Session: 11

Query and Criteria API

Objectives

- Explain Java Persistence Query Language (JPQL)
- List the characteristics of JPQL
- Explain Query and TypedQuery API
- Explain how to write the named queries in JPQL
- Explain how to execute the named queries in JPQL
- Explain the methods to tune the JPQL query results
- Explain polymorphic queries
- Explain Criteria API
- Understand how to develop queries using Criteria API
- Explain how to manage Criteria query results
- Explain Metamodel API

Introduction

- Querying is the process of extracting data from the database.
- Structured Query Language (SQL) is used to retrieve data from the databases.
- Java Persistence Query Language (JPQL) is equivalent to SQL and is used by JPA.
- **JPQL:**
  - Is a string-based query language which allows the developers to write queries over entity objects created in the enterprise applications.
  - Offers queries that follow object-oriented approach.
  - Makes the application independent of a particular database schema.

Characteristics of JPQL

- Some of the characteristics of JPQL are as follows:
  - Is a query specification language.
  - Enables writing static and dynamic queries on persisted entities.
  - Compiles to a target language such as SQL.
  - Allows the developer to write queries on Abstract Persistence Schema.
  - Uses SQL like syntax to retrieve objects or values based on abstract schema.
  - Allows defining queries in meta data annotations or deployment descriptor.

Creating and Executing JPQL Queries

- Following are the steps for creating and executing a JPQL query:

Obtain an instance of `EntityManager` in a `PersistenceContext`.

Create a `Query` instance through methods of `EntityManager`.

Execute the query.

Retrieve the results of the query through methods of `Query` interface.

JPQL Named Query 1-2

- Defines a predefined static query string.
- Are created when certain query has to be executed repeatedly.
- Created through `javax.persistence.NamedQuery` annotation that has four attributes namely, `name`, `query`, `hints`, and `lockMode`.

- Following code snippet shows the usage of named query:

```
@NamedQuery(name =
Account.All"
query = "Select a from
Account a")
public class Account
implements Serializable {
   .  . .
}
```


JPQL Named Query 2-2

- Following table lists the attributes and descriptions of the `@NamedQuery` annotation:

Description	Attribute
Refers to the query.	`Name`
Refers to the query string written in the JPQL text format.	`Query`
Refers to the hints and properties related with the query.	`Hints`
Used in query execution.	`LockMode`

Multiple Named Queries

- Developers can attach multiple named queries to the same entity class through `@NamedQueries` annotation.
- Following code snippet shows the usage of `@NamedQueries` annotation:

```
@NamedQueries({
  @NamedQuery(name="Account.findAll",
              query="SELECT a FROM Account a”),
  @NamedQuery(name="Account.findByName",
              query="SELECT a FROM Account a WHERE
                                    a.name = :name")
             })
public class Account implements Serializable {
  .  . .
}
```


JPQL Dynamic Query

- Dynamic queries are used when the application has to build queries at runtime.
- Created through the `createQuery()` method of the `EntityManager` interface.
- Following code snippet demonstrates creation of dynamic queries:

```
@Stateless
public class AccountStatelessSessionBean implements AccountBean {
. . .
@PersistenceContext
public EntityManager e;
public List displayAccounts() {
   Query Q = e.createQuery("select a from Account a", Account.class);
   List accountsList = Q.getResultList();
   . . .
}}
```


Query Interface 1-2

- `Query` interface provides various methods to:
  - set parameters to the query.
  - execute the queries.
  - retrieve the results of the query.
  - set the pagination properties of the result set.
  - control the flush mode.
- JPA 2.0 introduces `TypedQuery` interface which extends the `Query` interface.
  - It is usually used when you expect the query to return more specific result types

Query Interface 2-2

- Following table shows methods provided by `Query` interface for query execution:

Method	Description
`List getResultList()`	Used to extract the result set of a query.
`Object getSingleResult()`	Used to extract one object from the result set.
`int executeUpdate()`	Used to execute an update or delete statement on the database.
`Query setMaxResult()`	Used to set the maximum number of results that can be received from a query.
`Query setFirstResult()`	Used to set the position of the first result the query is set to receive.
`void setParameter()`	Is an overloaded method and is used to set parameters for query execution.
`void setFlushMode()`	Used to set the flush mode to execute the query.

Executing the Named Query

- Following code snippet demonstrates the execution of named queries:

```
@Stateless
public class AccountStatelessSessionBean implement AccountBean {
  @PersistenceContext(unitName="mypersistenceunit")
  private EntityManager e;
  public List findAccounts() {
    Query q = e.createNamedQuery("Account.findAll", Account.class);
    List accList = q.getResultList();
    return accList;
 }
}
```

- The `findAccounts()` method invokes the `createNamedQuery()` method on the entity manager, which returns the `Query` object.

Parameterized Queries

- JPQL allows execution of parameterized queries.
- Parameters are dynamically passed during execution for these queries.
- The `setParameter()` method of `Query` interface is used to set parameters.
- The parameters are passed in two ways:
  - Based on the name of the parameters
  - Based on the position of the parameters

Named Parameters

- Are those variables in the query which can assume any value during execution.
- Are prefixed with a colon.
- Are set through `setParameter()` method which has the following prototype:
  - `Query setParameter(String parameter_name, Object parameter_value);`
- Following code snippet demonstrates the usage of named parameter:

```
Query Q = e.createQuery("Select a from Account a where
accnum = :acno", Account.class);
Q.setParameter("acno", 10234);
```


Positional Parameters

- Refer to the parameters in the query based on a numerical position in `setParameter()` method.
- Are sequentially numbered.
- Are prefixed with '`?`'.
- Following code snippet demonstrates the usage of positional parameters:

```
. . .
Query Q = e.createQuery("Select a from Account a where
accnum = ?1 and acc_bal =?2");
Q.setParameter(1, 10234);
Q.setParameter(2,2000);
.  . .
```


Temporal Parameters

- Temporal parameters are also referred to as Date parameters.
- Following are the types of objects which are passed as temporal parameters:
  - `Date`
  - `Time`
  - `TimeStamp`
- Temporal parameters are passed through the following `setParameter()` method:

```
query.setParameter("date", new java.util.Date(),
TemporalType.DATE);
```


Tuning the Queries

- Query interface provides various tuning methods that are used to effect the result of the query execution.
  - `getResultList()`
  - `getSingleResult()`
- These methods can be invoked before the query execution to set the result.

Paging the Result

- Paging is the process of accessing a subset of results of the query.
- Following are the methods used for paging the result of a query:
  - `getMaxResults()` – is used limit the maximum number of results from query execution.
  - `setFirstResult()` – is used for processing the output of an executed query.
- Following code snippet shows the usage of `getMaxResults()` and `setFirstResult()` methods:

```
. . .
Query Q = e.createQuery("Select a from Account);
Q.setParameter("acno", 10234);
List results =
Q.setFirstResult('10234').setMaxResult(1).getResultList();
. . .
```


Query Hints

- Hints are set along with the queries to define additional information which can be used while executing a query.
- The `setHint()` method is used to define hints which has the following syntax:

```
Query setHint(String hint_name, Object value);
```

- Following code snippet demonstrates the usage of `setHint()` method:

```
. . .
Query Q = e.createQuery("Select Customer_name from
BankAccount where accnum = :acno");
Q.setParameter("acno", 10234);
Q.setHint("timeout", 1000);
… . .
```


Flush Mode

- When an application performs an operation on the database, the changes are synchronized with the database.
- The `EntityManager` invokes `flush()` method to synchronize the changes with the database.
- There are two flush modes – `AUTO` and `COMMIT`.
- The `setFlushMode()` method is used to set the flush mode.
- Following is the syntax of `setFlushMode()` method:

```
Query setFlushMode(FlushModeType
flushmode);
```


WHERE Clause

- Used to specify a condition on the result set.
- Can apply multiple conditions using `AND` and `OR` operators.
- Following are the clauses which can be used with the `WHERE` clause:

DISTINCT	IN	LIKE
BETWEEN	IS NULL	IS EMPTY

DISTINCT Clause

- Used along with the `WHERE` clause in `SELECT` statement to eliminate duplicates in the result set.
- Following code snippet demonstrates the usage of `DISTINCT` clause in the query:

```
. . .
TypedQuery<String> query = em.createQuery(
"Select DISTINCT a.Customer_name from Account a",
String.class);

List<String> results = query.getResultList();
. . .
```


IN Clause

- Used to retrieve a subset of entities from a set of entities.
- Used to write nested queries.
- Following code snippet demonstrates the usage of `IN` clause:

```
. . .
TypedQuery<String> query = em.createQuery(
"SELECT b.name from Account b
WHERE acc_no IN(SELECT a.acc_no from Account a
WHERE city = :city)", String.class);

List<String> results = query.getResultList();
…..
```


LIKE Clause

- Used to match patterns in the query.
- Following code snippet demonstrates the usage of the `LIKE` clause:

```
. . .
TypedQuery<String> query = em.createQuery(
  "SELECT AVG(acc_bal) from Account a
WHERE city LIKE “B%”
GROUP BY city", String.class);
  List<String> results = query.getResultList();
. . .
```


BETWEEN Clause

- Used to specify a range of a property value in the `WHERE` clause.
- Following code snippet demonstrates the usage of `BETWEEN` clause:

```
. . .
TypedQuery<String> query = em.createQuery("SELECT
a.acc_no, a.name FROM Account a
WHERE a.acc_bal BETWEEN :min_sal and :max_sal"
,String.class);

List<String> results = query.getResultList();
. . .
```


IS NULL Clause

- Used along with `WHERE` clause to check whether certain property has an associated value or not.
- Following code snippet shows the usage of `IS NULL` clause:

```
. . .
TypedQuery<String> query = em.createQuery(
  " SELECT a.name, a.address
FROM Account a WHERE a.email IS NULL", String.class);

List<String> results = query.getResultList();
. . .
```


IS EMPTY Clause

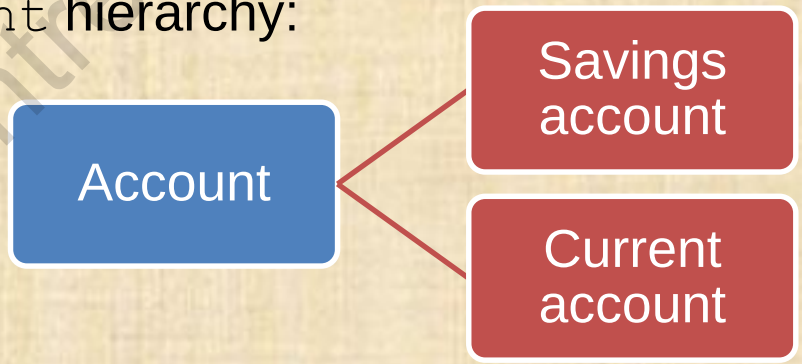
- Used to check the presence of data values in the database.
- Following code snippet demonstrates the usage of `IS EMPTY` clause:

```
. . .
TypedQuery<String> query = em.createQuery(
  " SELECT b.customer_name, b.customer_address
FROM BankAccount b WHERE b.cust_email IS EMPTY",
String.class);

List<String> results = query.getResultList();
. . .
```


Polymorphic Queries

- When a JPQL query is executed on an entity class, then the query executed is applicable to all its sub classes.
- The entity in the JPQL query is placed along with the 'where' clause in the query.
- By default all queries are polymorphic.
- Following figure shows the `Account` hierarchy:



- A query on `Account` entity is applicable to both the `SavingsAccount` entity and `CurrentAccount` entity.
- For example: `select customer_name from Account;`

Overview of Criteria API

- Used to define dynamic queries through query-defining objects.
- Are defined by instantiating Java objects.
- Offers better integration with Java language than JPQL queries as it operates on the database in terms of objects.
- Accepts values for each clause of the query through different methods and validates them.
- Are queries based on abstract schema of persistent entities, their relationships, and embedded objects.

Creating Criteria Query

- Criteria API creates queries by obtaining an instance of `javax.persistence.criteria.CriteriaQuery`.
- The `createQuery()` method is used to create the query.
- Following code snippet demonstrates the creation of query:

```
. . .
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery();
. . .
```


Query Root

- The query root in Criteria API query is entity where the navigation initiates.
- Created by invoking `from()` method of `CriteriaQuery` interface.
- The query root can be defined as:

```
Root c = cq.from(Customer.class);
```

- For queries which require multiple entities at the root, the from() method is invoked with multiple entity classes as parameter.

Using Criteria API 1-5

- Following are the steps involved in creating a criteria API query:

`CriteriaBulider` object created by `EntityManager`

`CriteriaBuilder` creates `CriteriaQuery` object to create `SELECT` queries

`from()` method is used to identify the entities on which the query is defined

Each clause of the query is defined as a method, a `where()` method is used to apply conditions

Using Criteria API 2-5

`select()` and `multiselect()` methods are used to return objects from the query

The `multiselect()` method is used in instances where multiple properties of the entity are to be selected

The query is executed by invoking the method `getResultList()`

Using Criteria API 3-5

- Following code snippet demonstrates various Criteria queries:

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

// Query for a List of objects
CriteriaQuery cq = cb.createQuery();
Root c = cq.from(Customer.class);
cq.where(cb.greaterThan(c.get("salary"), 100000));
Query query = em.createQuery(cq);

List<Employee> result = query.getResultList();
```


Using Criteria API 4-5

```
// Query for a single object
CriteriaQuery cq = cb.createQuery();
Root c = cq.from(Customer.class);
cq.where(cb.equal(c.get("id"), cb.parameter(Long.class,
"id")));
Query query = em.createQuery(cq);
query.setParameter("id", id);
Customer result2 = (Customer)query.getSingleResult();

// Query for a single data element
CriteriaQuery cq = cb.createQuery();
Root c = cq.from(Customer.class);
cq.select(cb.max(c.get("salary")));
Query query = em.createQuery(cq);
BigDecimal result3 = (BigDecimal)query.getSingleResult();
```


Using Criteria API 5-5

```
// Query for a List of data elements
CriteriaQuery cq = cb.createQuery();
Root c = cq.from(Customer.class);
cq.select(c.get("firstName"));
Query query = em.createQuery(cq);
List<String> result4 = query.getResultList();

// Query for a List of element arrays
CriteriaQuery cq = cb.createQuery();
Root c = cq.from(Customer.class);
cq.multiselect(c.get("firstName"), c.get("lastName"));
Query query = em.createQuery(cq);
List<Object[]> result5 = query.getResultList();
….
```


Managing Criteria Query Results 1-3

- The Criteria API provides methods such as `orderBy()` and `groupBy()` to implement `ORDER BY` and `GROUP BY` clauses.
- The `asc()` and `desc()` methods are used to sort the results.
- Following code snippet demonstrates the usage of `orderBy()` method:

```
…..
CriteriaQuery<BankAccount> cq =
cb.createQuery(BankAccount.class);
Root<BankAccount> BA = cq.from(BankAccount.class);
cq.select(BA);
cq.orderBy(cb.desc(BA.get(BankAccount_.acc_bal)));
….
```


Managing Criteria Query Results 2-3

- Following code snippet demonstrates the usage of `groupBy()` method:

```
…..
CriteriaQuery<BankAccount> cq =
cb.createQuery(BankAccount.class);
Root<BankAccount> BA = cq.from(BankAccount.class);
cq.select(BA);
cq.orderBy(cb.desc(BA.get(BankAccount_.acc_bal)));
….
```


Managing Criteria Query Results 3-3

- Following code snippet demonstrates the usage of `having()` method:

```
….
CriteriaQuery<BankAccount> cq =
cb.createQuery(BankAccount.class);
Root<BankAccount> BA = cq.from(BankAccount.class);
cq.groupBy(BA.get(BankAccount_.acc_type));
cq.having(cb.in(BA.get(BankAccount_.acc_type)).value(sa
vings));
….
```


Executing Criteria Query

`CriteriaBuilder` prepares for the execution of the query

A `TypedQuery` object must be created which is the return type of the query

`TypedQuery` object must be defined through the `createQuery()` method of `EntityManager`

`getSingleResult()` and `getResultList()` methods are used for executing queries

Querying Relationships Using Joins

- Criteria queries implement join operation on entities through `join()` method.
  - The `join` operation implies comparing the values of one attribute of an entity with a similar attribute of another entity.
- Following code snippet demonstrates the usage of `join()` method:

```
…..
CriteriaQuery<BankAccount> cq =
cb.createQuery(BankAccount.class);
Root<BankAccount> BA = cq.from(BankAccount.class);
Join<BankAccount, Customer> result =
cq.join(BankAccount_.Customer);

….
```


Using Metamodel API 1-4

- MetaModel API works along with the Criteria API to model persistence entity classes.
- JPA Metamodel API enables the developers to understand the database schema of the application data.
- Metamodel comprises information about the mapped attributes for an entity class, their corresponding data types, and so on.
- The Metamodel classes corresponding to the entity classes are of the form:
  - `javax.persistence.metamodel.EntityType<T>`

Using Metamodel API 2-4

- The developer can use Metamodel API to create the metamodel of the entities managed in the persistence unit of the application.
- Following code snippet shows an entity class:

```
. . .
@Entity
public class BankAccount{
@Id
protected Long acc_no;
protected String account_holder_name;
protected String acc_type;
protected Long acc_bal
...
}
```


Using Metamodel API 3-4

- Following code snippet shows the metamodel class for the `BankAccount` entity class:

```
...
@Static Metamodel(BankAccount.class)
public class BankAccount_ {
public static volatile SingularAttribute<BankAccount,
Long> acc_no;
public static volatile SingularAttribute<BankAccount,
String> account_holder_name;
public static volatile SingularAttribute<BankAccount,
String> acc_type;
public static volatile SingularAttribute<BankAccount,
Long> acc_bal;
.. .
}
```


Using Metamodel API 4-4

- A Metamodel class can be generated through `getModel()` method.
- Following code snippet demonstrates the usage of `getModel()` method to obtain the `BankAccount` entity:

```
CriteriaQuery cq =
cb.createQuery(BankAccount.class);
Root<BankAccount> c = cq.from(BankAccount.class);
EntityType<BankAccount> account_ =  c.getModel();
```


Summary

- Querying of databases is implemented through Java Persistence Query Language (JPQL) and Criteria API.
- EntityManager instance is responsible for executing both named queries and dynamic queries.
- Query API executes the queries created by the EntityManager and extracts the results.
- JPQL can be used to write queries equivalent to SQL queries. The developer can attach multiple named queries to the same entity class.
- JPA 2.0 introduced TypedQuery interface. It extends the Query interface and is used to control the execution of the query.
- Criteria queries are used to write object-based queries.
- Criteria API queries are based on the abstract schema of persistent entities, their relationships, and embedded objects.
- JPQL queries are preferred for simple static queries, whereas criteria API queries are used for dynamic queries.
- The TypedQuery object must be defined through the createQuery method invoked by the EntityManager.
- The MetaModel API works along with Criteria API to model persistence entity classes. These persistence entity classes are used by the Criteria queries.