What is JPA?

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.

JPA is a java EE specification(Interface) that defines how data persistence-related task are handled using ORM(object relational mapping)

There are multiple provider which provide implementation like Hibernate , Open JPA,JDO , EclipseLink etc .

For each technology the configuration basically consists of injecting DataSource bean into SessionFactory or EntityManagerFactory and helps in perfoming CRUD operations .

#JPA based repository uses EntityManagerfactory bean >> most convient way because JPA is specification so with little modification in future , we can migrate one ORM implementation to other .

#Hibernate based repository uses SessionFactory bean >> Hibernate is implementation of JPA.

Supports HQL and JPQL

Spring obtains the database connection through DataSource bean . Spring boot automatically create the datasource bean by reading the database details like driver , connection URl ,username and password present in application.properties

Graphical user interface

Description automatically generated

|  |  |
| --- | --- |
| **Units** | **Description** |
| **EntityManagerFactory** | This is a factory class of EntityManager. It creates and manages multiple EntityManager instances. |
| **EntityManager** | It is an Interface, it manages the persistence operations on objects. It works like factory for Query instance. |
| **Entity** | Entities are the persistence objects, stores as records in the database. |
| **EntityTransaction** | It has one-to-one relationship with EntityManager. For each EntityManager, operations are maintained by EntityTransaction class. |
| **Persistence** | This class contain static methods to obtain EntityManagerFactory instance. |
| **Query** | This interface is implemented by each JPA vendor to obtain relational objects that meet the criteria. |

JPARepositroy provided built in method:

A 1. findall() – returns all entity 2. findById() 3. Save(S Entity) 4. saveAll(Iterable<S> entities) 5.deleteById() 6. Delete(T entity) 7. Count() – return the number of entity 8.existById(ID id) - > checks whether an entity exist 8. dleleteAll() 9. findAllById(T entity) – return all entity 10. DeleteAll(T entity)

Q>. Pagination and sorting in JPA : it will help to find the given page and sorting of result . This is the best functionalities of jpa : ( see in google to know how to do )

Spring Data JPA – Query Approaches(@Query always take priority over other options)

Approaches : 1. Query creation based on the method name 2.Query creation using @NamedQuery : through naming convention 3. Using @Query : annotate the query method @Query

We can use given below method In interface which will exptends jpa repository

1. Query method name are derived by combing the property name of an entity with supported keywords such as “findby”,”readBy”or “queryBy”

Eg:

Aa a. Customer findByEmail(String email) , this method is equivalet to JPQL :slelect c from customer c where c.email=?

b. List<Customer> findByLastNameLike(String lastname) , this method is equal to JPQL : select c from customer c where c.lastName LIKE concat (‘%’,?,’%’)

c. . List<Customer> findByEmailOrContactNumber(String email, Stirng number) , this method is equal to JPQL : select c from customer c where c.email=? or c.contactNumber=?

d. List<Customer> findByFirstNameAndAddress\_City(String firstname, String city) , this method is equal to JPQL : select c from customer c where c.firstname=? and c;address.city=?

Note: Here there is two class Customer and Address , and Customer is dependent on Address , so Address class Is property of customer class . and city is property of Address class .

E .List<Customer> findByLastNameOrderByFirstASC (String lastname) , this method is equl to JPQL :

Select c from customer c where c.lastname=? order by c.firstName

f. List<customer> findByAddress\_CityIn(collection<String> cities) , this method is equal to JPQL : select c from customer c where c.address.city =?1

g.List<Customer> findByActiveTrue() , this method is equivalent to JPQL: select c from customer c where c.active=true

Note: Active is Boolean properties in customer class

h.List<Customer> findbyCreditPintsGreaterThanEqual(int points), this method is equivalent to JPQL: select c from customer c where c.creditPoints between ? 1

I .List<Customer> findbyCreditPintsGreaterThanEqual(int points1 , int point2), this method is equivalent to JPQL: select c from customer c where c.creditPoints between ? 1 and ? 2

1. @NamedQuery : - this is we use in entity class Like:

**@NamedQuery(name=”customer.findByAddress”,query=”select c from customer where c.address =?1”)**

**Public class Customer{**

**}**

And also we need to mention it in repository interface too .

**Public interface cusotmerRepository extends JpaRepository<Customer,long>{**

**Customer findByAddress(String address);**

**}**

1. @Query annotation: eg .

**Public interface cusotmerRepository extends JpaRepository<Customer,long>{**

**@Query(“select cus from Customer cus where cus.address=?1”)**

**Customer findByAddress(String address);**

**}**

Note : @Param in method in jpa repository interface :

 @Param annotation to give a method parameter a concrete name and bind the name in the query. That makes refactoring of code easier in case we have to add/remove additional parameters.

@Param works with both @Query and @NamedQuery.

Eg.

public interface EmployeeRepository extends JPARepositry<Employee, Long> {

@Query("SELECT e FROM Employee e WHERE e.dept = :dept AND "

+ "(SELECT COUNT(DISTINCT e2.salary) FROM Employee e2 "

+ "WHERE e.salary < e2.salary AND e2.dept = :dept) < :topSalNum "

+ "ORDER BY e.salary DESC")

List<Employee> findByDeptTopNSalaries(@Param("topSalNum") long topSalaryNum, @Param("dept") String dept);

}

# Spring data mongodb

Spring also support nosql database . MongoDB is noSql database . for this our custome repository interface extends MongoDBRepository . EG

*/\* ... \*/*

2@Document(collection = "books")

3public class Book {

4

5 @Id

6 private String id;

7 private String title;

The main parts of this class are the @Document annotation that indicates that this class represents a MongoDB Document of the books collection, and the @Id annotation which tells Spring that this field should be used as the unique identification for the document.

As already stated, this entity will hold basic information about a book publication, such as title, ISBN, author names, etc.

**Defining the base Repository**

Next step is defining the base interface for the Book repository.

1*/\* ... \*/*

2public interface BookRepository extends MongoRepository<Book, String> {

3

4 List<Book> findByTitleContainingOrderByTitle(String titleContains);

The repository extends MongoRepository interface, indicating Spring that this is a MongoDB specific repository and inheriting all of the methods available in the parent interfaces (PagingAndSortingRepository, CrudRepository…)

We are going to add a simple interface method findByTitleContainingOrderByTitle to highlight how Spring will auto-implement this method.