Upsert, Composite Key and @NoRepositoryBean in Spring JPA – 2024

How to implement Upsert – Update or Insert

* Without using EntityManager

@Override

**public** **void** saveOrUpdate(Product product) {

Product productToSaveOrUpdate = **null**;

Product dbProd = prodRepo.findByProdCode(product.getProdCode());

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

dbProd.setName(product.getName());

dbProd.setProdDesc(product.getProdDesc());

productToSaveOrUpdate = dbProd;

} **else** {

productToSaveOrUpdate = product;

}

prodRepo.save(productToSaveOrUpdate);

}

* Using EntityManager

@Autowired

**private** EntityManager em;

// Using EntityManager

@Transactional

@Override

**public** **void** upsert(Product product) {

String jpql = "SELECT p FROM Product p where p.prodCode=:prodCode";

TypedQuery<Product> query = em.createQuery(jpql, Product.**class**);

query.setParameter("prodCode", product.getProdCode());

Product dbProd = query.getResultList().get(0);

**if** (dbProd.getId() == **null**) {

em.persist(product);

} **else** {

dbProd.setName(product.getName());

dbProd.setProdDesc(product.getProdDesc());

em.merge(dbProd);

}

}

**How to create Composite Key in Spring JPA**

1. **Using @IdClass**
2. **Using @Embeddable and @EmbeddedId**

In order to define the composite primary keys, we should follow some rules:

* The composite primary key class must be public.
* It must have a no-arg constructor.
* It must define the equals() and hashCode() methods.
* It must be Serializable.

**Using @IdClass**

Department Id and project code should be a combined key to identify the employee.

**public** **class** EmpCompositeId **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = -8893256279041337074L;

**private** String departmentId;

**private** String projectCode;

}

@Entity(name="Employee") @Table(name="employee")

@Getter @Setter @NoArgsConstructor

**@IdClass(EmpCompositeId.class)** 🡸 Mark it

**public** **class** Employee {

@Id 🡸 Mark it

**private** String departmentId;

@Id 🡸 Mark it

**private** String projectCode;

**private** String name;

**public** Employee(String name, String departmentId, String projectCode) {

**// constructor code**

}

}

**Testing code**

**private** **void** saveCompositeKeyEmployee() {

Employee emp = **new** Employee("John","D-001","P-002");

empRepo.save(emp);

}

**Using @Embeddable and @EmbeddedId**

@Embeddable 🡸 Mark it

@Getter @Setter @NoArgsConstructor

**public** **class** ItemId **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 8094664891673130307L;

**private** **int** itemId;

**private** String itemCode;

**public** ItemId(**int** itemId, String itemCode) {

**// Constructor**

}

}

@Entity(name="Item") @Table(name="item")

@Getter @Setter @NoArgsConstructor

**public** **class** Item {

**@EmbeddedId** 🡸 Mark it

**private** ItemId itemId;

**private** String name;

**public** Item(ItemId itemId, String name) {

**this**.itemId = itemId;

**this**.name = name;

}

}

**Testing code**

**private** **void** saveCompositeItem() {

ItemId itemId = **new** ItemId(33, "I-007");

Item item = **new** Item(itemId, "Samsung External Hard Disk");

itemRepo.save(item);

}

**What is the use of @NoRepositoryBean in Spring JPA**

* The annotation**@NoRepositoryBean** is used to avoid creating repository proxies for interfaces.
* Use @NoRepositoryBean in Spring Data when you have a custom repository base class that you don't want to be instantiated as a Spring bean. This typically happens when you have common methods that you want to share

across multiple repositories.

* It's an annotation to indicate that a repository interface should not be considered for automatic implementation

by Spring Data JPA.

* It's handy for creating base repository interfaces with common methods that other repositories can extend

without Spring trying to create an instance of the base interface itself.

Practical use Case: Imagine you have a set of common CRUD operations that multiple repositories need to use, but you don’t want to create a separate bean for those operations. With @NoRepositoryBean, you can create a base repository class with those common methods. Then, you extend that base class in your actual repository interfaces. This keeps your code DRY and helps with maintainability.

**Example is given below**

**@NoRepositoryBean**

public interface ExtendedRepository<T, ID extends Serializable> extends JpaRepository<T, ID> {

public List<T> findByAttributeContainsText(String attributeName, String text);

}

**This is necessary because otherwise, the default Spring behavior is to create an implementation for all subinterfaces of *Repository.***

Instead, it's intended to be used as a superclass for other repository interfaces, providing common functionality that can be inherited.

**@NoRepositoryBean**  
public interface BaseRepository<T, ID> extends JpaRepository <T, ID> {  
 <S extends T> S save(S entity);  
 Optional<T> findById(ID id);  
}

**@Repository**  
public interface CustomProductRepository extends BaseRepository<Product, Long> {  
}