Usage of CriteriaBuilder in Spring JPA – 2025

**SpringBoot Version: 3.5.6, Java Version: 21**

**CriteriaBuilder** is part of the **JPA Criteria API**, which allows you to build **type-safe**, **dynamic queries** programmatically at runtime. It's especially useful when query conditions depend on user input or runtime logic.

**Advantages**

* Type safety: Avoids runtime errors due to incorrect field names.
* Dynamic query construction: Ideal for complex filters and conditional logic.

**Limitations**

* Verbose syntax: More boilerplate compared to JPQL or method queries.
* Steeper learning curve: Requires understanding of JPA internals.

🔄 **Common Use Cases**

* Building queries with multiple AND/OR conditions
* Filtering based on optional parameters
* Paginated and sorted queries
* Complex joins and subqueries

We should know about the following APIs in case of CriteriaBuilder in Spring JPA.

* **CriteriaBuilder**: Entry point to create queries and predicates. Factory for query parts
* **CriteriaQuery<T>**: Represents the actual query; T denotes the expected result type. Represents a query object
* **Root<T>**: Refers to the root entity from which attributes and joins are derived. Represents table/entity.
* **Predicate**: Represents conditions in WHERE or HAVING clauses. WHERE conditions
* **Expression<T>**: Used for calculations, conditions, or data transformations.

**Common Methods in CriteriaBuilder**

* cb.equal(x, y): Creates a condition for equality (x = y).
* cb.notEqual(x, y): Creates a condition for inequality (x != y).
* cb.greaterThan(x, y): Creates a condition for x > y.
* cb.lessThan(x, y): Creates a condition for x < y.
* cb.like(x, pattern): Creates a condition for pattern matching (LIKE).
* cb.isNull(x): Checks if x is NULL.
* cb.isNotNull(x): Checks if x is not NULL.
* cb.and(predicates...): Combines multiple predicates with AND.
* cb.or(predicates...): Combines multiple predicates with OR.

Example: Create a Table called “**employee**” with the fields id, name, age, salary, designation, city, status, email.

Create the service layer with the basic structure.

@Service

**public** **class** EmployeeServiceImpl {

**@Autowired**

**private EntityManager entityMgr**;

}

**Use Case-1: Find the employees where city name = “Bhubaneswar”**

**public** List<Employee> findByCity(String cityName) {

// Step 1: Get CriteriaBuilder

**CriteriaBuilder cb = entityMgr.getCriteriaBuilder();**

// Step 2: Create CriteriaQuery

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

// Step 3: Define the root (FROM clause)

**Root<Employee> emp = query.from(Employee.class);**

// Step 4: Create predicates (WHERE clause)

**Predicate cityPredicate = cb.equal(emp.get("city"), cityName);**

// Step 5: Apply predicates to the query

**query.select(emp).where(cityPredicate);**

// Step 6: Execute the query

**return** **entityMgr.createQuery(query).getResultList();**

}

**Use Case-2: Find the employees where city name = “Bhubaneswar” ignoring the case.**

**public** List<Employee> findByCityIgnoreCase(String cityName) {

CriteriaBuilder cb = entityMgr.getCriteriaBuilder(); // Step 1: Get CriteriaBuilder

// Step 2: Create CriteriaQuery

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

// Step 3: Define the root (FROM clause)

Root<Employee> empRoot = query.from(Employee.**class**);

Predicate cityPredicate = cb.equal(cb.lower(empRoot.get("city")), cityName.toLowerCase());

query.select(empRoot).where(cityPredicate);

**return** entityMgr.createQuery(query).getResultList();

}

**Use Case-3: Use of AND in query. Where city = ? and status = ? and age = ?**

**public** List<Employee> getEmpsUsingAnd(String city, String status, **int** age) {

// Where city = Rourkela and status = On Leave and age > 20

CriteriaBuilder cb = entityMgr.getCriteriaBuilder();

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

Root<Employee> empRoot = query.from(Employee.**class**);

Predicate cityPredicate = cb.equal(empRoot.get("city"), city);

Predicate statusPredicate = cb.like(empRoot.get("status"), status);

Predicate agePredicate = cb.greaterThan(empRoot.get("age"), age);

Predicate andPredicate = cb.and(cityPredicate, statusPredicate, agePredicate);

query.select(empRoot).where(andPredicate);

**return** entityMgr.createQuery(query).getResultList();

}

The above can be written using @Query annotation.

@Query("""

**SELECT e FROM Employee e WHERE**

**e.city = :city**

**AND e.status = :status**

**AND e.age > :age**

"""

)

List<Employee> getEmpsUsingAnd(String city, String status, **int** age);

**Use Case-4: Get employee with several where clauses.**

**public** List<Employee> getEmp1(String city, Integer minAge, Double maxSal, String status) {

// Step 1: Get CriteriaBuilder

CriteriaBuilder cb = entityMgr.getCriteriaBuilder();

// Step 2: Create CriteriaQuery

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

// Step 3: Define the root (FROM clause)

Root<Employee> empRoot = query.from(Employee.**class**);

List<Predicate> predicates = **new** ArrayList<Predicate>();

**if** (city != **null**) {

Predicate cityPredicate = cb.equal(empRoot.get("city"), city);

predicates.add(cityPredicate);

}

**if** (minAge != **null**) {

Predicate minAgePredicate = cb.greaterThan(empRoot.get("age"), minAge);

predicates.add(minAgePredicate);

}

**if** (Objects.*nonNull*(predicates)) {

Predicate maxSalPredicate = cb.le(empRoot.get("salary"), maxSal);

predicates.add(maxSalPredicate);

}

**if** (!Objects.*isNull*(status)) {

Predicate statusPredicate = cb.like(empRoot.get("status"), status);

predicates.add(statusPredicate);

}

query.select(empRoot).where(cb.and(predicates.toArray(**new** Predicate[0])));

**return** entityMgr.createQuery(query).getResultList();

}

The above code can be rewritten using @Query annotation.

@Repository

**public** **interface** EmployeeRepository **extends** CrudRepository<Employee, Long> {

@Query("""

**SELECT e FROM Employee e**

**WHERE (:city IS NULL OR e.city = :city)**

**AND (:minAge IS NULL OR e.age > :minAge)**

**AND (:maxSal IS NULL OR e.salary <= :maxSal)**

**AND (:status IS NULL OR e.status LIKE :status)**

""")

List<Employee> getEmpDetails(String city, Integer minAge, Double maxSal, String status);

}

**Use Case-5: Use of IN query. Where city in (“Rourkela”, “Bhubaneswar”)**

**public** List<Employee> getEmpsInCities(List<String> cityList) {

CriteriaBuilder cb = entityMgr.getCriteriaBuilder();

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

Root<Employee> empRoot = query.from(Employee.**class**);

Predicate cityInpredicate = empRoot.get("city").in(cityList);

query.select(empRoot).where(cityInpredicate);

**return** entityMgr.createQuery(query).getResultList();

}

Equivalent Code using @Query annotation.

@Query("""

SELECT e FROM Employee e WHERE e.city in :cityList

""")

List<Employee> getEmpsInCities(List<String> cityList);

**Use Case-6: Use of OR in query. Where city = ? OR status = ? OR age = ?**

**public** List<Employee> getEmpsUsingOR(**double** sal, String status, **int** age) {

CriteriaBuilder cb = entityMgr.getCriteriaBuilder();

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

Root<Employee> empRoot = query.from(Employee.**class**);

Predicate salPredicate = cb.greaterThan(empRoot.get("salary"), sal);

Predicate statusPredicate = cb.like(empRoot.get("status"), status);

Predicate agePredicate = cb.greaterThan(empRoot.get("age"), age);

Predicate andPredicate = cb.or(salPredicate, statusPredicate, agePredicate);

query.select(empRoot).where(andPredicate);

**return** entityMgr.createQuery(query).getResultList();

}

The above can be written using @Query annotation.

@Query("""

SELECT e FROM Employee e WHERE

e.salary > :sal

OR e.status LIKE :status

OR e.age > :age

"""

)

List<Employee> getEmpsUsingOR(**double** sal, String status, **int** age);

**Use Case-7: Use of NULL in query. Where city can be null.**

**public** List<Employee> getEmpsForNullCity(String city) {

CriteriaBuilder cb = entityMgr.getCriteriaBuilder();

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

Root<Employee> empRoot = query.from(Employee.**class**);

Predicate nullCityPredicate = empRoot.get("city").isNull();

query.select(empRoot).where(nullCityPredicate);

**return** entityMgr.createQuery(query).getResultList();

}

Equivalent code using @Query annotation.

@Query("""

SELECT e FROM Employee e

WHERE (e.city IS NULL)

""")

List<Employee> getEmpsForNullCity(String city);

**Use Case-8: Use of NOT NULL in query.**

**public** List<Employee> getEmpsForNullCity(String city) {

CriteriaBuilder cb = entityMgr.getCriteriaBuilder();

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

Root<Employee> empRoot = query.from(Employee.**class**);

Predicate nullCityPredicate = empRoot.get("city").isNull();

query.select(empRoot).where(nullCityPredicate);

**return** entityMgr.createQuery(query).getResultList();

}

Equivalent code using @Query Annotation

@Query("""

SELECT e FROM Employee e

WHERE (e.city IS NOT NULL)

""")

List<Employee> getEmpsForNotNullCity();

Use of Specification in Spring JPA

In Spring Data JPA, a **Specification** is a powerful way to define **dynamic, type-safe queries** using the **JPA Criteria API**. It encapsulates query logic into reusable components and is especially useful for filtering based on multiple optional parameters. It’s part of the **Spring Data JPA Criteria support**, introduced via the JpaSpecificationExecutor interface.

**public interface Specification<T> {**

**Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder cb);**

**}**

* T: The entity type.
* Root: Represents the entity in the query.
* CriteriaQuery: The query being built.
* CriteriaBuilder: Used to construct predicates and expressions.

How to Use Specification

1. **Extend JpaSpecificationExecutor in your repository**

public interface BookRepository extends JpaRepository, **JpaSpecificationExecutor** { }

2. **Create Specification**

public class BookSpecifications {

public static Specification<Book> hasAuthor(String author) {

return (root, query, cb) -> cb.equal(root.get("author"), author);

}

public static Specification<Book> titleContains(String keyword) {

return (root, query, cb) -> cb.like(root.get("title"), "%" + keyword + "%");

}

}

3. **Combine Specifications**

Specification<Book> spec = Specification

.where(BookSpecifications.hasAuthor("John"))

.and(BookSpecifications.titleContains("Spring"));

List<Book> results = bookRepository.findAll(spec);

**Advantages**

* **Modular**: Each filter is a separate method, making it easy to reuse.
* **Dynamic**: Combine conditions at runtime based on user input.
* **Type-safe**: Uses Criteria API, avoiding string-based errors.
* **Composable**: Chain multiple specs with .and(), .or(), .not().

**Considerations**

* Requires understanding of Criteria API.
* Can be verbose for simple queries.
* Not ideal for very complex joins or aggregations—QueryDSL may be better in those cases.

**Real-World Use Cases**

* Advanced search filters (e.g., e-commerce product filters)
* Admin dashboards with multiple optional filters
* REST APIs with dynamic query parameters

Complete example is given below.

@Repository

**public** **interface** EmpRepository **extends** CrudRepository<Employee, Long>, JpaSpecificationExecutor<Employee> {

}

Create a Specification Class

**public** **class** EmpSpecifications {

**public** **static** Specification<Employee> hasNameLike(String namePattern) {

**return** (root, query, criteriaBuilder)

-> namePattern == **null** ? **null** : criteriaBuilder.like(root.get("name"), "%" + namePattern + "%");

}

**public** **static** Specification<Employee> isWithinAgeRange(Integer minAge, Integer maxAge) {

**return** (root, query, criteriaBuilder)

-> criteriaBuilder.between(root.get("age"), minAge, maxAge);

}

**public** **static** Specification<Employee> hasStatus(String status) {

**return** (root, query, criteriaBuilder)

-> status == **null** ? **null** : criteriaBuilder.equal(root.get("status"), status);

}

}

In the implementation class like AutoRun, use it like this.

@Autowired

**private** EmpRepository empRepository;

Implementation is given below.

**public** **void** showEmpsBasedOnSpecification() {

List<Employee> emps = empRepository.findAll(EmpSpecifications.*hasNameLike*("Mishra"));

List<Employee> emps = empRepository.findAll(EmpSpecifications.*hasNameLike*("Mishra")

.and(EmpSpecifications.*isWithinAgeRange*(30, 40)));

emps.forEach(emp -> System.***out***.println("Emp: " + emp));

System.***out***.println("Total Count : " + emps.size());

// Combine Specifications Dynamically, use it in Service layer

Specification<Employee> spec = Specification // Specification.where() is deprecated

.~~where~~(EmpSpecifications.*hasNameLike*("Mishra")

.and(EmpSpecifications.*isWithinAgeRange*(30, 40))

.and(EmpSpecifications.*hasStatus*("Active")));

// Combine Specifications Dynamically, use it in Service layer **without using where()**

Specification<Employee> spec = EmpSpecifications.*hasNameLike*("Mishra")

.and(EmpSpecifications.*isWithinAgeRange*(30, 40))

// .and(EmpSpecifications.hasStatus("Active"));

.and(EmpSpecifications.*hasStatus*(**null**)); // This also works

List<Employee> emps = empRepository.findAll(spec);

emps.forEach(emp -> System.***out***.println("Emp: " + emp));

System.***out***.println("Total Count : " + emps.size());

}

**How to use JOIN with CriteriaBuilder in Spring JPA**

Example is given below.

First create 3 entities like Person, Department and Project.

@Entity @Table(name = "person")

@Getter @Setter @ToString

**public** **class** Person {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long id;

**private** String name;

@ManyToOne

**private** Department department;

@ManyToMany

@JoinTable(name = "person\_project", // Explicitly defines the join table name

joinColumns = @JoinColumn(name = "person\_id"), // Column in the join table referencing Person's ID

inverseJoinColumns = @JoinColumn(name = "project\_id") // Column in the join table referencing Project's ID

)

**private** Set<Project> projects;

}

@Entity @Table(name = "department")

@Getter @Setter

**public** **class** Department {

@Id @GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long id;

**private** String name;

@OneToMany(mappedBy = "department")

**private** List<Person> persons;

}

@Entity

@Table(name = "project")

@Getter @Setter

**public** **class** Project {

@Id @GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** Long id;

**private** String projectName;

@ManyToMany(mappedBy = "projects")

**private** Set<Person> persons;

}

Create the corresponding following tables in database.

A blue rectangle with black text

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

How to join 3 tables based on certain conditions.

Use the following code in the service layer.

**public** List<Person> findEmployeesByDepartmentAndProject() {

CriteriaBuilder cb = entityMgr.getCriteriaBuilder();

CriteriaQuery<Person> query = cb.createQuery(Person.**class**);

Root<Person> personRoot = query.from(Person.**class**);

// Join with Department

Join<Person, Department> departmentJoin = personRoot.join("department");

// Join with Project

Join<Person, Project> projectJoin = personRoot.join("projects");

// Conditions

Predicate departmentPredicate = cb.equal(departmentJoin.get("name"), "IT");

Predicate projectPredicate = cb.equal(projectJoin.get("projectName"), "Apollo");

**query.select(personRoot).where(cb.and(departmentPredicate, projectPredicate));**

// .distinct(true); // Because of potential duplicates in many-to-many

**return** entityMgr.createQuery(query).getResultList();

}

Notes and Leraning:

@JoinTable: Explain