# **Q: What is JPA?**

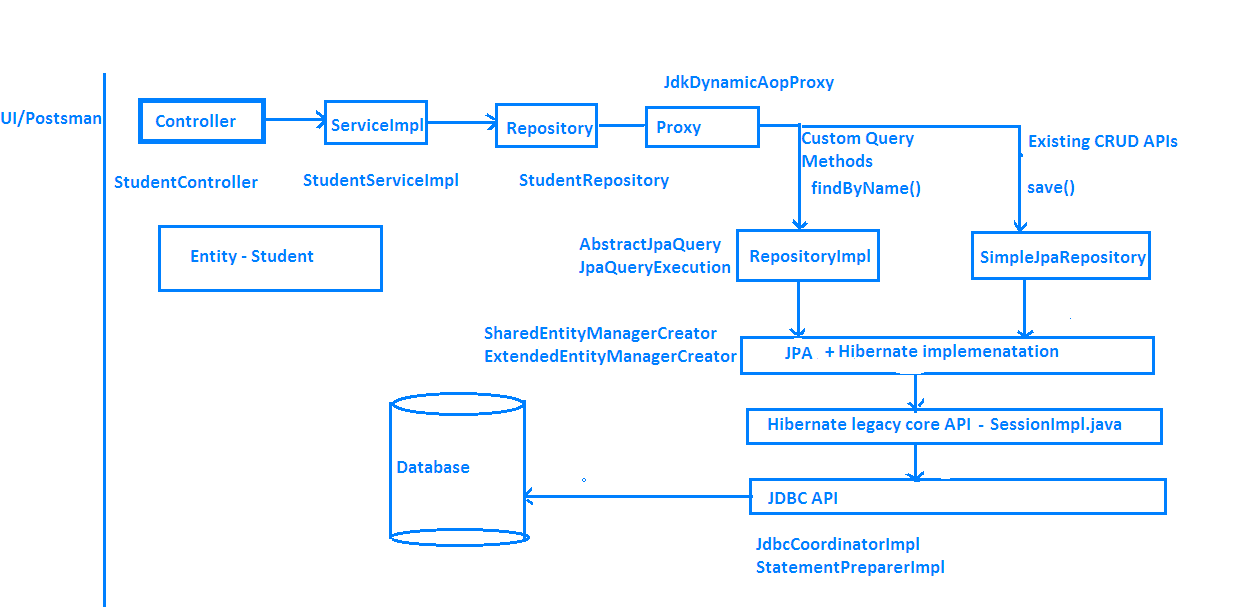
**A:** Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.  
The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition. Persistence in this context covers three areas:

the API itself, defined in the javax.persistence package

the Java Persistence Query Language (JPQL)

object/relational metadata

# **Explain the basic flow of Basic Spring Data JPA Flow.**

[](https://netsurfingzone.com/jpa/spring-data-jpa-interview-questions-and-answers/attachment/springdataflow1-2/)

# **Q: What's the difference between JPA and Hibernate ?**

**A:**JPA is the interface while Hibernate is the implementation.  
There are multiple popular implementations of JPA.

* Hibernate
* MyBatis
* TopLink
* Hibernate is one of the most popular open source implementations of the latest specification (JPA 2.1).
* JPA only describes rules and APIs and Hibernate implements these descriptions, but Hibernate has additional features that are not described in JPA.
* JPA defines the specification. It is an API.
* Hibernate provides additional features on top of JPA. But depending on them would mean a lock in to Hibernate.

# **What are the properties of an entity ?**

* Persistability: An object is called persistent if it is stored in the database and can be accessed anytime.
* Persistent Identity: In Java, each entity is unique and represents an object identity. Similarly, when the object identity is stored in a database, then it is represented as persistence identity. This object identity is equivalent to the primary key in the database.
* Transactionality: A transaction is a set of operations that either fail or succeed as a unit. Transactions are a fundamental part of persistence.
* Granularity: Entities should not be primitives, primitive wrappers or built-in objects with single dimensional state.

**Q: JPA EntityManager: Why use persist() over merge()?  
A:** Both persist and merge serve different purposes.

|  |  |
| --- | --- |
| Persist | Merge |
| Persist takes an entity instance, adds it to the context and makes that instance managed | Merge creates a new instance of your entity, copies the state from the supplied entity, and makes the new copy managed. |
| Insert a new register to the database | Find an attached object with the same id and update it. |
| You want the method always creates a new entity and never updates an entity. Otherwise, the method throws an exception as a consequence of primary key uniqueness violation. | You want the method either inserts or updates an entity in the database. |

**Q: Why does JPA have a @Transient annotation?  
Example - String test = "This is a test String and 'This is data we want'"**  
**A:** Java's transient keyword is used to denote that a field is not to be serialized, whereas JPA's @Transient annotation is used to indicate that a field is not to be persisted in the database, i.e. their semantics are different. Is it possible to set a default value for columns in JPA, and if, how is it done using annotations? Actually it is possible in JPA, although a little bit of a hack using the columnDefinition property of the @Column annotation, for example: @Column(name="Price", columnDefinition="Decimal(10,2) default '100.00'")

**In which case do you use JPA @JoinTable annotation?**  
**A**: It's the only solution to map a ManyToMany association : you need a join table between the to entities tables to map the association. It's also used for OneToMany (usually unidirectional) associations, when you don't want to add a foreign key in the table of the many side, and thus keep it independant of the one side. Search for @JoinTable in the hibernate documentation for explanations and examples.

**What is the naming convention for finder methods in the Spring data repository interface?**  
Answer:  This is another key feature of Spring Data JPA API which makes writing query method really easy. The finder method should use a special keyword, i.e. "find", followed by the name of the variable. For example, findByLastName().  
  
**Why is an interface not a class?**  
Answer: Interface is not a class because it does not contain concrete methods. It can contain only abstract methods.  
  
**Can we perform actual tasks like access, persist, and manage data with JPA?**  
Answer: No, we can't because JPA is only a Java specification.

# **What is the Spring data repository?**

Spring data repository is a very important feature of JPA. It helps in reducing a lot of boilerplate code. Moreover, it decreases the chance of errors significantly. This is also the key abstraction that is provided using the Repository interface. It takes the domain class to manage as well as the id type of the domain class as Type Arguments.  
These Repositories are Java interfaces that allow you as the developer to define a data access contract. The Spring Data JPA framework can then inspect that contract, and automatically build the interface implementation under the covers for you.

# **What is the difference between @Column and @Basic annotations in JPA?** A: @Basic signifies that an attribute is to be persisted and a standard mapping is to be used. It has parameters which allow you to specify whether the attribute is to be lazily loaded and whether it's nullable. @Column allows you to specify the name of the column in the database to which the attribute is to be persisted.

# **How can we create a custom repository in Spring data JPA?**

We can create custom repository extending any of these interfaces according to need.

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

public List<Student> findByNameAndRollNumber(String name, String rollNumber);

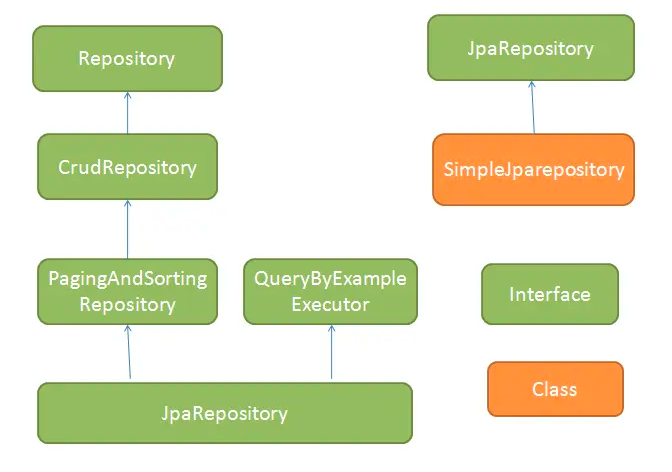
public List<Student> findByNameOrRollNumber(String name, String rollNumber);

public List<Student> findByNameAndRollNumberOrUniversity(String name, String rollNumber, String university);

}

The four major components are:

* Repository
* PagingAndSortingRepository
* CrudRepository
* JpaRepository
* QueryByExampleRepository
* Repository – Top-level interface defined in Spring Data Hierarchy. This is a marker interface i.e doesn’t contain any method. See more details about the Repository interface [here](https://netsurfingzone.com/jpa/difference-between-repository-and-crudrepository/).
* CrudRepository – The CrudRepository interface extends Repository interface, provides methods to perform CRUD operation. See more details about the CrudRepository interface [here](https://netsurfingzone.com/jpa/crudrepository-methods-example/).
* PagingAndSortingRepository – The PagingAndSortingRepository interface extends CrudRepository interface and provides additional methods to retrieve entities using the pagination and sorting.
* QueryByExampleExecutor – The QueryByExampleExecutor interface used to execute Query by Example.
* JpaRepository – The JpaRepository interface extends PagingAndSortingRepository and QueryByExampleExecutor interface, provides some additional batch methods. See more details [here](https://netsurfingzone.com/jpa/difference-between-crudrepository-and-jparepository-in-spring-data-jpa/).
* SimpleJpaRepository – The SimpleJpaRepository is the implementation class of the CrudRepository interface.
* QueryDslJpaRepository – This is a class.



**What is @Query used for?**  
Answer:  Spring Data API provides many ways to define SQL query which can be executed and Query annotations one of them. The @Query is an annotation that is used to execute both JPQL and native SQL queries.

**Give an example of using @Query annotation with JPQL.**  
Answer: Here is an example of @Query annotation from Spring Data Application which returns all active orders from the database:

@Query("**SELECT** **order** **FROM** Orders o **WHERE** o.Disabled= 0")

Collection<User> findAllActiveOrders();

and, here is another example, which returns matching employees from the database

@Query("select e from Employee e where se.name = ?1")

List<Employee> getEmployees(String name);

# **@NamedQuery vs @NamedNativeQuery in Spring Data JPA?**

@NamedQuery and @NamedNativeQuery annotations used with entity class.

@NamedQuery example.

package com.netsurfingzone.entity;

@Entity

@NamedQuery(name = "Student.findByName", query = "select s from Student s where s.name = ?1")

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

@NamedNativeQuery example.

package com.netsurfingzone.entity;

@Entity

@NamedNativeQuery(name = "Student.findByName", query = "select \* from Student where name = ?1", resultClass = Student.class)

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

# **Defining a JPA Entity Class?**

A JPA entity class is a Plain Old Java Object class.Eg. an ordinary Java class that is marked as having the ability to represent objects in the database. Conceptually this is similar to serializable classes, which are marked as having the ability to be serialized.

# **What is an Entitymanager?**

The EntityManager is an API that manages the lifecycle of entity instances. They manages a set of entities that are defined by a persistence unit. Each EntityManager instance is associated with a persistence context.  
The EntityManager API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities. The set of entities that can be managed by a given EntityManager instance is defined by a persistence unit.

# **What is difference between CrudRepository and JpaRepository interfaces in Spring Data JPA?**

CrudRepository provides CRUD functions. PagingAndSortingRepository provides methods to do pagination and sort records.  
JpaRepository provides JPA related methods such as flushing the persistence context and delete records in a batch.

| **CrudRepository** | **JpaRepository** |
| --- | --- |
| 1. CrudRepository extends Repository interface. | 1. JpaRepository extends PagingAndSortingRepository and QueryByExampleExecutor interface. |
| 2. CrudRepository provides methods to perform CRUD operations. | 2. JpaRepository provides additional methods like flush(), saveAndFlush(), deleteInBatch() etc. |
| 3. The saveAll(Iterable entities)  method of CrudRepository returns Iterable. | 3. The saveAll(Iterable entities)  method of JpaRepository returns List. |
| 4. If we have to perform mainly CRUD operation, define our repository using CrudRepository. | 4. If we have to perform CRUD as well as Batch operation define our repository extending JpaRepository. |

# **Difference between Repository and CrudRepository in Spring Data JPA?**

Repository interface.

The Repository is a top-level interface in hierarchy.

The Repository is a marker interface. It doesn’t have any method.

Repository interface has been defined as below.

@Indexed

public interface Repository<T, ID> {

}

CrudRepository interface.

The CrudRepository extends Repository interface. It has below methods to perform CRUD operation.

save(S entity)  
saveAll(Iterable<S> entities)  
findById(ID id)  
existsById(ID id)  
findAll()  
findAllById(Iterable<ID> ids)  
count()  
deleteById(ID id)  
delete(T entity)  
delete(Iterable<? extends T> entities)  
deleteAll()

# **How CrudRepository save() methods internally works in Spring Data JPA?**

The CrudRepository’s save() method is used to perform save as well as update operation both. The implementation has been given in SimpleJpaRepository.java, where persist() and merge() is getting called. If we try to save entity first time then persist() method will get invoked and if we try to update the same entity merge() will get invoked.

public S save(S entity) {

if (entityInformation.isNew(entity)) {

em.persist(entity);

return entity;

} else {

return em.merge(entity);

}

}

# Tell something about the CrudRepository saveAll() method.

The CrudRepository saveAll() method used to save multiple entities and internally annotated with @Transactional annotation. It internally uses save() method only as below.

@Transactional

public <S extends T> List<S> saveAll(Iterable<S> entities) {

List<S> result = new ArrayList<S>();

for (S entity : entities) {

result.add(save(entity));

}

return result;

}

# **How to write a query method for sorting using Spring Data JPA?**

Consider we have the following entity.

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

The first scenario **–** Retrieve the data on the basis of one field(i.e university) and sort on the basis of another field(name).

public List<Student> findByUniversity(String university) {  
List<Student> response = studentRepository.findByUniversityOrderByNameAsc(university);  
return response;  
}

The second scenario**–** Retrieve all record(rows)  and sort on the basis of some field(name).

public List<Student> findAll() {  
List<Student> response = (List<Student>) studentRepository.findAllByOrderByNameAsc();  
return response;  
}

# **How to implement projection using Spring Data JPA?**

We want to fetch any single column.

Define Interface with some method.

package com.netsurfingzone.repository;

public interface StudentName {

String getName();

}

Define Repository as below.

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

List<StudentName> findAllBy();

}

# **How to write query using @NamedQueries and @NamedNativeQueries in Spring Data JPA.**

Both @NamedQueries and @NamedNativeQueries annotations used with the entity.

Using @NamedQueries.

package com.netsurfingzone.entity;

@Entity

@NamedQueries({ @NamedQuery(name = "Student.findByName1", query = "select s from Student s where s.name = ?1"),

@NamedQuery(name = "Student.findByNameAndRollNumber", query = "select s from Student s where s.name = ?1 and s.rollNumber = ?2"),

@NamedQuery(name = "Student.findByNameOrRollNumber", query = "select s from Student s where s.name = ?1 or s.rollNumber = ?2") })

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

Using @NamedNativeQueries.

package com.netsurfingzone.entity;

@Entity

@NamedNativeQueries({

@NamedNativeQuery(name = "Student.findByName", query = "select \* from Student where name = ?1", resultClass = Student.class),

@NamedNativeQuery(name = "Student.findByNameAndRollNumber", query = "select \* from Student where name = ?1 and roll\_number = ?2", resultClass = Student.class),

@NamedNativeQuery(name = "Student.findByNameOrRollNumber", query = "select \* from Student where name = ?1 or roll\_number = ?2", resultClass = Student.class) })

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

Corresponding repository methods.

@Repository

public interface StudentRepository extends CrudRepository<Student, Serializable> {

List<Student> findByName(String name);

List<Student> findByNameAndRollNumber(String name, String rollNumber);

List<Student> findByNameOrRollNumber(String name, String rollNumber);

}

# **Difference between delete() vs deleteInBatch() Methods in Spring Data JPA.**

The delete() method has been defined in the CrudRepository interface with the following signature.

void delete(T entity);

The deleteInBatch() has been defined in the JpaRepository interface with the following signature.

void deleteInBatch(Iterable<T> entities);

The delete() method internally uses EntityManager’s remove() method as below.

@Transactional

public void delete(T entity) {

em.remove(em.contains(entity) ? entity : em.merge(entity));

}

The deleteInBatch() internally implemented as below.

public void deleteInBatch(Iterable<T> entities) {

applyAndBind(getQueryString(DELETE\_ALL\_QUERY\_STRING, entityInformation.getEntityName()), entities, em)

.executeUpdate();

}

Observe both methods implementation. The first one i.e delete() internally uses remove() method. Before call remove() method it calls contains()/merge() method. Then flow will go inside SessionImpl.java class(fireDelete() method) and a lot of stuff happens.

On the other hand, deleteInBatch() prepares the query and collect some other information and directly calls the executeUpdate() method.

With the help of delete() method, we can delete a single record at a time whereas using deleteInBatch() we can delete multiple records.

The delete() method is a little slower as compare deleteInBatch() as delete() does some extra stuff than deleteInBatch().

# **Difference between deleteAll() Vs deleteAllInBatch() in Spring Data JPA.**

The deleteAll() method is defined in CrudRepository has two overloaded versions.

First one – void deleteAll(Iterable<? extends T> entities)

Using the above method we can delete all entities which we pass as request data. The deleteAll() internally use delete() method only.

public void deleteAll(Iterable<? extends T> entities) {

for (T entity : entities) {

delete(entity);

}

}

Spring Data JPA deleteAll() Vs deleteAllInBatch()

Second one – void deleteAll()

The above one will delete all records that belong to that repository. The deleteAll() internally uses findAll() and delete() method as below.

public void deleteAll() {

for (T element : findAll()) {

delete(element);

}

}

The query generated in case of deleteAll(). Suppose we are deleting twelve entities.

The deleteAllInBatch() method is defined in the JpaRepository interface as below.

void deleteAllInBatch()

Using deleteAllInBatch() method we can delete all entities from database. No need to pass entities as parameters.For example, let’s say we have an entity Student.java and also we have a corresponding repository class called StudentRepository.java something like below.

package com.netsurfingzone.repository;

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

}

If we want to delete all student records we need to do studentRepository.deleteAllInBatch().  See the below sample code.

@Transactional

public void deleteAllInBatch() {

studentRepository.deleteAllInBatch();

}

Note – Internally deleteAllInBatch()   uses createQuery() method as below.

public void deleteAllInBatch() {

em.createQuery(getDeleteAllQueryString()).executeUpdate();

}

In case of deleteAllInBatch() the query will generate as below.

Hibernate: delete from student

Both deleteAll() and deleteAllInBatch() use to delete all entities. The deleteAll() internally uses CrudRepository delete() method which further uses EntityManger remove() method which perform some other stuff. But in case of deleteAllInBatch(), EntityManger createQuery() method will preapare the query and it directly calls executeUpdate() method.

# **What will happen when we define wrong Query Methods in Spring Data JPA?**

Suppose we have an entity called Student.java and Student class has fields called id and name. We want all student records for the given name.

Our query method would be List<Student> findByName(String name);

Just think by mistake developer writes findName(). instead of findByName(). In this case, while deployment we will get an error.

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name ‘studentController’: Unsatisfied dependency expressed through field ‘studentService’; nested exception is org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name ‘studentServiceImpl’: Unsatisfied dependency expressed through field ‘studentRepository’; nested exception is org.springframework.beans.factory.BeanCreationException: Error creating bean with name ‘studentRepository’: Invocation of init method failed; nested exception is java.lang.IllegalArgumentException: Failed to create query for method public abstract java.util.List com.netsurfingzone.repository.StudentRepository.findName(java.lang.String)! No property findName found for type Student!

Spring scans all repository methods while deployment itself and checks query methods have been written properly or not.

# **How to define case insensitive search Query Methods in Spring Data JPA?**

public List<Student> findByNameIgnoreCase(String name);

# **List of important keywords and corresponding Query Methods.**

| **Keyword** | **Query methods** | **JPQL** |
| --- | --- | --- |
| And | findByLastnameAndFirstname | ...where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | ...where x.lastname = ?1 or x.firstname = ?2 |
| Is, Equals | findByFirstnameEquals | ...where x.firstname = ?1 |
| Between | findByStartDateBetween | ...where x.startDate between ?1 and ? |
| LessThan | findByAgeLessThan | ...where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | ...where x.age <= ?1< td> |
| GreaterThan | findByAgeGreaterThan | ...where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | ...where x.age >= ?1 |
| After | findByStartDateAfter | ...where x.startDate > ?1 |
| Before | findByStartDateBefore | ...where x.startDate < ?1 |
| IsNull | findByAgeIsNull | ...where x.age is null |
| IsNotNull, NotNull | findByAge(Is)NotNull | ...where x.age not null |
| Like | findByFirstnameLike | ...where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | ...where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | ...where x.firstname like ?1 (parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | ...where x.firstname like ?1 (parameter bound with prepended %) |
| Containing | findByFirstnameContaining | ...where x.firstname like ?1 (parameter bound wrapped in %) |
| OrderBy | findByAgeOrderByLastnameDesc | ...where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | ...where x.lastname <> ?1 |
| In | findByAgeIn(Collection ages) | ...where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection ages) | ...where x.age not in ?1 |
| True | findByActiveTrue() | ...where x.active = true |
| False | findByActiveFalse() | ...where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | ...where UPPER(x.firstame) = UPPER(?1) |

# **What are the important methods of Crudrepository to perform CRUD operations in Spring Data JPA?**

The Crudrepository contains below methods.

* save(S entity) – Used to save a single entity at a time. See an example of the save() method [here](https://netsurfingzone.com/hibernate/spring-data-crudrepository-save-method/).
* saveAll(Iterable<S> – we can save multiple entities at a time. See a complete example of the CrudRepository saveAll() method from scratch [here](https://netsurfingzone.com/hibernate/spring-data-crudrepository-saveall-and-findall/).
* findById(ID id) – Use to get entity basis of id. See an example of the findById() method [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-finbyid/).
* existsById(ID id) – Used to check whether an entity is already exited in DB for given Id. See an example of the existsById() method [here](https://netsurfingzone.com/jpa/spring-data-crudrepository-existsbyid/).
* findAll() – Find all entity of particular type. See an example of the findAll() [here](https://netsurfingzone.com/hibernate/spring-data-crudrepository-saveall-and-findall/).
* findAllById(Iterable<ID> ids)  – Return all entity of given ids. See an example [here](https://netsurfingzone.com/jpa/crudrepository-findallbyid-example-using-spring-boot/).
* count() – Returns the number of entities. See an example of the count() method [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-count-example/).
* deleteById(ID id) – Delete the entity on basis of id. See a complete example deleteById() method [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-deletebyid-example/).
* delete(T entity) – delete the entity which one we are passing. See a complete example [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-delete-and-deleteall/).
* delete(Iterable<? extends T> entities) – delete multiple entities which we are passing.  See a complete example [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-delete-and-deleteall/).
* deleteAll() – delete all entities. See a complete example [here](https://netsurfingzone.com/jpa/spring-data-jpa-crudrepository-delete-and-deleteall/).

# **How you will write custom method in the repository in Spring Data JPA? What are rules to define Query methods(query creation from method names)?**

Consider we have an entity called Student.java and we have some records in the database as below.

Student.java

package com.netsurfingzone.entity;

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

}

To write query methods first we need to define repository interface.

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

}

Rules to define Query methods.

Rule 1 – The name of the query method must start with findBy or getBy  or queryBy or countBy or readBy prefix. The findBy is mostly used by the developer.

For example findByName(String name), getByName(String name), queryByName(String name), countByName(String name), readByName(String name),

Note – All the above query methods will return all students whose name is “what ever name we provide as parameter”.

Rule 2 – The first character of field name should capital letter. Although if we write the first character of the field in small then it will work but we should use camelcase for the method name.

Both are valid query method defined below but we should follow the first way.

public List<Student> findByName(String name);

public List<Student> findByname(String name);

Rule 3 – While using findBy or getBy or queryBy or countBy or readBy the character B must be in capital letter, else we will get an exception while deployment.

Caused by: org.springframework.data.mapping.PropertyReferenceException: No property querybyName found for type Student!

Invalid query method.

public List<Student> findbyName(String name);

Rule 4 – We can write the query method using multiple fields using predefined keywords(eg. And, Or etc) but these keywords are case sensitive. We must use “And” instead of “and”.

**Write query method using @Query.**

Writing JPQL using Spring Data Jpa @Query.

@Query("select s from Student s where s.name = ?1")  
List<Student> getStudents(String name);

See a complete example of the JPQL using Spring Boot and Oracle [here](https://netsurfingzone.com/jpa/spring-data-jpa-jpql-and-native-query-example/).

# How to write named parameters in Spring Data JPA?

@Param and @Query annotations used to define Named Parameters.

@Query("select s from Student s where s.name = :name")  
List<Student> findByName(@Param("name") String name);

@Query("select s from Student s where s.name = :name")  
List<Student> findByName(@Param("name") String name);

@Query("select s from Student s where s.name = :name and s.rollNumber = :rollNumber")  
List<Student> findByNameAndRollNumber(@Param("name") String name, @Param("rollNumber") String rollNumber);

# **What data types are allowed in the Attributes of the Entity Class?**

The fields or properties must be of the following Java language types:

* Java primitive types
* java.lang.String
* java.math.BigInteger
* java.math.BigDecimal
* java.util.Date
* java.util.Calendar
* java.sql.Date
* java.sql.Time
* java.sql.TimeStamp

# **What is the Embeddable class?**

Embeddable classes are used to represent the state of an entity but don't have a persistent identity of their own, unlike entity classes. Instances of an embeddable class share the identity of the entity that owns it.It exist only as the state of another entity.  
In general, such a class serves to make the definition of common attributes for several Entity, we can assume that JPA simply embeds into the Entity instead of an object of this class, the attributes it contains.

# **What requirements does JPA set for Embeddable classes?**

Such classes must satisfy the same rules as the Entity classes, except that they do not have to contain a primary key and be marked with the Entity annotation.  
The Embeddable class must be marked with the Embeddable annotation or described in the XML configuration file JPA.

**What is PlatformTransactionMangaer?**  
Answer: PlatformTransactionMangaer is an interface that extends TransactionManager. It is the central interface in Spring's transaction infrastructure.

**Differentiate between findById() and getOne().**  
Answer: The findById() is available in CrudRepository while getOne() is available in JpaRepository. The findById() returns null if record does not exist while the getOne() will throw an exception called EntityNotFoundException. 

|  |  |
| --- | --- |
| findById() | getOne() |
| 1. The findById() method is available in CrudRepository interface. | 1. The getOne() method is available in JpaRepositpry interface. |
| 2. The findById() method will return null if the record doesn’t exist in the database. | 2. The getOne() method throw EntityNotFoundException if the record doesn’t exist in the database. |
| 3. Internally findById() method use EntityManger find() method. | 3. Internally getOne() method use EntityManger getReference() method. |
| 4. Calling findById() returns a eager fetched entity. | 4. Calling getOne() returns a lazily fetched entity. |

# **What are the constraints on an entity class?**

An entity class must fulfill the following requirements:

* The class must have a no-argument constructor.
* The class can't be final.
* The class must be annotated with @Entity annotation.
* The class must implement a Serializable interface if value passes an empty instance as a detached object.

# **What type of objects can be stored in the JPA collections mapping?**

Following are the type of objects that JPA allows to store: -

* Basic Types
* Entities
* Embeddable

# **What type of collections can be used in JPA?**

To store multivalued entity associations and a collection of objects, following types of Java collections is used: -

* List
* Set
* Map

# **What are the types of cascade supported by JPA?**

Following is the list of cascade type: -

* **PERSIST:** In this cascade operation, if the parent entity is persisted then all its related entity will also be persisted.
* **MERGE:** In this cascade operation, if the parent entity is merged, then all its related entity will also be merged.
* **DETACH:** In this cascade operation, if the parent entity is detached, then all its related entity will also be detached.
* **REFRESH:** In this cascade operation, if the parent entity is refreshed, then all its related entity will also be refreshed.
* **REMOVE:** In this cascade operation, if the parent entity is removed, then all its related entity will also be removed.
* **ALL** In this case, all the above cascade operations can be applied to the entities related to the parent entity.

# **What is JPQL?**

The Java Persistence Query language (JPQL) is a part of JPA specification that defines searches against persistence entities. It is an object-oriented query language which is used to perform database operations on persistent entities. Instead of the database table, JPQL uses entity object model to operate the SQL queries. Here, the role of JPA is to transform JPQL into SQL. Thus, it provides an easy platform for developers to handle SQL tasks. JPQL is an extension of Entity JavaBeans Query Language (EJBQL).

# **29) What are the features of JPQL?**

Some of the essential features of JPQL are: -

* It is simple and robust.
* It is a platform-independent query language.
* JPQL queries can be declared statically into metadata or can also be dynamically built in code.
* It can be used with any database such as MySQL, Oracle.

# **30) What is the Criteria API?**

The Criteria API is a specification that provides type-safe and portable criteria queries written using Java programming language APIs. It is one of the most common ways of constructing queries for entities and their persistent state. It is just an alternative method for defining JPA queries. Criteria API defines a platform-independent criteria queries, written in Java programming language. It was introduced in JPA 2.0. The main purpose behind this is to provide a type-safe way to express a query.

# **What are the different types of identifier generation?**

Following are the types of id generation strategy required to specify with @GeneratedValue annotation: -

* Automatic Id generation - In this case, the application doesn't care about the kind of id generation and hand over this task to the provider. If any value is not specified explicitly, the generation type defaults to auto.
* Id generation using a table - The identifiers can also be generated using a database table.
* Id generation using a database sequence - Databases support an internal mechanism for id generation called sequences. To customize the database sequence name, we can use the JPA @SequenceGenerator annotation.
* Id generation using a database identity - In this approach, whenever a row is inserted into the table, a unique identifier is assigned to the identity column that can be used to generate the identifiers for the objects.

# **Explain persistence life cycle of an object?**

In persistence life cycle, the object lies in the following states: -

* Transient - The object is called to be in the transient state when it is just declared by using the new keyword. When an object remains in the transient state, it doesn't contain any identifier(primary key) in the database.
* Persistence - In this state, an object is associated with the session and either saved to a database or retrieved from the database. When an object remains in the persistence state, It contains a row of the database and consists of an identifier value. We can make an object persistent by associating it with the hibernate session.
* Detached - The object enters into a detached state when the hibernate session is closed. The changes made to the detached objects are not saved to the database.

# **What is an orphan removal in mappings?**

If a target entity in one-to-one or one-to-many mapping is removed from the mapping, then remove operation can be cascaded to the target entity. Such target entities are known as orphans, and the orphanRemoval attribute can be used to specify that orphaned entities should be removed.

# **What are the steps to persist an entity object?**

The following steps are performed to persist an entity object.

* Create an entity manager factory object. The **EntityManagerFactory** interface present in **java.persistence** package is used to provide an entity manager.
  1. EntityManagerFactory emf=Persistence.createEntityManagerFactory("Student\_details");
* Obtain an entity manager from the factory.
  1. EntityManager em=emf.createEntityManager();
  2. Initialize an entity manager.
  3. em.getTransaction().begin();
  4. Persist the data into the relational database.
  5. em.persist(s1);
  6. Closing the transaction
  7. em.getTransaction().commit();
  8. Release the factory resources.
  9. emf.close();
  10. em.close();

# **What is the difference between FetchType.Eager and FetchType.Lazy?**

FetchType attribute indicates how whether records will be eagerly or lazily loaded from the database. When records are eagerly loaded, JPA returns these objects regardless of whether they are accessed by the client or not. When records are lazily loaded the actual objects are only retrieved when directly accessed. This can save memory and processing when appropriate.

# **How to enable Spring Data JPA features.**

First, we need to define some configuration class let’s say JpaConfig.java and then we need to use @EnableJpaRepositories annotation with that class. Also, provide basePackages value where (we have our repositories). Let’s see below example.

package com.netsurfingzone.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@Configuration

@EnableJpaRepositories(basePackages = "com.netsurfingzone.repository")

public class JpaConfig {

}

# **Define a simple repository method which returns all records/entities for a given name?**

We can define a repository or query method as below. All three methods would do the same task.

import java.io.Serializable;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.netsurfingzone.entity.Student;

@Repository

public interface StudentRepository extends JpaRepository<Student, Serializable> {

public List<Student> findByName(String name);

public List<Student> findByNameIs(String name);

public List<Student> findByNameEquals(String name);

}

# **How to define Query Methods for the nested property.**

Consider we have two entities Student.java and Address.java. Student and Address entities are in one to one relationship and we want to fetch all students from the database who belongs to city pune.

Student.java

package com.netsurfingzone.entity;

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "name")

private String name;

@Column(name = "roll\_number")

private String rollNumber;

@Column(name = "university")

String university;

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "address\_id")

Address address;

}

Address.java

package com.netsurfingzone.entity;

@Entity

public class Address {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int id;

@Column(name = "house\_number")

private String houseNumber;

@Column(name = "city")

private String city;

}

Now we want all Students records which belong to city pune.

Query method to fetch all students who belong to city pune using Spring Data JPA.

List<Student> findByAddressCity(String city);