`CriteriaBuilder` in Spring Boot

In Spring Boot, `CriteriaBuilder` is typically used for building dynamic and type-safe queries in the context of the **Java Persistence API (JPA)**. Spring Boot integrates well with JPA via `Spring Data JPA`, which is a high-level framework to handle database operations. `CriteriaBuilder` is part of JPA's Criteria API, which allows you to write dynamic queries without resorting to JPQL (Java Persistence Query Language) or SQL, ensuring better performance, maintainability, and type safety.

How `CriteriaBuilder` Works in Spring Boot

1. **Context**: In Spring Boot, you use JPA for ORM (Object Relational Mapping) to manage database entities. The `CriteriaBuilder` allows you to programmatically build queries instead of writing static JPQL.
2. **Integration**: You typically inject the `EntityManager` into a Spring service or repository to access `CriteriaBuilder`, and then use it to construct queries.
3. **Type Safety**: Since the queries are built using Java objects and properties, you avoid runtime errors due to incorrect field names or data types, as these are checked during compile time.

Steps Involved in Using `CriteriaBuilder` in Spring Boot

1. **Inject `EntityManager`**: The `EntityManager` is used to create `CriteriaBuilder` and `CriteriaQuery`. In Spring Boot, you inject it into your service or repository.
2. **Create `CriteriaQuery`**: The `CriteriaQuery` object defines the structure of your query, specifying the entity class to query and its properties.
3. **Define `Predicate` (Conditions)**: `Predicate` represents conditions that are applied to the query (e.g., equality, comparisons, or complex logical expressions).
4. **Execute the Query**: Once the query is defined, execute it using `EntityManager.createQuery()` to get the results.

Use Cases for `CriteriaBuilder` in Spring Boot

1. **Dynamic Queries**: When you don't know the full structure of your query at compile-time (e.g., the query structure depends on user input or filtering criteria).
2. **Complex Queries**: For queries involving joins, groupings, and aggregations.
3. **Avoiding String-Based Queries**: When you want compile-time checks for correctness, especially in large systems where queries might change over time.
4. **Filtering and Sorting**: Creating dynamic filter conditions and sorting without manually writing out JPQL or SQL queries.

Example Scenario

Let’s consider a simple example where we have an entity `Employee` with fields like `id`, `name`, `age`, and `department`. We want to fetch employees who are older than 30 years and belong to the "HR" department.

1. Define the `Employee` Entity

java
Copy code
```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Employee {
    @Id
    private Long id;
    private String name;
    private int age;
    private String department;

    // Getters and setters
}
```

2. Create a Service to Use `CriteriaBuilder`

In a service, you can inject the `EntityManager` and use `CriteriaBuilder` to build the dynamic query.

java
Copy code
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import javax.persistence.EntityManager;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Predicate;
import javax.persistence.criteria.Root;
import javax.persistence.TypedQuery;
import java.util.List;

@Service
public class EmployeeService {

    @Autowired
    private EntityManager entityManager;

    public List<Employee> getEmployeesAboveAgeAndInDepartment(int
age, String department) {
        // Create CriteriaBuilder instance
        CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();

        // Create CriteriaQuery for Employee entity
        CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);

        // Define root for the query
        Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);

        // Define predicates (conditions)
        Predicate agePredicate =
criteriaBuilder.greaterThan(employeeRoot.get("age"), age);
        Predicate departmentPredicate =
criteriaBuilder.equal(employeeRoot.get("department"), department);

        // Combine predicates using AND
        criteriaQuery.where(criteriaBuilder.and(agePredicate,
departmentPredicate));

        // Execute query
        TypedQuery<Employee> query =
entityManager.createQuery(criteriaQuery);
        return query.getResultList();
    }
}
```

3. Use the Service in a Controller

You can now create a Spring MVC controller that uses this service to fetch and return employee data.

java
Copy code
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/employees")
    public List<Employee> getEmployees(@RequestParam int age,
@RequestParam String department) {
        return
employeeService.getEmployeesAboveAgeAndInDepartment(age,
department);
    }
}
```

4. Run the Application

Now, when you run your Spring Boot application and make a GET request to `/employees?age=30&department=HR`, it will return the list of employees who are older than 30 years and work in the HR department.

Advanced Use Cases for `CriteriaBuilder`

Joins: When you need to join multiple tables (entities), you can use `CriteriaBuilder` to define the joins.
java
Copy code
```
Join<Employee, Department> departmentJoin =
employeeRoot.join("department", JoinType.INNER);
```

1.

Aggregations: For grouping results and performing aggregations like `count`, `avg`, `max`, `min`, etc.
java
Copy code
```
CriteriaQuery<Object[]> criteriaQuery =
criteriaBuilder.createQuery(Object[].class);
```

```
criteriaQuery.multiselect(employeeRoot.get("department"),
criteriaBuilder.count(employeeRoot));
criteriaQuery.groupBy(employeeRoot.get("department"));
```

2.

Sorting: Sorting results using `CriteriaBuilder` is easy:
java
Copy code

```
criteriaQuery.orderBy(criteriaBuilder.asc(employeeRoot.get("name")))
;
```

3.

Dynamic Filters: Build filters dynamically based on user input. For instance, if users choose multiple filter criteria, you can dynamically add them to the query.
java
Copy code

```
Predicate dynamicPredicate = criteriaBuilder.conjunction();
if (age != null) {
    dynamicPredicate = criteriaBuilder.and(dynamicPredicate,
criteriaBuilder.equal(employeeRoot.get("age"), age));
}
if (department != null) {
    dynamicPredicate = criteriaBuilder.and(dynamicPredicate,
criteriaBuilder.equal(employeeRoot.get("department"), department));
}
criteriaQuery.where(dynamicPredicate);
```

4.

Pagination: If you want to paginate the results, you can use the `setFirstResult` and `setMaxResults` methods to apply pagination to the query.
java
Copy code

```
query.setFirstResult(pageNumber * pageSize);
query.setMaxResults(pageSize);
```

5.

Advantages of Using `CriteriaBuilder` in Spring Boot

1. **Type Safety**: Reduces runtime errors because the fields and properties are checked at compile-time.
2. **Refactorability**: Since it uses Java objects and not raw strings, renaming fields or making changes to entity classes will be reflected automatically.

3. **Maintainability**: Helps keep queries clean and readable, especially as they become more complex (with joins, groupings, etc.).
4. **Dynamic Queries**: It's much easier to create dynamic queries based on various conditions that are evaluated at runtime, such as user inputs or system configurations.

Disadvantages

- **Verbosity**: The Criteria API can sometimes be more verbose than using JPQL or SQL directly.
- **Learning Curve**: Understanding the Criteria API can be a bit challenging for developers who are accustomed to writing traditional JPQL or SQL queries.

Writing `CriteriaBuilder` queries in JPA allows you to construct dynamic, type-safe queries without directly writing JPQL or SQL. Below are examples of how to use `CriteriaBuilder` for various types of queries.

Key Components of `CriteriaBuilder`:

1. **`CriteriaBuilder`**: Used to create conditions, predicates, and the query itself.
2. **`CriteriaQuery`**: Defines the structure of the query (what entity to query).
3. **`Root`**: Represents the root entity of the query.
4. **`Predicate`**: Represents the conditions (filters) applied to the query.
5. **`Join`**: Allows joining other entities to the root entity.
6. **`TypedQuery`**: Used to execute the query and get results.

Let's walk through different examples to understand how to use `CriteriaBuilder` effectively.

Example 1: Simple Query with `CriteriaBuilder`

Scenario:

We have an `Employee` entity with fields like `id`, `name`, `age`, and `department`. We want to fetch employees whose age is greater than 30.

Step-by-Step Explanation:

1. **Create the Entity Class**

java
Copy code
```java
@Entity
public class Employee {
    @Id
    private Long id;
    private String name;
    private int age;
    private String department;

    // Getters and setters
}
```

2. **Build the CriteriaQuery**

java
Copy code

```
import javax.persistence.EntityManager;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Predicate;
import javax.persistence.criteria.Root;
import javax.persistence.TypedQuery;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class EmployeeService {

    @Autowired
    private EntityManager entityManager;

    public List<Employee> getEmployeesAboveAge(int age) {
        // Create CriteriaBuilder instance
        CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();

        // Create a CriteriaQuery for Employee entity
        CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);

        // Define the root of the query (i.e., the Employee entity)
        Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);

        // Create a Predicate (filter) for the age condition
        Predicate agePredicate =
criteriaBuilder.greaterThan(employeeRoot.get("age"), age);

        // Apply the predicate to the query
        criteriaQuery.where(agePredicate);

        // Execute the query
        TypedQuery<Employee> query =
entityManager.createQuery(criteriaQuery);
        return query.getResultList();
    }
}
```

How it works:

- **CriteriaBuilder** is used to create the query.
- **CriteriaQuery** is used to define the structure of the query.
- **Root** represents the `Employee` entity.
- **Predicate** is used to filter the employees by age.
- The query is executed, and a list of employees is returned.

Example 2: Query with Multiple Conditions

Scenario:

Now, let's say we want to fetch employees who are older than 30 and belong to the "HR" department.

Modified Code:
java
Copy code

```
public List<Employee> getEmployeesAboveAgeAndInDepartment(int age,
String department) {
    CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);
    Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);

    // Predicate for age
    Predicate agePredicate =
criteriaBuilder.greaterThan(employeeRoot.get("age"), age);

    // Predicate for department
    Predicate departmentPredicate =
criteriaBuilder.equal(employeeRoot.get("department"), department);

    // Combine both predicates using AND
    criteriaQuery.where(criteriaBuilder.and(agePredicate,
departmentPredicate));

    // Execute the query
    TypedQuery<Employee> query =
entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

How it works:

- **`criteriaBuilder.and()`** is used to combine multiple conditions with an AND operator.
- The query will fetch employees who satisfy both the `age` and `department` conditions.

Example 3: Query with Sorting

Scenario:

We want to get employees older than 30 and sort them by their name.

Modified Code:
java
Copy code
```
public List<Employee> getEmployeesAboveAgeSortedByName(int age) {
    CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);
    Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);
```

```
    // Predicate for age
    Predicate agePredicate =
criteriaBuilder.greaterThan(employeeRoot.get("age"), age);
    criteriaQuery.where(agePredicate);

    // Sorting by name

criteriaQuery.orderBy(criteriaBuilder.asc(employeeRoot.get("name")))
;

    // Execute the query
    TypedQuery<Employee> query =
entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

How it works:

- **`criteriaQuery.orderBy()`** is used to specify the sorting criteria.
- The query will first filter by age and then sort the results in ascending order based on the `name` field.

Example 4: Query with Joins

Scenario:

Now, let's say there is a `Department` entity, and each `Employee` is associated with a `Department`. We want to fetch employees whose department is "HR".

Step 1: Define the `Department` Entity

java
Copy code

```
@Entity
public class Department {
    @Id
    private Long id;
    private String name;

    // Getters and setters
}
```

Step 2: Build the Query with a Join
java
Copy code
```java
public List<Employee> getEmployeesInDepartment(String
departmentName) {
    CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);
    Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);

    // Join with the Department entity
    Join<Employee, Department> departmentJoin =
employeeRoot.join("department");

    // Predicate for department name
    Predicate departmentPredicate =
criteriaBuilder.equal(departmentJoin.get("name"), departmentName);
    criteriaQuery.where(departmentPredicate);

    // Execute the query
    TypedQuery<Employee> query =
entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

How it works:

- **`employeeRoot.join("department")`** creates an inner join between `Employee` and `Department`.
- The query filters employees who belong to a specific department (`"HR"` in this case).
- You can also use `JoinType.LEFT` to specify a left join.

Example 5: Query with Grouping and Aggregation

Scenario:

Let’s say we want to find out how many employees there are in each department.

Code:

java
Copy code
```
public List<Object[]> getEmployeeCountByDepartment() {
    CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    CriteriaQuery<Object[]> criteriaQuery =
criteriaBuilder.createQuery(Object[].class);
    Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);

    // Group by department
    criteriaQuery.groupBy(employeeRoot.get("department"));

    // Aggregate function: count the number of employees in each
department
    criteriaQuery.multiselect(employeeRoot.get("department"),
criteriaBuilder.count(employeeRoot));

    // Execute the query
    TypedQuery<Object[]> query =
entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

How it works:

- **`criteriaQuery.groupBy()`** groups the employees by department.
- **`criteriaBuilder.count()`** counts the number of employees in each department.
- The result is an array of objects where the first element is the department name and the second is the count of employees in that department.

Example 6: Dynamic Query with Multiple Conditions

Scenario:

We want to build a dynamic query based on user input. The user can filter employees by age, department, and name.

Code:
java
Copy code

```
public List<Employee> getEmployeesWithDynamicFilters(Integer age,
String department, String name) {
    CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);
    Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);

    // Start with an empty condition (true)
    Predicate predicate = criteriaBuilder.conjunction();

    // Add conditions dynamically based on the input
    if (age != null) {
        predicate = criteriaBuilder.and(predicate,
criteriaBuilder.greaterThan(employeeRoot.get("age"), age));
    }
    if (department != null) {
        predicate = criteriaBuilder.and(predicate,
criteriaBuilder.equal(employeeRoot.get("department"), department));
    }
    if (name != null) {
        predicate = criteriaBuilder.and(predicate,
criteriaBuilder.like(employeeRoot.get("name"), "%" + name + "%"));
    }

    // Apply the conditions
    criteriaQuery.where(predicate);

    // Execute the query
    TypedQuery<Employee> query =
entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

How it works:

- **`criteriaBuilder.conjunction()`** starts with a condition that is always true.
- **`criteriaBuilder.and()`** adds conditions based on user input.
- The query is dynamically constructed and filtered based on available inputs (`age`, `department`, `name`).

CriteriaBuilder with Joins

In JPA, `CriteriaBuilder` can be used to create queries with joins, which is helpful when you want to retrieve data from multiple related entities. Using joins, you can combine data from different tables (or entities) based on their relationships.

Here are a few examples to show how to use `CriteriaBuilder` with joins.

Scenario 1: Simple Inner Join between Two Entities

Let's say we have two entities: `Employee` and `Department`. Each `Employee` belongs to a `Department`, and we want to fetch employees who work in the "HR" department.

1. Define the Entity Classes

java
Copy code
```java
@Entity
public class Department {
    @Id
    private Long id;
    private String name;

    // Getters and setters
}

@Entity
public class Employee {
    @Id
    private Long id;
    private String name;
    private int age;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    // Getters and setters
}
```

2. Build the Query Using `CriteriaBuilder` with Join

java

Copy code
```
import javax.persistence.criteria.Join;

public List<Employee> getEmployeesInDepartment(String
departmentName) {
    CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);
    Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);

    // Perform an inner join with the Department entity
    Join<Employee, Department> departmentJoin =
employeeRoot.join("department");

    // Define the condition for the department name
    Predicate departmentPredicate =
criteriaBuilder.equal(departmentJoin.get("name"), departmentName);
    criteriaQuery.where(departmentPredicate);

    // Execute the query
    TypedQuery<Employee> query =
entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

Explanation:

- **`employeeRoot.join("department")`** creates an inner join with the `Department` entity. The `"department"` is the name of the relationship field in the `Employee` entity.
- The condition **`criteriaBuilder.equal(departmentJoin.get("name"), departmentName)`** filters the employees based on the department name.
- This query will return all employees who belong to the "HR" department.

Scenario 2: Left Join between Two Entities

In some cases, you may want to fetch all employees, including those who might not have a department assigned. This would require a **left join** to include even those employees who have no matching department.

Code Example:
java
Copy code
```
public List<Employee> getEmployeesWithOptionalDepartments() {
    CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);
    Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);

    // Perform a left join with the Department entity
    Join<Employee, Department> departmentJoin =
employeeRoot.join("department", JoinType.LEFT);

    // No specific filter, just fetch all employees (with or without
a department)
    criteriaQuery.select(employeeRoot);

    // Execute the query
    TypedQuery<Employee> query =
entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

Explanation:

- **`JoinType.LEFT`** specifies that we are performing a left join with the `Department` entity.
- The query fetches all employees, even those who don't belong to a department (i.e., `null` department).

Scenario 3: Multiple Joins (Inner Join + Left Join)

If you have multiple relationships, you might want to use multiple joins in a single query. For example, let's say an `Employee` is related to both a `Department` and a `Manager`, and you want to fetch employees along with their department and manager (if present).

Code Example:
java
Copy code

```
@Entity
public class Manager {
    @Id
    private Long id;
    private String name;

    // Getters and setters
}

@Entity
public class Employee {
    @Id
    private Long id;
    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    @ManyToOne
    @JoinColumn(name = "manager_id")
    private Manager manager;

    // Getters and setters
}
```

Build the Query:

java
Copy code

```
public List<Employee> getEmployeesWithDepartmentAndManager(String
departmentName) {
    CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);
    Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);

    // Join the Department entity
    Join<Employee, Department> departmentJoin =
employeeRoot.join("department");
```

```
    // Left join the Manager entity
    Join<Employee, Manager> managerJoin =
employeeRoot.join("manager", JoinType.LEFT);

    // Define conditions for department name
    Predicate departmentPredicate =
criteriaBuilder.equal(departmentJoin.get("name"), departmentName);
    criteriaQuery.where(departmentPredicate);

    // Execute the query
    TypedQuery<Employee> query =
entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

Explanation:

- **`employeeRoot.join("department")`** creates an inner join between `Employee` and `Department`.
- **`employeeRoot.join("manager", JoinType.LEFT)`** creates a left join between `Employee` and `Manager`.
- The query fetches employees along with their department and manager (if available).

Scenario 4: Join with Aggregation

If you need to aggregate data (e.g., count the number of employees in each department), you can use `CriteriaBuilder` along with `join` to achieve this.

Code Example:
java
Copy code
```
public List<Object[]> getEmployeeCountByDepartment() {
    CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    CriteriaQuery<Object[]> criteriaQuery =
criteriaBuilder.createQuery(Object[].class);
    Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);

    // Perform a join with the Department entity
    Join<Employee, Department> departmentJoin =
employeeRoot.join("department");

    // Group by department
    criteriaQuery.groupBy(departmentJoin.get("name"));

    // Aggregate: Count the number of employees in each department
    criteriaQuery.multiselect(departmentJoin.get("name"),
criteriaBuilder.count(employeeRoot));

    // Execute the query
    TypedQuery<Object[]> query =
entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

Explanation:

- **`criteriaQuery.groupBy(departmentJoin.get("name"))`** groups the results by the department name.
- **`criteriaBuilder.count(employeeRoot)`** counts the number of employees in each department.
- The query returns an array of objects where the first element is the department name and the second is the employee count.

Scenario 5: Self Join (Join an Entity with Itself)

In some cases, you may want to join an entity with itself. For example, let's say you want to find employees who share the same manager.

Code Example:
java
Copy code

```
public List<Employee> getEmployeesWithSameManager(Long managerId) {
    CriteriaBuilder criteriaBuilder =
entityManager.getCriteriaBuilder();
    CriteriaQuery<Employee> criteriaQuery =
criteriaBuilder.createQuery(Employee.class);
    Root<Employee> employeeRoot =
criteriaQuery.from(Employee.class);
```

```
    // Self-join on the Employee entity (joining employees with the
same manager)
    Join<Employee, Employee> managerJoin =
employeeRoot.join("manager");

    // Define condition for the manager
    Predicate managerPredicate =
criteriaBuilder.equal(managerJoin.get("id"), managerId);
    criteriaQuery.where(managerPredicate);

    // Execute the query
    TypedQuery<Employee> query =
entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```

Explanation:

- **`employeeRoot.join("manager")`** joins the `Employee` entity with itself by joining the `manager` field.
- **`criteriaBuilder.equal(managerJoin.get("id"), managerId)`** filters the employees by their manager's ID.
- The query fetches all employees who have the same manager.