Projections in Spring JPA – 2024

**Scalar Projections**

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

@Repository

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

@Query("SELECT org.name, org.website FROM Organisation org")

List<Object[]> getNameAndWebsite();

}

**Testing code**

**public** **void** showOrgDetails() {

List<Object[]> objectList = orgRepo.getNameAndWebsite();

objectList.forEach(object -> {

System.***out***.println("Name: " + object[0]);

System.***out***.println("Website: " + object[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 @ToString

**public** **class** OrgNameWebsiteDto {

**private** String name;

**private** String website;

**public** OrgNameWebsiteDto(String name, String website) {

**//constructor code**

}

}

Now repository class looks like this.

@Repository

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

@Query("SELECT new com.ddlab.rnd.dto.OrgNameWebsiteDto(org.name, org.website) from Organisation org where org.name=:name")

OrgNameWebsiteDto getNameAndWebsiteByDTO(String name);

}

Testing Code

**public void showOrgDetailsByDTO() {**

**OrgNameWebsiteDto dto = orgRepo.getNameAndWebsiteByDTO("Microsoft");**

**System.out.println("Name: " + dto.getName());**

**System.out.println("Website: " + dto.getWebsite());**

**}**

Instead of doing all these Spring JPA provides Projections. We have different types of projections.

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

**Closed Projections**

Let’s declare a projection interface

**public** **interface** OrgSummaryView {

String getName();

String getWebsite();

String getCeoName();

}

**Repository class is like this.**

@Repository

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

@Query("SELECT org FROM Organisation org where org.name=:name")

OrgSummaryView getOrgSummary(String name);

}

**public** **void** showOrgDetailsByOrgSummaryView() {

OrgSummaryView orgSummaryView = orgRepo.getOrgSummary("Microsoft");

System.***out***.println("Name: " + orgSummaryView.getName());

System.***out***.println("Website: " + orgSummaryView.getWebsite());

System.***out***.println("CEO Name: " + orgSummaryView.getCeoName());

}

You can also extend the the **OrgSummaryView 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** OrgSummaryView {

String getName();

String getWebsite();

String getCeoName();

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 org FROM Organisation org join fetch org.departments where org.name=:name")

@Query("SELECT org FROM Organisation org where org.name=:name")

OrgSummaryView getOrgSummary(String name);

}

**Testing code**

**public** **void** showOrgDetailsByOrgSummaryView() {

OrgSummaryView orgSummaryView = orgRepo.getOrgSummary("Microsoft");

System.***out***.println("Name: " + orgSummaryView.getName());

System.***out***.println("Website: " + orgSummaryView.getWebsite());

System.***out***.println("CEO Name: " + orgSummaryView.getCeoName());

orgSummaryView.getDepartments().forEach( department -> {

System.***out***.println("Department Name: "+department.getName());

});

}

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

// Without using @Query

OrgSummaryView findByCeoName(String name);

}

**Testing code given below.**

**public** **void** showOrgDetailsByCeoName() {

OrgSummaryView orgSummaryView = orgRepo.findByCeoName("Satya Nadel");

System.***out***.println("Name: " + orgSummaryView.getName());

System.***out***.println("Website: " + orgSummaryView.getWebsite());

System.***out***.println("CEO Name: " + orgSummaryView.getCeoName());

orgSummaryView.getDepartments().forEach( department -> {

System.***out***.println("Department Name: "+department.getName());

});

}

**Complete repository class is given below.**

@Repository

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

@Query("SELECT org.name, org.website FROM Organisation org")

List<Object[]> getNameAndWebsite();

@Query("SELECT new com.ddlab.rnd.dto.OrgNameWebsiteDto(org.name, org.website) from Organisation org where org.name=:name")

OrgNameWebsiteDto getNameAndWebsiteByDTO(String name);

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

@Query("SELECT org FROM Organisation org where org.name=:name")

OrgSummaryView getOrgSummary(String name);

// Without using @Query

OrgSummaryView findByCeoName(String name);

}

**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();

}

**Create the Repository class**

**import** **org.springframework.stereotype.Repository**;

**import** com.ddlab.rnd.entity.Employee;

**import** com.ddlab.rnd.view.EmployeeView;

@Repository

**public** **interface** EmployeeRepository **extends** **org.springframework.data.repository.Repository**<Employee, Long> {

// Do not use findById, it will throw exception, as it will return Employee object

EmployeeView findEmpById(Long id);

}

**Testing code**

**public** **void** showEmpDetails() {

EmployeeView empView = empRepo.findEmpById(5L);

System.***out***.println("Emp Full Name: "+empView.getFullName());

}

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 Repository to get the information. You can use java record class to achieve the result.

**public** **record** BranchRecord(String name, String location) {

}

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

Repository class is given below.

@org.springframework.stereotype.Repository

**public** **interface** BranchRepository **extends** Repository<Branch, Long> {

BranchRecord getBranchDetailByBranchCode(String branchCode);

}

**Testing code**

**private** **void** showBranchDetail() {

BranchRecord branchRecord = branchRepo.getBranchDetailByBranchCode("007A");

System.***out***.println("Branch Name: "+branchRecord.name());

System.***out***.println("Branch Location: "+branchRecord.location());

}

**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 a Employee view for first name and salary.

**public** **interface** EmployeeNameSalaryView {

String getFirstName();

**int** getSal();

}

**Repository class is like this**.

**import** org.springframework.data.jpa.repository.Query;

**import** org.springframework.data.repository.Repository;

**import** com.ddlab.rnd.entity.Employee;

@org.springframework.stereotype.Repository

**public** **interface** EmployeeDynamicRepository **extends** Repository<Employee, Long> {

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

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

}

**Testing code**

**private** **void** showEmpDynamicView() {

List<EmployeeNameSalaryView> empNameSalViewList = empDynamicRepo.getEmployeeAsPerNameLike("Ram", EmployeeNameSalaryView.**class**);

empNameSalViewList.forEach( empNameSalView -> {

System.***out***.println("Employee First Name: "+empNameSalView.getFirstName());

System.***out***.println("Employee Salary: "+empNameSalView.getSal());

});

}

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

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