CS544
Enterprise Application Architecture

Lesson 6 – JPA Queries

Frameworks and Best Practices Used in Designing Large-Scale Software Systems

Payman Salek, M.S.

Original Material: Prof. Rene de Jong – July 2022

Query techniques

- Query creation from method names
- Using @Query
- Using named queries
- Using native queries

METHOD BASED QUERY

Query creation from method names

```
public interface UserRepository extends JpaRepository<User, Long> {
  User findByEmailAddress(String emailAddress);
  List<User> findByLastname(String lastname);
  List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);

}
```

Query methods rules

- The name of our query method must start with one of the following prefixes:
  - find…By, read…By, query…By, count…By, and get…By.
- If we want to specify the selected property, we must add the name of the property before the first By word.
  - findTitleBy
- If we want to limit the number of returned query results, we can add the First or the Top keyword before the first By word.
  - If we want to get more than one result, we have to append the optional numeric value to the First and the Top keywords.
  - findTopBy, findTop1By, findFirstBy, findFirst2By
- If we want to select unique results, we have to add the Distinct keyword before the first By word.
  - findTitleDistinctBy, findDistinctTitleBy
- We must add the search criteria of our query method after the first By word.
  - findByEmailAddressAndLastname
- If our query method specifies x search conditions, we must add x method parameters to it.
  - The number of method parameters must be equal than the number of search conditions.
  - The method parameters must be given in the same order than the search conditions.

Query method examples

```
Dog findById(Long id);
Dog readById(Long id);
Dog getById(Long id);
Dog queryById(Long id);
```

These are all the same

```
Integer countByName(String name);
```

```
List<Dog> findByAgeAndHeight(Integer age, double height);
List<Dog> findByAgeAndNameAndColor(Integer age, String name, String color);
List<Dog> findByNameOrAge(String name, Integer age);
List<Dog> findByNameIgnoreCaseAndColor(String name, String color);
```

```
Dog findFirstByName(String name);
Dog findTopByName(String name);

List<Dog> findTop10ByColor(String color);
```

Query method examples

```
Dog findFirstByName(String name);
Dog findTopByName(String name);

List<Dog> findTop10ByColor(String color);
```

These 2 are the same

```
List<Dog> findByNameContaining(String subName);
List<Dog> findByNameStartingWith(String subName);
```

```
List<Dog> findByHeightLessThan(double height);
List<Dog> findByAgeLessThanOrHeightGreaterThan(Integer age, double height);
List<Dog> findByAgeGreaterThanAndAgeLessThan(Integer ageStart, Integer ageEnd);
List<Dog> findByAgeGreaterThanEqual(Integer age);
List<Dog> findByDateOfBirthBetween(Date start, Date end);
```

Supported keywords

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull	findByAgeIsNull	… where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false

Query method examples

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    List<Customer> findByLastName(String lastName);
    Optional<Customer> findByEmail(String email);
    Customer findByFirstNameAndLastName(String fistName, String lastName);
    List<Customer> findFirst2By();
}
```

```
Optional<Customer> custopt = custemerrepository.findByEmail("dpalmer@gmail.com");
if (custopt.isPresent()) {
  Customer thecustomer = custopt.get();
  System.out.println(thecustomer);
}

Customer cust = custemerrepository.findByFirstNameAndLastName("Chloe", "O'Brian");
System.out.println(cust);

for (Customer cust2 : custemerrepository.findFirst2By()) {
  System.out.println(cust2);
}
```

Main point

- One can create queries in the repository by defining methods according to a certain convention in the repository interface.

  Science of Consciousness: Through the daily practice of transcending one's thoughts get more powerful witch leads to more fulfillment. Thoughts leads to Action, Action leads to Achievement, Achievement leads to Fulfilment

@QUERY

Using @Query

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    List<Customer> findByLastName(String lastName);

    @Query("select c from Customer c where c.email = ?1")
    Customer findByEmail(String email);
}
```

The method name does not have any significance

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    List<Customer> findByLastName(String lastName);

    @Query("select c from Customer c where c.email = :email ")
    Customer findByEmail((@Param("email") String email);
}
```

Named parameter

JPQL examples

```
select b from Book b where b.price > 15
```

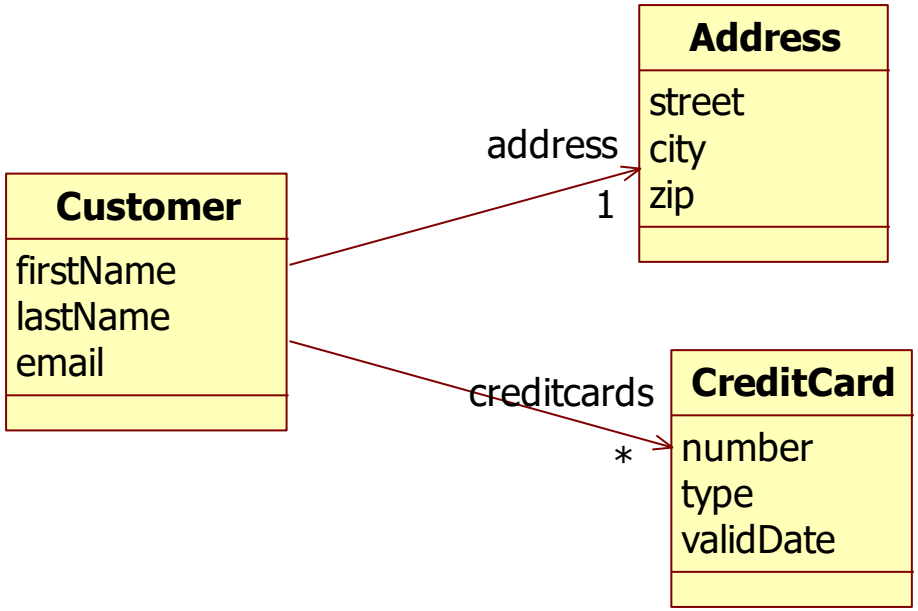
```
select b.title from Book b
```

```
select b from Book b where b.price between 10 and 15
```

```
select b from Book b where b.title like '%love%'
```

```
select b from Book b order by b.price asc
```

JPQL examples




```
select c from Customer c where c.address.city = 'Boston'
```

```
select c from Customer c JOIN c.creditcards cr where cr.number= '127865439867'
```

@Query: finding a property

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

  @Query("select c.lastName from Customer c where c.firstName= :firstName")
  String findLastNameByFirstName(@Param("firstName") String firstName);

}
```

NAMED QUERY

Named query

Named query

```
@Entity
@NamedQuery(name="Employee.findByFirstName", query="select e from Employee e where
e.firstname = :name")
public class Employee {

   @Id
   @GeneratedValue
   private int id;
   private String firstname;
   private String lastname;
   …

}
```

Must be a unique name

Multiple named queries

```
@Entity
@NamedQueries({
   @NamedQuery(name="Employee.findByFirstName", query=" select e from Employee e where
e.firstname = :name"),
   @NamedQuery(name="Employee.findByLastName", query=" select e from Employee e where
e.lastname = :name")
})
public class Employee {

   @Id
   @GeneratedValue
   private int id;
   private String firstname;
   private String lastname;
   …
}
```

Using named queries

```
@Entity
@NamedQuery(name = "Customer.findByEmail", query = "select c from Customer c
where c.email = ?1")
public class Customer {
...
```

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {


    Customer findByEmail(String email);
}
```

Named queries with named parameters

```
@Entity
@NamedQuery(name = "Customer.findByEmail", query = "select c from Customer c
where c.email = :email")
public class Customer {
```

Named parameter

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {


    Customer findByEmail(@Param("email") String email);
}
```

@Param

NATIVE QUERY

Using native queries

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    List<Customer> findByLastName(String lastName);

    @Query(value = "SELECT * FROM customer WHERE EMAIL = ?1", nativeQuery = true)
    Customer findByEmail(String email);
}
```

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    List<Customer> findByLastName(String lastName);

    @Query(value = "SELECT * FROM customer WHERE EMAIL = :email", nativeQuery = true)
    Customer findByEmail(@Param("email") String email);
}
```

Named parameter

Modifying statements

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    List<Customer> findByLastName(String lastName);

    @Modifying
    @Transactional
    @Query("update Customer cust set cust.firstName = :firstname where
            cust.lastName = :lastname")
    int setFixedFirstnameFor(@Param("firstname") String firstName,
                             @Param("lastname") String lastName);
}
```

@Modifying for update, delete statements

Modifying method must be transactional

Bulk update and delete

```
update Customer c set c.status = 'Gold' where c.orders > :numberoforders
```

```
delete Customer c where c.status = :status
```

OPTIMIZATION

Entities

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private long id;
  private String firstname;
   private String lastname;

   @ManyToOne(cascade={CascadeType.PERSIST})
   private Address address;
```

```
@Entity
public class Address {
  @Id
  @GeneratedValue
  private long id;
  private String street;
   private String city;
   private String zip;
```

ManyToOne

```
public void run(String... args) throws Exception {
  Address a1 = new Address("mainstreet 1", "Chicago", "58902");
  Customer c1 = new Customer("Frank", "Brown");
  c1.setAddress(a1);
  customerRepository.save(c1);

  Address a2 = new Address("mainstreet 4", "New York", "21345");
  Customer c2 = new Customer("Frank", "Johnson");
  c2.setAddress(a2);
  customerRepository.save(c2);

  List<Customer> customerList = customerRepository.findByFirstname("Frank");
  customerList.stream().forEach(c -> System.out.println(c));
}
```

```
List<Customer> findByFirstname(String name);
```

ManyToOne

```
Hibernate: select customer0_.id as id1_2_, customer0_.address_id as address_4_2_,
customer0_.firstname as firstnam2_2_, customer0_.lastname as lastname3_2_ from
customer customer0_ where customer0_.firstname=?
Hibernate: select address0_.id as id1_0_0_, address0_.city as city2_0_0_,
address0_.street as street3_0_0_, address0_.zip as zip4_0_0_ from address addre
where address0_.id=?
Hibernate: select address0_.id as id1_0_0_, address0_.city as city2_0_0_,
address0_.street as street3_0_0_, address0_.zip as zip4_0_0_ from address addre
where address0_.id=?
Customer{id=1, firstname='Frank', lastname='Brown', address=Address{id=2,
street='mainstreet 1', city='Chicago', zip='58902'}}
Customer{id=3, firstname='Frank', lastname='Johnson', address=Address{id=4,
street='mainstreet 4', city='New York', zip='21345'}}
```

Load all customers

Load address for customer1

Load address for customer2

addresses are loaded eagerly

ManyToOne

- Always make ManyToOne relations lazy.

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private long id;
  private String firstname;
   private String lastname;

   @OneToMany (cascade={CascadeType.PERSIST})
   private Collection<CreditCard> creditcard=new ArrayList<CreditCard>();

   @ManyToOne(cascade={CascadeType.PERSIST}, fetch=FetchType.LAZY)
   private Address address;
```

Lazy

ManyToOne

- Always make ManyToOne relations lazy.

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private long id;
  private String firstname;
   private String lastname;

  @ManyToOne(cascade={CascadeType.PERSIST}, fetch=FetchType.LAZY)
   private Address address;
```

```
@Query("select c from Customer c")
List<Customer> findByFirstnameLazy(String name);
```

```
Hibernate: select customer0_.id as id1_2_, customer0_.address_id as address_4_2_,
customer0_.firstname as firstnam2_2_, customer0_.lastname as lastname3_2_ from
customer customer0_
Customer{id=1, firstname='Frank', lastname='Brown'}
Customer{id=3, firstname='Frank', lastname='Johnson'}
```

ManyToOne with join fetch

- Always make ManyToOne relations lazy and use join fetch to load the related object.

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private long id;
  private String firstname;
   private String  lastname;

  @ManyToOne(cascade={CascadeType.PERSIST}, fetch=FetchType.LAZY)
   private Address address;
```

```
@Query("select c from Customer c join fetch c.address")
List<Customer> findByFirstnameEager(String name);
```

1 query to get all customers and their addresses

```
Hibernate: select customer0_.id as id1_2_0_, address1_.id as id1_0_1_,
customer0_.address_id as address_4_2_0_, customer0_.firstname as firstnam2_2_0_,
customer0_.lastname as lastname3_2_0_, address1_.city as city2_0_1_, address1_.street
as street3_0_1_, address1_.zip as zip4_0_1_ from customer customer0_ inner join
address address1_ on customer0_.address_id=address1_.id
Customer{id=1, firstname='Frank', lastname='Brown'}
Customer{id=3, firstname='Frank', lastname='Johnson'}
```

OneToMany

```
@Override
  public void run(String... args) throws Exception {
    CreditCard creditCard1 = new CreditCard("123", "Frank Brown", new Date());
    CreditCard creditCard2 = new CreditCard("345", "Frank Brown", new Date());
    Customer c1 = new Customer("Frank", "Brown");
    c1.getCreditcard().add(creditCard1);
    c1.getCreditcard().add(creditCard2);
    customerRepository.save(c1);

    CreditCard creditCard11 = new CreditCard("123", "Frank Johnson", new Date());
    CreditCard creditCard22 = new CreditCard("345", "Frank Johnson", new Date());
    Customer c2 = new Customer("Frank", "Johnson");
    c2.getCreditcard().add(creditCard11);
    c2.getCreditcard().add(creditCard22);
    customerRepository.save(c2);

    List<Customer> customerList2 = customerRepository.findByFirstnameLazy("Frank");
    customerList2.stream().forEach(c -> System.out.println(c));
  }
```

```
@Query("select c from Customer c")
List<Customer> findByFirstnameLazy(String name);
```

OneToMany

```
Hibernate: select customer0_.id as id1_2_, customer0_.firstname as firstnam2_2_,
customer0_.lastname as lastname3_2_ from customer customer0_
Hibernate: select creditcard0_.customer_id as customer1_3_0_,
creditcard0_.creditcards_id as creditca2_3_0_, creditcard1_.id as id1_1_1_,
creditcard1_.expiration as expirati2_1_1_, creditcard1_.name as name3_1_1_,
creditcard1_.number as number4_1_1_ from customer_creditcards creditcard0_ inner
credit_card creditcard1_ on creditcard0_.creditcards_id=creditcard1_.id where
creditcard0_.customer_id=?
Hibernate: select creditcard0_.customer_id as customer1_3_0_,
creditcard0_.creditcards_id as creditca2_3_0_, creditcard1_.id as id1_1_1_,
creditcard1_.expiration as expirati2_1_1_, creditcard1_.name as name3_1_1_,
creditcard1_.number as number4_1_1_ from customer_creditcards creditcard0_ inner join
credit_card creditcard1_ on creditcard0_.creditcards_id=creditcard1_.id where
creditcard0_.customer_id=?
Customer{id=1, firstname='Frank', lastname='Brown', creditcards=[CreditCard{id=2,
number='123', name='Frank Brown', expiration=2022-04-01 21:42:57.818},
CreditCard{id=3, number='345', name='Frank Brown', expiration=2022-04-01
21:42:57.818}]}
Customer{id=4, firstname='Frank', lastname='Johnson', creditcards=[CreditCard{id=5,
number='123', name='Frank Johnson', expiration=2022-04-01 21:42:57.947},
CreditCard{id=6, number='345', name='Frank Johnson', expiration=2022-04-01
21:42:57.947}]}
```

Load all customers

Load all creditcards for customer1

Load all creditcards for customer2

OneToMany with join fetch

```
@Override
  public void run(String... args) throws Exception {
    CreditCard creditCard1 = new CreditCard("123", "Frank Brown", new Date());
    CreditCard creditCard2 = new CreditCard("345", "Frank Brown", new Date());
    Customer c1 = new Customer("Frank", "Brown");
    c1.getCreditcard().add(creditCard1);
    c1.getCreditcard().add(creditCard2);
    customerRepository.save(c1);

    CreditCard creditCard11 = new CreditCard("123", "Frank Johnson", new Date());
    CreditCard creditCard22 = new CreditCard("345", "Frank Johnson", new Date());
    Customer c2 = new Customer("Frank", "Johnson");
    c2.getCreditcard().add(creditCard11);
    c2.getCreditcard().add(creditCard22);
    customerRepository.save(c2);

    List<Customer> customerList2 = customerRepository.findByFirstnameEager("Frank");
    customerList2.stream().forEach(c -> System.out.println(c));
  }
```

```
@Query("select c from Customer c join fetch c.creditcards")
List<Customer> findByFirstnameEager(String name);
```

OneToMany with join fetch

```
Hibernate: select customer0_.id as id1_2_0_, creditcard2_.id as id1_1_1_,
customer0_.firstname as firstnam2_2_0_, customer0_.lastname as lastname3_2_0
creditcard2_.expiration as expirati2_1_1_, creditcard2_.name as name3_1_1_,
creditcard2_.number as number4_1_1_, creditcard1_.customer_id as customer1_3_0__,
creditcard1_.creditcards_id as creditca2_3_0__ from customer customer0_ inner join
customer_creditcards creditcard1_ on customer0_.id=creditcard1_.customer_id inner join
credit_card creditcard2_ on creditcard1_.creditcards_id=creditcard2_.id
Customer{id=1, firstname='Frank', lastname='Brown', creditcards=[CreditCard{id=2,
number='123', name='Frank Brown', expiration=2022-04-01 21:48:35.213},
CreditCard{id=3, number='345', name='Frank Brown', expiration=2022-04-01
21:48:35.213}]}
Customer{id=1, firstname='Frank', lastname='Brown', creditcards=[CreditCard{id=2,
number='123', name='Frank Brown', expiration=2022-04-01 21:48:35.213},
CreditCard{id=3, number='345', name='Frank Brown', expiration=2022-04-01
21:48:35.213}]}
Customer{id=4, firstname='Frank', lastname='Johnson', creditcards=[CreditCard{id=5,
number='123', name='Frank Johnson', expiration=2022-04-01 21:48:35.337},
CreditCard{id=6, number='345', name='Frank Johnson', expiration=2022-04-01
21:48:35.337}]}
Customer{id=4, firstname='Frank', lastname='Johnson', creditcards=[CreditCard{id=5,
number='123', name='Frank Johnson', expiration=2022-04-01 21:48:35.337},
CreditCard{id=6, number='345', name='Frank Johnson', expiration=2022-04-01
21:48:35.337}]}
```

Load all customers and all creditcards in 1 query

Cartesian product

OneToMany with distinct join fetch

```
@Override
  public void run(String... args) throws Exception {
    CreditCard creditCard1 = new CreditCard("123", "Frank Brown", new Date());
    CreditCard creditCard2 = new CreditCard("345", "Frank Brown", new Date());
    Customer c1 = new Customer("Frank", "Brown");
    c1.getCreditcard().add(creditCard1);
    c1.getCreditcard().add(creditCard2);
    customerRepository.save(c1);

    CreditCard creditCard11 = new CreditCard("123", "Frank Johnson", new Date());
    CreditCard creditCard22 = new CreditCard("345", "Frank Johnson", new Date());
    Customer c2 = new Customer("Frank", "Johnson");
    c2.getCreditcard().add(creditCard11);
    c2.getCreditcard().add(creditCard22);
    customerRepository.save(c2);

    List<Customer> customerList2 = customerRepository.findByFirstnameEager("Frank");
    customerList2.stream().forEach(c -> System.out.println(c));
  }
```

```
@Query("select distinct c from Customer c join fetch c.creditcards")
List<Customer> findByFirstnameEager(String name);
```

OneToMany with distinct join fetch

```
Hibernate: select distinct customer0_.id as id1_2_0_, creditcard2_.id as id1_1_1_,
customer0_.firstname as firstnam2_2_0_, customer0_.lastname as lastname3_2_0_,
creditcard2_.expiration as expirati2_1_1_, creditcard2_.name as name3_1_1_,
creditcard2_.number as number4_1_1_, creditcard1_.customer_id as customer1_3_0__,
creditcard1_.creditcards_id as creditca2_3_0__ from customer customer0_ inner join
customer_creditcards creditcard1_ on customer0_.id=creditcard1_.customer_id inner join
credit_card creditcard2_ on creditcard1_.creditcards_id=creditcard2_.id
Customer{id=1, firstname='Frank', lastname='Brown', creditcards=[CreditCard{id=2,
number='123', name='Frank Brown', expiration=2022-04-01 21:50:43.899},
CreditCard{id=3, number='345', name='Frank Brown', expiration=2022-04-01
21:50:43.899}]}
Customer{id=4, firstname='Frank', lastname='Johnson', creditcards=[CreditCard{id=5,
number='123', name='Frank Johnson', expiration=2022-04-01 21:50:44.015},
CreditCard{id=6, number='345', name='Frank Johnson', expiration=2022-04-01
21:50:44.015}]}
```

Summary

- Always make ManyToOne relations lazy
  - Use join fetch to fetch them eagerly
- OneToMany relations are already lazy
  - Use distinct join fetch to fetch them eagerly
- Always check how often the ORM goes to the database

Main point

- When using JPA it is important to optimize the mapping and queries to get the most optimal database access performance.

  Science of Consciousness: Nature always takes the most optimal path of least action.

Connecting the parts of knowledge with the wholeness of knowledge

1. Spring provides different ways to add queries to an enterprise application.
2. JPA optimization helps to get better performance

3. **Transcendental consciousness** is the field of all possibilities.
4. **Wholeness moving within itself:** In Unity Consciousness, we experience the unity within all diversity in creation.