**What are** **different ways you can work with native queries in spring Data JPA?**

In **Spring Data JPA**, there are several ways to work with **native queries**. Native queries allow you to use raw SQL directly, which is particularly useful for complex queries that might be difficult to express using JPQL or when interacting with database-specific features.

Here are the different ways to work with native queries in Spring Data JPA:

**1. Using @Query Annotation with Native Query**

You can define native queries directly in repository interfaces using the @Query annotation and set the nativeQuery attribute to true.

**Example:**

public interface ProductRepository extends JpaRepository<Product, Long> {

@Query(value = "SELECT \* FROM products WHERE category = ?1", nativeQuery = true)

List<Product> findByCategory(String category);

}

 **Pros:** Easy to define and manage native SQL queries directly in the repository layer.

 **Cons:** Can become cumbersome for complex queries. SQL is hardcoded and not reusable across different methods.

**2. Using EntityManager and createNativeQuery()**

You can use the EntityManager directly to create and execute native queries. This is the most flexible method and allows for complex queries with dynamic parameters.

**Example:**

@Autowired

private EntityManager entityManager;

public List<Product> findProductsByCategory(String category) {

Query query = entityManager.createNativeQuery("SELECT \* FROM products WHERE category = ?", Product.class);

query.setParameter(1, category);

return query.getResultList();

}

Here's how it works:

1. **Query Creation:** The query is constructed using entityManager.createNativeQuery(), which doesn't actually execute the query yet but prepares it.
2. **Setting Parameters:** You set parameters using queryManager.setParameter(), which also doesn't execute the query but configures it.
3. **Query Execution:** When you call queryManager.getResultList(), the query is fired against the database, retrieving the result set.

 **Pros:** Full flexibility, allows mapping to entity classes or custom DTOs, and provides more control over the execution.

 **Cons:** More verbose and requires manual handling of result sets and types.

**3) Using Native Queries with Pagination**

Spring Data JPA supports paginating native queries. You can use native queries with pagination by extending PagingAndSortingRepository or using the Pageable parameter.

**Example:**

public interface ProductRepository extends JpaRepository<Product, Long> {

@Query(value = "SELECT \* FROM products WHERE category = ?1", nativeQuery = true)

Page<Product> findByCategory(String category, Pageable pageable);

}

 **Pros:** Easy pagination handling of large result sets.

 **Cons:** Performance can degrade if pagination isn't optimized.

**Using EntityManager with createNativeQuery() (for Complex Queries)**

For more complex scenarios where you need fine-grained control over the native query, or the query has dynamic parts (e.g., conditional clauses, complex joins), using the EntityManager's createNativeQuery() method gives you flexibility. This also works well when mapping custom result sets to DTOs.

**Ideal Use Case:**

* Complex queries with dynamic conditions, pagination, or advanced SQL features.
* Custom result mapping to DTOs.