Spring Data series: Spring Data in Nutshell

# Spring Data Overview

In this Series, we will deep dive into Spring Data Architecture. In this Article, we will concentrate on features of Spring Data.

**Why Spring Data?**

Spring data is an Umbrella which takes care of SQL and NOSQL databases and reduces the effort to use them. As Spring community’s motto is to reduce boilerplate code so Spring community invented **Spring** **data** which is take one step forward than database template. In JPA template although spring hides the creation EntityManagerFactory managing Entity manager but the problem is we still have to maintain a DAO layer and provide CRUD operations for each entity. CRUD operations are similar in nature you have framed it in your mind but still, have to write or copy and paste same code over and over for different Entities.

Spring address this problem and introduce Spring data which says you don’t have to write redundant code for CRUD just define an interface, Spring creates the actual implementation on the fly. Pure magic.

**Spring Data Repository:**

Spring data introduce Repository concept, Repository act like Adapter It takes the Entity and makes Persistence layer specific call, it may be JPA or MongoDB etc. The beauty of Repository is we just to create Interface which will extend Spring Data’s **Repository** **marker interface**. By doing so you instruct spring to find those interfaces which will extend Repository interface and add them as a Spring bean, by creating a Proxy and delegate the call to actual implementation. But as a developer we don’t have to write actual implementation just we need to mention what methods we need Spring takes care of that by **Create and Derivation** approach.

**Choosing underlying Persistence Layer:**

As I told earlier Spring data is like umbrella it contains different types of persistence layers like JPA,MongoDB so as a client you need to instruct Spring data to which persistence layer you want to use. So Spring data provide different Annotations for different Repository So If you want to use JPA just annotated your main class with **@EnableJpaRepositories** or If you prefer MongoDB then **@EnableMongoRepositories**. Now Spring Data understand you need JPA repository or Mongo repository so it can create JPA or MONGO Specific calls. Actually, Behind the Scene Spring data has an implementation of each Repository like **SimpleJpaRepository** or **SimpleMongoRepository**, Spring data internally delegates the call to actual implementation.

**Different Repository:**

Spring Data provide various Repository interfaces which extend the Marker interface Repository. You can choose them based on your need. Just create a custom interface and extend one of the Repository interfaces,

**Two important Repository interfaces**

**1. CRUDRepository**

**2. PagingAndSortingRepository**

**1 .** **CRUDRepository** : Use when you need to perform CRUD operation (Create,Read,Update,Delete)

*public interface CrudRepository<T, ID extends Serializable>*

*extends Repository<T, ID> {*

*<S extends T> S save(S entity);*

*T findOne(ID primaryKey);*

*Iterable<T> findAll();*

*Long count();*

*void delete(T entity);*

*boolean exists(ID primaryKey);*

*} // … more functionality omitted.*

1. **PagingAndSortingRepository :** Use when you want to use paging and sorting.

*public interface PagingAndSortingRepository<T, ID extends Serializable>*

*extends CrudRepository<T, ID> {*

*Iterable<T> findAll(Sort sort);*

*Page<T> findAll(Pageable pageable);*

*}*

**Assumption**

1. We use JPA as Repository
2. We use Mysql Dbas persistence layer

So introduce Spring Data in your project following Steps are needed.

1. Add Spring data JPA dependencies into Pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

1. Add target database driver. We use MySql so we add following

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

1. Create a Person Entity
2. Create a Person table in MySql database
3. Create a PersonRepository which extends CRUDRepository.
4. Put **@EnableJpaRepositories** overmain class

Now run application data will successfully save in data base.

**Java Code:**

**As I use Spring boot I use database details in application.properties**

spring.datasource.url=jdbc:mysql://localhost:3306/testdb

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

package com.example.cab;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class Person {

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

private Long id;

private String name;

private String country;

private String gender;

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;

}

@Override

public String toString() {

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

}

}

package com.example.cab;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

public interface PersonRepositary extends CrudRepository<Person, Long> {

List<Person> findByCountryContains(String country);

}

**package** com.example.cab;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.context.annotation.Bean;

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

@SpringBootApplication

@EnableJpaRepositories

**public** **class** PersonApplication {

**private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(PersonApplication.**class**);

@Bean

**public** CommandLineRunner demo(PersonRepositary repository) {

Person p = **new** Person();

p.setName("Samir mitra");

p.setCountry("America");

p.setGender("male");

repository.save(p);

***log***.info("Person created" + p);

**return** **null**;

}

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(PersonApplication.**class**, args);

}

}

Output: **Person createdPerson [id=4, name=Samir mitra, country=America, gender=male]**

**Adding Extra filter query in Interface:** Sometimes you want to create a find query based on a property which is not in **CRUDRepository** interface. You have provision to do so just declare a method in your custom interface which must have to obey a Syntax. then Spring data create the query for you on the fly.

Syntax is **Queryname<java property><Operation>,**

like we create a query in Person Repository

***List<Person> findByCountryContains(String country);***

If you run application Output will be

Hibernate:

select

person0\_.id as id1\_0\_,

person0\_.country as country2\_0\_,

person0\_.gender as gender3\_0\_,

person0\_.name as name4\_0\_

from

person person0\_

where

person0\_.country like ?

2016-09-24 21:37:32.334 INFO 6384 --- [ main] com.example.cab.PersonApplication : Person Person [id=1, name=Shamik mitra, country=India, gender=male]

2016-09-24 21:37:32.334 INFO 6384 --- [ main] com.example.cab.PersonApplication : Person Person [id=2, name=Swastika mitra, country=India, gender=Female]

2016-09-24 21:37:32.334 INFO 6384 --- [ main] com.example.cab.PersonApplication : Person Person [id=3, name=Swastika mitra, country=India, gender=Female]

**Delete Operation:**

**When the discussion comes to delete a curious mind often asks, Spring data support soft delete?** The answer is no Spring data only Support physical delete as of now.

**Use Handcrafted Query:** If you want you do not use Spring data generated query you can use your handcrafted query by the @Query interface, one use case maybe you don’t want physical delete so you override the Spring Data query by your handcraft soft delete query.

**Just redefine the method and add @Query interface and pass the Query.**

**@Query("select p from Person p where p.country like ?1")List<Person> findByCountryContains(String country);**

**Query DSL :** Although when you search entity by properties in Spring Data you just declare them in the interface but if you want to create a query which will search the entity dynamically. I mean to say your entity can be searched by the different combination of properties

**what do you do?**

One approach is declaring a method in all possible combination, but this is a bad idea your interface will be long and with addition and deletion of property combination need to be modified.

Criteria API is the Solution; we use Criteria API for the same reason.

We can introduce Predicate and Query Dsl for it

**Query DSL: Spring data use Query’s which is quite similar to criteria API but different approach. Just like the JPA Criteria API it uses a Java 6 annotation processor to generate meta-model objects but produces a much more approachable API.**

To enable it

**Add the following plugin in pom.xml**

<plugin>

<groupId>com.mysema.maven</groupId>

<artifactId>maven-apt-plugin</artifactId>

<version>1.0</version>

<executions>

<execution>

<phase>generate-sources</phase>

<goals>

<goal>process</goal>

</goals>

<configuration>

<outputDirectory>target/generated-sources</outputDirectory>

<processor>com.mysema.query.apt.jpa.JPAAnnotationProcessor</processor>

</configuration>

</execution>

</executions>

</plugin>

In output Directory, it creates a special Class **QPerson** and using **QPerson.person** we can access every property and build our query.

Later I will discuss details on every bullet point with an example.

To follow it Please keep eye on ***javaonfly.blogspot.in***

**Benefits of Spring Data:**

1. Get rid from boiler plate Dao classes.

2. Easily switch from one to another repository.

3. You don’t have to an expert of JPA or Mongo Query Spring will take care of it in an optimized way.

4. Highly flexible and support to add your custom query.