Projections in Spring JPA – 2024

**In Spring JPA Projection is a way to retrieve specific fields or properties from an entity rather than whole/entire entity.**

**The followings are the ways to achieve.**

1. **Scalar Projection (Using Object[] or DTO class)**
2. **Closed Projection (Using Interface Based)**
3. **Open Projection (Using @Value with expression, Class based using Java Record)**
4. **Dynamic Projection (Passing Class<T> type)**

**Scalar Projections**

If we want to get only two fields like name and website, we can define like this.

@Repository  
public interface EmpRepo extends CrudRepository<Emp, Long> {  
 @Query("select e.firstName, e.lastName from Emp e where e.name like %:name%")  
 List<Object[]> getEmpFirstAndLastNameLike(String name);  
}

**Testing code**

public void showFirstNameLastName() {  
 List<Object[]> objList = empRepository.getEmpFirstAndLastNameLike("Hari");  
 objList.forEach( obj -> {  
 System.out.println("First Name: "+obj[0]);  
 System.out.println("Last Name: "+obj[1]);  
 });  
}

A query result stored in an *Object[]* is hard to use. We need to know the position of each field.

**DTO Projections – Sometimes, it is called as class projections**

To eliminate the above problem, we can define a DTO. A DTO class typically only defines a set of attributes, getter and setter methods for each of them. Code is given below.

**DTO Class**

@Getter @Setter @AllArgsConstructor @ToString  
public class EmpFirstLastNameDTO {  
 private String firstName;  
 private String lastName;  
}

Now repository class looks like this.

@Repository  
public interface EmpRepo extends CrudRepository<Emp, Long> {  
 @Query("select new com.ddlab.rnd.dto.EmpFirstLastNameDTO(e.firstName, e.lastName) " +  
 "from Emp e where e.location =:location")  
 List<EmpFirstLastNameDTO> getFirstAndLastNameUsingDTO(String location);  
}

**Testing Code**

public void showFirstNameLastNameByDTO() {  
 List<EmpFirstLastNameDTO> dtoList = empRepository.getFirstAndLastNameUsingDTO("Chennai");  
 dtoList.forEach( dto -> {  
 System.out.println("First Name: "+dto.getFirstName());  
 System.out.println("Last Name: "+dto.getLastName());  
 });  
}

**For More Clarity, Complete Repository is given below.**

**Interface-Based Projections:** It has also following types. **Closed Projections, Open Projections**

**Closed Projections**

Let’s declare a projection interface

**public** **interface** EmpView {

String getFirstName();

String getLastName();

String getLocation();

}

**Repository class is like this.**

@Repository

**public** **interface** EmpRepo **extends** CrudRepository<Emp, Long> {

@Query("select e from Emp e")

List<EmpView> getEmpView();

}

**Testing Method**

**public** **void** showEmpView() {

List<EmpView> empViewList = empRepository.getEmpView();

empViewList.forEach( empView -> {

System.***out***.println(empView.getFirstName()+", "+empView.getLastName()+", "+empView.getLocation());

});

}

You can also extend the **EmpView to include other objects like Department** with limited data.

You have to define another interface like this.

**public** **interface** DepartmentView {

String getName();

}

Now OrgSummaryView will include the department list.

**public** **interface** EmpView {

String getFirstName();

String getLastName();

String getLocation();

List<DepartmentView> getDepartments();

}

**Now repository code looks like this.**

@Repository

**public** **interface** OrgRepository **extends** CrudRepository<Organisation, Long> {

// Below query to avoid N+1 issue

// @Query("SELECT e FROM Emp e join fetch e.departments where e.name=:name")

@Query("SELECT e FROM Emp e where e.name=:name")

EmpView getEmpSummaryView(String name);

}

Behind the scenes,**Spring creates a proxy instance of the projection interface for each entity object, and all calls to the proxy are forwarded to that object. Note that recursive projections only work if we traverse from the owning side to the inverse side. If we do it the other way around, the nested projection would be set to *null*.**

**Also you can use without writing any specific query also. The code is given below.**

@Repository

**public** **interface** EmpRepo **extends** CrudRepository<Emp, Long> {

// Without using Query

List<EmpView> findByLocation(String location);

}

**Testing code given below.**

**public** **void** showEmpViewWithoutQuery() {

List<EmpView> empViewList = empRepository.findByLocation("Bangalore");

empViewList.forEach( empView -> {

System.***out***.println(empView.getFirstName()+", "+empView.getLastName()+", "+empView.getLocation());

});

}

**Open Projections**

**These projections enable us to define interface methods with unmatched names and with return values computed at runtime.**

Create the interface as

**public** **interface** EmployeeView {

@Value("#{target.firstName + ' ' + target.lastName}")

String getFullName();

@Value("#{target.location}")

String getLocation();

@Value("#{target.maritalStatus}")

String getMaritalStatus();

}

There is no need to create a another Repository which extends **org.springframework.data.repository.Repository. You can use simple CrudRepository.**

Repository class is given below. To find by Marital Status

@Repository

**public** **interface** EmpRepo **extends** CrudRepository<Emp, Long> {

List<EmployeeView> findByMaritalStatus(String status);

}

**Testing code**

**public** **void** showEmpViewByMaritalStatus() {

List<EmployeeView> empViewList = empRepository.findByMaritalStatus("Married");

empViewList.forEach( empView -> {

System.out.println(empView.getFullName()+", "+empView.getMaritalStatus()+", "+empView.getLocation());

});

}

You can also create a Repository class which can extend to **org.springframework.data.repository.Repository**

You can have the above same **EmployeeView** to get the specific fields.

**Create the Repository class**

@Repository

**public** **interface** SpecificEmpRepo **extends** **org.springframework.data.repository.Repository**<Emp, Long> {

@Query("select e from Emp e")

List<EmployeeView> fetchAllEmps();

}

**Testing code**

@Autowired

**private** SpecificEmpRepo specificEmpRepo;

**public** **void** showSpecificEmpView() {

List<EmployeeView> empViewList = specificEmpRepo.fetchAllEmps();

empViewList.forEach( empView -> {

System.out.println(empView.getFullName()+", "+empView.getMaritalStatus()+", "+empView.getLocation());

});

}

Note: While defining EmployeeView, you cannot write methods specific to **CrudRepository** or **JPARepository** like findById() or findByName(). Provide a different name to get the projection.

**Class-Based Projections**

You can use java record class in the with CrudRepository or with **data.repository.Repository** to get the information. Specifically there is no need to use data.repository.Repository.

**public record EmpRecord(String firstName, String lastName) {**

**}**

Note: Here you have to declare the name and location variable exactly as declared in the Emp class.

Repository class is given below.

@Repository

**public** **interface** EmpRepo **extends** CrudRepository<Emp, Long> {

List<EmpRecord> findByGender(String gender);

}

**Testing code**

**public** **void** showEmpByGender() {

List<EmpRecord> recordList = empRepository.findByGender("Female");

recordList.forEach( emp -> {

System.***out***.println(emp.firstName()+" "+emp.lastName());

});

}

**Dynamic Projections**

Sometimes we have multiple views related to the same entity object. If we have common queries for some of these views, we would have to define separate functions to fetch each of these views for the same queries. Defining a separate repository or function for different views can be cumbersome and requires lots of code duplication.

Let use create an Employee view for first name and salary.

**public** **interface** EmpFinancialView {

String getFirstName();

String getSalary();

String getPanNo();

}

**In this case, there is no need to extend org.springframework.data.repository.Repository while creating a repository class, you can simply define CrudRepository. If you want you can define org.springframework.data.repository.Repository.**

**Repository class is like this**.

@Repository  
public interface EmpRepo extends CrudRepository<Emp, Long> {  
   
 @Query("Select e from Emp e where e.firstName like %:firstName%")  
 <T> List<T> getEmpFinancialView(String firstName, Class<T> type);  
  
 @Query("Select e from Emp e where e.firstName like %:firstName%")  
 **List<EmpFinancialView>** getEmpFinancialView11(String firstName, Class<**EmpFinancialView**> type);  
}

The above both the methods are right and it works.

**Testing code**

**public** **void** showEmpFinancialStatus() {

List<EmpFinancialView> financeViewList =

empRepository.getEmpFinancialView11("Hari", EmpFinancialView.**class**);

financeViewList.forEach( emp -> {

System.***out***.println(emp.getFirstName()+" "+emp.getPanNo()+" "+emp.getSalary());

});

}

Complete Repository class is given below for more clarity.

@Repository

**public** **interface** EmpRepo **extends** CrudRepository<Emp, Long> {

@Query("select e.firstName, e.lastName from Emp e where e.name like %:name%")

List<Object[]> getEmpFirstAndLastNameLike(String name);

@Query("select new com.ddlab.rnd.dto.EmpFirstLastNameDTO(e.firstName, e.lastName) " +

"from Emp e where e.location =:location")

List<EmpFirstLastNameDTO> getFirstAndLastNameUsingDTO(String location);

@Query("select e from Emp e")

List<EmpView> getEmpView();

// Without using Query

List<EmpView> findByLocation(String location);

List<EmployeeView> findByMaritalStatus(String status);

List<EmpRecord> findByGender(String gender);

@Query("Select e from Emp e where e.firstName like %:firstName%")

<T> List<T> getEmpFinancialView(String firstName, Class<T> type);

@Query("Select e from Emp e where e.firstName like %:firstName%")

List<EmpFinancialView> getEmpFinancialView11(String firstName, Class<EmpFinancialView> type);

}

For the above examples, the entity details given below in github.

<https://github.com/debjava/projection-spring-jpa-2024>