**JPA Repository Interface**

Extending these interfaces allows Spring to find your repository interfaces and create proxy objects for them, that you can inject later on into your beans.

It provides you with methods that allow you to perform some common operations without the need to declare these methods yourself.

**PagingAndSortingRepository** extends **CrudRepository**

**JpaRepository** extends **PagingAndSortingRepository**

The **CrudRepository** interface provides methods for CRUD operations, so it allows you to create, read, update and delete records without having to define your own methods. The **PagingAndSortingRepository** provides additional methods to retrieve entities using pagination and sorting. Finally the JpaRepository add some more functionality that is specific to JPA.

In short JpaRepository

* has additional JPA specific methods that support for example [Query By Example](http://docs.spring.io/spring-data/jpa/docs/current/reference/html/#query-by-example) , deleting in batches, manual flushing changes to database
* querying methods return List's instead of Iterable's

**Query methods** are methods that find information from the database and are declared on the repository interface. For example, if we want to create a database query that finds the Todo object that has a specific id, we can create the query method by adding the findById() method to the TodoRepositoryinterface.

We can create query methods that use this strategy by following these rules:

* The name of our query method must start with one of the following prefixes: find…By, read…By, query…By, count…By, and get…By.
* If we want to limit the number of returned query results, we can add the First or the Top keyword before the first By word. If we want to get more than one result, we have to append the optional numeric value to the First and the Top keywords. For example, findTopBy, findTop1By, findFirstBy, and findFirst1By all return the first entity that matches with the specified search criteria.
* If we want to select unique results, we have to add the Distinct keyword before the first By word. For example, findTitleDistinctBy or findDistinctTitleBy means that we want to select all unique titles that are found from the database.

|  |  |
| --- | --- |
|  | interface TodoRepository extends Repository<Todo, Long> {      //This is a query method.      Todo findById(Long id);  } |

interface TodoRepository extends Repository<Todo, Long> {

  // Returns the found todo entry by using its id as search criteria. If no todo entry is found, this method returns null.

public Todo findById(Long id);

// Returns an Optional which contains the found todo entry by using its id as search criteria. If no to entry is found, this method returns an empty Optional.

public Optional<Todo> findById(Long id);

// Returns the found todo entry whose title or description is given as a method parameter. If no todo entry is found, this method returns an empty list.

public List<Todo> findByTitleOrDescription(String title, String description);

// Returns the found todo entry whose title or description is given as a method parameter. If no todo entry is found, this method returns an empty list.

public List<Todo> findByTitleOrDescription(String title, String description);

// Returns the number of todo entry whose title is given s a method parameter

public long countByTitle(String title);

// Returns the distinct todo entries whose title is given as a method parameter. If no todo entries is found, this method returns an empty list.

public List<Todo> findDistinctByTitle(String title);

//Returns the first three todo entries whose title is given as a method parameter. If no todo entries is found, this method returns an empty list.

public List<Todo> findFirst3ByTitleOrderByTitleAsc(String title);

// Returns the first three todo entries whose title is given as a method parameter. If no todo entries is found, this method returns an empty list.

public List<Todo> findTop3ByTitleOrderByTitleAsc(String title);

}

| **Keyword** | **Sample** | **JPQL snippet** |
| --- | --- | --- |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is,Equals | findByFirstname,findByFirstnameIs,findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| 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<Age> ages) | … where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection<Age> 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) |

Returning values from query method:

interface TodoRepository extends Repository<Todo, Long> {

    @Query("SELECT t.title FROM Todo t where t.id = :id")

    String findTitleById(@Param("id") Long id);

    @Query("SELECT t.title FROM Todo t where t.id = :id")

    Optional<String> findTitleById(@Param("id") Long id);

    Todo findById(Long id);

    Optional<Todo> findById(Long id);

 List<Todo> findByTitle(String title);

}

**Passing parameter to query method:**

interface TodoRepository extends Repository<Todo, Long> {

    @Query("SELECT t FROM Todo t where t.title = :title AND t.description = :description")

    public Optional<Todo> findByTitleAndDescription(@Param("title") String title,

                                                     @Param("description") String description);

    @Query( value = "SELECT \* FROM todos t where t.title = :title AND t.description = :description",  nativeQuery=true  )

    public Optional<Todo> findByTitleAndDescription(@Param("title") String title,

                                                    @Param("description") String description);

}

**CURD Operation:**

**OneToMany Mapping:**

@Entity

@TableGenerator(name="student",table="SEQ\_TABLE",pkColumnName="TBL\_NAME",

valueColumnName="TBL\_SEQ",initialValue=103,allocationSize=1)

public class **Student** implements Serializable{

@Id

@GeneratedValue(generator="student",strategy=GenerationType.TABLE)

private Long sId;

private String name;

**@OneToMany(cascade=CascadeType.ALL,orphanRemoval=true) //orphanRemoval - This will remove the child records, when the parent records get removed**

@JoinColumn(name="sId") //Avoid storing null value for foreign key column in child table.

private List<School> school;

}

@Entity

@TableGenerator(name="school",table="SEQ\_TABLE",pkColumnName="TBL\_NAME",

valueColumnName="TBL\_SEQ",initialValue=204,allocationSize=1)

public class **School** implements Serializable{

@Id

@GeneratedValue(generator="school",strategy=GenerationType.TABLE)

private Long slId;

private String name;

private String address;

**@ManyToOne**

@JoinColumn(name="sId",insertable = false, updatable = false)

private Student student;

}

public interface **StudentRepository** extends **Repository<Student,Long>**{

public List<Student> **findBySId**(Long sId);

public Student **save**(Student student);

//Same for Update – It will update if the record found, else it will create new one.

public void **delete**(Student student);

}

@PostMapping(path="/save-student-info")

public String saveStudentInfo(@RequestBody Student student){

Student std = studentRepository.save(student);

}

**Deleting Child records:**

@PostMapping(path="/delete-school-info")

public String deleteSchoolInfo(@RequestBody Student student){

if(student.getsId()!=null){

if(!studentRepository.findBySId(student.getsId()).isEmpty()){

List<School> school = student.getSchool();

Iterator<School> schoolItr = school.iterator();

while(schoolItr.hasNext())

schoolRepository.delete(schoolItr.next());

return "Deleted";

}else{

return "Record not found for Delete";

}

}else{

return "ID Missing";

}

}

public interface **SchoolRepository** extends Repository<School,Long>{

public void **delete**(School school);

}

**GenerationType.IDENTITY:**

It relies on an auto-incremented database column and lets the database generate a new value with each insert operation. It is not suitable for batch update.

CREATE TABLE Students(Student\_ID int NOT NULL **AUTO\_INCREMENT**,

LastName varchar(255) NOT NULL, FirstName varchar(255), PRIMARY KEY (Student\_ID))

**GenerationType.SEQUENCE:**

It uses database sequence to generate unique values. It requires additional select statements to get the next value from a database sequence. We need to create a sequence generator in database and refer that name in the code.

@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="course\_seq")

@SequenceGenerator(name="course\_seq", sequenceName="course\_sequence", allocationSize=20)

private int id;

**GenerationType.TABLE:**

@Entity

**@TableGenerator**(name="student", table="SEQ\_TABLE", pkColumnName="TBL\_NAME", valueColumnName="TBL\_SEQ", initialValue=103,allocationSize=1)

public class Student implements Serializable{

@Id

@GeneratedValue(generator="student",strategy=GenerationType.TABLE)

private Long sId;

}

**Composite Primary key:**

A composite primary key consist of multiple primary key fields. When an entity has multiple primary key fields, JPA requires defining a special class to define the composite key and then attached to the entity class using the @EmbededId annotation.

**@Embeddable**

public class EmployeeIdentity implements Serializable {

@NotNull

private String employeeId;

@NotNull

private String companyId;

}

@Entity

@Table(name = "employees")

public class Employee {

**@EmbeddedId**

private EmployeeIdentity employeeIdentity;

private String name;

}

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, **EmployeeIdentity**> {

public Employee findById(EmployeeIdentity empId);

//Retrieve by single primary key

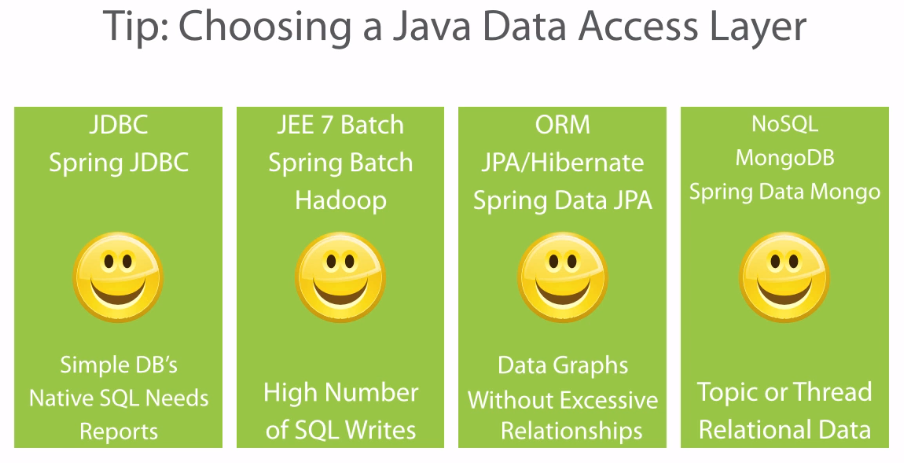
List<Employee> findByEmployeeIdentityCompanyId(String companyId);

}

// Retrieving an Employee Record with the composite primary key

employeeRepository.findById(new EmployeeIdentity("E-123", "D-457"));

**Choosing Data Access Layer**

****