



JVA-000 Java Persistence with Hibernate

Module 6
JPQL and Criteria API

Objectives

- Observe Java Persistence Query Language and its facilities
- Observe Query and Criteria APIs provided by JPA
- Learn how to use JPQL and Query API for selecting, updating and removing the entities

Java Persistence Query Language

Java Persistence Query Language (JPQL) is the query language defined by JPA

JPQL is similar to SQL, but operates on:

- Objects (instead of tables)
- Attributes (instead of columns)
- Relationships (instead of references)

JPQL can be used for:

- Reading (SELECT statement)
- Bulk updates (UPDATE statement)
- Deletes (DELETE statement)

JPQL: Reading an Objects

SELECT queries are used to read objects from the database

SELECT can return:

- Single object or data element
- List of objects or data elements
- Array of multiple objects and data

JPQL: Reading an Objects

```
Query query = null;
```

```
/** Query for a List of objects */
```

```
query = em.createQuery("select e from Employee e where e.salary > 1000");  
List<Employee> employees = query.getResultList();
```

```
/** Query for a single object */
```

```
query = em.createQuery("select e from Employee e where e.id = 1");  
Employee employee = (Employee) query.getSingleResult();
```

```
/** Query for a single data element */
```

```
query = em.createQuery("select max(e.salary) from Employee e");  
BigDecimal employeeSalary = (BigDecimal) query.getSingleResult();
```

```
/** Query for a List of data elements */
```

```
query = em.createQuery("select e.firstName from Employee e");  
List<String> employeeFirstName = query.getResultList();
```

```
/** Query for a List of element arrays */
```

```
query = em.createQuery("select e.firstName, e.lastName from Employee e");  
List<Object[]> employeeInfo = query.getResultList();
```

Sample of using select statements for reading the data

JPQL: Reading an Objects

Syntax of SELECT statement:

```
select_statement ::=  
    select_clause  
    from_clause  
    [where_clause]  
    [groupby_clause]  
    [having_clause]  
    [orderby_clause]
```

JPQL: Reading an Objects

SELECT clause can contain:

- Object and attributes expressions
- Functions and aggregation functions
- Constructors

```
-- query for a entity objects --  
SELECT e FROM Employee e  
  
-- query for count of entities --  
SELECT COUNT(e) FROM Employee e  
  
-- query for max value of entity attribute --  
SELECT MAX(e.salary) FROM Employee e  
  
-- query for entity attributes --  
SELECT e.name, e.salary FROM Employee e  
  
-- query for new objects created from entity's attributes --  
SELECT NEW edu.entities.Employee(e.name, e.salary) FROM Employee e
```

JPQL: Reading an Objects

FROM clause - defines what is being queried (entity name)

Can contain:

- **JOIN** – produces inner join
- **LEFT JOIN** – produces outer join
- **ON** – defines additional conditions to join entities (JPA 2.1)

```
SELECT e, a FROM Employee e, MailAddress a WHERE e.address = a.address
```

```
SELECT e FROM Employee e JOIN e.address a WHERE a.city = 'Moscow'
```

```
SELECT e FROM Employee e LEFT JOIN MailAddress a ON e.address = a.address
```


JPQL: Reading an Objects

WHERE clause - defines conditions to filter what to return

```
SELECT e FROM Employee e WHERE e.salary > 1000
```

```
SELECT e FROM Employee e WHERE e.salary BETWEEN 500 AND 1000
```

```
SELECT e FROM Employee e WHERE e.name IS NULL
```

```
SELECT e FROM Employee e WHERE e.name LIKE 'A%' AND e.salary < 100
```

```
SELECT e FROM Employee e WHERE e.name IN ('John', 'Fred')
```

JPQL also supports sub-queries in **WHERE** clause

```
SELECT e FROM Employee e  
WHERE e.address IN (select a.address FROM MailAddress a WHERE a.id = 10)
```

JPQL: Reading an Objects

ORDER BY clause - allows the ordering of the results to be specified

GROUP BY clause - allows for summary information to be computed on a set of objects (normally is used with aggregation functions)

HAVING clause - allows for the results of a **GROUP BY** to be filtered

```
SELECT NEW samples.GroupInfo(c.location, COUNT(c))  
FROM Company c  
GROUP BY c.location  
HAVING COUNT(c) > 1  
ORDER BY c.location DESC
```

JPQL: Updating an Objects

UPDATE statement:

- Used to perform bulk update of entities
- Operates on a single entity type
- Equivalent to the SQL UPDATE statement, but with JPQL conditional expressions
- Doesn't allow joins, but supports sub-selects (in WHERE)
- Can only update attributes of the object or its embeddables but cannot update relationships

The persistence context is not updated to reflect results of update operations.

JPQL: Removing an Objects

DELETE statement:

- Used to perform bulk removal of an entities
- Equivalent to the SQL DELETE statement, but with JPQL conditional expressions
- Doesn't allow joins, but supports sub-selects (in WHERE)
- DELETE is polymorphic: any entity subclass instances that meet the criteria of the DELETE will be deleted

The persistence context may not be updated to reflect results of update operations.

JPQL: UPDATE and DELETE statements sample

```
UPDATE Employee e SET e.salary = 2000 WHERE e.salary = 1000
```

```
DELETE FROM Employee e WHERE e.department IS NULL
```

Examples of JPQL UPDATE and DELETE statements

JPQL: Supported Functions

JPQL support the following functions:

- **ABS** – returns absolute value
- **COALESCE** – evaluates to the first non null argument value
- **CONCAT** – concatenates two or more string values
- **CURRENT_DATE** - the current date on the database
- **CURRENT_TIME** - the current time on the database
- **LENGTH** – returns character/byte length of character/binary value
- **LOCATE** – returns the index of the string within the string
- **LOWER** – convert the string value to lower case
- **SUBSTRING** – returns the substring from the string
- **TRIM** – trims leading, trailing, or both spaces
- **UPPER** – converts the string value to upper case

Query API

Queries are represented in JPA 2 by interfaces:

- **`javax.persistence.Query`** – old interface which is also available in JPA 1.0.

Should be used mainly when the query result type is unknown or when a query returns polymorphic results.

- **`javax.persistence.TypedQuery`** – new interface introduced in JPA 2.0 (extends `javax.persistence.Query`)

Processes the query results in a type safe manner.

Should be used when a more specific result type is expected.

Query API

The **EntityManager** serves as a factory for both **Query** and **TypedQuery**:

```
/** create EntityManager */  
EntityManager em = entityManagerFactory.createEntityManager();
```

```
/** create old-style query object */  
Query query = em.createQuery("SELECT e FROM Employee");
```

Old-style query object that works with query results as abstract Object (downcasting is needed)

```
/** create JPA 2.0 style query object */  
TypedQuery<Employee> typedQuery =  
    em.createQuery("SELECT e FROM Employee", Employee.class);
```

New-style (JPA 2.0) query object that provides type-safe results fetching.

When building a **TypedQuery** instance the expected result type has to be passed as an additional argument

Query API

JPA provides static and dynamic queries:

- **Dynamic** queries – using of **JPQL** or **Criteria API**
- **Static** queries – using **@NamedQuery** and **@NamedQueries** to statically define query for entity

Using named queries (*instead of dynamic queries*) may improve code organization by separating the JPQL query strings from the Java code.

Query API

```
@Entity
@NamedQueries(
{
    @NamedQuery(
        name = "Employee.findAll",
        query = "SELECT e FROM Employee e ORDER BY e.name"),
    @NamedQuery(
        name = "Employee.findByName",
        query = "SELECT e FROM Employee e WHERE e.name = :name")
})
public class Employee {
    ...
}
```

Statically define named queries for the entity

```
public void sample(EntityManager em) {

    /** create JPA 1.0 old-style query object */
    Query query =
        em.createNamedQuery("Employee.findAll");

    /** create JPA 2.0 new-style query object */
    TypedQuery<Employee> typedQuery =
        em.createNamedQuery("Employee.findAll", Employee.class);
}
```

Create query objects for statically defined queries

Sample of using static named queries

Running JPA Queries

The **Query** interface defines two methods for running **SELECT** queries:

- **Query.getSingleResult()** – for use when exactly one result object is expected.
- **Query.getResultList()** – for general use in any other case.

Similarly, the **TypedQuery** interface defines the methods:

- **TypedQuery.getSingleResult()** – for use when exactly one result object is expected.
- **TypedQuery.getResultList()** – for general use in any other case.

Running JPA Queries

In addition, the **Query** interface defines a method for running **DELETE** and **UPDATE** queries:

- **Query.executeUpdate()** – for running the only **DELETE** and **UPDATE** queries.

Running JPA Queries

The following query retrieves all the `Employee` objects in the database. Because multiple result objects are expected, the query should be run using the `getResultList()` method.

```
Query query = em.createQuery("SELECT e FROM Employee");  
List employees = query.getResultList();  
  
TypedQuery<Employee> typedQuery =  
    em.createQuery("SELECT e FROM Employee", Employee.class);  
List<Employee> employees = typedQuery.getResultList();
```

Running JPA Queries

The following aggregate query always returns a single result object, which is a Long object reflecting the number of Employee objects in the database.

```
Query query = em.createQuery("SELECT COUNT(e) FROM Employee");
Long employeesCount = (Long) query.getSingleResult();

TypedQuery<Long> typedQuery =
    em.createQuery("SELECT COUNT(e) FROM Employee", Long.class);
Long employeesCount = typedQuery.getSingleResult();
```

Running JPA Queries

DELETE and UPDATE queries are executed using the `executeUpdate()` method.

```
Query deleteQuery = em.createQuery("DELETE FROM Employee");  
int countDeleted = deleteQuery.executeUpdate();  
  
Query updateQuery = em.createQuery("UPDATE Employee SET salary = 0");  
int countUpdated = deleteQuery.executeUpdate();
```

On Success – the method returns the number of objects that have been updated/deleted by the query

A `TransactionRequiredException` is thrown if no transaction is active.

Query Parameters

Query parameters enable the definition of reusable queries.



Running the same query multiple times with different parameter values (arguments) is more efficient than using a new query string for every query execution, because it eliminates the need for repeated query compilations.

JPA supports queries with parameters:

- Named Parameters (:name)
- Ordinal Parameters (?index)

Query Parameters: Named parameters

The following method retrieves an Employee object from the database by its name:

```
private Employee getEmployeeByName(EntityManager em, String employeeName) {  
    TypedQuery<Employee> query = em.createQuery(  
        "SELECT e FROM Employee e WHERE e.name = :name",  Define the query  
        Employee.class);                                with parameters  
  
    query.setParameter("name", employeeName);  Set the query parameter  
    return query.getSingleResult();           value and execute the query  
}
```

The WHERE clause reduces the query results to Employee objects whose name field value is equal to :name, which is a parameter that serves as a placeholder for a real value.

Query Parameters: Named parameters

Named parameters is identified in a query string by a colon (:) followed by a valid JPQL identifier (*that serves as the parameter name*)

Before the query can be executed a parameter value has to be set using the `setParameter()` method:

```
Query setParameter(String, Object)
```

```
TypedQuery<X> setParameter(String, Object)
```

Queries can include multiple parameters and each parameter can have one or more occurrences in the query string

Query Parameters: Ordinal parameters

The following method is equivalent to the method described previously but with an ordinal parameter:

```
private Employee getEmployeeByName(EntityManager em, String employeeName) {  
    TypedQuery<Employee> query = em.createQuery(  
        "SELECT e FROM Employee e WHERE e.name = ?1",  
        Employee.class);  
    query.setParameter(1, employeeName);  
    return query.getSingleResult();  
}
```

Define the query with parameters

Set the query parameter value and execute the query

The form of ordinal parameters is a question mark (?) followed by a positive integer number.

Besides the notation difference, named parameters and ordinal parameters are identical.

JPA Criteria API

JPA Criteria API provides an alternative way for defining JPA queries *(useful for building dynamic queries whose exact structure is known at runtime)*

The `CriteriaBuilder` interface serves as the main factory of criteria queries and criteria query elements

The `CriteriaBuilder` can be obtained by:

```
EntityManagerFactory.getCriteriaBuilder()
```

```
EntityManager.getCriteriaBuilder()
```

JPA Criteria API

The following query string represents a minimal JPQL query:

```
SELECT e FROM Employee e
```

An equivalent query can be built using the JPA criteria API as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();  
  
CriteriaQuery<Employee> query = cb.createQuery(Employee.class);  
Root<Employee> c = query.from(Employee.class);  
query.select(c);
```

JPA Criteria API

The following query string represents a JPQL query with a parameter:

```
SELECT e FROM Employee e WHERE e.name = :name
```

An equivalent query can be built using the JPA criteria API as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();  
  
CriteriaQuery<Employee> query = cb.createQuery(Employee.class);  
Root<Employee> fromClause = query.from(Employee.class);  
  
ParameterExpression<String> param = cb.parameter(String.class, "name");  
query.select(fromClause).where(cb.equal(fromClause.get("name"), param));
```

JPA Criteria API

The criteria API provides several ways for setting the SELECT clause:

- Single Selection
- Multiple Selection

Multiple selection can be implemented via:

- CriteriaBuilder's array
- CriteriaBuilder's construct

JPA Criteria API

Example of **single select** query constructed with help of Criteria API:

```
CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Department> criteriaQuery = cb.createQuery(Department.class);  
Root<Employee> fromClause = criteriaQuery.from(Employee.class);  
Selection<Department> selection = fromClause.get("department");  
criteriaQuery.select(selection).distinct(true);  
TypedQuery<Department> query = em.createQuery(criteriaQuery);  
List<Department> departments = query.getResultList();  
  
for (Department department : departments) {  
    System.out.println(department.getName());  
}
```

← Result will be of type Department

← And we're going to select from Employee

← Persistent field to select is "department"

← And we don't want to have duplicates in result

← Run the query and process result

And the resulting JPQL query:

```
SELECT DISTINCT e.department FROM Employee e
```


JPA Criteria API

Example of **multi select** query constructed with help of Criteria API with result of **array type**:

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

CriteriaQuery<Object[]> criteriaQuery = cb.createQuery(Object[].class);
Root<Employee> fromClause = criteriaQuery.from(Employee.class);
Selection<Object[]> selectionDepartmentName = fromClause.get("department").get("name");
Selection<Object[]> selectionEmployeeName = fromClause.get("name");
criteriaQuery.select(cb.array(selectionDepartmentName, selectionEmployeeName));

TypedQuery<Object[]> query = em.createQuery(criteriaQuery);
List<Object[]> results = query.getResultList();

for (Object[] res : results) {
    System.out.println(String.format("%1$s - %2$s", res[0], res[1]));
}
```

← Result will be of type Object[]

← And we're going to select from Employee

← The persistent fields to select are name and department.name

← Run the query and process result

And the resulting JPQL query:

```
SELECT e.department.name, e.name FROM Employee e
```

JPA Criteria API

Example of **multi select** query constructed with help of Criteria API with result of **user defined type**:

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

CriteriaQuery<SelectionResult> criteriaQuery = cb.createQuery(SelectionResult.class);

Root<Employee> fromClause = criteriaQuery.from(Employee.class);

Selection<?> selectionDepartmentName = fromClause.get("department").get("name");
Selection<?> selectionEmployeeName = fromClause.get("name");
criteriaQuery.select(
    cb.construct(
        SelectionResult.class, selectionDepartmentName, selectionEmployeeName
    )
);

TypedQuery<SelectionResult> query = em.createQuery(criteriaQuery);
List<SelectionResult> results = query.getResultList();

for (SelectionResult res : results) {
    System.out.println(String.valueOf(res));
}
```

Result will be of user defined type SelectionResult

And we're going to select from Employee

Specify persistent fields to select and user defined type to create for holding the result

Run the query and process result

And the resulting JPQL query:

```
SELECT NEW jpa.runtime.SelectionResult(e.department.name, e.name) FROM Employee e
```

JPA Criteria API

FROM query identification variables are represented in criteria queries by sub interfaces of From:


- Range variables are represented by the Root interface
- Join variables are represented by the Join interface (and its sub interfaces)


JPA Criteria API


Example of select query with JOIN constructed with help of Criteria API:

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


CriteriaQuery<Object[]> criteriaQuery = cb.createQuery(Object[].class);

Root<Department> fromClause = criteriaQuery.from(Department.class);  Select from Department

Join<Department, Company> joinClause = fromClause.join("company", JoinType.LEFT);  And join the Company entity

Selection<?> companyName = joinClause.get("name");
Selection<?> departmentName = fromClause.get("name");  Specify what to select

criteriaQuery.select(cb.array(companyName, departmentName));

TypedQuery<Object[]> query = em.createQuery(criteriaQuery);
List<Object[]> results = query.getResultList();  Run the query and process result

for (Object[] res : results) {
    System.out.println(String.format("%1$s - %2$s", res[0], res[1]));
}
```

And the resulting JPQL query:

```
SELECT c.name, d.name FROM Department d LEFT OUTER JOIN d.company c
```

JPA Criteria API

The `CriteriaQuery` interface provides two `where` methods for setting the `WHERE` clause:

- **Single Restriction:** `where()` method takes one `Expression<Boolean>` argument and uses it as the `WHERE` clause content
- **Multiple Restrictions:** `where()` method takes a variable number of arguments of `Predicate` type and uses an `AND` conjunction as the `WHERE` clause content

JPA Criteria API

Here is an example of single condition in WHERE clause:

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

CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
Root<Employee> fromClause = query.from(Employee.class);

ParameterExpression<String> param = cb.parameter(String.class, "name");
query.select(fromClause).where(cb.equal(fromClause.get("name"), param));
```

The resulting JPQL query is:

```
SELECT e FROM Employee e WHERE e.name = :name
```

JPA Criteria API

Example of multi conditions in WHERE clause:

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

CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
Root<Employee> fromClause = query.from(Employee.class);

ParameterExpression<Long> param1 = cb.parameter(Long.class, "id");
ParameterExpression<String> param2 = cb.parameter(String.class, "name");

query.select(fromClause).where(
    cb.equal(fromClause.get("id"), param1),
    cb.equal(fromClause.get("name"), param2)
);

query.select(fromClause).where(
    cb.or(
        cb.equal(fromClause.get("id"), param1),
        cb.equal(fromClause.get("name"), param2)
    )
);
```

Prepare parameter's expressions

Case #1: combine conditions by AND operand

Case #2: combine conditions by OR operand

The resulting JPQL queries are:

Case #1: `SELECT e FROM Employee e WHERE e.id = :id AND e.name = :name`
Case #2: `SELECT e FROM Employee e WHERE e.id = :id OR e.name = :name`



Practical work

Exercise 6:

Working with JPQL and Criteria API



Thank you for your attention!

Questions?