Mindtree

Welcome to possible

Hibernate query language

Objectives

- How to use JP-QL: The Object Query Language
- How to use Criteria Queries
- How to use Native query

JP-QL: The Object Query Language

- The Java Persistence Query Language (JP-QL) has been heavily inspired by HQL, the native Hibernate Query Language.
- Both JP-QL and HQL a therefore very close to SQL, but portable and independent of the database schema.
- HQL is a strict superset of JP-QL and you use the same query API for both types of queries
- **Case Sensitivity**
  - Queries are case-insensitive, except for names of Java classes and properties. So SeLeCT is the same as sELEct is the same as SELECT but org.hibernate.eg.FOO is not org.hibernate.eg.Foo and foo.barSet is not foo.BARSET.

Java Persistence Query Language (JP-QL)

- Comparison between SQL and JP-QL

TVEHICLE table contents

VEHICLE_ID	REG_NO	DAILY_RENTAL	FUEL_TYPE	MILEAGE	MANUFACTURER	DESCRIPTION	CATEGORY_ID
1	KA-04-Z-1234	1200.99	Diesel	14	Toyota	Innova, 6 Seater	1
2	AP-08-XY-9822	800.88	Diesel	21	Tata	Indica, 4 Seater	1
3	JK-02-AA-120	450	Petrol	16	Maruthi	Omni Van , All in One	1
4	TN-09-EF-3411	3200	Diesel	5	Ashok Leyland	Luxury Bus	3
5	KA-08-AE-672	4000	Diesel	6	Tata	Heavy Duty Truck	2
6	KA-02-GH-788	900	Petrol	15	Hyundai	Hyndai Accent	1

com.mindtree.entity.Vehicle

-vehicleId: Integer
+registrationNumber: String
+name: String
+manufacturer: String
+description: String
+mileagePerLiter: int
+fuelType: String
+dailyRentalAmount: double
+category: Category

Description	SQL	JP-QL
Retrieve all vehicles	select * from TVEHICLE	Select v from Vehicle v
Retrieve all Petrol vehicles	select * from TVEHICLE where FUEL_TYPE = 'Petrol'	Select v from Vehicle v where v.fuelType = ‘Petrol’
Retrieve selective columns (scalar values)	select REG_NO,MILEAGE from TVEHICLE	select registrationNumber, mileagePerLiter from Vehicle
Retrieve vehicles of a category	select * from TVEHILCE where CATEGORY_ID = 1	Select v from Vehicle v where v.category = 1

Java Persistence Query Language (JP-QL)

- JP-QL queries are represented with an instance of **javax.persistence.Query**.
- The Query interface offers methods for parameter binding, result set handling, and for the execution of the actual query.
- Query is obtained using the current EntityManager
- Query is usually executed by invoking a getResultList() method.

SELECT Syntax :

```
SELECT [<result>]
[FROM <candidate-class(es)>]
[WHERE <filter>]
[GROUP BY <grouping>]
[HAVING <having>]
[ORDER BY <ordering>]
```

Java Persistence Query Language (JP-QL)

- Example:

```
EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("JPAService");
EntityManager em = emf.createEntityManager();

javax.persistence.Query query = em
        .createQuery("select c from Customer c");


List<Customer> customers = query.getResultList();

for (Customer customer : customers) {
    System.out.println(customer.getCustomerName() + ","
            + customer.getCity());
}
```

Java Persistence Query Language (JP-QL)

- Named parameters are query parameters that are prefixed with a colon **(:)**. Named parameters in a query are bound to an argument by the following method:
  - javax.persistence.Query.setParameter(String name, Object value)

```
String strQuery = "select c from Customer c "
        + "where c.contactTitle =:title and c.city = :cty" ;
javax.persistence.Query query = em
        .createQuery(strQuery);
query.setParameter("title", "owner");
query.setParameter("cty", "Seattle");

List<Customer> customers = query.getResultList();
```

Java Persistence Query Language (JP-QL)

- **Positional Parameters in Queries.**
  - **You may use positional parameters instead of named parameters in queries.**
  - **Positional parameters are prefixed with a question mark (?) followed the numeric position of the parameter in the query.**
  - **The Query.setParameter(integer position, Object value) method is used to set the parameter values.**

```
String strQuery = "select c from Customer c "
        + "where c.contactTitle =?1 and c.city = ?2" ;
javax.persistence.Query query = em
        .createQuery(strQuery);
query.setParameter(1, "owner");
query.setParameter(2, "Seattle");
```

Java Persistence Query Language (JP-QL)

- Range of Results:

  Query setFirstResult(int startPosition)

  - Set the position of the first result to retrieve.
  - Parameters: startPosition - the start position of the first result, numbered from 0

  Query setMaxResults(int maxResult)

  - Set the maximum number of results to retrieve.

```
EntityManager em = emf.createEntityManager();

String strQuery = "select c from Customer c ";
javax.persistence.Query query = em.createQuery(strQuery);
query.setFirstResult(0);
query.setMaxResults(5);
```

Java Persistence Query Language (JP-QL)

- Named Queries
  - The createNamedQuery method is used to create static queries, or queries that are defined in metadata by using the javax.persistence.NamedQuery annotation.

```
@Entity
@Table(name="customers")
@NamedQueries({
    @NamedQuery(name="ParisCustomers",
        query = "select c from Customer c where c.city = 'Paris'"),
    @NamedQuery(name="FetchSalesManagers",
        query = "select c from Customer c where c.contactTitle = 'Sales Manager'")
})
public class Customer {
```

```
javax.persistence.Query query = em.createNamedQuery("FetchSalesManagers");
List<Customer> customers = query.getResultList();
```

JP-QL joins

- Joins illustrated using Product and Category

```java
@Entity
@Table(name="Categories")
public class Category {
    @Id
    @Column(name = "CATEGORY_ID")
    private int categoryId;

    @Column(name = "CATEGORY_NAME")
    private String categoryName;
```

CATEGORY_ID	CATEGORY_NAME
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood

```java
@Entity
@Table(name="products")
public class Product {
    @Id
    @Column(name = "PRODUCT_ID")
    private int productId;

    @Column(name = "PRODUCT_NAME")
    private String productName;

    @Column(name ="UNIT_PRICE")
    private double unitPrice;

    @ManyToOne
    @JoinColumn(name ="CATEGORY_ID")
    private Category category;
```

PRODUCT_...	PRODUCT_NAME	UNIT_PRICE	CATEGORY_ID
1	Chai	18.0000	1
2	Chang	19.0000	1
3	Aniseed Syrup	10.0000	2
4	Chef Anton's Cajun Seasoning	22.0000	2
5	Chef Anton's Gumbo Mix	21.3500	2
6	Grandma's Boysenberry Spread	25.0000	2
7	Uncle Bob's Organic Dried Pears	30.0000	7
8	Northwoods Cranberry Sauce	40.0000	2
9	Mishi Kobe Niku	97.0000	6
10	Ikura	31.0000	8
11	Queso Cabrales	21.0000	4
12	Queso Manchego La Pastora	38.0000	4
13	Konbu	6.0000	8
14	Tofu	23.2500	7
15	Genen Shouyu	15.5000	2
16	Pavlova	17.4500	3

JP-QL joins

- Inner Join: Inner join retrieves result by combining column values of two tables based upon the join-predicate.

```
String strQuery = "select p,c from Product p inner join p.category c";
javax.persistence.Query query = em.createQuery(strQuery);

List<Object[]> list = query.getResultList();

for(int i= 0 ; i < list.size(); i++) {
    Object[] objects = list.get(i);
    if(objects[0] != null) {
        Product p = (Product)objects[0];
        System.out.print(p.getProductName() + ", ");
    }
    if(objects[1] != null) {
        Category c = (Category)objects[1];
        System.out.println(c.getCategoryName());
    }
}
```

First entity will be Product and second entity will be Category

JP-QL joins

- Right Outer join
  - A **right outer join** (or **right join**) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table will appear in the joined table at least once.

```
String strQuery = "select p,c from Product p right outer join p.category c";
javax.persistence.Query query = em.createQuery(strQuery);
```

JP-QL Sub Queries

- Fetch all products whose UNIT_PRICE is above average UNIT_PRICE.

```
EntityManager em = emf.createEntityManager();
String strQuery = "select p from Product p "
        + " where p.unitPrice >= (select avg(unitPrice) from Product)";
javax.persistence.Query query = em.createQuery(strQuery);

List<Product> list = query.getResultList();
for(Product p : list) {
    System.out.println(p.getProductName() +"," + p.getUnitPrice());
}
```

- Fetch only seafood whose UNIT_PRICE is above average UNIT_PRICE

```
String strQuery = "select p from Product p "
        + " where p.unitPrice >= (select avg(unitPrice) from Product)"
        + " and p.category.categoryName ='Seafood'";
```

Criteria API to Create Queries

- The Criteria API is used to define queries for entities and their persistent state by creating query-defining objects.
- Criteria queries are written using Java programming language APIs, are typesafe, and are portable.
- Criteria API work regardless of the underlying data store
- The Criteria API and JPQL are closely related and are designed to allow similar operations in their queries

Criteria API

- The basic semantics of a Criteria query consists of a
  - SELECT clause,
  - FROM clause,
  - and an optional WHERE clause

Criteria queries set these clauses by using Java programming language objects, so the query can be created in a typesafe manner.

Criteria API

- Creating a Criteria Query
  - The javax.persistence.criteria.CriteriaBuilder interface is used to construct
    - Criteria queries
    - Selections
    - Expressions
    - Predicates
    - Ordering
- To obtain an instance of the CriteriaBuilder interface, call the getCriteriaBuilder method on either an EntityManager or an EntityManagerFactory instance.

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
```

Criteria API

- javax.persistence.criteria.CriteriaQuery
  - CriteriaQuery objects define a particular query that will navigate over one or more entities.
  - Obtain CriteriaQuery instances by calling one of the CriteriaBuilder.createQuery methods.
  - For creating typesafe queries, call the CriteriaBuilder.createQuery method as follows:

```
CriteriaQuery<Product> cq = cb.createQuery(Product.class);
```

Criteria API

- Query Roots
  - For a particular CriteriaQuery object, the root entity of the query, from which all navigation originates, is called the query root.
  - It is similar to the FROM clause in a JPQL query.

- Create the query root by calling the from method on the CriteriaQuery instance.
- The following code sets the query root to the Product entity:
- CriteriaQuery< Product > cq = cb.createQuery(Product.class);
- Root< Product > pet = cq.from(Product.class);

Criteria API

- Code Snippet:

```
CriteriaBuilder builder = emf.getCriteriaBuilder();
CriteriaQuery<Product> criteria = builder.createQuery(Product.class);
Root<Product> productRoot = criteria.from(Product.class);
criteria.select(productRoot);
List<Product> list = em.createQuery(criteria).getResultList();
```

Criteria API

- Restricting Criteria Query Results
  - The results of a query can be restricted on the CriteriaQuery object according to conditions set by calling the CriteriaQuery.where method. Calling the where method is analogous to setting the WHERE clause in a JPQL query.

Criteria API

- Restricting Criteria Query Results
  - **Using the Metamodel API to Model Entity Classes.**
    - The Metamodel API is used to create a metamodel of the managed entities in a particular persistence unit. For each entity class in a particular package, a metamodel class is created with a trailing underscore and with attributes that correspond to the persistent fields or properties of the entity class.

```
@StaticMetamodel(Product.class)
public class Product_ {
    public static volatile SingularAttribute<Product, Integer> productId;
      public static volatile SingularAttribute<Product, String> productName;
      public static volatile SingularAttribute<Product, Double> unitPrice;
      public static volatile SingularAttribute<Product, Category> category;

}
```

Criteria API

- Restricting Criteria Query Results
  - Populate the "where" method of the CriteiraQuery with "Predicate" objects.

```
EntityManager em = emf.createEntityManager();
CriteriaBuilder builder = emf.getCriteriaBuilder();
CriteriaQuery<Product> criteria = builder.createQuery(Product.class);
Root<Product> productRoot = criteria.from(Product.class);
criteria.select(productRoot);
Predicate predicate = builder.ge(productRoot.get(Product_.unitPrice), 100);
criteria.where(predicate);
List<Product> list = em.createQuery(criteria).getResultList();
```

Criteria API

- Conditional Methods in the CriteriaBuilder Interface

Conditional Method	Description
`equal`	Tests whether two expressions are equal
`notEqual`	Tests whether two expressions are not equal
`gt`	Tests whether the first numeric expression is greater than the second numeric expression
`ge`	Tests whether the first numeric expression is greater than or equal to the second numeric expression
`lt`	Tests whether the first numeric expression is less than the second numeric expression
`le`	Tests whether the first numeric expression is less than or equal to the second numeric expression
`between`	Tests whether the first expression is between the second and third expression in value
`like`	Tests whether the expression matches a given pattern

Criteria API

- Examples:

The following code uses the `CriteriaBuilder.gt` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date someDate = new Date(...);
cq.where(cb.gt(pet.get(Pet_.birthday), date));
```

The following code uses the `CriteriaBuilder.between` method:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
Date firstDate = new Date(...);
Date secondDate = new Date(...);
cq.where(cb.between(pet.get(Pet_.birthday), firstDate, secondDate));
```

Criteria API

- Compound Predicate Methods in the CriteriaBuilder Interface
- Examples:

The following code shows the use of compound predicates in queries:

```
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(cb.equal(pet.get(Pet_.name), "Fido")
    .and(cb.equal(pet.get(Pet_.color), "brown")));
```

Native query

- You may also express queries in the native SQL dialect of your database. This is useful if you want to utilize database specific features

```
EntityManager em = emf.createEntityManager();
String strQuery = "select product_id,product_name from products";
javax.persistence.Query query = em.createNativeQuery(strQuery);
List<Object[]> list = query.getResultList();
System.out.println(list);
for(Object[] data : list) {
    System.out.println(data[0] +"," + data[1] );
}
```

Welcome to possible

India | USA | UK | Germany | Sweden | Belgium | France | Switzerland | UAE | Singapore | Australia | Japan | China