

**@Repository**

public interface ServiceSetupRepository

extends JpaRepository<ServiceSetup, String>, QuerydslPredicateExecutor<ServiceSetup> {

}

**Is writing @Repository mandatory here?**

No, the @Repository annotation is **not strictly mandatory** for Spring Data JPA repositories like ServiceSetupRepository. The annotation is **optional** because Spring automatically detects repositories that extend JpaRepository or other Spring Data interfaces and treats them as repositories.

This is because **Spring automatically recognizes** interfaces that extend JpaRepository and manages them as repository beans.

**However, here are a few points to consider:**

* **Automatic Detection**: Spring scans for components based on the presence of specific annotations. While @Repository is not needed for **exception translation**, using it can help explicitly mark the class as a repository.
* **Custom Repositories**: If you create custom implementations for your repository interfaces, it’s a good idea to use the @Repository annotation on those custom classes to ensure they are recognized as Spring beans.
* **Readability and Maintenance**: Even if it works without the annotation, adding @Repository can improve the readability of your code, making it clear to others (or yourself in the future) that this interface is intended to be a repository.

**Custom Repositories in Spring Data**

Custom repositories in Spring Data refer to situations where you need to implement **custom business logic** or **query methods** that cannot be easily expressed using the predefined methods provided by JpaRepository or other Spring Data interfaces.

By default, Spring Data JPA provides repository interfaces like JpaRepository, CrudRepository, etc., offering basic CRUD operations and some query derivation mechanisms. However, there are cases where you might need **more complex queries**, **non-standard operations**, or **specific performance optimizations**, which may not be achievable through the built-in methods.

**In such cases, you can create a custom repository. Here's how custom repositories work:**

**1. Creating Custom Repository Interface**

You define a custom interface that declares the additional methods you need. This interface will act as an **extension** to the default repository.

public interface CustomServiceSetupRepository {

List<ServiceSetup> findCustomSetups(String param);

}

**2. Implementing Custom Repository Logic**

You provide the actual implementation of the custom methods by implementing the custom interface in a separate class. The implementation class must follow this naming convention:

* It should be named as <BaseRepositoryName>Impl.
* The implementation class should be annotated with @Repository.

@Repository

public class CustomServiceSetupRepositoryImpl implements CustomServiceSetupRepository {

@PersistenceContext

private EntityManager entityManager;

@Override

public List<ServiceSetup> findCustomSetups(String param) {

// Use JPA Criteria API, JPQL, or native SQL for custom queries

String query = "SELECT s FROM ServiceSetup s WHERE s.someField = :param";

return entityManager.createQuery(query, ServiceSetup.class)

.setParameter("param", param)

.getResultList();

}

}

**3. Extending the Custom Repository**

Now, extend your standard repository with both JpaRepository and your custom repository interface:

@Repository

public interface ServiceSetupRepository

extends JpaRepository<ServiceSetup, String>, CustomServiceSetupRepository {

}

This allows ServiceSetupRepository to inherit all the standard CRUD methods from JpaRepository while also using the custom methods from CustomServiceSetupRepository.

**Why Use Custom Repositories?**

* **Custom Query Logic**: You can implement queries that are more complex than what Spring Data JPA can generate through method naming conventions or @Query annotations.
* **Optimizations**: You can apply performance optimizations or specific database operations that are not part of the default repository methods.
* **Non-JPA Operations**: You can include logic that uses other mechanisms like JDBC, stored procedures, or non-relational databases while still keeping the repository structure in place.

**Example Use Case**

* If you need to perform a **bulk update** or **custom aggregation** that can't be easily expressed with JPQL or method-based query derivation.
* If you want to mix different technologies within the same repository, like performing some operations via JPA and others via native queries.

**Summary**

Custom repositories allow you to extend the standard repository behavior with your own implementations, offering **flexibility** for more complex scenarios while still leveraging Spring Data’s powerful abstractions.