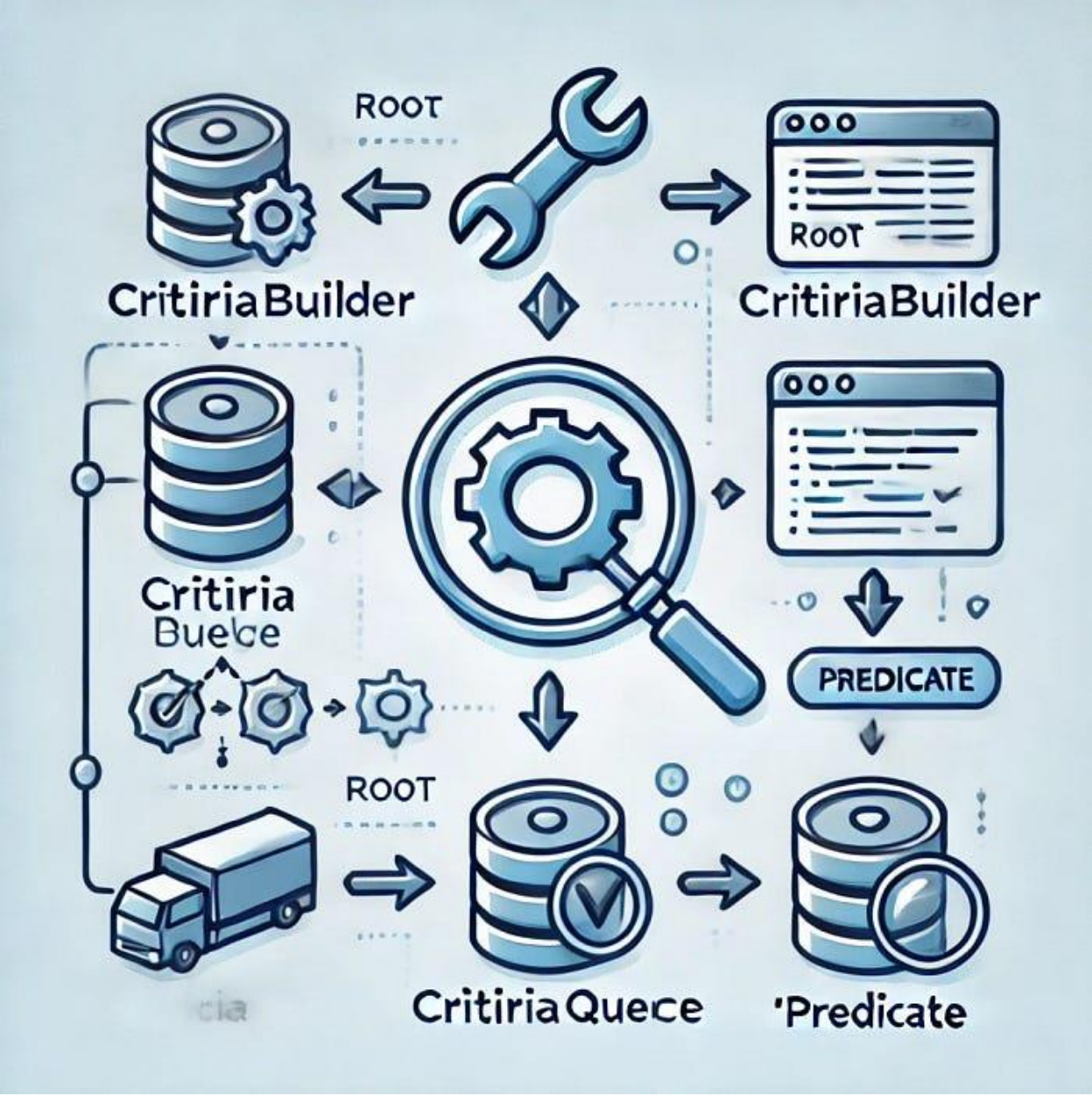
CriteriaQuery in Java



In the realm of **Java Persistence API (JPA), CriteriaQuery** stands out as a robust, type-safe mechanism for building dynamic queries.

Introduced in JPA 2.0, the Criteria API provides an **alternative** to the **traditional JPQL (Java Persistence Query Language) and SQL queries**, offering developers a **fluent, programmatic way** to construct queries.

This article delves deep into the workings of **CriteriaQuery**, exploring its components, usage, and benefits.

What is CriteriaQuery?

CriteriaQuery is part of the JPA Criteria API, a comprehensive set of classes and interfaces designed to construct and execute queries in a type-safe manner.

Unlike **JPQL,** which is **string-based** and **prone to errors that are only caught at runtime, CriteriaQuery** allows for **compile-time checking of query constructs,** reducing the **risk of runtime errors** and enhancing code maintainability.

Core Components of CriteriaQuery

1. **CriteriaBuilder:** This is the starting point for constructing queries using the Criteria API. The **CriteriaBuilder interface** provides methods to create various types of **criteria,** such as **CriteriaQuery,**

Predicate, Expression, etc.

2. **CriteriaQuery<T>:** Represents a specific query structure. It is a generic interface where T represents the result type of the query. It supports various methods to define the query's **select, where, order by, group by, and having clauses.**
3. **Root<T>:** Represents the entity that is being queried. It acts as the root from which all paths in the query are derived.
4. **Predicate:** Used to form **conditional expressions** in the query. Predicates can be combined using logical operators such as **AND, OR, and NOT.**
5. **Expression<T>:** Represents an expression over the data in the query. It can be used to refer to entity attributes, perform **arithmetic operations, or apply functions.**

How to Use CriteriaQuery

1. Creating a Simple Query

To create a simple query using **CriteriaQuery**, follow these steps:

- Obtain a **CriteriaBuilder** instance from the **EntityManager.**
- Create a **CriteriaQuery** instance.
- Define the **root** entity from which the query will start.

- Set the **query's selection** criteria and other clauses.

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

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

Root<Employee> employee = cq.from(Employee.class);

cq.select(employee).where(cb.equal(employee.get("department"),
"Sales"));

TypedQuery<Employee> query = entityManager.createQuery(cq);

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

2. Complex Queries with Multiple Conditions

CriteriaQuery can handle complex queries involving multiple conditions, joins, groupings, and more.

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

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

Root<Employee> employee = cq.from(Employee.class);
Join<Employee, Department> department =
employee.join("department");

cq.select(employee)
```

```
    .where(cb.and(
        cb.equal(department.get("name"), "Sales"),
        cb.greaterThan(employee.get("salary"), 50000)
    ));

TypedQuery<Employee> query = entityManager.createQuery(cq);

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

3. Ordering and Grouping

CriteriaQuery allows for specifying ordering and grouping of results.

```
cq.orderBy(cb.asc(employee.get("name")));

cq.groupBy(department.get("name"));
```

Advantages of Using CriteriaQuery

1. **Type Safety:** Compile-time checking of query constructs prevents many runtime errors.
2. **Dynamic Query Building:** Queries can be constructed dynamically based on user input or other runtime conditions.
3. **Ease of Refactoring:** Changes in entity attributes are easily reflected in the queries, as they are constructed using Java's type system.

4. **Readability and Maintainability:** The fluent API style makes the query construction process more readable and easier to maintain.

CriteriaQuery is a powerful feature of JPA that offers a type-safe, programmatic way to construct complex and dynamic queries.

It provides numerous benefits over traditional JPQL, including improved type safety, easier maintenance, and dynamic query capabilities.

By understanding and leveraging **CriteriaQuery**, developers can write more robust, efficient, and maintainable code for interacting with databases.

==========T

This Page is Intentionally Left Blank

CriteriaBuilder Methods

The **most commonly used methods in CriteriaBuilder** — this class is central to building dynamic and type-safe queries in JPA using the Criteria API.

CriteriaBuilder – Key Methods Cheat Sheet

Here's a categorized list of useful methods with examples and SQL equivalents.

---

1. Comparison Methods

Method	Description	SQL Equivalent
equal(x, y)	Equals	x = y
notEqual(x, y)	Not equals	x != y
greaterThan(x, y)	Greater than	x > y
greaterThanOrEqualTo(x, y)	Greater than or equal to	x >= y
lessThan(x, y)	Less than	x < y
lessThanOrEqualTo(x, y)	Less than or equal to	x <= y

Example:

```
cb.greaterThan(user.get("age"), 25);
```

---

2. String Methods

Method	Description	SQL Equivalent
like(x, pattern)	String matching	x LIKE pattern
notLike(x, pattern)	NOT LIKE	x NOT LIKE pattern

concat(x, y)	Concatenate strings	`x
lower(x)	Lowercase	LOWER(x)
upper(x)	Uppercase	UPPER(x)
length(x)	String length	LENGTH(x)

Example:

```
cb.like(cb.lower(user.get("username")), "%shek%");
```

---

3. Logical Methods

Method	Description
and(p1, p2, ...)	Combine with AND
or(p1, p2, ...)	Combine with OR
not(p)	Negate a predicate

Example:

```
cb.and(
   cb.equal(user.get("role"), "ADMIN"),
   cb.like(user.get("email"), "%@gmail.com")
);
```

---

4. Arithmetic Methods

Method	Description	SQL Equivalent
sum(x, y)	Sum	x + y
diff(x, y)	Subtraction	x - y
prod(x, y)	Multiplication	x * y
quot(x, y)	Division	x / y

mod(x, y)	Modulo	MOD(x, y)

Example:

cb.greaterThan(cb.sum(order.get("price"), order.get("tax")), 100);

---

5. Aggregation / Grouping Methods

Method	Description
count(x)	Count records
max(x)	Maximum value
min(x)	Minimum value
avg(x)	Average value
sum(x)	Sum
groupBy(expr...)	Grouping
having(expr)	Filtering on aggregates

Example:

cq.select(cb.count(user)).where(cb.like(user.get("email"), "%@gmail.com"));

---

5. Date/Time Methods (JPA 2.1+)

Method	Description
currentDate()	Current date
currentTime()	Current time
currentTimestamp()	Current timestamp

Example:

```
cb.lessThan(order.get("orderDate"), cb.currentDate());
```

Real-World Example

```
Predicate p1 = cb.like(user.get("username"), "%venkat%");
Predicate p2 = cb.greaterThan(user.get("age"), 25);
Predicate p3 = cb.lessThanOrEqualTo(user.get("age"), 40);

cq.select(user)
  .where(cb.and(p1, cb.or(p2, p3)))
  .orderBy(cb.asc(user.get("username")));
```

SQL Equivalent:

```
SELECT * FROM user
WHERE username LIKE '%venkat%'
  AND (age > 25 OR age <= 40)
ORDER BY username ASC
```

CriteriaQuery methods

CriteriaQuery methods — these are essential when you're working with the **JPA Criteria API** and need to construct the structure of your query: **what to select**, **how to filter**, **group**, **sort**, etc.

What is CriteriaQuery<T>?

CriteriaQuery<T> is the main object that represents a **JPA query** in a type-safe way. It defines the **query result type**, the **FROM clause**, the **SELECT**, and optional **WHERE**, **GROUP BY**, **ORDER BY**, and more.

Common CriteriaQuery Methods

Let's break them down by category with examples:

---

1. select(...)

Defines what the query should return.

```
cq.select(root); // SELECT u FROM User u
```

You can also select specific fields or **build DTOs**:

```
cq.select(cb.construct(UserDTO.class, root.get("id"),
root.get("username")));
```

---

2. where(...)

Adds conditions (predicates) to the WHERE clause.

```
cq.where(cb.equal(root.get("email"), "venkat@gmail.com"));
```

With multiple conditions:

```
cq.where(cb.and(
    cb.like(root.get("username"), "%venkat%"),
    cb.greaterThan(root.get("age"), 25)
));
```

---

3. from(...)

Defines the root entity (i.e., the FROM clause).

Typically used like this:

```
Root<User> root = cq.from(User.class); // FROM User u
```

You can join other entities too:

```
Join<User, Address> addressJoin = root.join("address");
```

---

4. orderBy(...)

Sets the ORDER BY clause.

```
cq.orderBy(cb.asc(root.get("username")));
cq.orderBy(cb.desc(root.get("createdDate")));
```

Multiple orders:

```
cq.orderBy(cb.asc(root.get("role")), cb.desc(root.get("username")));
```

---

5. groupBy(...)

Sets GROUP BY expressions.

```
cq.groupBy(root.get("department"));
```

---

6. having(...)

Adds HAVING clause to filter grouped data.

```
cq.groupBy(root.get("role"))
  .having(cb.gt(cb.count(root), 10));
```

---

distinct(...)

Eliminates duplicate results.

```
cq.distinct(true);
```

---

8. multiselect(...)

Select multiple fields (alternative to select).

```
cq.multiselect(root.get("username"), root.get("email"));
```

Useful for returning Object[] or mapping to DTOs.

---

9. subquery(...)

Creates subqueries inside your query.

```
Subquery<Long> sub = cq.subquery(Long.class);
Root<User> subRoot = sub.from(User.class);

sub.select(cb.count(subRoot)).where(cb.equal(subRoot.get("role"),
"ADMIN"));
```

```
cq.where(cb.greaterThan(sub, 5L));
```

Sample Full Query

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

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

Root<User> root = cq.from(User.class);

cq.select(root)
  .where(cb.and(
     cb.like(root.get("username"), "%Venkat%"),
     cb.greaterThanOrEqualTo(root.get("age"), 25)
  ))
  .orderBy(cb.asc(root.get("username")))
  .distinct(true);

List<User> result = em.createQuery(cq).getResultList();
```

Output SQL (approx.)

```
SELECT DISTINCT *
FROM user
WHERE username LIKE '%venkat%' AND age >= 25
ORDER BY username ASC;
```

Methods in Root

The **Root<T> interface** represents the **main entity/table** in a criteria query — like the root of a SQL FROM clause.

It extends From<T, T> and is the starting point for constructing path expressions to access entity fields, joins, etc.

What is Root<T>?

Think of it as:

Root<User> root = criteriaQuery.from(User.class);

This is conceptually:

SELECT * FROM User u

Now you use root to access fields like u.id, u.username, etc.

Commonly Used Root<T> Methods

Method	Purpose / SQL Equivalent
get(String attributeName)	Access a field/column (e.g. user.get("email") → u.email)
join(String attributeName)	Perform an inner join
join(String, JoinType)	Join with type: INNER, LEFT, RIGHT
isNull() / isNotNull()	Check if a column is null/not null
alias(String name)	Assign an alias to this path
type()	Access the entity type (for polymorphic queries)
getModel()	Get metadata about the entity

Examples

1. Basic Field Access

```
cb.like(root.get("username"), "%venkat%");
```

SQL: WHERE username LIKE '%venkat%'

---

2. Access Nested Entity via Join

Assume User has a Department:

```
Join<User, Department> dept = root.join("department");
cb.equal(dept.get("name"), "HR");
```

SQL: **INNER JOIN department d ON u.department_id = d.id WHERE d.name = 'HR'**

---

3. Left Join

```
Join<User, Address> address = root.join("address", JoinType.LEFT);
```

---

4. Alias Usage

Helpful in complex queries or projections.

```
root.alias("u");
```

---

5. Type Discriminator (for Inheritance)

```
Expression<Class<?>> type = root.type();
cb.equal(type, Admin.class);
```

---

6. Get Path to Nested Fields

If you have something like user.address.city, you can do:

```
root.get("address").get("city");
```

Root<T> in Multiselect or DTO Queries

```
cq.multiselect(
    root.get("id"),
    root.get("username"),
    root.get("email")
);
```

Or for DTO mapping:

```
cq.select(cb.construct(UserDTO.class,
    root.get("id"),
    root.get("username"),
    root.get("email")
));
```

Summary Table

Method	Example	Purpose
get("field")	root.get("email")	Get column
join("relation")	root.join("department")	Inner join
join("rel", JoinType)	root.join("address", LEFT)	Left/right join
alias("name")	root.alias("u")	Give alias
getModel()	root.getModel()	Get metadata about entity class
type()	root.type()	Polymorphic (inheritance) use

Summary

CriteriaBuilder – Methods

Method	Description / SQL Equivalent
equal(x, y)	x = y
notEqual(x, y)	x != y
greaterThan(x, y)	x > y
greaterThanOrEqualTo(x, y)	x >= y
lessThan(x, y)	x < y
lessThanOrEqualTo(x, y)	x <= y
like(x, pattern)	LIKE string match
notLike(x, pattern)	Negated LIKE
and(p1, p2, ...)	Logical AND
or(p1, p2, ...)	Logical OR
not(p)	Logical NOT
isNull(x)	x IS NULL
isNotNull(x)	x IS NOT NULL
count(x)	Aggregate count
sum(x)	Aggregate sum
avg(x)	Aggregate average
min(x)	Aggregate min
max(x)	Aggregate max
currentDate() / currentTime()	Current date/time/timestamp
concat(x, y)	Concatenate strings
lower(x) / upper(x)	Case conversion
length(x)	String length
construct(Class, args...)	Used to build DTOs from selected fields

CriteriaQuery<T> – Methods

Method	Description / SQL Equivalent
select(x)	Defines what to select (entire entity or field)
multiselect(...)	Select multiple fields
from(Entity.class)	Define FROM clause
where(...)	Add WHERE conditions
orderBy(...)	Add ORDER BY clause
groupBy(...)	Add GROUP BY clause
having(...)	Add HAVING clause after grouping
distinct(true/false)	Enable/disable DISTINCT result
subquery(Class<T>)	Create a subquery

Root<T> – Methods

Method	Description / SQL Equivalent
get("field")	Access entity attribute (u.email)
join("relation")	Join to related entity (INNER JOIN)
join("rel", JoinType.X)	Join with type: INNER, LEFT, RIGHT
alias("aliasName")	Set alias (used in advanced queries)
type()	Get entity type (useful in inheritance)
getModel()	Metadata for the entity

Quick Visual Reference

Component	Role	Think of it like...
CriteriaBuilder	Query **builder / factory**	SQL expression writer
CriteriaQuery	**Structure** of the SQL query	SELECT + WHERE + GROUP BY
Root<T>	Represents **entity/table**	FROM clause + fields