SPringData series part 2: Property TRAVERSAL

# property Traversal

We are aware of the fact that if we want to create a query based on entity bean properties which are not present Repository interface (CrudRepository,JpaRepository etc). We can do it very easily in SpringData. Just we need to declare a method in our custom Repository which must have to obey a pattern. then Spring data create the query for us on the fly.

Pattern is Queryname<java property><Operation>,

**In this Article, we will discuss the details of this Query derivation technique.**

**SpringData** has an inbuilt data stores(JPA,MongoDB) specific QureyTransalorFactory ,which will translate the method written in the custom repository to store specific query.

Let say we have a Person Entity and has a Person Repository and underlying dataStore is JPA

package com.example.person;

import java.util.ArrayList;

import java.util.List;

import javax.persistence.CascadeType;

import javax.persistence.Entity;

import javax.persistence.FetchType;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.OneToMany;

@Entity

public class Person {

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

private Long id;

private String name;

private String country;

private String gender;

@OneToMany(mappedBy="person",targetEntity=Hobby.class,

fetch=FetchType.EAGER,cascade=CascadeType.ALL)

List<Hobby> hobby;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCountry() {

return country;

}

public void setCountry(String country) {

this.country = country;

}

public String getGender() {

return gender;

}

public void setGender(String gender) {

this.gender = gender;

}

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public List<Hobby> getHobby() {

return hobby;

}

public void setHobby(List<Hobby> hobby) {

this.hobby = hobby;

}

public void addHobby(Hobby ihobby)

{

if(hobby == null)

{

hobby = new ArrayList<Hobby>();

}

hobby.add(ihobby);

}

@Override

public String toString() {

return "Person [id=" + id + ", name=" + name + ", country=" + country + ", gender=" + gender + "]";

}

}

package com.example.person;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.ManyToOne;

@Entity

public class Hobby {

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

private Long id;

@ManyToOne

@JoinColumn(name="person\_id")

private Person person;

private String name;

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public Person getPerson() {

return person;

}

public void setPerson(Person person) {

this.person = person;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

**Repository**

package com.example.repo;

import java.util.List;

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

import org.springframework.data.repository.CrudRepository;

import com.example.person.Person;

public interface PersonRepositary extends CrudRepository<Person, Long> {

List<Person> findByCountryContains(String country);

}

Now, When QueryTranslator encounters the method findByCountryContains, it performs following Steps

1. Strip the section findBy.

2. Now try to find an exact match for the stripped section and Person entity’s property. As Person does not have any property called CountryContains it goes to Step 3.

3. Again query translator split the rest section based on camelCase pattern from the tail so it has now two tokens (Country and Contains)

4. Now query translator tries to match country with Person entity properties now it finds an exact match so it's taken this phrase as one of the filter criteria.

5. As Contains is a predefined combining criterion based on the underlying data store. So query translator understands the same has and put a like check.

6. As underlying Store is JPA now query translator generate a query

***Select p from person p where p.country like ?1***

Spring Data’s **QueryTranslator** is very powerful it can also derive a query using nested bean property.

Let check the Person bean again, we have a mapping between Person and Hobby it is a one to many relationships.

suppose we want to create a query which will fetch person based on a hobby.

To do this ,we just have to create a method in Person repository,

package com.example.repo;

import java.util.List;

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

import org.springframework.data.repository.CrudRepository;

import com.example.person.Person;

public interface PersonRepositary extends CrudRepository<Person, Long> {

List<Person> findByCountryContains(String country);

**List<Person> findPersonByHobbyName(String hobby);**

}

Now query translator tries to find a match “**HobbyName**” as it is not matched, again it strips this phrase to two tokens hobby and name then it tries to match hobby now it finds a List in Person entity then it goes into the Hobby Entity and found a property called name so it creates a query like following

Hibernate:

select

person0\_.id as id1\_1\_,

person0\_.country as country2\_1\_,

person0\_.gender as gender3\_1\_,

person0\_.name as name4\_1\_

from

person person0\_

left outer join

hobby hobby1\_

on person0\_.id=hobby1\_.person\_id

where

hobby1\_.name=?

Query translator is powerful but it has one shortcoming, let say we have added an additional property **hobbyName** in person now Query translator finds a match hobbyName in first place so it tries to filter against this property

So then query would be

**Select p from Person p where p.hobbyName=?1**

**Which is wrong so overcome this situation we can introduce a \_(underscore) to demarcation the traversal path**

Method name will be,

**List<Person> findPersonByHobby\_Name(String hobby);**

Now **QueryTranslator** understands hobby and name are two different tokens.