

JPA Using Hibernate

By

Praveen Oruganti

Blog: <https://praveenorugantitech.blogspot.com>

Facebook Group: <https://www.facebook.com/groups/268426377837151>

Github repo: <https://github.com/praveenoruganti>

Email: praveenorugantitech@gmail.com

Major drawbacks of JDBC

We create SQL queries and are database dependent so that persistence logic is database dependent.

JDBC exceptions are checked exceptions hence we need to put the code in try and catch blocks

In JDBC we use Resultset Object to store the data selected from database and it is not transferable over the network.

If database table structure is modified then queries needs to be modified accordingly.

Let's see simple JDBC program using mysql

```
try (Connection con = DriverManager.getConnection("jdbc:mysql://root:password@localhost:3306/praveendb", "root",
    "password");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from employee");
    PreparedStatement ps = con.prepareStatement(
        "insert into employee(employee_id,email,employee_name,gender,salary) values(?,?,?,?,?)");
    CallableStatement cs = con.prepareCall("{call INSEREMP(?,?,?,?,?)}"); {
    Class.forName("com.mysql.cj.jdbc.Driver");
    int row = 0;
    while (rs.next()) {
        System.out.println(rs.getInt(1) + " " + rs.getString(2) + " " + rs.getString(3) + " " + rs.getString(4)
            + " " + rs.getDouble(5));
        row++;
    }
    ps.setInt(1, row);
    ps.setString(2, "mnp3pk" + row + "@gmail.com");
    ps.setString(3, "Praveen " + row);
    ps.setString(4, "Male");
    ps.setDouble(5, 90000);
    ps.executeUpdate();
    row = row + 1;
    cs.setInt(1, row);
    cs.setString(2, "mnp3pk" + row + "@gmail.com");
    cs.setString(3, "Praveen " + row);
    cs.setString(4, "Male");
    cs.setDouble(5, 90000);
    cs.execute();
} catch (SQLException | ClassNotFoundException e) {
    e.printStackTrace();
}

create table employee(EMPLOYEE_ID INT, email VARCHAR(50),employee_name VARCHAR(50),gender VARCHAR(50),salary DOUBLE);
DELIMITER //
CREATE PROCEDURE INSEREMP (IN EMPLOYEE_ID INT, IN email VARCHAR(50),IN employee_name VARCHAR(50),IN gender VARCHAR(50),IN salary DOUBLE)
BEGIN
insert into employee values(employee_id,email,employee_name,gender,salary);
END //
```

Here comes ORM.....

2 | Praveen Oruganti

Blog: <https://praveenorugantitech.blogspot.com>

Facebook Group: <https://www.facebook.com/groups/268426377837151>

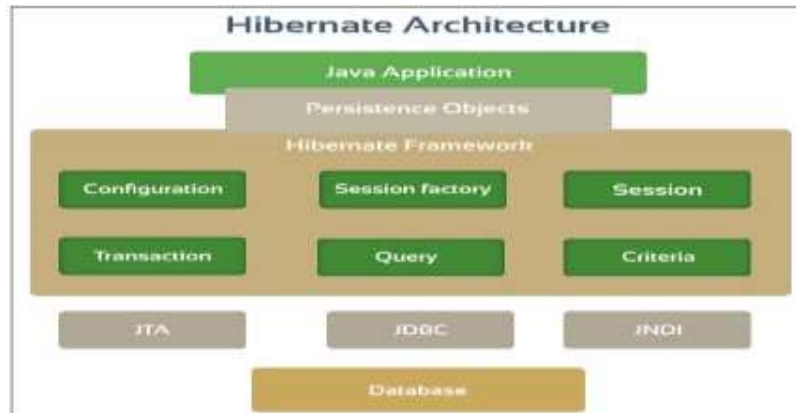
Github repo: <https://github.com/praveenoruganti>

Email : praveenorugantitech@gmail.com

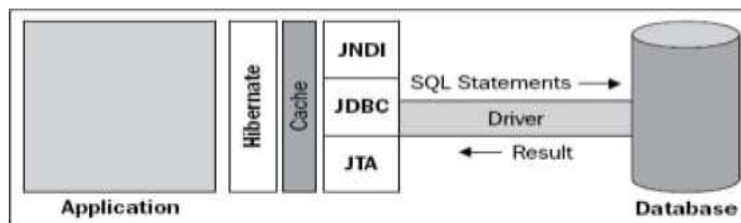
Hibernate

Hibernate is an open source, non-invasive framework, ORM Tool and is used in the data layer of application which also implements JPA.

Hibernate Architecture



Full execution chain



JTA(Java Transaction API) is a standard for transactions, allowing for management of multiple transactions among multiple databases

JPA(Java Persistence API) is a specification of how the object relation mapping should be done.

JNDI(Java Naming And Directory Interface) is the directory which contains the list of objects defined on the server. When the user requests for a connection pool, the code would look for the object on the JNDI and would get the requested connection pool from the datasource. JDBC is the one which does all the interactions with the database. An application will lookup the JNDI to get a connection pool for it to interact with the database.

Three important objects of Hibernate Persistence logic development

- ✓ Configuration Object
- ✓ SessionFactory Object

3 | Praveen Oruganti

Blog: <https://praveenorugantitech.blogspot.com>

Facebook Group: <https://www.facebook.com/groups/268426377837151>

Github repo: <https://github.com/praveenoruganti>

Email : praveenorugantitech@gmail.com

- ✓ Session Object

Configuration Object

- ✓ It maintains inmemory data like db properties and mapping class.
- ✓ **How to build configuration object in hibernate**
org.hibernate.cfg.Configuration class
// Activate Hibernate framework(bootstrapping)
Configuration cfg= new Configuration();
cfg.addAnnotatedClass(Message.class);
// To load property file hibernate.properties
org.hibernate.service.ServiceRegistry serviceRegistry= new
StandardService.RegistryBuilder().applySettings(cfg.properties()).build();
- ✓ The configuration object is usually created once during application initialization.

SessionFactory Object

- ✓ Factory to create session objects.
- ✓ Designed around factory pattern.
- ✓ obj of a class that implements org.hibernate.SessionFactory()
- ✓ Immutable object
- ✓ Threadsafe object
- ✓ It is heavy weight object
- ✓ **How to build SessionFactory Object obviously it is by using Configuration object**
SessionFactory= Configuration.buildSessionFactory(serviceRegistry);
- ✓ It gathers inmemory data from configuration object(JDBC properties)
- ✓ It creates JDBC connection pool having set of connection object.
- ✓ It represents second level cache of hibernate.
- ✓ one sessionFactory created per database.

Session Object

- ✓ It is created based on sessionFactory Object
- ✓ Opens connection/session with database through hibernate framework
- ✓ It is base for Program/App to give domain class object based persistence instructions to Hibernate framework.
- ✓ It is lightweight object
- ✓ It is not thread safe object
- ✓ It represents first level cache of hibernate
org.hibernate.Session session= HibernateUtil.getSessionFactory().openSession();

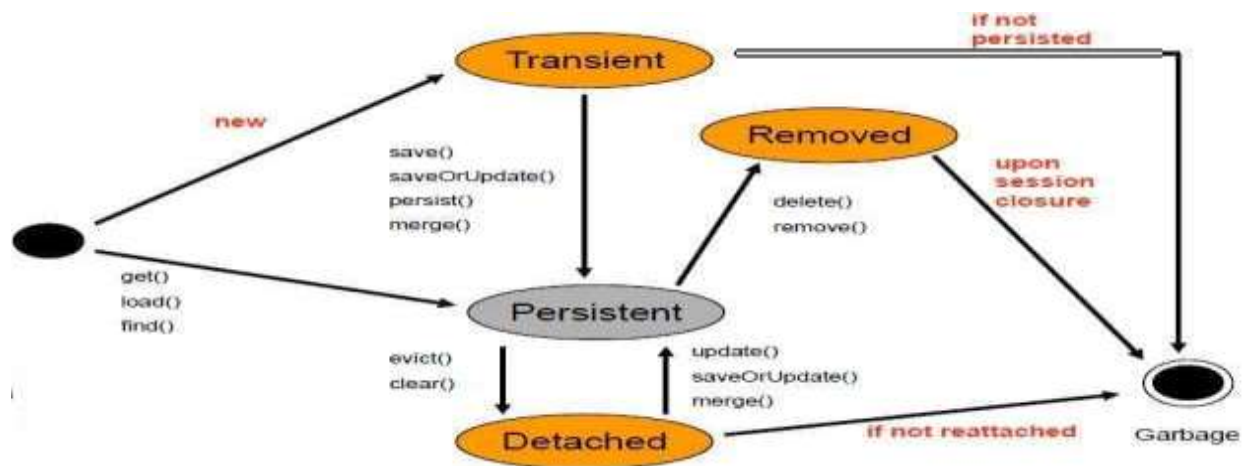
```

session.save(obj);
session.delete(obj);
session.update(obj);
session.load/get(domain class,id);
load -> return proxy object(Lazy) -> object not found exception
get-> directly hits database(Eagerly)->null

```

Unlike sessionFactory, the session object will be created on demand. session is a lightweight object.session provides a physical connectivity between your application and database.

Object states



- create an object new()-> moved to Transient
- session.save()-> moved from Transient to Persistent
- session.close()-> moved from Persistent to Detached.

The lifecycle of a session object is bounded by the beginning and end of a logical transaction.

1. **beginTransaction()**

Begin a unit of work and return the associated transaction object once CRUD operations are done on session. call commit() method on returned transaction object. In case of any issues, call rollback() error on the transaction object.

2. **save()**

persist the given transient instance, first assigning a generated identifier. This method stores the given object in the DB.

Before storing it checks for generated identifier, declaration and process it first and then it will store in DB.

3. **update()**

update the persistence instance with the identifier of the given detached instance. it updates the database record.

4. **saveorUpdate()**

either save() or update() the given instance depending on the resolution of the unsaved value checks. First hibernate checks the existence of instance of entity and if not available then inserts the data into the database and if available then updates the data.

5. **createQuery()**

create a new instance Query for the given HQL query string

6. **createSQLQuery()** -> native SQL

7. **merge()**

Retrieves a fresh version of the object from the database and based on that, as well as modifications made to the object being passed in, schedules update statements to modify the existing object in the database.

8. **get()**

Retrieves an object instance, or null if not found

9. **lock()**

Reattaches a detached object to a session without scheduling an update

Also used to 'lock' records in the database

10. **delete()**

Schedule an object for removal from the database

11. **clear()**

Removes all objects from persistence context, changing all their states from persistent to detached

Lazy Vs Eager Loading

- ✓ Eager Loading is a design pattern in which data initialization occurs on the spot. It means that collections are fetched fully at the time their parent is fetched (fetch immediately)
- ✓ Lazy Loading is a design pattern which is used to defer initialization of an object until the point at which it is needed. This can effectively contribute to application's performance.

Transaction Object

A transaction simply represents a unit of work. In such case, if one step fails, the whole transaction fails(which is termed as atomicity).

A transaction can be described by ACID(Atomicity, Consistency, Isolation and Durability)

Methods in transaction

a. void begin() b. void commit() c. void rollback() d.void setTimeout(int seconds) e. boolean isAlive() f. void registerSynchronization(synchronization s) g. boolean wasCommitted() h. boolean wasRolledBack()

```
Transaction transobj=null;
```

```
Session sessionobj=null;
```

```
try{
```

```
sessionobj=HibernateUtil.buildSessionFactory();
```

```
transobj= sessionobj.beginTransaction();
```

```
// perform some operation
```

```
transobj.commit();
```

```
}catch(HibernateException e){
```

```
if(transobj!=null){
```

```
transobj.rollback();
```

```
}
```

```
System.out.println(e);
```

```
}finally{
```

```
sessionobj.close();
```

```
}
```

Query Object -> HQL(Hibernate Query Language)

HQL uses classname instead of table name and property names instead of column name.

A HQL Query may consist of following elements

a) clauses b) Aggregate function c) subqueries clauses

a) clauses

1.from 2. as 3. select 4. where 5. orderby 6. groupby 7. update 8. delete 9. insert

1. from clause

This clause is used to load a complete persistent object into memory.

```
String hql="From Employee";
```

7 | Praveen Oruganti

Blog: <https://praveenorugantitech.blogspot.com>

Facebook Group: <https://www.facebook.com/groups/268426377837151>

Github repo: <https://github.com/praveenoruganti>

Email : praveenorugantitech@gmail.com

```
Query query=Session.createQuery(hql);
```

```
List results=query.list();
```

2. As clause

```
"From Employee as E";
```

3. Select clause

```
String hql="Select e.firstName FROM Employee e";
```

4. Where clause

```
"FROM Employee e where e.id=100";
```

5. order by clause

```
"FROM Person p where p.id>10 order by p.salary desc"
```

6. group by clause

```
"SELECT SUM(p.salary),p.firstName FROM PERSON P GROUP BY p.firstName"
```

7. update clause

```
String hql="Update Employee set salary=:salary"+"where id=:empld";
```

```
Query query=session.createQuery(hql);
```

```
query.setParameter("salary",10000);
```

```
query.setParameter("empld",10);
```

```
int result= query.executeUpdate();
```

8. Delete clause

```
String hql="Delete from employee"+" where id=:empld";
```

```
Query query=session.createQuery(hql);
```

```
query.setParameter("empld",10);
```

```
int result= query.executeUpdate();
```

b) Aggregate functions

i) avg(), sum(),min(),max()

ii) count(*), count(distinct ..), count(all ..)

we need to use named parameters to avoid SQL injection.

Pagination using query

a) setFirstResult(int startposition) b) setMaxResults(int maxResult)

Native SQL

```
Query sqlquery= session.createSQLQuery("select * from Books where author =?");
```



```
List results = sqlquery.setString(0,"Praveen").list();
```

Criteria Object -> Hibernate Criteria Query Language

Criteria is only used for Select(fetch results) in object oriented approach.

```
public Criteria createCriteria(class c)
```

methods:

- a) Criteria add(Criterion c) is used to add restrictions
- b) Criteria addOrder(Order o) specifies ordering
- c) Criteria setFirstResult(int firstResult) specifies the first number of record to be retrieved.
- d) Criteria setMaxResult(int totalResult) specifies the total number of records to be retrieved.
- e) public List list() returns list containing object
- f) public Criteria setProjection(Projection projection) specifies the projection

Restrictions class

simpleExpressionlt(String propertyName, Object value) sets the less than constraint to the given property

- ✓ le(String propertyName, Object value) – less than or equal
- ✓ gt(String propertyName, Object value) – greater than
- ✓ ge-> greater than or equal
- ✓ ne-> not equal
- ✓ eq->equal
- ✓ between(String propertyName, Object low, Object high)
- ✓ like

Order class

- 1. order asc(String propertyName) -> ascending
- 2. order desc(String propertyName) -> descending

Examples

- 1. get all records

```
Criteria c=session.createCriteria(Employee.class);
```

```
List list= c.list();
```

- 2.get the 10th to 20th record

```
c.setFirstResult(10);
c.setMaxResult(20);
3.get the records whose salary is greater than 10000
c.add(Restriction.gt("salary",10000);
4.get the records in ascending order on the basis of salary
c.addOrder(Order.asc("salary"));
```

with Projection

we can fetch the data of a particular column by projection such as name etc

```
c.setProjection(Projections.proeperty("name"));
```

Multiple projections

```
ProjectionList p1= Projections.ProjectionList();
p1.add(Projects.property("name"));
p1.add(Projects.property("price"));
c.setProjection(p1);
```

Criteria preferences over HQL

- 1.Criteria is safe from vulnerable to SQL injection as queries are fixed or parametrized
2. HQL is ideal for static queries
- 3.HQL does not support pagination but it can be achieved by criteria for pagination
- 4.Criteria helps us to build queries in a cleaner way and most of our errors are solved during compile time.

only problem with Criteria

HQL supports both select and non select where as Criteria supports select only
Criteria is slower than HQL.

Hibernate Named Query

While executing either HQL, Native SQL Queries if we want to execute the same queries multiple times and in more than one client program application then we can use the named queries mechanism

```
@NamedQuery(name="@HQL_GET_ALL_ADDRESS", query=" from Address")
```

Advantages

- 1.Named queries are compiled when session factory is instantiated

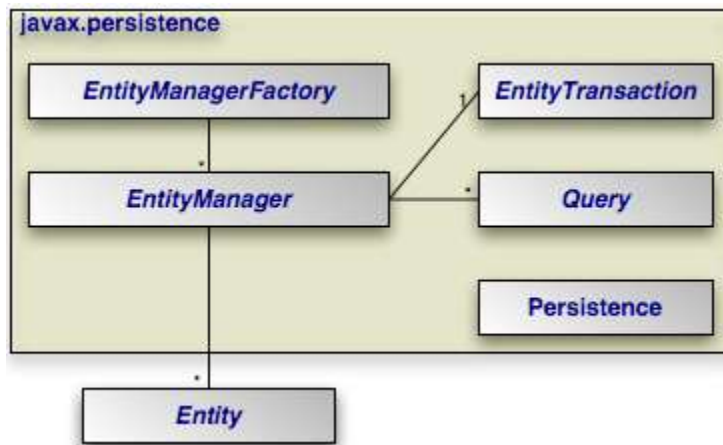
2.Easier to maintain

Disadvantages

Static queries and not customizable at runtime.

JPA

The Java Persistence API provides a specification for persisting, reading, and managing data from your Java object to relational tables in the database.



Persistence: The javax.persistence.Persistence class contains static helper methods to obtain EntityManagerFactory instances in a vendor-neutral fashion.

EntityManagerFactory: The javax.persistence.EntityManagerFactory class is a factory for EntityManager s.

EntityManager : The javax.persistence.EntityManager is the primary JPA interface used by applications. Each EntityManager manages a set of persistent objects, and has APIs to insert new objects and delete existing ones. When used outside the container, there is a one-to-one relationship between an EntityManager and an EntityTransaction. EntityManagers also act as factories for Query instances.

Entity : Entites are persistent objects that represent datastore records.

EntityTransaction: Each EntityManager has a one-to-one relation with a single javax.persistence.EntityTransaction. EntityTransactions allow operations on persistent data to be grouped into units of work that either completely succeed or completely fail,

leaving the datastore in its original state. These all-or-nothing operations are important for maintaining data integrity.

Query : The `javax.persistence.Query` interface is implemented by each JPA vendor to find persistent objects that meet certain criteria. JPA standardizes support for queries using both the Java Persistence Query Language (JPQL) and the Structured Query Language (SQL). You obtain Query instances from an `EntityManager`.

Persistence Context and Entity Manager

The Persistence Context and the Entity Manager are two core concepts of the Java Persistence API (JPA). In short: The Persistence Context is responsible for JPA entity management: When an application loads an entity from the database, the entity is in fact stored in the Persistence Context, so the entity becomes managed by the Persistence Context. Any further change made over that same entity will be monitored by the Persistence Context.

The Persistence Context will also flush changed entities to the database when appropriate. When a transaction commits, the associated Persistence Context will also flush any eventual pending changes to the Database. These are some of the operations that are handled by the Persistence Context.

The Entity Manager is an interface for the application to interact with the Persistence Context.

JPA Annotations

1. `javax.persistence.Entity`

`@Entity` annotation defines that a class can be mapped to a table and that is it, it just a marker like for example `Serializable` interface.

When we create a new entity we have to do atleast two things

a) `@Entity` b) `@Id`

Anything else is optional

2. `javax.persistence.Table`

`@Table` specifies the primary table for the annotated entity and is optional.

`@Table(name="CUST",schema="RECORDS")`

3. **javax.persistence.Column**

@Column is used to specify the mapped column for a persistent property or field and is optional.

@Column(name="DESC",nullable=false,length=512,columnDefinition="CLOB NOT NULL", table="CUST",updatable=true)

@Column(name="ORDER_COST",updatable=false,precision=12,scale=2)

4. **javax.persistence.Id**

@Id specifies the primary key of an entity and is mandatory

5. **javax.persistence.GeneratedValue**

@GeneratedValue(strategy=GenerationType.SEQUENCE) provides for the specification of generation strategies for the values of primary keys.

TABLE,SEQUENCE,IDENTITY,AUTO and it is optional.

6. **@Version**

we control versioning or concurrency using this annotation

As we know once an object saved in a database, we can modify the object any number of times, if we want to know how many number of times that an object is modified then we need to apply this versioning concept.

7. **@OrderBy**

Sort your data using @OrderBy annotation

@OrderBy("id asc")

private Set employee_addressl

8. **@Transient**

Every non static and non transient property of an entity is considered persistent unless you annotate it as @Transient

9. **@Lob**

Large objects are declared with @Lob

10. @Embeddable

An Entity has a meaning on its own where as value object doesn't have meaning on its own.

For value objects we will use @Embeddable

11. @EmbeddedId

If primary key is a combination of multiple attributes then we need to use @EmbeddedId

12. @ElementCollection

For saving collections we use this.

@ElementCollection

```
private Set<Address> listofAddress= new HashSet<>();
```

JPA using Hibernate

JPA(Java Persistence API) is an interface and Hibernate is the implementation of it.

All the entities saved using EntityManager(interface) will be saved in the persistence context.

It's the persistence context which keeps track of all the different entities which are changed during the transaction and also it keeps track of all the changes that needs to be stored back to the database.

Below are the JPA methods

- ✓ **find()** will fetch the entity from database based on primary key id
- ✓ **persist** will insert the entity in database and will not return anything
- ✓ **merge** will get the fresh object from database and merges the changes if anything happened to the entity into the persistence context.
- ✓ **remove** will first check whether the entity is available in database based on the primary key id and will remove the entity in the database accordingly.
- ✓ **flush()** whatever is there will be saved to database.
- ✓ **detach()** persistence context will not track the changes.
- ✓ **clear()** Clear the persistence context, causing all managed entities to become detached. Changes made to entities that have not been flushed to the database will not be persisted.
- ✓ **refresh()** : All the changes done will be wiped off and will fetch the entity data present from the database.

Here with the sample JPA repository

```
package com.praveen.spring.jpa.repository;

import java.util.List;

import javax.persistence.EntityManager;
import javax.transaction.Transactional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import com.praveen.spring.jpa.entity.Course;
import com.praveen.spring.jpa.entity.Review;

@Repository
@Transactional
public class CourseRepository {

    @Autowired
    EntityManager em;

    public Course findById(Long id) {
        return em.find(Course.class, id);
    }

    public void deleteById(Long id) {
        Course course=findById(id);
        em.remove(course);
    }

    public void save(Course course) {
        if(course.getId()==null) {
            em.persist(course); //insert
        }else {
            em.merge(course); //update
        }
    }

    public void addHardcodedReviewsForCourse() {
        Course course=findById(10004L);
        Review review1=new Review("Superb Course","FOUR");
        Review review2=new Review("Excellent Course","FIVE");
        course.addReview(review1);
        course.addReview(review2);
        review1.setCourse(course);
        review2.setCourse(course);
        em.persist(review1);
        em.persist(review2);
    }

    public void addReviewsForCourse(Long courseId,List<Review> reviews) {
        Course course=findById(courseId);
        for(Review review:reviews) {
            course.addReview(review);
            review.setCourse(course);
            em.persist(review);
        }
    }
}
```

JPQL(Java Persistence Query Language)

In JPQL we query from entities whereas SQL we query from tables.

SQL: Select * from Course

JPQL: Select c from Course c

Here with few JPQL queries

```
package com.praveen.spring.jpa.repository;

import java.util.List;

@RunWith(SpringRunner.class)
@SpringBootTest(classes=PraveenorugantiSpringbootJpaHibernateApplication.class)
public class JPQLTest {
    private Logger logger= LoggerFactory.getLogger(this.getClass());

    @Autowired
    public EntityManager em;

    @Test
    @DirtyContext
    public void jpqlTest() {
        //Query q= em.createQuery("Select c from Course c");
        Query q= em.createNamedQuery("query_get_all_courses");
        logger.info("Select c from Course c -> {}",q.getResultList());
        //Output: Select c from Course c -> [Course[CORE JAVA], Course[SPRING], Course[JPA HIBERNATE], Course[SQL in 25 Days], Course[Maven in 25 Days]]
    }

    @Test
    @DirtyContext
    public void jpqlTypedTest() {
        //TypedQuery<Course> q= em.createQuery("Select c from Course c",Course.class);

        TypedQuery<Course> q= em.createNamedQuery("query_get_all_courses",Course.class);
        List<Course> qlist=q.getResultList();
        logger.info("Select c from Course c -> {}",qlist);
        //Output: Select c from Course c -> [Course[CORE JAVA], Course[SPRING], Course[JPA HIBERNATE], Course[SQL in 25 Days], Course[Maven in 25 Days]]
    }

    @Test
    @DirtyContext
    public void jpqlWhere() {
        //TypedQuery<Course> q= em.createQuery("Select c from Course c where name like '%25%' ",Course.class);
        TypedQuery<Course> q= em.createNamedQuery("query_get_like25",Course.class);
        List<Course> qlist=q.getResultList();
        logger.info("Select c from Course c where name like '%25%' -> {}",qlist);
        //Output: Select c from Course c where name like '%25%' -> [Course[SQL in 25 Days], Course[Maven in 25 Days]]
    }
}
```

Here with the named queries

```
package com.praveen.spring.jpa.entity;

import java.time.LocalDateTime;

@Entity
@NamedQueries(value = { @NamedQuery(name = "query_get_all_courses", query = "Select c from Course c"),
    @NamedQuery(name = "query_get_like25", query = "Select c from Course c where name like '%25%'") })
//@NamedQuery(name="query_get_all_courses",query="select c from Course c")
@Cacheable
public class Course {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @CreationTimestamp
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime lastUpdatedAt;

    // @JsonIgnore
    @OneToMany(mappedBy="course") // by default it is lazy fetch
    private List<Review> reviews = new ArrayList<Review>();

    @JsonIgnore
    @ManyToMany(mappedBy="courses") // by default it is lazy fetch
    private List<Student> students= new ArrayList<Student>();
}
```


Native Queries

```
package com.praveen.spring.jpa.repository;

import java.util.List;

@RunWith(SpringRunner.class)
@SpringBootTest(classes=PraveenorugantiSpringbootJpaHibernateApplication.class)
public class NativeQueryTest {
    private Logger log= LoggerFactory.getLogger(this.getClass());

    @Autowired
    public EntityManager em;

    @Test
    @DirtyContext
    public void nativequeryTest() {
        Query q= em.createNativeQuery("select * from Course",Course.class);
        log.info("select * from Course -> {}",q.getResultList());
        //Output: select * from Course -> [Course[CORE JAVA], Course[SPRING], Course[JPA HIBERNATE], Course[SQL in 25 Days], Course[Mayen in 25 Days]]
    }

    @Test
    @DirtyContext
    public void nativequerywhere() {
        Query q= em.createNativeQuery("select * from Course where name like '%25'",Course.class);
        log.info("Select c from Course c where name like '%25%' -> {}",q.getResultList());
        //Output: Select c from Course c where name like '%25%' -> [Course[SQL in 25 Days], Course[Mayen in 25 Days]]
    }

    @Test
    @Transactional
    public void native_queries_to_update() {
        Query query = em.createNativeQuery("Update COURSE set last_updated_date=sysdate()");
        int noOfRowsUpdated = query.executeUpdate();
        log.info("noOfRowsUpdated -> {}", noOfRowsUpdated);
    }

    @Test
    public void native_queries_with_parameter() {
        Query query = em.createNativeQuery("SELECT * FROM COURSE where id = ?", Course.class);
        query.setParameter(1, 10001L);
        List resultList = query.getResultList();
        log.info("SELECT * FROM COURSE where id = ? -> {}", resultList);
    }

    @Test
    public void native_queries_with_named_parameter() {
        Query query = em.createNativeQuery("SELECT * FROM COURSE where id = :id", Course.class);
        query.setParameter("id", 10001L);
        List resultList = query.getResultList();
        log.info("SELECT * FROM COURSE where id = :id -> {}", resultList);
    }
}
```

@DirtyContext is used in test class to rollback whatever updates you have written in your test method.

JPA Using Hibernate Mapping

Hibernate mappings are one of the key features of Hibernate. They establish the relationship between two database tables as attributes in your model. That allows you to easily navigate the associations in your model and Criteria queries.

You can establish either unidirectional or bidirectional i.e. you can either model them as an attribute on only one of the associated entities or on both. It will not impact your

database mapping tables, but it defines in which direction you can use the relationship in your model and Criteria queries.

The relationship that can be established between entities are

- ✓ **One To One** – It represent one to one relationship between two tables.
- ✓ **One To Many/Many To One** – It represents the one to many / many to one relationship between two tables.
- ✓ **Many To Many** – It represents the many to many relationship between two tables.

OneToOne Mapping

Let's consider an example for One to One Mapping i.e... A Student has a Passport.

Any OneToOne is an Eager mapping which leads to performance implications.

In order to avoid this we need to use fetch lazy.

Now to establish bidirectional relation mapping using OneToOne mapping.

As student is owner, we need to initialize mappedBy="passport" in Passport.java

Let's see how we can define this in Student and Passport Entity classes.

```
@Entity
public class Passport {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable=false)
    private String number;

    @OneToOne(fetch=FetchType.LAZY,mappedBy="passport") // by default it is eager fetch
    private Student student;

}

@Entity
public class Student {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable=false)
    private String name;

    @OneToOne(fetch=FetchType.LAZY) // by default it is eager fetch
    private Passport passport; // SELECT * FROM STUDENT,PASSPORT WHERE STUDENT.PASSPORT_ID= PASSPORT.ID

}
```

SELECT * FROM STUDENT;

id	name	passport_id
20001	Praveen	40001
20002	Prasad	40002
20003	Kiran	40003

SELECT * FROM PASSPORT;

id	number
40001	E123456
40002	N123457
40003	L123890

SELECT * FROM STUDENT, PASSPORT WHERE STUDENT.PASSPORT_ID= PASSPORT.ID;

id	name	passport_id	id	number
20001	Praveen	40001	40001	E123456
20002	Prasad	40002	40002	N123457
20003	Kiran	40003	40003	L123890

One To Many/Many To One

Let's consider an example, A Course has multiple reviews (One To Many) and multiple reviews can be mapped to Single Course (Many To One)

By default for One To Many it is lazy fetch and for Many To One it is eager fetch.

Let's see how we can define this in Course and Review Entity classes.

```
@Entity
public class Review {

    @Id
    @GeneratedValue
    private Long id;

    private String description;

    private String rating;

    @ManyToOne(fetch=FetchType.EAGER) // by default it is eager fetch
    private Course course;
```

```

@Entity
@NamedQueries(value = { @NamedQuery(name = "query_get_all_courses", query = "Select c from Course c"),
    @NamedQuery(name = "query_get_like25", query = "Select c from Course c where name like '%25%') })
//@NamedQuery(name="query_get_all_courses",query="Select c from Course c")
@Cacheable
public class Course {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @CreationTimestamp
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime lastUpdatedAt;

    // @JsonIgnore
    @OneToMany(mappedBy="course") // by default it is lazy fetch
    private List<Review> reviews = new ArrayList<Review>();
}

```

SELECT * FROM COURSE;

id	created_date	last_updated_date	name
10001	2019-10-12 00:19:49	2019-10-12 00:19:49	CORE JAVA
10002	2019-10-12 00:19:49	2019-10-12 00:19:49	SPRING
10003	2019-10-12 00:19:49	2019-10-12 00:19:49	JPA HIBERNATE
10004	2019-10-12 00:19:49	2019-10-12 00:19:49	SQL in 25 Days
10005	2019-10-12 00:19:49	2019-10-12 00:19:49	Maven in 25 Days

SELECT * FROM REVIEW;

id	description	rating	course_id
50001	Great Course	FIVE	10003
50002	Wonderful Course	FOUR	10003
50003	Awesome Course	FIVE	10005

Many To Many

Let's consider an example, A student can register in multiple courses and each course can have multiple students.

By Default for Many to Many it is lazy fetch.

Let's see how we can define this in Student and Course Entity classes.

```
@Entity
@NamedQueries(value = { @NamedQuery(name = "query_get_all_courses", query = "Select c from Course c"),
    @NamedQuery(name = "query_get_like25", query = "Select c from Course c where name like '%25%'") })
//@NamedQuery(name="query_get_all_courses",query="Select c from Course c")
@Cacheable
public class Course {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @CreationTimestamp
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime lastUpdatedAt;

    @JsonIgnore
    @ManyToMany(mappedBy="courses") // by default it is lazy fetch
    private List<Student> students= new ArrayList<Student>();
}
```

```
@Entity
public class Student {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable=false)
    private String name;

    @ManyToMany // by default it is lazy fetch
    @JoinTable(name="STUDENT_COURSE",
        joinColumns= @JoinColumn(name="STUDENT_ID"),
        inverseJoinColumns=@JoinColumn(name="COURSE_ID"))
    public List<Course> courses= new ArrayList<Course>();
}
```

SELECT * FROM COURSE;

id	created_date	last_updated_date	name
10001	2019-10-12 00:19:49	2019-10-12 00:19:49	CORE JAVA
10002	2019-10-12 00:19:49	2019-10-12 00:19:49	SPRING
10003	2019-10-12 00:19:49	2019-10-12 00:19:49	JPA HIBERNATE
10004	2019-10-12 00:19:49	2019-10-12 00:19:49	SQL in 25 Days
10005	2019-10-12 00:19:49	2019-10-12 00:19:49	Maven in 25 Days

SELECT * FROM STUDENT;

id	name	passport_id
20001	Praveen	40001
20002	Prasad	40002
20003	Kiran	40003

As we provided for ManyToMany a new table will be created called STUDENT_COURSE and it will populate the students who are opted for courses.

SELECT * FROM STUDENT_COURSE;

student_id	course_id
20001	10003
20002	10003
20003	10003
20001	10005

You can find the complete code for JPA Hibernate relationships in <https://github.com/praveenoruganti/praveenoruganti-springboot-jpa-hibernate-master>

@JoinColumn vs mappedBy

mappedBy attribute is used to tell hibernate which entity is the owner side of the relation. This is used mostly in OneToMany and ManyToOne relationship. This is generally used for bidirectional relationship.

@JoinColumn is used for joining the columns and act as foreign key.

How to map composite key

- ✓ We use JPA's @Embeddable annotation to declare that a class is meant to be embedded by other entities.
- ✓ The @EmbeddedId is used to instruct Hibernate that the Employee entity uses a compound key.
- ✓ When we define the value object for the entity class we use @Embeddable.
- ✓ When we use value type object in entity class we use @Embedded

```
@Embeddable
@NoArgsConstructor
@AllArgsConstructor
@Data
@EqualsAndHashCode
public class EmployeeId implements Serializable {
    private static final long serialVersionUID = 5433558671941980196L;

    @Column(name = "company_id")
    private Long companyId;
    @Column(name = "employee_number")
    private Long employeeNumber;
}
```

22 | Praveen Oruganti

Blog: <https://praveenorugantitech.blogspot.com>

Facebook Group: <https://www.facebook.com/groups/268426377837151>

Github repo: <https://github.com/praveenoruganti>

Email : praveenorugantitech@gmail.com

```

@Entity(name = "Employee")
@Table(name = "employee")
@Data
public class Employee {
    @EmbeddedId
    private EmployeeId id;
    private String name;
}

@Entity(name = "Phone")
@Table(name = "phone")
@Data
public class Phone {
    @Id
    @Column(name = "number")
    private String number;
    @ManyToOne
    @JoinColumns({ @JoinColumn(name = "company_id", referencedColumnName = "company_id"),
        @JoinColumn(name = "employee_number", referencedColumnName = "employee_number") })
    private Employee employee;
}

```

JPA Using Hibernate Inheritance Mapping

We can map the inheritance hierarchy classes with the table of the database.

For example, let's consider FullTimeEmployee and PartTimeEmployee by extending abstract class Employee.

FullTimeEmployee works based on monthly salary and PartTimeEmployee works based on hourly wage.

Let's create abstract Entity class Employee

```

@Entity
@Data
@NoArgsConstructor
public abstract class Employee {
    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    public Employee(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return String.format("Employee[%s]", name);
    }
}

```

Let's create an entity class FullTimeEmployee who works based on monthly salary.

```
@Entity
public class FullTimeEmployee extends Employee {

    private BigDecimal salary;

    protected FullTimeEmployee() {

    }

    public FullTimeEmployee(String name, BigDecimal salary) {
        super(name);
        this.salary = salary;
    }

    public BigDecimal getSalary() {
        return salary;
    }

}
```

Let's create an entity class PartTimeEmployee who works based on hourly wage.

```
@Entity
public class PartTimeEmployee extends Employee {

    private BigDecimal hourlyWage;

    protected PartTimeEmployee() {

    }

    public PartTimeEmployee(String name, BigDecimal hourlyWage) {
        super(name);
        this.hourlyWage = hourlyWage;
    }

    public BigDecimal getHourlyWage() {
        return hourlyWage;
    }

}
```

You can choose between 4 strategies which map the inheritance structure of your domain model to different table structures.

1. Single Table Inheritance Strategy

By default, the strategy for inheritance is SINGLE_TABLE

The single table strategy maps all entities of the inheritance structure to the same database table. This approach makes polymorphic queries very efficient and provides the best performance.

But it also has some drawbacks. The attributes of all entities are mapped to the same database table. Each record uses only a subset of the available columns and sets the rest of them to null. You can, therefore, not use not null constraints on any column that isn't mapped to all entities. That can create data integrity issues, and your database administrator might not be too happy about it.

When you persist all entities in the same table, Hibernate needs a way to determine the entity class each record represents. This information is stored in a discriminator column which is not an entity attribute. You can either define the column name with a `@DiscriminatorColumn` annotation on the superclass or Hibernate will use `DTYPE` as its default name.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="EmployeeType")
@Data
@NoArgsConstructor
public abstract class Employee {
    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    public Employee(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return String.format("Employee[%s]", name);
    }
}
```

If you see now, `EMPLOYEE_TYPE` name is reflecting in the table.

The problem with single table strategy is there will be many null able columns.

SELECT * FROM EMPLOYEE;

id	name	salary	hourly_wage	employee_type
1	Praveen	100000.00	NULL	FullTimeEmployee
2	Naveen	NULL	50.00	PartTimeEmployee

2. Table per class Inheritance strategy

Each concrete subclass will be having one table for this approach

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@Data
@NoArgsConstructor
public abstract class Employee {
    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    public Employee(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return String.format("Employee[%s]", name);
    }
}
```

The problem with the table per class strategy is that the common columns are repeated in both the tables here.

```
SELECT * FROM FULL_TIME_EMPLOYEE;
```

ID	NAME	SALARY
1	Praveen	100000.00

(1 row, 11 ms)

```
SELECT * FROM PART_TIME_EMPLOYEE;
```

ID	NAME	HOURLY_WAGE
2	Naveen	50.00

(1 row, 5 ms)

3. Joined Inheritance Strategy

There will be separate tables will be created for super and sub classes.

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Data
@NoArgsConstructor
public abstract class Employee {
    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    public Employee(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return String.format("Employee[%s]", name);
    }
}
```

Problem with this is Join with tables

SELECT * FROM EMPLOYEE;

ID	NAME
1	Praveen
2	Naveen

(2 rows, 3 ms)

SELECT * FROM FULL_TIME_EMPLOYEE;

SALARY	ID
100000.00	1

(1 row, 6 ms)

SELECT * FROM PART_TIME_EMPLOYEE;

HOURLY_WAGE	ID
50.00	2

(1 row, 8 ms)

4. Mapped Super Class Inheritance Strategy

Here Employee will not be treated as Entity and a separate table will be created for each concrete class.

There is no inheritance Strategy is used here.

```
@MappedSuperclass
@Data
@NoArgConstructor
public abstract class Employee {
    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    public Employee(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return String.format("Employee[%s]", name);
    }
}

@Repository
@Transactional
public class EmployeeRepository {
    @Autowired
    EntityManager em;

    public void insert(Employee employee) {
        em.persist(employee);
    }

    public List<Employee> retrieveAllEmployees() {
        return em.createQuery("Select e from Employee e", Employee.class).getResultList();
    }

    public List<PartTimeEmployee> retrievePartTimeEmployees() {
        return em.createQuery("Select e from PartTimeEmployee e", PartTimeEmployee.class).getResultList();
    }

    public List<FullTimeEmployee> retrieveFullTimeEmployees() {
        return em.createQuery("Select e from FullTimeEmployee e", FullTimeEmployee.class).getResultList();
    }
}
```

27 | Praveen Oruganti

Blog: <https://praveenorugantitech.blogspot.com>

Facebook Group: <https://www.facebook.com/groups/268426377837151>

Github repo: <https://github.com/praveenoruganti>

Email : praveenorugantitech@gmail.com

SELECT * FROM FULL_TIME_EMPLOYEE;

ID	NAME	SALARY
1	Praveen	100000.00

(1 row, 10 ms)

SELECT * FROM PART_TIME_EMPLOYEE;

ID	NAME	HOURLY_WAGE
2	Naveen	50.00

(1 row, 28 ms)

We can use CommandLineRunner to test the above 4 strategies.

```
@SpringBootApplication
public class PraveenorugantiSpringbootJpaHibernateApplication implements CommandLineRunner {
    private Logger log = LoggerFactory.getLogger(this.getClass());

    @Autowired
    EmployeeRepository employeeRepository;

    public static void main(String[] args) {
        SpringApplication.run(PraveenorugantiSpringbootJpaHibernateApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {

        employeeRepository.insert(new FullTimeEmployee("Praveen", new BigDecimal("100000")));
        employeeRepository.insert(new PartTimeEmployee("Naveen", new BigDecimal("50")));

        log.info("PartTime Employees {}"+employeeRepository.retrievePartTimeEmployees());
        log.info("FullTime Employees {}"+employeeRepository.retrieveFullTimeEmployees());

    }
}
```

How to choose between 4 strategies??

- ✓ If you are really concerned about data integrity, we need to go with Joined inheritance strategy.
- ✓ If you are concerned about performance, we need to go with Single table inheritance strategy.
- ✓ I will not consider the other 2 options i.e.. Table per class and MappedSuperClass at all as there are repeated columns as that doesn't meet 3rd normal form.

You can find the complete code for JPA Hibernate inheritance strategies in <https://github.com/praveenoruganti/praveenoruganti-springboot-jpa-hibernate-master>

Spring Data JPA Using Hibernate

The problem with JPA Using Hibernate which we learnt in our above sections is that there are lot of duplication between Repositories.

The other problem is proliferation of data stores i.e... many are coming up like SQL, NO SQL etc.

Spring Data provides the abstraction support to handle the above problems.

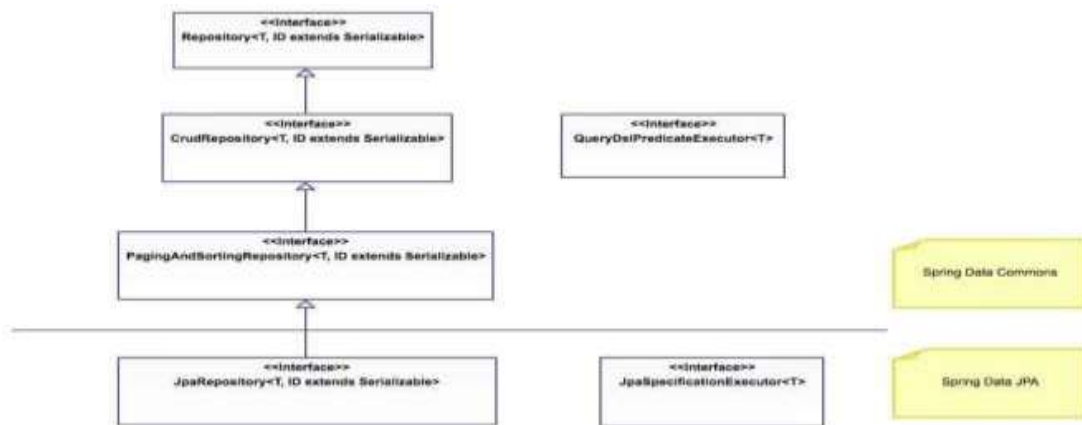
What is Spring Data JPA

- ✓ Provides repository support for the Java Persistence API (JPA).
- ✓ The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.
- ✓ Spring Data JPA is not a JPA provider. It is a library/framework that adds an extra layer of abstraction on the top of our JPA provider (like Hibernate).
- ✓ Hibernate is a JPA implementation, while Spring Data JPA is a JPA Data Access Abstraction.

Let's see how Hibernate, EclipseLink and Spring Data JPA categorizes using below diagram.



Core API



Where T represents domain type and ID represents type of domain types id
All CRUD methods are transactional by default.

Difference between CrudRepository and JpaRepository interfaces in Spring Data JPA

- ✓ 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.

Add the below dependency in pom.xml and by default Hibernate jars are available in spring data jpa

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Create interface CourseSpringDataRepository by extending JpaRepository<Course,Long>

```
package com.praveen.spring.jpa.repository;

import java.util.List;

public interface CourseSpringDataRepository extends JpaRepository<Course, Long> {

    List<Course> findByName(String name);

    List<Course> deleteByName(String name);

    @Query("Select c from Course c Where c.name like '%25 Days%'")
    List<Course> courseWith25DaysInName();

    @Query(value = "Select * from Course c Where c.name like '%25 Days'", nativeQuery = true)
    List<Course> courseWith25DaysInNameNative();

    @Query(value = "SELECT c from Course c where c.name LIKE '%' || :keyword || '%'",
    List<Course> search(@Param("keyword") String keyword);

}
```

Now create test class for testing the above methods

```
package com.praveen.spring.jpa.repository;

import static org.junit.Assert.assertEquals;

@RunWith(SpringRunner.class)
@SpringBootTest(classes=PraveenorugantiSpringbootJpaHibernateApplication.class)
public class CourseSpringDataRepositoryTest {

    private Logger log= LoggerFactory.getLogger(this.getClass());

    @Autowired
    public CourseSpringDataRepository courseSpringDataRepository;

    @Test
    @DirtiesContext
    public void findById_CoursePresent() {
        Optional<Course> courseOptional=courseSpringDataRepository.findById(10003L);
        assertTrue(courseOptional.isPresent());
    }

    @Test
    @DirtiesContext
    public void findByName() {
        log.info("FindByName -> {}",courseSpringDataRepository.findByName("JPA HIBERNATE"));
    }

    @Test
    @DirtiesContext
    public void courseWith25DaysInName() {
        log.info("courseWith25DaysInName -> {}",courseSpringDataRepository.courseWith25DaysInName());
    }

}
```

```

@Test
@DirtiesContext
public void courseWith25DaysInNameNative() {
    log.info("courseWith25DaysInNameNative -> {}", courseSpringDataRepository.courseWith25DaysInNameNative());
}

@Test
@DirtiesContext
public void search() {
    log.info("search for 25 Days -> {}", courseSpringDataRepository.search("25 Days"));
}

@Test
@DirtiesContext
public void save() {
    log.info("save Test is running");
    Optional<Course> course=courseSpringDataRepository.findById(10003L);
    //assertEquals("JPA HIBERNATE", course.get().getName());
    course.get().setName("SQL");
    courseSpringDataRepository.save(course.get());
    assertEquals("SQL", course.get().getName());
}

@Test
public void sort() {
    Sort sort= new Sort(Sort.Direction.DESC, "name");
    log.info("Sorted Courses -> {}", courseSpringDataRepository.findAll(sort));
    log.info("Count -> {}", courseSpringDataRepository.count());
}

@Test
public void pagination() {
    PageRequest pageRequest= PageRequest.of(0, 2);
    Page<Course> firstPage= courseSpringDataRepository.findAll(pageRequest);
    log.info("First Page -> {}", firstPage.getContent());

    Pageable secondPageable = firstPage.nextPageable();
    Page<Course> secondPage= courseSpringDataRepository.findAll(secondPageable);
    log.info("Second Page -> {}", secondPage.getContent());

    Pageable thirdPageable = secondPage.nextPageable();
    Page<Course> thirdPage= courseSpringDataRepository.findAll(thirdPageable);
    log.info("Third Page -> {}", thirdPage.getContent());
}
}

```


Spring Data JPA REST

To expose repository as rest we can use Spring Data JPA REST and this is not recommended for production

Add the below dependency in pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

Now add annotation @RepositoryRestResource(path="courses")

```
package com.praveen.spring.jpa.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import com.praveen.spring.jpa.entity.Course;

@RepositoryRestResource(path="courses")
public interface CourseSpringDataRepository extends JpaRepository<Course, Long> {
    List<Course> findByName(String name);

    List<Course> deleteByName(String name);

    @Query("Select c from Course c Where c.name like '%25 Days%'")
    List<Course> courseWith25DaysInName();

    @Query(value = "Select * from Course c Where c.name like '%25 Days'", nativeQuery = true)
    List<Course> courseWith25DaysInNameNative();

    @Query(value = "SELECT c from Course c where c.name LIKE '%' || :keyword || '%'",
    List<Course> search(@Param("keyword") String keyword);
}
```

To avoid infinite loop add @JsonIgnore for relationships

```
package com.praveen.spring.jpa.entity;

import java.time.LocalDateTime;

@Entity
@NamedQueries(value = { @NamedQuery(name = "query_get_all_courses", query = "Select c from Course c"),
    @NamedQuery(name = "query_get_like25", query = "Select c from Course c where name like '%25%') })
//@NamedQuery(name="query_get_all_courses",query="Select c from Course c")
@Cacheable
public class Course {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @CreationTimestamp
    private LocalDateTime createdAt;

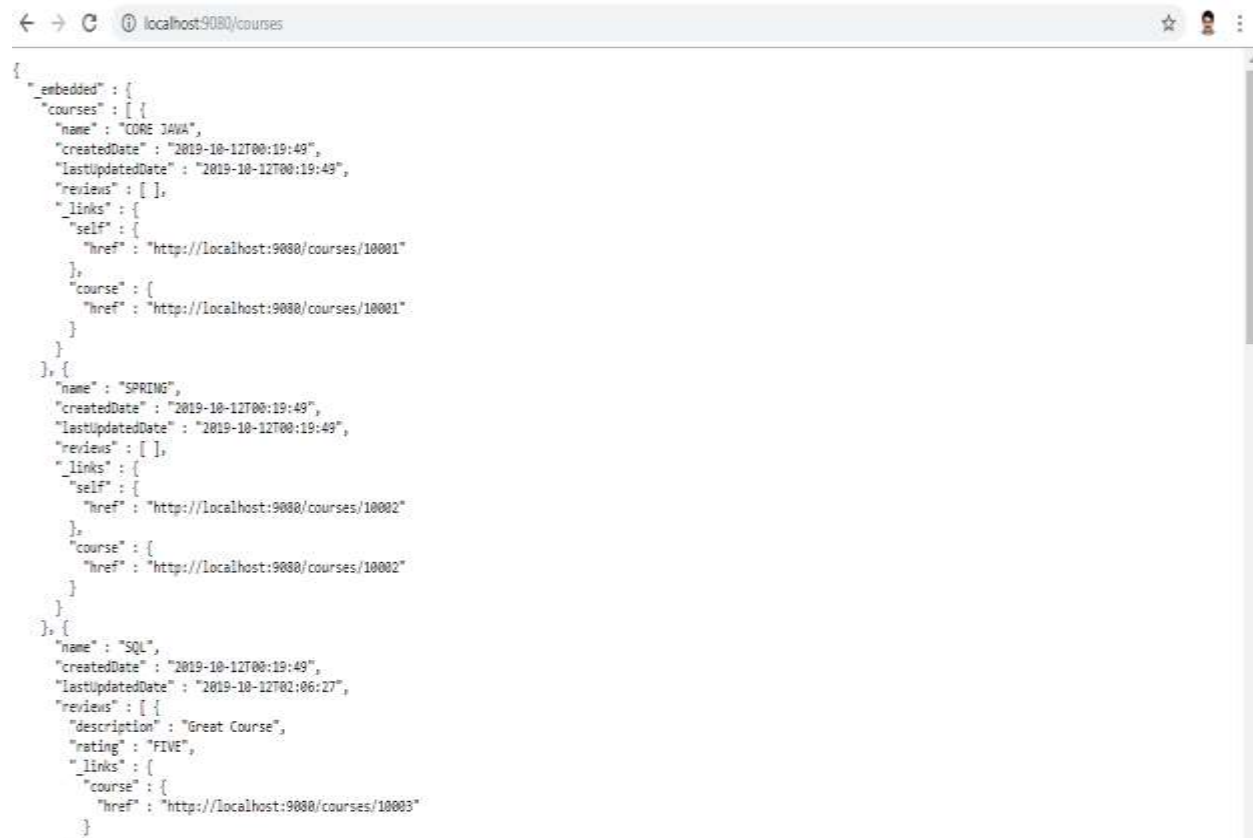
    @UpdateTimestamp
    private LocalDateTime lastUpdatedAt;

    // @JsonIgnore
    @OneToMany(mappedBy="course") // by default it is lazy fetch
    private List<Review> reviews = new ArrayList<Review>();

    @JsonIgnore
    @ManyToMany(mappedBy="courses") // by default it is lazy fetch
    private List<Student> students= new ArrayList<Student>();

    protected Course() {

    }
}
```



```
{
  "embedded": {
    "courses": [ {
      "name": "CORE JAVA",
      "createdAt": "2019-10-12T00:19:49",
      "lastUpdatedAt": "2019-10-12T00:19:49",
      "reviews": [ ],
      "_links": {
        "self": {
          "href": "http://localhost:9080/courses/10001"
        },
        "course": {
          "href": "http://localhost:9080/courses/10001"
        }
      }
    }, {
      "name": "SPRING",
      "createdAt": "2019-10-12T00:19:49",
      "lastUpdatedAt": "2019-10-12T00:19:49",
      "reviews": [ ],
      "_links": {
        "self": {
          "href": "http://localhost:9080/courses/10002"
        },
        "course": {
          "href": "http://localhost:9080/courses/10002"
        }
      }
    }, {
      "name": "SQL",
      "createdAt": "2019-10-12T00:19:49",
      "lastUpdatedAt": "2019-10-12T02:06:27",
      "reviews": [ {
        "description": "Great Course",
        "rating": "FIVE"
      } ],
      "_links": {
        "self": {
          "href": "http://localhost:9080/courses/10003"
        },
        "course": {
          "href": "http://localhost:9080/courses/10003"
        }
      }
    }
  ]
}
```

34 | Praveen Oruganti

Blog: <https://praveenorugantitech.blogspot.com>

Facebook Group: <https://www.facebook.com/groups/268426377837151>

Github repo: <https://github.com/praveenoruganti>

Email: praveenorugantitech@gmail.com

How to use Cache in JPA Hibernate

There are 2 types of Cache i.e.. First Level Cache and Second Level Cache.

- ✓ First Level Cache will be present in Persistent Context and it is available for that User request.
- ✓ Second Level Cache can be done using ehcache and generally not frequently changed data will be stored in second level cache.

We need to add the below dependency in pom.xml

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-ehcache</artifactId>
</dependency>
```

Make sure you configure below cache properties in application.yml

```
jpa:
  database-platform: org.hibernate.dialect.MySQLDialect
  show-sql: true
  properties:
    javax:
      persistence:
        sharedCache:
          mode: ENABLE_SELECTIVE
    hibernate:
      generate_statistics: true
#      ddl-auto: create
      format-sql: true
      batch_size: 10
      cache:
        use_second_level_cache: true
        region:
          factory_class: org.hibernate.cache.ehcache.EhCacheRegionFactory
```

For example, let's consider Course is not changing frequently.

Just we need to include @Cacheable on top of Course Entity

```
package com.praveen.spring.jpa.entity;

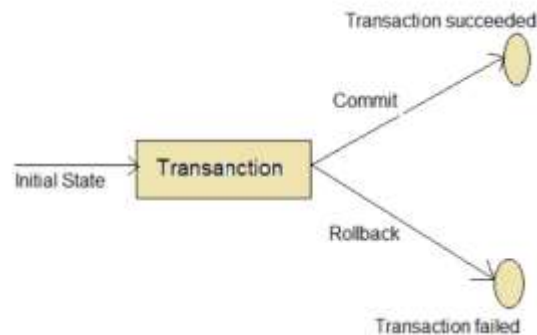
import java.time.LocalDateTime;

@Entity
@NamedQueries(value = { @NamedQuery(name = "query_get_all_courses", query = "Select c from Course c"),
  @NamedQuery(name = "query_get_like25", query = "Select c from Course c where name like '%25%'") })
//@NamedQuery(name="query_get_all_courses",query="Select c from Course c")
@Cacheable
public class Course {

  @Id
  @GeneratedValue
  private Long id;
```

Spring Transaction Management

A transaction is a logical unit of work that contains one or more statements. A transaction is an atomic unit. The effects of all the statements in a transaction can be either all committed or all rolled back.



Transaction management is an important part of enterprise applications to ensure data integrity and consistency.

The concept of transactions can be described as ACID(Atomicity, Consistency, Isolation, Durability) property:

Atomicity:

Atomicity requires that each transaction is “all or nothing” which means either the sequence of a transaction is successful or unsuccessful.

If one part of a transaction fails then entire transaction fails.

Consistency:

The database value should be consistent from state to state while any data written to the database for any combination of constraints, cascade, triggers etc.

Isolation:

Each transaction must be executed in isolation in concurrent environment to prevent data corruption.

Durability:

Once a transaction has been committed, the results of this transaction have to be made permanent even in the event of power loss, crashes, or errors.

The Spring transaction management

- ✓ We will go over on how does @Transactional really work under the hood.
- ✓ how to use features like propagation and isolation
- ✓ what are the main pitfalls and how to avoid them



Any bean in Spring is annotated with @Transactional will be intercepted by Proxy. This type of transaction is called declarative transactions. This proxy can be jdk proxy or cglib proxy.

There are 5 types of Isolation levels are available

	dirty reads	non-repeatable reads	phantom reads
READ_UNCOMMITTED	yes	yes	yes
READ_COMMITTED	no	yes	yes
REPEATABLE_READ	no	no	yes
SERIALIZABLE	no	no	no

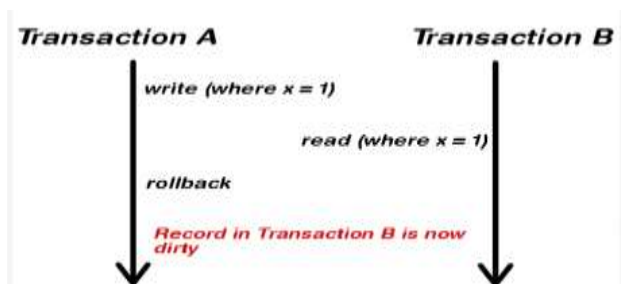
DEFAULT

Use the default isolation level of the underlying datastore. All other levels correspond to the JDBC isolation levels.

READ_UNCOMMITTED

This level allows a row changed by one transaction to be read by another transaction before any changes in that row have been committed (a "dirty read"). If any of the changes are rolled back, the second transaction will have retrieved an invalid row.

Dirty Read



In this example Transaction A writes a record. Meanwhile Transaction B reads that same record before Transaction A commits. Later Transaction A decides to rollback and now we have changes in Transaction B that are inconsistent. This is a dirty read. Transaction B was running in READ_UNCOMMITTED isolation level so it was able to read Transaction A changes before a commit occurred.

Note: READ_UNCOMMITTED is also vulnerable to non-repeatable reads and phantom reads.

READ COMMITTED

This level only prohibits a transaction from reading a row with uncommitted changes in it.

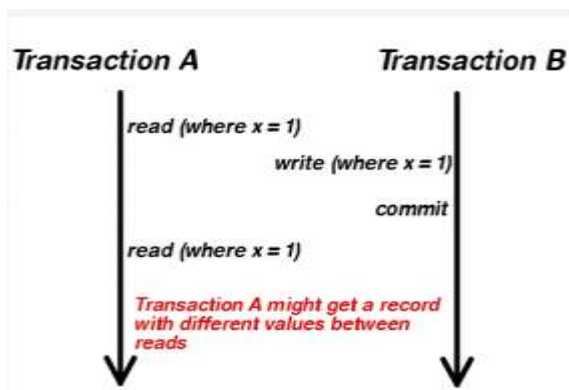
Isolation level in a transactional method

```
@Autowired
private TestDAO testDAO;

@Transactional(isolation=Isolation.READ_COMMITTED)
public void someTransactionalMethod(User user) {

    // Interact with testDAO
}
```

Non-repeatable read



In this example Transaction A reads some record. Then Transaction B writes that same record and commits. Later Transaction A reads that same record again and may get different values because Transaction B made changes to that record and committed. This is a non-repeatable read.

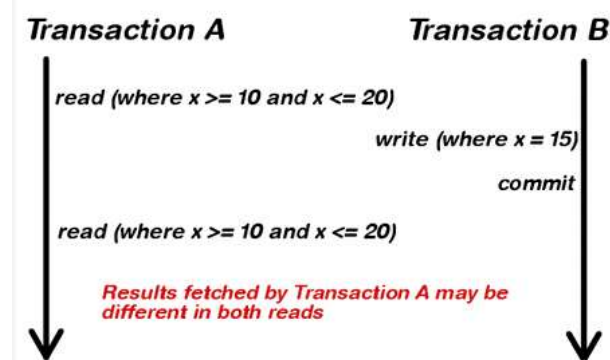
Note: READ_COMMITTED is also vulnerable to phantom reads.

REPEATABLE READ

This level prohibits a transaction from reading a row with uncommitted changes in it, and it also prohibits the situation where one transaction reads a row, a second transaction alters the row, and the first transaction rereads the row, getting different values the second time (a "non-repeatable read").

This eliminates both the dirty read and the non-repeatable read issues.

Phantom read



In this example Transaction A reads a range of records. Meanwhile Transaction B inserts a new record in the same range that Transaction A initially fetched and commits. Later Transaction A reads the same range again and will also get the record that Transaction B just inserted. This is a phantom read: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records)

SERIALIZABLE

This level includes the prohibitions in {`ISOLATION_REPEATABLE_READ`} and further prohibits the situation where one transaction reads all rows that satisfy a {`WHERE`} condition, a second transaction inserts a row that satisfies that {`WHERE`} condition, and the first transaction rereads for the same condition, retrieving the additional "phantom" row in the second read.

There are 7 types of propagation levels

REQUIRED

Support a current transaction, create a new one if none exists. Analogous to EJB transaction attribute of the same name. This is the default setting of a transaction annotation.

Outer bean

```
@Autowired
private TestDAO testDAO;

@Autowired
private InnerBean innerBean;

@Override
@Transactional(propagation=Propagation.REQUIRED)
public void testRequired(User user) {
    testDAO.insertUser(user);
    try{
        innerBean.testRequired();
    } catch(RuntimeException e){
        // handle exception
    }
}
```

Inner bean

```
@Override
@Transactional(propagation=Propagation.REQUIRED)
public void testRequired() {
    throw new RuntimeException("Rollback this transaction!");
}
```

Note that the inner method throws a `RuntimeException` and is annotated with `REQUIRED` behavior. This means that it will use the same transaction as the outer bean, so the outer transaction will fail to commit and will also rollback.

Note 1: The only exceptions that set a transaction to rollback state by default are the unchecked exceptions (like `RuntimeException`). If you want checked exceptions to also set transactions to rollback you must configure them to do so.

Note 2: When using declarative transactions, ie by using only annotations, and calling methods from the same bean directly (self-invocation), the `@Transactional` annotation will be ignored by the container. If you want to enable transaction management in self-invocations you must configure the transactions using aspect.

SUPPORTS

Support a current transaction, execute non-transactionally if none exists. Analogous to EJB transaction attribute of the same name.

MANDATORY

Support a current transaction, throw an exception if none exists. Analogous to EJB transaction attribute of the same name.

REQUIRES_NEW

Create a new transaction, and suspend the current transaction if one exists. Analogous to the EJB transaction attribute of the same name.

```
Outer bean

@Autowired
private TestDAO testDAO;

@Autowired
private InnerBean innerBean;

@Override
@Transactional(propagation=Propagation.REQUIRED)
public void testRequiresNew(User user) {
    testDAO.insertUser(user);
    try{
        innerBean.testRequiresNew();
    } catch(RuntimeException e){
        // handle exception
    }
}

Inner bean

@Override
@Transactional(propagation=Propagation.REQUIRES_NEW)
public void testRequiresNew() {
    throw new RuntimeException("Rollback this transaction!");
}
```

The inner method is annotated with `REQUIRES_NEW` and throws a `RuntimeException` so it will set its transaction to rollback but will not affect the outer transaction. The outer transaction is paused when the inner transaction starts and then resumes after the inner transaction is concluded. They run independently of each other so the outer transaction may commit successfully.

NOT_SUPPORTED

Execute non-transactionally, suspend the current transaction if one exists. Analogous to EJB transaction attribute of the same name.

NEVER

Execute non-transactionally, throw an exception if a transaction exists. Analogous to EJB transaction attribute of the same name.

NESTED

Execute within a nested transaction if a current transaction exists, behave like {@code REQUIRED} otherwise. There is no analogous feature in EJB.

Important points

- ✓ `@Transactional(isolation=Isolation.READ_COMMITTED,propagation=Propagation.REQUIRED,readOnly=false,timeout=100,rollbackFor=Exception.class)`
- ✓ `@Transactional` on a class applies to each method on the service. It is a shortcut. Typically, you can set `@Transactional(readOnly = true)` on a service class, if you know that all methods will access the repository layer. You can then override the behavior with `@Transactional` on methods performing changes in your model.
- ✓ Rolls back the `RuntimeException` but does not rollback the checked exception

JPA and Transaction Management

It's important to notice that JPA on itself does not provide any type of declarative transaction management. When using JPA outside of a dependency injection container, transactions need to be handled programmatically by the developer:

```
UserTransaction utx = entityManager.getTransaction();
try {
    utx.begin();
    businessLogic();
    utx.commit();
} catch (Exception ex) {
    utx.rollback();
    throw ex;
}
```

This way of managing transactions makes the scope of the transaction very clear in the code, but it has several disadvantages:

- ✓ it's repetitive and error prone
- ✓ any error can have a very high impact
- ✓ errors are hard to debug and reproduce
- ✓ this decreases the readability of the code base

Using Spring @Transactional

With Spring @Transactional, the above code gets reduced to simply this:

```
@Transactional
public void businessLogic() {
    ... use entity manager inside a transaction ...
}
```

This is much more convenient and readable, and is currently the recommended way to handle transactions in Spring.

By using @Transactional, many important aspects such as transaction propagation are handled automatically. In this case if another transactional method is called by businessLogic(), that method will have the option of joining the ongoing transaction.

One potential downside is that this powerful mechanism hides what is going on under the hood, making it hard to debug when things don't work.

What does @Transactional mean?

One of the key points about @Transactional is that there are two separate concepts to consider, each with its own scope and life cycle:

1. the persistence context
2. the database transaction

The transactional annotation itself defines the scope of a single database transaction.

The database transaction happens inside the scope of a persistence context.

The persistence context is in JPA the EntityManager, implemented internally using an Hibernate Session (when using Hibernate as the persistence provider).

The persistence context is just a synchronizer object that tracks the state of a limited set of Java objects and makes sure that changes on those objects are eventually persisted back into the database.

This is a very different notion than the one of a database transaction. One EntityManager can be used across several database transactions, and it actually often is.

When does an EntityManager span multiple database transactions?

The most frequent case is when the application is using the Open Session In View pattern to deal with lazy initialization exceptions, see this previous blog post for it's pros and cons.

In such case the queries that run in the view layer are in separate database transactions than the one used for the business logic, but they are made via the same entity manager.

Another case is when the persistence context is marked by the developer as `PersistenceContextType.EXTENDED`, which means that it can survive multiple requests.

What defines the EntityManager vs Transaction relation?

This is actually a choice of the application developer, but the most frequent way to use the JPA Entity Manager is with the "Entity Manager per application transaction" pattern. This is the most common way to inject an entity manager.

@PersistenceContext
private EntityManager em;

Here we are by default in "Entity Manager per transaction" mode. In this mode, if we use this Entity Manager inside a `@Transactional` method, then the method will run in a single database transaction.

How does @PersistenceContext work?

One question that comes to mind is, how can `@PersistenceContext` inject an entity manager only once at container startup time, given that entity managers are so short lived, and that there are usually multiple per request.

The answer is that it can't: EntityManager is an interface, and what gets injected in the spring bean is not the entity manager itself but a context aware proxy that will delegate to a concrete entity manager at runtime.

Usually the concrete class used for the proxy is `SharedEntityManagerInvocationHandler`, this can be confirmed with the help a debugger.

How does @Transactional work then?

The persistence context proxy that implements EntityManager is not the only

component needed for making declarative transaction management work. Actually three separate components are needed:

- ✓ The EntityManager Proxy itself
- ✓ The Transactional Aspect
- ✓ The Transaction Manager

The Transactional Aspect

The Transactional Aspect is an 'around' aspect that gets called both before and after the annotated business method. The concrete class for implementing the aspect is TransactionInterceptor.

The Transactional Aspect has two main responsibilities:

- ✓ At the 'before' moment, the aspect provides a hook point for determining if the business method about to be called should run in the scope of an ongoing database transaction, or if a new separate transaction should be started.
- ✓ At the 'after' moment, the aspect needs to decide if the transaction should be committed, rolled back or left running.
- ✓ At the 'before' moment the Transactional Aspect itself does not contain any decision logic, the decision to start a new transaction if needed is delegated to the Transaction Manager.

The Transaction Manager

The transaction manager needs to provide an answer to two questions:

- ✓ should a new Entity Manager be created?
- ✓ should a new database transaction be started?

This needs to be decided at the moment the Transactional Aspect 'before' logic is called.

The transaction manager will decide based on the fact that one transaction is already ongoing or not the propagation attribute of the transactional method (for example **REQUIRES_NEW** always starts a new transaction)

If the transaction manager decides to create a new transaction, then it will:

- ✓ create a new entity manager
- ✓ bind the entity manager to the current thread
- ✓ grab a connection from the DB connection pool
- ✓ bind the connection to the current thread

The entity manager and the connection are both bound to the current thread using ThreadLocal variables.

They are stored in the thread while the transaction is running, and it's up to the Transaction Manager to clean them up when no longer needed.

Any parts of the program that need the current entity manager or connection can retrieve them from the thread. One program component that does exactly that is the EntityManager proxy.

The EntityManager proxy

The EntityManager proxy (that we have introduced before) is the last piece of the puzzle. When the business method calls for example entityManager.persist(), this call is not invoking the entity manager directly. Instead the business method calls the proxy, which retrieves the current entity manager from the thread, where the Transaction Manager put it.

Knowing now what are the moving parts of the @Transactional mechanism, let's go over the usual Spring configuration needed to make this work.

Putting It All Together

Let's go over how to setup the three components needed to make the transactional annotation work correctly. We start by defining the entity manager factory. This will allow the injection of Entity Manager proxies via the persistence context annotation:

```
@Configuration
public class EntityManagerFactoriesConfiguration {
    @Autowired
    private DataSource dataSource;
    @Bean(name = "entityManagerFactory")
    public LocalContainerEntityManagerFactoryBean emf() {
        LocalContainerEntityManagerFactoryBean emf = ...
        emf.setDataSource(dataSource);
        emf.setPackagesToScan(new String[] {"your.package"});
        emf.setJpaVendorAdapter(
            new HibernateJpaVendorAdapter());
        return emf;
    }
}
```

The next step is to configure the Transaction Manager and to apply the Transactional Aspect in @Transactional annotated classes:

```
@Configuration
@EnableTransactionManagement
public class TransactionManagersConfig {
    @Autowired
    EntityManagerFactory emf;
    @Autowired
    private DataSource dataSource;
    @Bean(name = "transactionManager")
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager tm = new JpaTransactionManager();
        tm.setEntityManagerFactory(emf);
        tm.setDataSource(dataSource);
        return tm;
    }
}
```

The annotation @EnableTransactionManagement tells Spring that classes with the @Transactional annotation should be wrapped with the Transactional Aspect. With this the @Transactional is now ready to be used.

Important points

- ✓ Any change made within a transaction to an entity (retrieved within the same transaction) will automatically be populated to the database at the end of the transaction, without the need of explicit manual updates.
- ✓ Don't write redundant JPA calls or @Transactional just "for more safety". It may bring more risk than safety.
- ✓ @Transactional works only when the annotated method is public and invoked from another bean. Otherwise, the annotation will be silently ignored.
- ✓ As a rule of thumb: don't use readOnly = true parameter until it's really necessary.
- ✓ By default, only unchecked exceptions trigger rollbacks, checked exceptions do not. It can be customized with rollbackFor and noRollbackFor parameters.

You can refer code in <https://github.com/praveenoruganti/praveenoruganti-springboot-jpa-hibernate-master>

Please check out my other ebooks in

<https://github.com/praveenoruganti/PraveenOruganti-Tech-Ebooks>