**Spring Data**

Spring provides CrudRepository implementation class automatically at runtime. It contains methods such as save, findById, delete, count etc. If we want to add extra methods, we need to declare it in our interface. All the methods of CrudRepository are annotated with @Transactional in implementation class by default at runtime.

## CrudRepository Interface:

CrudRepository is an interface and extends Spring data Repository interface. CrudRepository provides generic CRUD operation on a repository for a specific type. It has generic methods for CRUD operation. To use CrudRepository we have to create our interface and extend CrudRepository. We need not to implement our interface, its implementation will be created automatically at runtime. Find some of CrudRepository methods.   
  
<S extends T> S save(S entity): Saves and updates the current entity and returns that entity.   
Optional<T> findById(ID primaryKey): Returns the entity for the given id.   
Iterable<T> findAll(): Returns all entities.   
long count(): Returns the count.   
void delete(T entity): Deletes the given entity.   
boolean existsById(ID primaryKey): Checks if the entity for the given id exists or not.   
  
CrudRepository has subinterface as PagingAndSortingRepository that provide additional methods to retrieve entities using the pagination and sorting abstraction.

## Configure JPA repositories

We need to configure JPA repository location so that Spring data can find our all defined repository interfaces that are extending CrudRepository. We need to configure package name of our repository classes. Here we will provide JPA repository configuration using JavaConfig as well as XML configuration.   
  
**a.** **JavaConfig**We need to annotate our JavaConfig class with @EnableJpaRepositories and pass repository package name. Find the code snippet.

|  |
| --- |
| # Local mysql database configuration.  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  spring.datasource.url=jdbc:<mysql://localhost:3306/sampledb>  spring.datasource.username=root  spring.datasource.password=    # Hibernate configuration.  spring.jpa.show-sql=true  spring.jpa.hibernate.ddl-auto=validate  spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect    # Ehcache configuration.  spring.cache.cache-names=productsCache  spring.cache.type=ehcache  spring.cache.ehcache.config=classpath:ehcache.xml |

@Configuration

@EnableJpaRepositories("com.concretepage.repository")

public class JPAConfig {

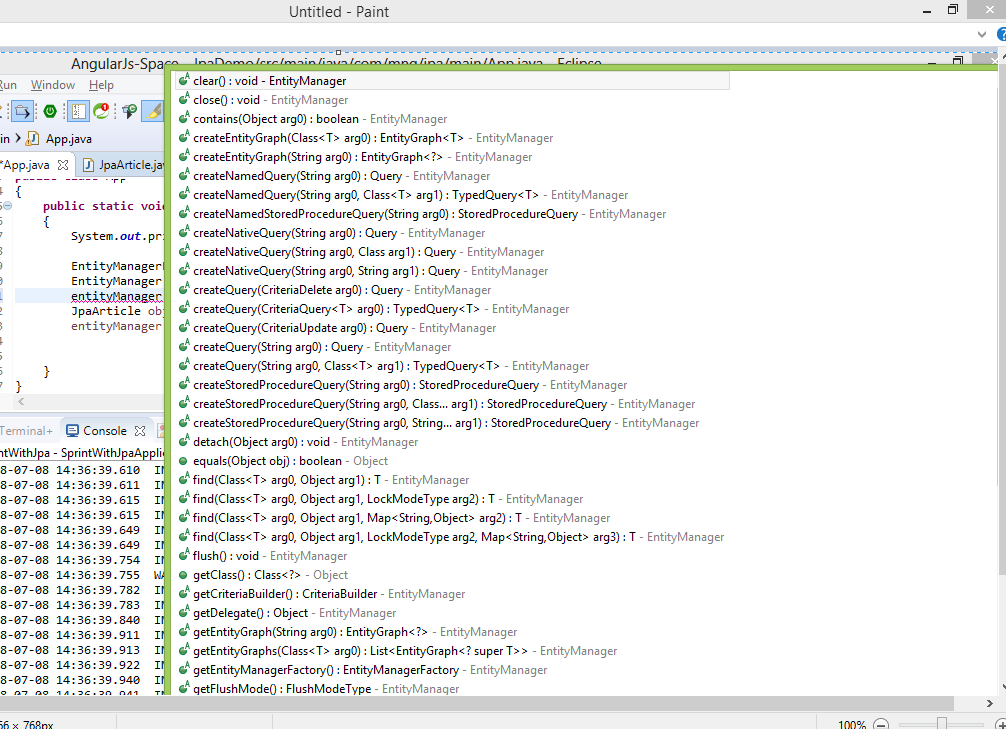
}

**Xml file Configuration:**

|  |
| --- |
| @Configuration  @EnableTransactionManagement  @EnableJpaRepositories("com.concretepage.repository")  @PropertySource("classpath:database.properties")  public class JPAConfig {  @Autowired  @Bean(name = "entityManagerFactory")  **public** LocalContainerEntityManagerFactoryBean getEntityManager() {  LocalContainerEntityManagerFactoryBean obj = **new** LocalContainerEntityManagerFactoryBean();  obj.setPersistenceUnitName("PERSISTENCE");  **return** obj;  }    @Bean(name = "transactionManager")  **public** JpaTransactionManager getJpaTransactionManager() {  JpaTransactionManager jpaTransactionManager = **new** JpaTransactionManager();  jpaTransactionManager.setEntityManagerFactory(getEntityManager().getObject());  **return** jpaTransactionManager;  }  **Xml Configuration:**  <persistence xmlns=*"http://xmlns.jcp.org/xml/ns/persistence"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/persistence*  *http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_1.xsd"*  version=*"2.1"*>  <persistence-unit name=*"PERSISTENCE"*>  <description> Hibernate JPA Configuration Example</description>  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>  <properties>  <property name=*"javax.persistence.jdbc.driver"* value=*"oracle.jdbc.driver.OracleDriver"* />  <property name=*"javax.persistence.jdbc.url"* value=*"jdbc:oracle:thin:@localhost:1521:xe"* />  <property name=*"javax.persistence.jdbc.user"* value=*"SYSTEM"* />  <property name=*"javax.persistence.jdbc.password"* value=*"nagendra123"* />  <property name=*"hibernate.show\_sql"* value=*"true"* />  <property name=*"hibernate.hbm2ddl.auto"* value=*"update"* />  </properties>  </persistence-unit>  </persistence> |

|  |
| --- |
| @NoRepositoryBean  **public** **interface** JpaRepository<T, ID> **extends** PagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> {  /\*  \* (non-Javadoc)  \* @see org.springframework.data.repository.CrudRepository#findAll()  \*/  List<T> findAll();  /\*  \* (non-Javadoc)  \* @see org.springframework.data.repository.PagingAndSortingRepository#findAll(org.springframework.data.domain.Sort)  \*/  List<T> findAll(Sort sort);  /\*  \* (non-Javadoc)  \* @see org.springframework.data.repository.CrudRepository#findAll(java.lang.Iterable)  \*/  List<T> findAllById(Iterable<ID> ids);  /\*  \* (non-Javadoc)  \* @see org.springframework.data.repository.CrudRepository#save(java.lang.Iterable)  \*/  <S **extends** T> List<S> saveAll(Iterable<S> entities);  /\*\*  \* Flushes all pending changes to the database.  \*/  **void** flush();  /\*\*  \* Saves an entity and flushes changes instantly.  \*  \* **@param** entity  \* **@return** the saved entity  \*/  <S **extends** T> S saveAndFlush(S entity);  /\*\*  \* Deletes the given entities in a batch which means it will create a single {@link Query}. Assume that we will clear  \* the {@link javax.persistence.EntityManager} after the call.  \*  \* **@param** entities  \*/  **void** deleteInBatch(Iterable<T> entities);  /\*\*  \* Deletes all entities in a batch call.  \*/  **void** deleteAllInBatch();  /\*\*  \* Returns a reference to the entity with the given identifier.  \*  \* **@param** id must not be {@literal null}.  \* **@return** a reference to the entity with the given identifier.  \* **@see** EntityManager#getReference(Class, Object)  \* **@throws** javax.persistence.EntityNotFoundException if no entity exists for given {@code id}.  \*/  T getOne(ID id);  /\*  \* (non-Javadoc)  \* @see org.springframework.data.repository.query.QueryByExampleExecutor#findAll(org.springframework.data.domain.Example)  \*/  @Override  <S **extends** T> List<S> findAll(Example<S> example);  /\*  \* (non-Javadoc)  \* @see org.springframework.data.repository.query.QueryByExampleExecutor#findAll(org.springframework.data.domain.Example, org.springframework.data.domain.Sort)  \*/  @Override  <S **extends** T> List<S> findAll(Example<S> example, Sort sort);  } |

**Entity Manager Methods details for custom implementation methods:**



|  |
| --- |
| import org.springframework.data.jpa.repository.JpaRepository;  import org.springframework.stereotype.Repository;  import com.mng.jpa.model.JpaArticle;  @Repository  public interface ArticleRepository extends JpaRepository<JpaArticle, Long>, JpaCustomArticleRepository{  }  import java.util.List;  import com.mng.jpa.model.JpaArticle;  public interface JpaCustomArticleRepository {  List<JpaArticle> findByName(String name);  }  **import** javax.persistence.EntityManager;  **import** javax.persistence.Query;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Repository;  **import** com.mng.jpa.model.JpaArticle;  @Repository  **public** **class** JpaCustomArticleRepositoryImpl **implements** JpaCustomArticleRepository {  @Autowired  **private** EntityManager entityManager;    @Override  **public** List<JpaArticle> findByName(String name) {  Query query = entityManager.createNativeQuery("select articleId, title, category from JpaArticle where title = ?");  query.setParameter(1, name);  **return** query.getResultList();  }  } |

**Entity Manager Operations:**

|  |
| --- |
| **public** **class** App  {  **public** **static** **void** main( String[] args )  {  System.***out***.println( "Hello World!" );    EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("PERSISTENCE");  EntityManager entityManager = factory.createEntityManager();    JpaArticle obj = entityManager.find(JpaArticle.**class**, 1);  System.***out***.println(obj);    EntityTransaction transaction = entityManager.getTransaction();  transaction.begin();    JpaArticle jpaArticle = **new** JpaArticle();  jpaArticle.setArticleId(101);  jpaArticle.setCategory("JPA");  jpaArticle.setTitle("hibernate");    entityManager.persist(jpaArticle);  transaction.commit();    System.***out***.println(entityManager.find(JpaArticle.**class**, 101));    EntityTransaction transaction1 = entityManager.getTransaction();  transaction1.begin();  entityManager.remove(jpaArticle);  transaction1.commit();    EntityTransaction transaction2 = entityManager.getTransaction();  transaction2.begin();    JpaArticle jpaArticle1 = **new** JpaArticle();  jpaArticle1.setArticleId(59);  jpaArticle1.setCategory("nag");  jpaArticle1.setTitle("nagendra");    entityManager.merge(jpaArticle1);  transaction2.commit();    System.***out***.println("updated data: "+entityManager.find(JpaArticle.**class**, 59));  }  } |

You can make an Embedded class, which contains your two keys, and then have a reference to that class as EmbeddedId in your Entity.

You would need the [@EmbeddedId](https://docs.oracle.com/javaee/6/api/javax/persistence/EmbeddedId.html) and [@Embeddable](https://docs.oracle.com/javaee/6/api/javax/persistence/Embeddable.html) annotations.

@Entity

public class Customer {

@EmbeddedId

private CustomerId customerId;

private String company;

private String name;

@OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)

     private Set<OrderDetail> orderDetails;

    ...

}

@Embeddable

public class CustomerId implements Serializable{

private static final long serialVersionUID = 1L;

@Column(name = "customer\_id")

private int customerId;

@Column(name = "brandcode")

private String brandcode;

    ...

}

**ElementCollection**

|  |
| --- |
| @Entity  @Table(name="employee")  **public** **class** Employee {    @Id  @Column(name="emp\_id")  **private** Integer id;    @Column(name="emp\_name")  **private** String name;    @Column(name="emp\_email")  **private** String email;    @Column(name="emp\_salary")  **private** Integer salary;    @ElementCollection  **private** Collection<Address> address = **new** HashSet<>();  }  @Embeddable  **public** **class** Address {  @Column(name="street\_name")  **private** String street;    @Column(name="city\_name")  **private** String city;    @Column(name="pin\_code")  **private** String pincode;    @Column(name="state\_name")  **private** String state;  }  **Customize properties:**  @ElementCollection  @JoinTable(name="address\_table", joinColumns=@JoinColumn(name="employee\_id"))  **private** Collection<Address> address = **new** HashSet<>(); |

**Relationship mapping**

**one-to-Many:**  according to database terminology, one row of table related with multiple rows of other table

According to hibernate, one object of one pojo class related to multiple objects of other pojo

I mean, one [parent] to many [Children], example of one-to-many is some thing category books contains different type of books, one vendor contains lot of customers bla bla.

To achieve one-to-many between two pojo classes in the hibernate, then the following two changes are required

In the parent pojo class, we need to take a collection property, the collection can be either Set,List,Map (We will see the example on separate collection later)

Example: we have two POJO classes called Vendor and customer, where the relationship is one vendor is associated with many customers.

In this vendor to customer relationship, vendor pojo class acts like a parent class, customer pojo class acts like a child class.

|  |
| --- |
| **Parent class Employee:**  @Entity  @Table(name = "employee\_table")  @DynamicUpdate  **public** **class** Employee {  @Id  @Column(name = "emp\_id", length = 10)  **private** Integer id;  @Column(name = "emp\_name", length = 10)  **private** String name;  @Column(name = "emp\_email", length = 15)  **private** String email;  @Column(name = "emp\_salary0", length = 10)  **private** Integer salary;  @OneToMany(cascade = CascadeType.***ALL***)  @JoinTable(name = "employee\_address\_table",  joinColumns = @JoinColumn(name = "emp\_id"), // in third table parent column  inverseJoinColumns = @JoinColumn(name = "address\_Id"))+  , // in third table child column  **private** List<Address> address = **new** ArrayList<>();  //setter and getters  }  **Child Class Address:**  @Entity  @Table(name = "address\_table")  **public** **class** Address {    @Id  @Column(name = "address\_Id")  **private** Integer addressId;  @Column(name="street\_name", length=15)  **private** String street;    @Column(name="city\_name", length=10)  **private** String city;    @Column(name="pin\_code", length=6)  **private** String pincode;    @Column(name="state\_name", length=10)  **private** String state;  }  **Client Application:**  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("PERSISTENCE");  EntityManager entityManager = factory.createEntityManager();  EntityTransaction transaction = entityManager.getTransaction();  transaction.begin();  Employee e1 = **new** Employee();  e1.setEmail("nage@gmail.com");  e1.setId(101);  e1.setName("Nagendra");  e1.setSalary(5000);  Address address1 = **new** Address();  address1.setAddressId(101);  address1.setCity("KLDM");  address1.setPincode("56006");  address1.setState("Ap");  address1.setStreet("10 fitrd cross");    Address address2 = **new** Address();  address2.setAddressId(102);  address2.setCity("banglore");  address2.setPincode("51567");  address2.setState("karnataka");  address2.setStreet("3 rd cross");    e1.getAddress().add(address1);  e1.getAddress().add(address2);    entityManager.persist(e1);  transaction.commit();    System.***out***.println("updated data: "+entityManager.find(Employee.**class**, 101));  }  **Output of the Tables:** |

**Without Third Table or Join Table:**

|  |
| --- |
| **Pareant Employee class:**  @Entity  @Table(name = "employee\_table")  @DynamicUpdate  **public** **class** Employee {  @Id  @Column(name = "employee\_id", length = 10)  **private** Integer id;  @Column(name = "emp\_name", length = 10)  **private** String name;  @Column(name = "emp\_email", length = 15)  **private** String email;  @Column(name = "emp\_salary0", length = 10)  **private** Integer salary;  @OneToMany(fetch = FetchType.EAGER,  cascade = CascadeType.***ALL***, mappedBy ="employee") // maintain the owanership of table with child class  **private** List<Address> address = **new** ArrayList<>();  **public** Integer getId() {  **return** id;  }  **Child Address class:**  Entity  @Table(name = "address\_table")  **public** **class** Address {    @Id  @Column(name = "address\_Id")  **private** Integer addressId;  @Column(name="street\_name", length=15)  **private** String street;    @Column(name="city\_name", length=10)  **private** String city;    @Column(name="pin\_code", length=6)  **private** String pincode;    @Column(name="state\_name", length=10)  **private** String state;    @ManyToOne  @JoinColumn(name="employee\_id") // forien key cloumn name, insted of dfault name  **private** Employee employee;  }  **Client application Code:**  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("PERSISTENCE");  EntityManager entityManager = factory.createEntityManager();  EntityTransaction transaction = entityManager.getTransaction();  transaction.begin();  Employee e1 = **new** Employee();  e1.setEmail("nage@gmail.com");  e1.setId(101);  e1.setName("Nagendra");  e1.setSalary(5000);  Address address1 = **new** Address();  address1.setAddressId(101);  address1.setCity("KLDM");  address1.setPincode("56006");  address1.setState("Ap");  address1.setStreet("10 fitrd cross");  address1.setEmployee(e1);  Address address2 = **new** Address();  address2.setAddressId(102);  address2.setCity("banglore");  address2.setPincode("51567");  address2.setState("karnataka");  address2.setStreet("3 rd cross");  address2.setEmployee(e1);  e1.getAddress().add(address1);  e1.getAddress().add(address2);  entityManager.persist(e1);  transaction.commit();  // Featching table  Employee emp = entityManager.find(Employee.**class**, 101);  System.***out***.println(  "Employee Result: " + emp.getId() + " " + emp.getName() + "" + emp.getEmail() + "" + emp.getSalary());  System.***out***.println(emp.getAddress().toString());  **for** (Address obj : emp.getAddress()) {  System.***out***.println("Address Result: " + obj.getAddressId() + " " + obj.getCity() + "" + obj.getState() + ""  + obj.getStreet());  }  } |

**Many to One Mapping:**

In the many to one relationship, the relationship is applied from child object to parent object, but in one to may parent object to child object right..! just remember

Many to one is similar to one to many but with the little changes

many to one relationship from child to parent we need reference of a parent class in a child class.

In the child pojo class, create an additional property of type parent for storing the parent object in child object

**OneToOne Mapping:**

* One object is associated with one object only
* In this relationship, one object of the one pojo class contains association with one object of the another pojo class
* To apply one to one relationship between two pojo class objects it is possible by without taking a separate foreign key column in the child table of the database
* To apply one to one relationship, we copy the primary key value of parent object into primary key value of the child object.  So that the relationship between two objects is one to one
* If we want to copy parent object primary key value into child object primary key, we need to use a special generator class given by hibernate called foreign
* Actually this foreign generator is only used in one to one relationship only
* We are going to apply one to one between Employee and address pojo classes, here the relation is one Employee is assigned for one address only
* In order to get one to one relationship between Employee and address, we are copying primary key value of student into primary key value of address

<https://www.baeldung.com/jpa-one-to-one>

|  |
| --- |
| **Address child:**  @Entity  @Table(name = "address\_table")  **public** **class** Address {    @Id  @Column(name = "address\_Id")  **private** Integer addressId;  @Column(name="street\_name", length=15)  **private** String street;    @Column(name="city\_name", length=10)  **private** String city;    @Column(name="pin\_code", length=6)  **private** String pincode;    @Column(name="state\_name", length=10)  **private** String state;  **public** **void** setAddressId(Integer addressId) {  **this**.addressId = addressId;  }  **Employee Parent:**  @Entity  @Table(name="employee\_table")  @DynamicUpdate  **public** **class** Employee {    @Id  @Column(name="emp\_id", length=10)  **private** Integer id;    @Column(name="emp\_name", length=10)  **private** String name;    @Column(name="emp\_email", length=15)  **private** String email;    @Column(name="emp\_salary0", length=10)  **private** Integer salary;    @OneToOne(cascade=CascadeType.***ALL***)  @JoinColumn(name="address\_Id") // assign the name for foreign key  **private** Address address;  }  **Ouput:** |

**Many To Many Relationship:**

Applying many to many relationship between two pojo class objects is nothing but applying one to many relationship on both sides, which tends to Bi-Directional i mean many to many.

**Example:**

Let us see this, if we apply many to many association between two pojo class objects student and course, provided the relationship is one student may joined in multiple courses and one course contains lot of students (joined by multiple students)

Remember, when ever we are applying many to many relationship between two pojo class objects, on both sides  we need a collection property [As we are applying one to many from both the sides]

**Note Points:**

While applying many to many relationship between pojo classes,  a mediator table is mandatory in the database, to store primary key as foreign key both sides, we call this table as Join table

In many to many relationship join table contain foreign keys only

|  |
| --- |
| **Parent class Employee:**  @Entity  @Table(name = "employee\_table")  @DynamicUpdate  **public** **class** Employee {  @Id  @Column(name = "employee\_id", length = 10)  **private** Integer id;  @Column(name = "emp\_name", length = 10)  **private** String name;  @Column(name = "emp\_email", length = 15)  **private** String email;  @Column(name = "emp\_salary0", length = 10)  **private** Integer salary;  @ManyToMany(cascade = CascadeType.***ALL***, fetch=FetchType.***EAGER***) // fetch=FetchType.EAGER or fetch=FetchType.LAZY  **private** List<Address> address = **new** ArrayList<>();  **}**  **Child Address class:**  @Entity  @Table(name = "address\_table")  **public** **class** Address {    @Id  @Column(name = "address\_Id")  **private** Integer addressId;  @Column(name="street\_name", length=15)  **private** String street;    @Column(name="city\_name", length=10)  **private** String city;    @Column(name="pin\_code", length=6)  **private** String pincode;    @Column(name="state\_name", length=10)  **private** String state;    @ManyToMany(cascade=CascadeType.***ALL***)  **private** Collection<Employee> employee = **new** ArrayList<>();  **}**  **Tables are created with joins tables both sides**    **Reduce the on join table with custom join table, columns using @JoinTable**  **Parent Employee class:**  @Entity  @Table(name = "employee\_table")  @DynamicUpdate  **public** **class** Employee {  @Id  @Column(name = "employee\_id", length = 10)  **private** Integer id;  @Column(name = "emp\_name", length = 10)  **private** String name;  @Column(name = "emp\_email", length = 15)  **private** String email;  @Column(name = "emp\_salary0", length = 10)  **private** Integer salary;  @ManyToMany(cascade = CascadeType.***ALL***, fetch=FetchType.***EAGER***) // fetch=FetchType.EAGER or fetch=FetchType.LAZY    @JoinTable(name = "employee\_address\_table",  joinColumns = @JoinColumn(name = "emp\_id"), // in third table parent column  inverseJoinColumns = @JoinColumn(name = "address\_Id"))  **private** List<Address> address = **new** ArrayList<>();  **}**  **Child Address class:**  @Entity  @Table(name = "address\_table")  **public** **class** Address {    @Id  @Column(name = "address\_Id")  **private** Integer addressId;  @Column(name="street\_name", length=15)  **private** String street;    @Column(name="city\_name", length=10)  **private** String city;    @Column(name="pin\_code", length=6)  **private** String pincode;    @Column(name="state\_name", length=10)  **private** String state;    @ManyToMany(mappedBy="address")  **private** Collection<Employee> employee = **new** ArrayList<>();  **}**  **Client Application:**  EntityManagerFactory factory = Persistence.*createEntityManagerFactory*("PERSISTENCE");  EntityManager entityManager = factory.createEntityManager();  EntityTransaction transaction = entityManager.getTransaction();  transaction.begin();  Employee e1 = **new** Employee();  e1.setEmail("nage@gmail.com");  e1.setId(101);  e1.setName("Nagendra");  e1.setSalary(5000);    Employee e2 = **new** Employee();  e2.setEmail("scala@gmail.com");  e2.setId(102);  e2.setName("scala");  e2.setSalary(5000);  Address address1 = **new** Address();  address1.setAddressId(101);  address1.setCity("KLDM");  address1.setPincode("56006");  address1.setState("Ap");  address1.setStreet("10 fitrd cross");  Address address2 = **new** Address();  address2.setAddressId(102);  address2.setCity("banglore");  address2.setPincode("51567");  address2.setState("karnataka");  address2.setStreet("3 rd cross");  Address address3 = **new** Address();  address3.setAddressId(103);  address3.setCity("Hyderbad");  address3.setPincode("65656");  address3.setState("TS");  address3.setStreet("5 rd cross");    // one Employee contains mutlipLe address.  e1.getAddress().add(address1);  e1.getAddress().add(address2);  e1.getAddress().add(address3);    // multiple address contains same employee  address1.getEmployee().add(e1);  address2.getEmployee().add(e1);  address3.getEmployee().add(e1);    entityManager.persist(e1);  entityManager.persist(e2);  transaction.commit();  } |

Primary Key Generation Strategies

**GenerationType.AUTO**

This GenerationType indicates that the persistence provider should automatically pick an appropriate strategy for the particular database. This is the default GenerationType, i.e. if we just use @GeneratedValue annotation then this value of GenerationType will be used.

This generator automatically created sequence table in db and selects value and insert into table

**By default stragey**

|  |
| --- |
| @Entity  @Table(name="demo")  @DynamicUpdate  **public** **class** DemoTable {  @Id  @Column(name = "id", length = 5)  @GeneratedValue(strategy=GenerationType.***AUTO***, generator="demo\_generator")  @SequenceGenerator(name="demo\_generator", initialValue=1, allocationSize=1)  **private** Integer id;    @Column(name = "name", length = 10)  **private** String name;    @Column(name = "age", length = 3)  **private** Integer age;  }  Hibernate:  select  demo\_generator.nextval  from  dual  Hibernate:  insert  into  demo  (age, name, id)  values  (?, ?, ?)  **Output:** |

create sequence demo\_generator1 start with 1 increment by 1

**IDENTITY Generator:**

**This generator class to be specified if tha data base table primary colum is auto-increment column.**

The IDENTITY generator allows an integer/bigint column to be auto-incremented on demand. The increment process happens outside of the current running transaction, so a roll-back may end-up discarding already assigned values (value gaps may happen).

One thing to note is that IDENTITY generation disables batch updates.

The increment process is very efficient since it uses a database internal lightweight locking mechanism as opposed to the more heavyweight transactional course-grain locks.

The only drawback is that we can’t know the newly assigned value prior to executing the INSERT statement. This restriction is hinderingthe “transactional write behind” flushing strategy adopted by Hibernate. For this reason Hibernates disables the JDBC batch support for entities using the IDENTITY generator.

In case of GenerationType.auto extra table is created for id value, but GenerationType.Identity not created any additional tables simple mention GenerationType.Identity.

Identity is generator is database dependent generator, only support my sql, db2, sql server databases, but not oracle.

Sequence and identity generator classes opposite, it means sequence works on oracle, but not my sql, but identity works on my sql db not in oracle db

@Id

@Column(name = "id", length = 5)

@GeneratedValue(strategy=GenerationType. Identity)

**private** Integer id;

**IDENTITY SEQUENCE:**

Sequence generator first selects the next value from sequence of database, then assign that value for id of object.

If sequence is exists in db;

create sequence demo\_sequnce start with 1 increment by 3

@Id

@Column(name = "id", length = 5)

@GeneratedValue(strategy=GenerationType.***SEQUENCE***)

@SequenceGenerator(name="demo\_sequnce1")

**private** Integer id;

We need to additional configuration required in case of sequence doesn’t exist in oracle db

@Id

@Column(name = "id", length = 5)

@GeneratedValue(strategy=GenerationType.***SEQUENCE***, generator = "demo\_sequnce2")

@SequenceGenerator(name="demo\_sequnce2", initialValue=1, allocationSize=3)

**private** Integer id;

**GenerationType.TABLE:**

The *GenerationType.TABLE* gets only rarely used nowadays. It simulates a sequence by storing and updating its current value in a database table which requires the use of pessimistic locks which put all transactions into a sequential order. This slows down your application, and you should, therefore, prefer the *GenerationType.SEQUENCE*, if your database supports sequences, which most popular databases do.

|  |
| --- |
| @Id |
|  | @GeneratedValue(strategy = GenerationType.TABLE) |
|  | @Column(name = "id", updatable = false, nullable = false) |
|  | private Long id; |

 You can use the @TableGenerator annotation in a similar way as the already explained @SequenceGenerator annotation to specify the database table which Hibernate shall use to simulate the sequence.

|  |
| --- |
| @Id |
|  | @GeneratedValue(strategy = GenerationType.TABLE, generator = "book\_generator") |
|  | @TableGenerator(name="book\_generator", table="id\_generator", schema="bookstore") |
|  | @Column(name = "id", updatable = false, nullable = false) |
|  | private Long id; |

<http://shengwangi.blogspot.com/2014/12/how-to-configure-primary-key-generator-in-jpa-for-oracle.html>

**Update:** Suppose we are dealing with any employee object in the same session then we should use update() or saveOrUpdate() method.

**Update:** if you are sure that the session does not contains an already persistent instance with the same identifier, then use update to save the data in hibernate

**Merge:** Suppose we are creating a session and load an employee object. Now object in session cache. If we close the session at this point and we edit state of object and tried to save using update() it will throw exception. To make object persistent we need to open another session. Now we load same object again in current session. So if we want to update present object with previous object changes we have to use merge() method. Merge method will merge changes of both states of object and will save in database.

**Merge:** if you want to save your modifications at any time with out knowing about the state of an session, then use merge() in hibernate.

**Cascading** is a phenomenon involving one object propagating to other objects via a relationship. It is transitive in nature and the cascade attribute in hibernate defines the relationship between the entities. The cascading types supported by the hibernate framework are as follow:

* CascadeType.PERSIST: It means that the save() and persist() operations in the hibernate cascade to the related entities
* CascadeType.MERGE: It means that the related entities are joined when the owning entity is joined
* CascadeType.REMOVE: It means that the related entities are deleted when the owning entity is deleted
* CascadeType.DETACH: It detaches all the related entities if a manual detach occurs
* CascadeType.REFRESH: It works similar to the refresh() operation in the hibernate
* CascadeType.ALL: It is an alternative for performing all the above cascade operations in the hibernate framework

**Get():**

get method performs early loading on a object. It means an object is loaded immediately from the database, irrespective of wether thate object is loaded or not.

Get() returns null if object not found in db

**Load():**

it performs lazy loading, , load the object from the data base, only if we accessed in the application.

Load() throws object not found the exception

**Cache Level**

* By default, for each hibernate application, the first level cache is automatically been enabled
* As a programmer, we no need to have any settings to enable the first level cache and also we cannot disable this first level cache
* the first level cache is associated with the session object and scope of the cache is limited to one session only
* When we load an object for the first time from the database then the object will be loaded from the database and the loaded object will be stored in the cache memory maintained by that session object
* If we load the same object once again, with in the same session, then the object will be loaded from the local cache memory not from the database
* If we load the same object by opening other session then again the object will loads from the database and the loaded object will be stored in the cache memory maintained by this new session

First level cache will be enabled by default, but for enable second level cache we need to follow some settings, let us see few points regarding this..

Second level cache was introduced in hibernate 3.0

* When ever we are loading any object from the database,  then hibernate verify whether that object is available in the local cache memory of that particular session [**means first level cache**], if not available then hibernate verify whether the object is available in global cache or factory cache [ **second level cache** ], if not available then hibernate will hit the database and loads the object from there, and then first stores in the local cache of the session [ first level ] then in the global cache [ second level cache ]
* When another session need to load the same object from the database,  then hibernate copies that object from global cache [ second level cache ] into the local cache of this new session

 Second level cache in the hibernate is of  from **4** vendors…

* Easy Hibernate [EHCache] Cache from hibernate framework
* Open Symphony [OS] cache from Open Symphony
* SwarmCache
* TreeCache from JBoss

**Life cycle of Sates**

## Transient & Persistent states:

* When ever an object of a pojo class is created then it will be in the Transient state
* When the object is in a Transient state it doesn’t represent any row of the database, i mean not associated with any Session object, if we speak more we can say no relation with the database its just an normal object
* If we modify the data of a pojo class object, when it is in transient state then it doesn’t effect on the database table
* When the object is in persistent state, then it represent one row of the database, if the object is in persistent state then it is associated with the unique Session
* if we want to move an object from persistent to detached state, we need to do either closing that session or need to clear the cache of the session
* if we want to move an object from persistent state into transient state then we need to delete that object permanently from the database

## Transient:

One newly created object,with out having any relation with the database, means never persistent, not associated with any Session object

## Persistent:

Having the relation with the database, associated with a unique Session object

## Detached:

previously having relation with the database [persistent ], now not associated with any Session

see the next sessions for the better understanding of the life cycle states of pojo class object(s) the hibernate

save() method can return that primary key id value which is generated by hibernate and we can see it by

long s = session.save(k);

In this same case, persist() will never give any value back to the client,

## . @Component

This is a general purpose stereotype annotation available with Spring framework.This annotation shows that the class is a spring managed bean/ component.

@Component

public class ApplicationConfigurations{

...

}

Copy

*Spring only scan @Component and does not look for @Service and @Repository.This happens because they annotate these other annotations themselves with @Component.*

@Component

public @interface Service {

….

}

@Component

public @interface Repository {

….

}

Copy

In another words we can say @Service and @Repository are special types of @Component annotation.

## 2. @Repository

This annotation classifies a Spring bean as a data repository.@Repository annotation also catch platform specific exceptions and re-throw specific them Spring unchecked exception. Spring provides [PersistenceExceptionTranslationPostProcessor](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/dao/annotation/PersistenceExceptionTranslationPostProcessor.html" \t "_blank), class to handle this.

<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

Copy

This bean post processor adds an advisor to any bean that’s annotated with @Repository.

## 3. @Service

Java class annotated with @Service annotation shows that the bean is part of the service layer which holds the business logic. This annotation only classify bean as service level beans and have no additional processing like @Repository.

## Summary

In this short post we saw the **difference between @Component, @Repository and @Service annotations in Spring**. We talked about each annotation with the area where we should use these annotations. At the end these annotations are almost similar with few additional features but they help us classify the beans and it is always better to choose annotation based on the layer conventions.

Nov 13, 2017 10:53:04 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh

INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@736e9adb: startup date [Mon Nov 13 22:53:04 CST 2017]; root of context hierarchy

Nov 13, 2017 10:53:04 PM org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseFactory initDatabase

INFO: Starting embedded database: url='jdbc:h2:mem:testdb;DB\_CLOSE\_DELAY=-1;DB\_CLOSE\_ON\_EXIT=false', username='sa'

Nov 13, 2017 10:53:04 PM org.springframework.jdbc.datasource.init.ScriptUtils executeSqlScript

INFO: Executing SQL script from class path resource [createEmployeeTable.sql]

Nov 13, 2017 10:53:04 PM org.springframework.jdbc.datasource.init.ScriptUtils executeSqlScript

INFO: Executed SQL script from class path resource [createEmployeeTable.sql] in 16 ms.

Nov 13, 2017 10:53:04 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [org/springframework/jdbc/support/sql-error-codes.xml]

Nov 13, 2017 10:53:04 PM org.springframework.jdbc.support.SQLErrorCodesFactory <init>

INFO: SQLErrorCodes loaded: [DB2, Derby, H2, HSQL, Informix, MS-SQL, MySQL, Oracle, PostgreSQL, Sybase, Hana]

org.springframework.dao.DataIntegrityViolationException: PreparedStatementCallback; SQL [insert into EMPLOYEE (NAME) values (?)NULL not allowed for column "DEPT"; SQL statement:

insert into EMPLOYEE (NAME) values (?) [23502-196]; nested exception is org.h2.jdbc.JdbcSQLException: NULL not allowed for column "DEPT"; SQL statement:

In above output, the H2 JdbcSQLException is translated to DataIntegrityViolationException which is a subclass of DataAccessException. Check out more subclasses of this exception [here](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/dao/package-summary.html).

public class **DuplicateKeyException**

extends [DataIntegrityViolationException](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/dao/DataIntegrityViolationException.html)

Exception thrown when an attempt to insert or update data results in violation of an primary key or unique constraint. Note that this is not necessarily a purely relational concept; unique primary keys are required by most database types.

## The @Repository annotation

As seen in above example, we are using @Repository annotation instead of of @Component in our DAO implementation. This annotation guarantees that all Spring DAO support, including the exception translation is provided in a consistent way.

## Service Layer

@Service

public class EmployeeService {

@Autowired

EmployeeDao employeeDao;

public void saveEmployee(Employee employee) {

try {

employeeDao.save(employee);

} catch (DataAccessException dae) {

System.err.println(dae);

}

}

}

As seen above, in our service layer we are handling data technology specific exception by using spring's DataAccessException which is translated from the underlying exception. There are two advantages; first we don't have to catch any H2 or JDBC related exceptions in our DAO (even we don't have to have the knowledge of those exceptions), second if in future we want to migrate from JDBC to other technology e.g. JPA then we just need to change our DAO implementation without changing our service layer.

**What is different b/w JpaRepository and CrudRepository**

JpaRepository extends PagingAndSortingRepository that extends CrudRepository.

CrudRepository mainly provides CRUD operations.

PagingAndSortingRepository provide methods to perform pagination and sorting of records.

JpaRepository provides JPA related methods such as flushing the persistence context and deleting of records in batch.

Due to their inheritance nature, JpaRepository will have all the behaviors of CrudRepository and PagingAndSortingRepository. So if you don't need the repository to have the functions provided by JpaRepository and PagingAndSortingRepository , use CrudRepository.