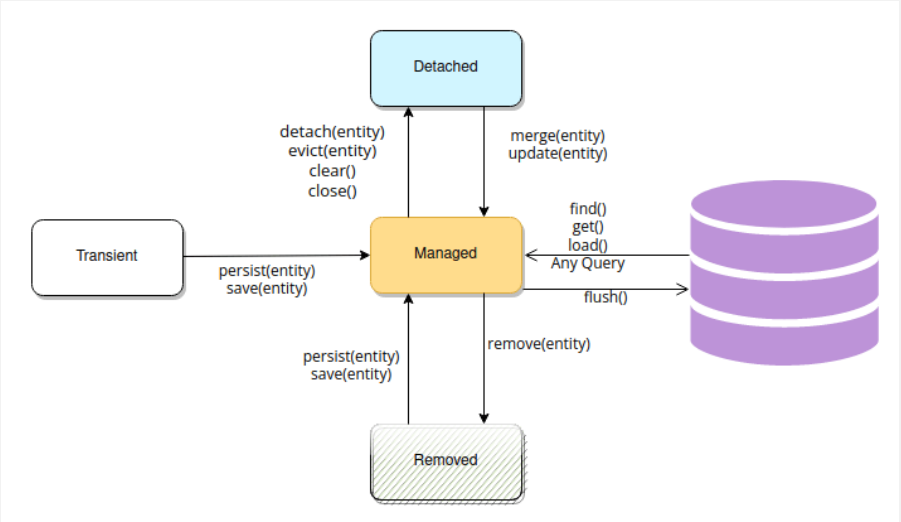
```java
@Entity
public class Client {
  @OneToMany(mappedBy = "owner")
  @JsonIdentityReference(alwaysAsId = true)
  Set<Pet> pets;

@JsonIdentityInfo(generator =
ObjectIdGenerators.PropertyGenerator.class,
property = "name")
@Entity
private class Pet {
  @ManyToOne
  private Client owner;
```

```
{
    "name" : "Jacques Chirac",
    "address" : "Champs Elysée”
    "pets” : [ 1473 ]
}
{
  "id" : 1473,
  "name" : ”Medor",
  "birth" : ”01/05/1980",
  "owner” : {
    "name" : "Jacques Chirac",
    "address" : "Champs Elysée”
    "pets” : [ 1473 ]
  }
}
```

CASCADING

CASCADE TYPES

- *Not only remove (as in DB), but all entity state changes*
- *Operations that will be cascaded (apply to relation ends) :*
  - *ALL*
  - *PERSIST*
  - *MERGE (updates)*
  - *REMOVE*
  - *REFRESH*
  - *DETACH*



CASCADE PERSITANCE ?

```
Client JC = new Client("Jacques Chirac", "Champs Elysées");
Pet medor = new Pet("Medor", "01/05/1980");
JC.pets.add(medor);
repo.save(JC);
```

Error: org.hibernate.TransientPropertyValueException: Not-null property references a transient value

```
@Entity
public class Client {
  @OneToMany(mappedBy = "owner", cascade = CascadeType.PERSIST)
  Set<Pet> pets;
```

SEVERAL CASCADE TYPES

```
public class Client {
  @OneToMany(mappedBy = "owner",
             cascade = { CascadeType.DETACH, CascadeType.MERGE })
  Set<Pet> pets;
```

TRANSACTIONS

TRANSACTIONS

- Same principles than DB transactions
- Essentials for maintaining data integrity and consistency
  - Especially in large scale / distributed application
  - Even in the context of REST services
- Two levels
  - “low” : BD transaction mapping : EntityManager
  - “high” : Jakarta transaction API (for any kind of operation)
- In case of exception/error : rollback !

TRANSACTIONS : USING ENTITY MANAGER

```
public void changeOwner(String oldClientId, String newClientId, int petId) {
  entityManager.getTransaction().begin();
  Client o = entityManager.find(Client.class, oldClientId),
  n = entityManager.find(Client.class, oldClientId);
  Pet pet = entityManager.find(Pet.class, petId);
  o.pets.remove(pet);
  n.pets.add(pet);
  entityManager.merge(pet);                    // sufficient even without cascade
  entityManager.getTransaction().commit();
}
```

JAKARTA TRANSACTION API

- Annotation `@Transactional`
  - On methods
  - On class : all methods are transactional
- Encapsulate method calls
- In Spring
  - Associate isolation level : *DEFAULT, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE*
  - Associate a propagation level (propagation between transactional methods)

JAKARTA TRANSACTION API EXAMPLE

```
@Transactional
public void changeOwner(String oldClientId, String newClientId, int petId) {
  Client o = clientRepositiory.findByName(oldClientId),
    n = clientRepositiory.findByName(newClientId);
  Pet pet = petRepository.findById(petId);
  o.pets.remove(pet);
  n.pets.add(pet);
  petRepository.save(pet); // sufficient even without cascade
  emailSender.send(...);
}
```

JPA QUERY

SPRING QUERY METHODS

- Methods automatically defined
- Non JPA/hibernate-standard

```
public interface ClientRepository extends CrudRepository<Client, String> {
  Client findByName(String name);
  Client findByAddress(String addr);
}
```

SPRING QUERY METHODS EXAMPLES

- findDistinctByLastnameAndFirstname(last, first)
- findByLastnameAndFirstname(last, first)
- findByLastnameOrFirstname(last, first)
- streamByAgeLessThan(max)
- countByStartDateBetween(start,end)
- findByFirstnameEndingWith(ending)
- deleteByLastnameNot(name)
- existsByAgeIn(set)
- …
- https://docs.spring.io/spring-data/jpa/reference/repositories/query-keywords-reference.html

JPQL

- Almost similar than SQL
  - Some limitations
  - Powerfull implicit joins
- entityManager. createQuery()
- Annotation @Query

```
public interface ClientRepository extends CrudRepository<Client, String> {
  @Query("select c from Client c join Pet p where p.name = 'Medor'")
  Client findBoomers();
```

QUERY PARAMETERS

```
@Query("select c from Client c join Pet p where p.name = ?1")
List<Client> findByPetName(String petName);

@Query("select avg(size(c.pets)) from Client c " +
   "where p.address.city = ?1 and p.age > ?2")
double avgPetNumByClientFromAndOlderThan(String city, int age);
```

DATA SOURCES

TO BE DEFINED

- Using persistence.xml
- Or Spring data sources, e.g.

```
@Bean
public DataSource dataSource() {
  DriverManagerDataSource dataSource = new DriverManagerDataSource();

  dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
  dataSource.setUsername("mysqluser");
  dataSource.setPassword("mysqlpass");
  dataSource.setUrl("jdbc:mysql://localhost:3306/myDb?createDatabaseIfNotExist=true");
  return dataSource;
}
```