C ### Criteria API in JPA

The \*\*Criteria API\*\* is a powerful and type-safe way to construct queries programmatically in JPA. Unlike JPQL (which is string-based), the Criteria API uses Java objects to build queries dynamically. This allows for more flexible, type-safe query creation, particularly in scenarios where queries need to be constructed at runtime.

---

### Why Use the Criteria API?

1. \*\*Type Safety\*\*: Since it operates on entity classes and attributes, the Criteria API offers compile-time checking and prevents errors like misspelled field names that can occur with JPQL.

2. \*\*Dynamic Queries\*\*: It is particularly useful for building complex queries dynamically, such as when filtering criteria are determined at runtime.

3. \*\*Database Agnostic\*\*: Just like JPQL, the Criteria API abstracts the underlying SQL syntax, making your queries portable across different databases.

---

### Components of Criteria API

1. \*\*`CriteriaBuilder`\*\*: This is the starting point for creating Criteria API queries. It is responsible for constructing different types of query objects.

2. \*\*`CriteriaQuery<T>`\*\*: Represents a query that returns results of type `T`.

3. \*\*`Root<T>`\*\*: Represents the entity on which the query is built.

4. \*\*`Predicate`\*\*: Represents a filter condition (like a `WHERE` clause in SQL).

5. \*\*`Path<T>`\*\*: Represents the attributes of entities used in the query.

6. \*\*`TypedQuery<T>`\*\*: Used to execute the query and return results.

---

### Basic Steps to Build a Query with Criteria API

1. \*\*Obtain an instance of `EntityManager`\*\*:

The `EntityManager` is the entry point to JPA and provides the methods needed to work with the Criteria API.

2. \*\*Get a `CriteriaBuilder`\*\*:

The `CriteriaBuilder` is used to construct the query and create expressions and predicates.

3. \*\*Create a `CriteriaQuery<T>`\*\*:

The `CriteriaQuery` defines the type of the query result (for example, `User.class`).

4. \*\*Define the Root\*\*:

The `Root<T>` represents the entity in the query (like the `FROM` clause in SQL).

5. \*\*Build Predicates for Filters\*\*:

Predicates represent the conditions in the `WHERE` clause.

6. \*\*Execute the Query\*\*:

Finally, you execute the query using `TypedQuery<T>`.

---

### Example: Simple Criteria Query

Assume we have a `User` entity, and we want to retrieve all users where `age > 18`.

```java

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private int age;

// Getters and Setters

}

```

Now, we will use the Criteria API to fetch all users who are older than 18.

```java

public List<User> getUsersAboveAge(int minAge) {

// Step 1: Obtain an EntityManager

EntityManager entityManager = ...; // Inject or retrieve from context

// Step 2: Create CriteriaBuilder

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

// Step 3: Create CriteriaQuery

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

// Step 4: Define the Root (FROM clause)

Root<User> user = query.from(User.class);

// Step 5: Build the Predicate (WHERE clause)

Predicate ageGreaterThan = cb.greaterThan(user.get("age"), minAge);

// Step 6: Build the query with the WHERE clause

query.select(user).where(ageGreaterThan);

// Step 7: Execute the query

TypedQuery<User> typedQuery = entityManager.createQuery(query);

return typedQuery.getResultList();

}

```

- \*\*CriteriaBuilder\*\*: Used to construct different query elements.

- \*\*CriteriaQuery\*\*: Represents the query that will be executed.

- \*\*Root\*\*: Represents the entity class from which data is selected (i.e., the `FROM` part).

- \*\*Predicate\*\*: Represents the condition (i.e., `WHERE` clause).

---

### Using Multiple Conditions

You can add multiple conditions to a query by combining predicates with the `and()` or `or()` methods provided by the `CriteriaBuilder`.

#### Example: Fetch users who are older than 18 and whose names start with "A".

```java

public List<User> getUsersByAgeAndName(int minAge, String nameStart) {

EntityManager entityManager = ...;

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

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

Root<User> user = query.from(User.class);

// Define multiple predicates

Predicate ageGreaterThan = cb.greaterThan(user.get("age"), minAge);

Predicate nameStartsWith = cb.like(user.get("name"), nameStart + "%");

// Combine predicates with AND

query.select(user).where(cb.and(ageGreaterThan, nameStartsWith));

TypedQuery<User> typedQuery = entityManager.createQuery(query);

return typedQuery.getResultList();

}

```

In this example, we combine two predicates (`age > 18` and `name LIKE 'A%'`) using `cb.and()`.

---

### Sorting with Criteria API

You can easily add sorting to your queries using the `CriteriaBuilder`'s `asc()` and `desc()` methods.

#### Example: Fetch all users and sort them by age in descending order.

```java

public List<User> getAllUsersSortedByAge() {

EntityManager entityManager = ...;

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

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

Root<User> user = query.from(User.class);

// Add sorting by age descending

query.select(user).orderBy(cb.desc(user.get("age")));

TypedQuery<User> typedQuery = entityManager.createQuery(query);

return typedQuery.getResultList();

}

```

---

### Pagination with Criteria API

Pagination can be implemented by setting the starting position and the maximum number of results using `TypedQuery`.

#### Example: Fetch paginated results for users.

```java

public List<User> getUsersPaginated(int pageNumber, int pageSize) {

EntityManager entityManager = ...;

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

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

Root<User> user = query.from(User.class);

query.select(user);

TypedQuery<User> typedQuery = entityManager.createQuery(query);

typedQuery.setFirstResult(pageNumber \* pageSize); // Set starting point

typedQuery.setMaxResults(pageSize); // Set max results (page size)

return typedQuery.getResultList();

}

```

---

### Joins in Criteria API

You can perform joins between entities using the `join()` method.

#### Example: Fetch users and their associated orders (assuming a `User` entity has a `List<Order>` relationship).

```java

public List<User> getUsersWithOrders() {

EntityManager entityManager = ...;

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

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

Root<User> user = query.from(User.class);

// Perform a join on the "orders" relationship

user.join("orders");

query.select(user);

TypedQuery<User> typedQuery = entityManager.createQuery(query);

return typedQuery.getResultList();

}

```

In this case, we assume the `User` entity has a collection of `Order` objects mapped, and we join the two entities.

---

### Aggregations in Criteria API

You can use aggregation functions like `COUNT`, `SUM`, `AVG`, etc., via the `CriteriaBuilder`.

#### Example: Count the number of users.

```java

public long countUsers() {

EntityManager entityManager = ...;

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

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

Root<User> user = query.from(User.class);

query.select(cb.count(user));

TypedQuery<Long> typedQuery = entityManager.createQuery(query);

return typedQuery.getSingleResult();

}

```

---

### Conclusion

The \*\*Criteria API\*\* is a robust and flexible tool in JPA for constructing type-safe, dynamic queries. It is ideal for scenarios where queries need to be built at runtime based on user input or other conditions. While the syntax can be more verbose than JPQL, the benefits of type safety, flexibility, and compile-time checks make it a valuable choice for complex query requirements.