SPRING DATA JPA

Design by: DieuNT1

Agenda

Lesson Objectives

1
- Understand Spring Data JPA Framework and its core technologies.

2
- Setting up a Spring Data JPA Project

3
- Able define custom query methods in Spring Data JPA repositories using method naming conventions.

3
- Understand how to write more complex queries using JPQL (Java Persistence Query Language) or native SQL queries

4
- Understand how transactions work in Spring Data JPA and how to configure transaction management

Section 1

Introduction

Introduction

Spring Data is a module of Spring Framework.
The goal of Spring Data repository abstraction is to significantly **reduce the amount of boilerplate code required** to implement *data access layers* for various persistence stores.

- Java Persistence API (JPA) is Java's standard API specification for object-relational mapping.
- Spring Data JPA is a part of Spring Data and it supports **Hibernate, Eclipse Link, or any other JPA provider**.

What Spring Data JPA?

Spring Data JPA

- ✓ It is NOT a JPA provider.
- ✓ It is a library/framework that **adds an extra layer of abstraction** on the top of our JPA provider (like Hibernate).

- **That means it uses all features defined by the JPA specification:**
  - ✓ The **entity and association mappings**,
  - ✓ The **entity lifecycle management**,
  - ✓ and **JPA's query** capabilities.

- Spring Data JPA adds its own features like a no-code implementation of the **repository pattern** and the creation of database queries from method names.

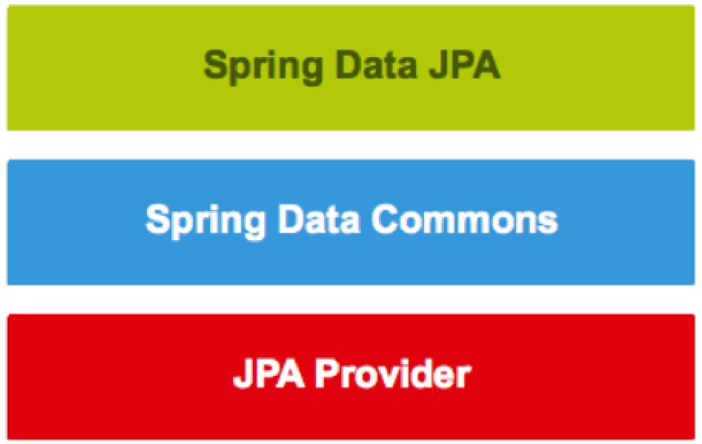
Introduction to Spring Data JPA

3 favorite features that Spring Data adds on top of JPA:

No-code Repositories	Spring Data JPA provides you a set of **repository interfaces** which you only need to extend to define a specific repository for one of your entities.
Reduced boilerplate code	Spring Data JPA provides a default implementation for each method defined by one of its repository interfaces.
Generated queries	With a simple query, you just need to define a method on your repository interface with a name that starts with *find…By*. Spring then parses the method name and creates a query for it.

What Spring Data JPA?

- If we decide to use Spring Data JPA, the repository layer of our application contains three layers that are described in the following:



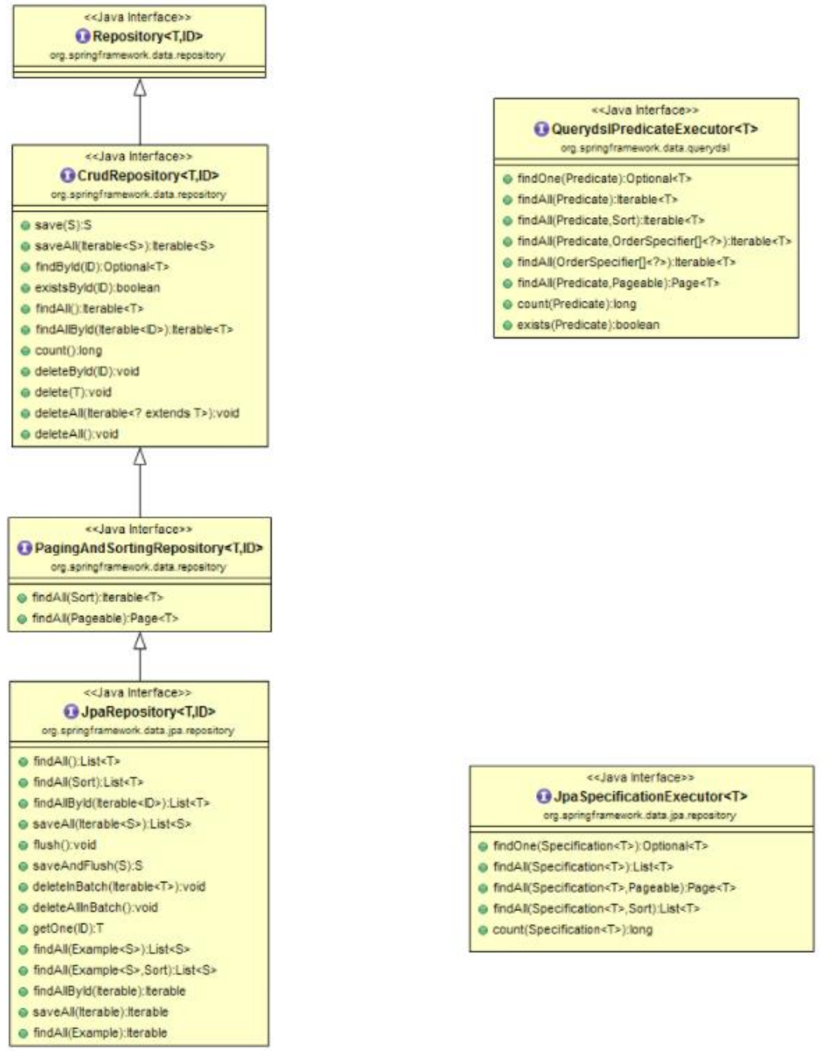
✓ *Spring Data JPA* provides support for creating JPA repositories by extending the Spring Data repository interfaces.

✓ *Spring Data Commons* provides the infrastructure that is shared by the datastore-specific Spring Data projects.

✓ *The JPA Provider* (like hibernate) implements the Java Persistence API.

Spring Data Repositories/Interfaces

- **Spring Data Commons** and **Spring Data JPA**



<<Java Interface>>
Repository<T,ID>
org.springframework.data.repository
<<Java Interface>>
CrudRepository<T,ID>
org.springframework.data.repository
save(S):S
saveAll(Iterable<S>):Iterable<S>
findById(ID):Optional<T>
existsById(ID):boolean
findAll():Iterable<T>
findAllById(Iterable<ID>):Iterable<T>
count():long
deleteById(ID):void
delete(T):void
deleteAll(Iterable<? extends T>):void
deleteAll():void
<<Java Interface>>
PagingAndSortingRepository<T,ID>
org.springframework.data.repository
findAll(Sort):Iterable<T>
findAll(Pageable):Page<T>
<<Java Interface>>
JpaRepository<T,ID>
org.springframework.data.jpa.repository
findAll():List<T>
findAll(Sort):List<T>
findAllById(Iterable<ID>):List<T>
saveAll(Iterable<S>):List<S>
flush():void
saveAndFlush(S):S
deleteInBatch(Iterable<T>):void
deleteAllInBatch():void
getOne(ID):T
findAll(Example<S>):List<S>
findAll(Example<S>,Sort):List<S>
findAllById(Iterable):Iterable
saveAll(Iterable):Iterable
findAll(Example):Iterable
<<Java Interface>>
QuerydslPredicateExecutor<T>
org.springframework.data.querydsl
findOne(Predicate):Optional<T>
findAll(Predicate):Iterable<T>
findAll(Predicate,Sort):Iterable<T>
findAll(Predicate,OrderSpecifier[]<?>):Iterable<T>
findAll(OrderSpecifier[]<?>):Iterable<T>
findAll(Predicate,Pageable):Page<T>
count(Predicate):long
exists(Predicate):boolean
<<Java Interface>>
JpaSpecificationExecutor<T>
org.springframework.data.jpa.repository
findOne(Specification<T>):Optional<T>
findAll(Specification<T>):List<T>
findAll(Specification<T>,Pageable):Page<T>
findAll(Specification<T>,Sort):List<T>
count(Specification<T>):long

Spring Data Repositories/Interfaces

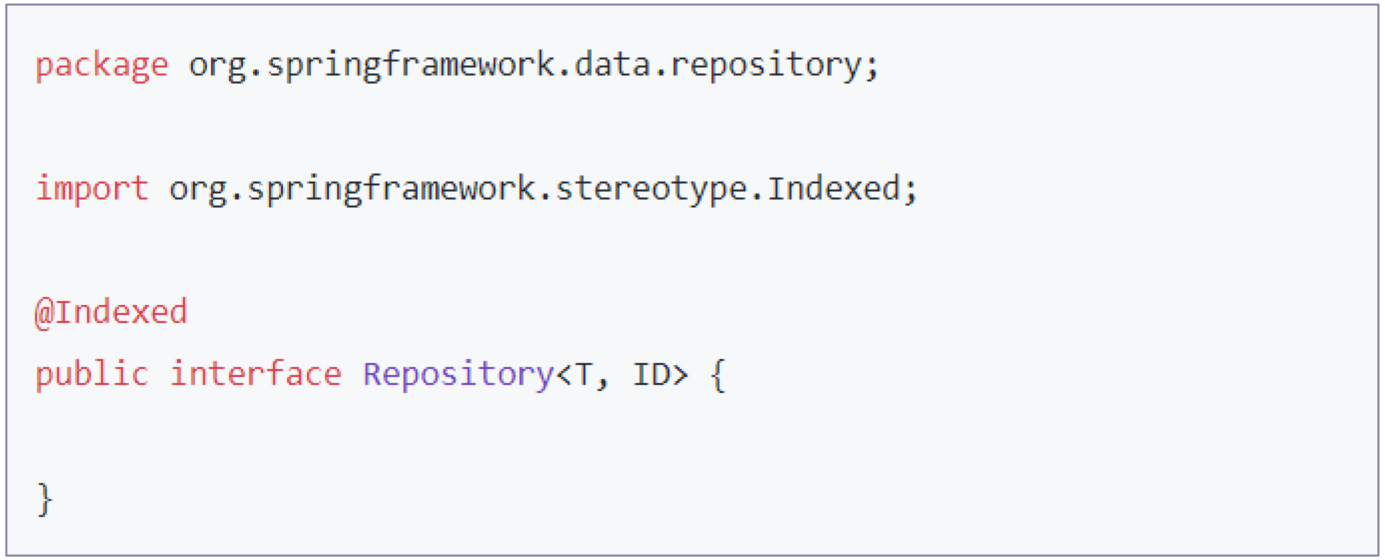
- It contains technology-neutral **repository interfaces** as well as a **metadata model** for persisting Java classes.
- Spring Data Commons project provides the following interfaces:
  - ✓ **Repository**<T, ID extends Serializable> interface
  - ✓ **CrudRepository**<T, ID extends Serializable> interface
  - ✓ **PagingAndSortingRepository**<T, ID extends Serializable> interface
  - ✓ **QueryDslPredicateExecutor** interface

Section 2

Spring Data Repositories Interfaces

Repository Interface

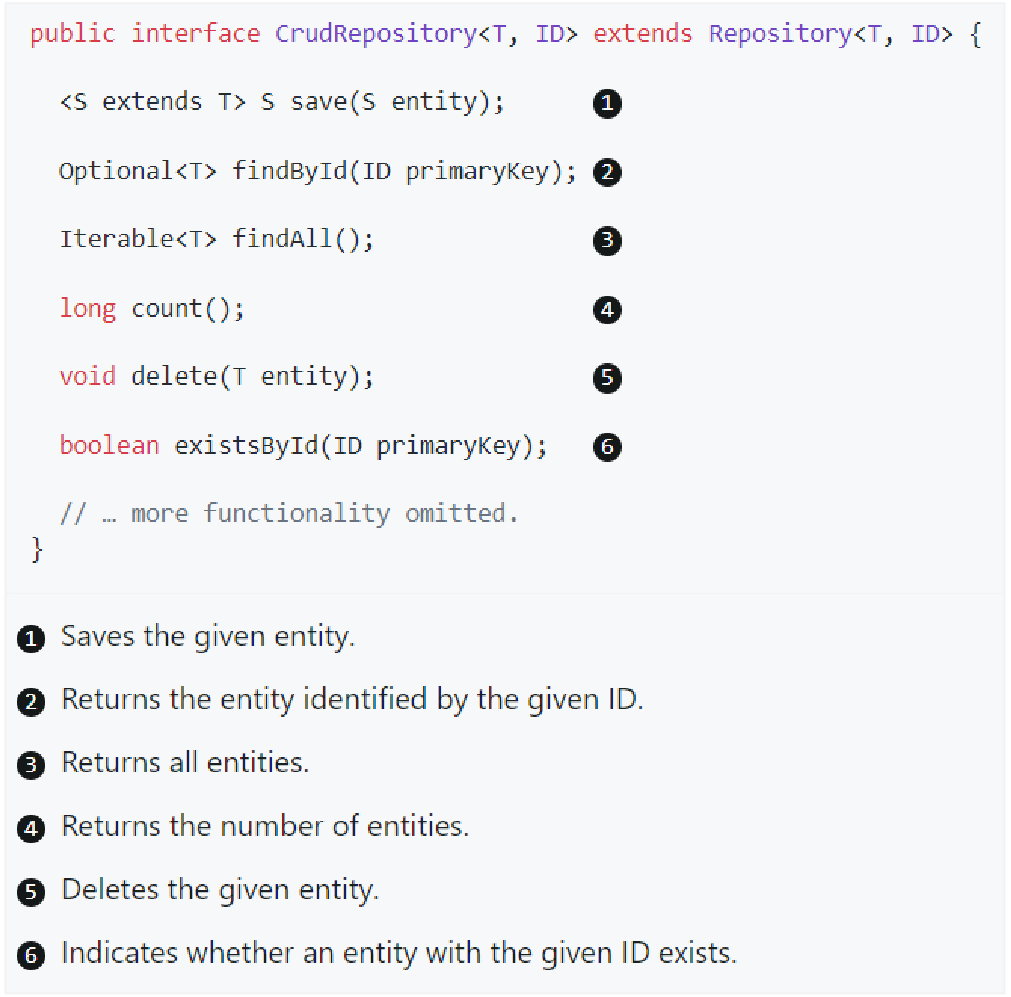
- The `Repository<T, ID extends Serializable>` interface is a marker interface that has two purposes:
  - ✓ It captures the type of the managed entity and the type of the entity’s id.
  - ✓ It helps the Spring container to discover the “concrete” repository interfaces during classpath scanning.
  - ✓ Let's look at the source code of the Repository interface.



```
package org.springframework.data.repository;

import org.springframework.stereotype.Indexed;

@Indexed
public interface Repository<T, ID> {

}
```

CrudRepository Interface

- The `CrudRepository<T, ID extends Serializable>` *interface* provides CRUD operations for the managed entity.



```
public interface CrudRepository<T, ID> extends Repository<T, ID> {

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

  Optional<T> findById(ID primaryKey); ❷

  Iterable<T> findAll();               ❸

  long count();                        ❹

  void delete(T entity);               ❺

  boolean existsById(ID primaryKey);   ❻

  // … more functionality omitted.
}
```

❶ Saves the given entity.

❷ Returns the entity identified by the given ID.

❸ Returns all entities.

❹ Returns the number of entities.

❺ Deletes the given entity.

❻ Indicates whether an entity with the given ID exists.

CrudRepository Interface

- Let's look at the usage of each method with description.
  - ✓ *long count()* - Returns the number of entities available.
  - ✓ *void delete(T entity)* - Deletes a given entity.
  - ✓ *void deleteAll()* - Deletes all entities managed by the repository.
  - ✓ *void deleteAll(Iterable<? extends T> entities)* - Deletes the given entities.
  - ✓ *void deleteById(ID id)* - Deletes the entity with the given id.
  - ✓ *boolean existsById(ID id)* - Returns whether an entity with the given id exists.
  - ✓ *Iterable findAll()* - Returns all instances of the type.
  - ✓ *Iterable findAllById(Iterable ids)* - Returns all instances of the type with the given IDs.
  - ✓ *Optional findById(ID id)* - Retrieves an entity by its id.
  - ✓ *save(S entity)* - Saves a given entity.
  - ✓ *Iterable saveAll(Iterable entities)* - Saves all given entities.

ListCrudRepository Interface

- With version 3.0 we also introduced `ListCrudRepository` which is very similar to the CrudRepository:

Those methods that return multiple entities it returns a List instead of an Iterable which you might find easier to use.

Modifier and Type	Method	Description
List<T>	findAll()	Returns all instances of the type.
List<T>	findAllById(Iterable<ID> ids)	Returns all instances of the type T with the given IDs.
<S extends T> List<S>	saveAll(Iterable<S> entities)	Saves all given entities.

PagingAndSortingRepository inteface

- The *PagingAndSortingRepository<T, ID extends Serializable> interface* is an extension of ***CrudRepository*** to provide additional methods to retrieve entities using the pagination and sorting abstraction.

```
package org.springframework.data.repository;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;

@NoRepositoryBean
public interface PagingAndSortingRepository < T, ID > extends CrudRepository < T, ID > {

    /**
     * Returns all entities sorted by the given options.
     */
    Iterable < T > findAll(Sort sort);

    /**
     * Returns a {@link Page} of entities meeting the paging restriction provided in the {@code Pageable} object.
     */
    Page < T > findAll(Pageable pageable);
}
```

PagingAndSortingRepository inteface

- **Example:** To access the first page of `User` by a page size of 20, you could do something like the following:

```
PagingAndSortingRepository<User, Long> repository = // … get access to a bean
Page<User> users = repository.findAll(PageRequest.of(1, 20));
```

Pagination and Sorting

- Example to access our *Products*, we'll need a *ProductRepository*:

```java
@Repository
public interface ProductRepository extends
        PagingAndSortingRepository<Product, Integer> {

    List<Product> findAllByPrice(double price, Pageable pageable);
}
```

- Create or obtain a *PageRequest* object, which is an implementation of the *Pageable* interface
- Pass the *PageRequest* object as an argument to the repository method we intend to use
- We can create a *PageRequest* object by passing in the requested page number and the page size.

```java
Pageable firstPageWithTwoElements = PageRequest.of(0, 2);

Pageable secondPageWithFiveElements = PageRequest.of(1, 5);
```

*Note: here **the page count starts at zero!***

Pagination and Sorting

- Similarly, to just have our query results sorted, we can simply pass an instance of *Sort* to the method:

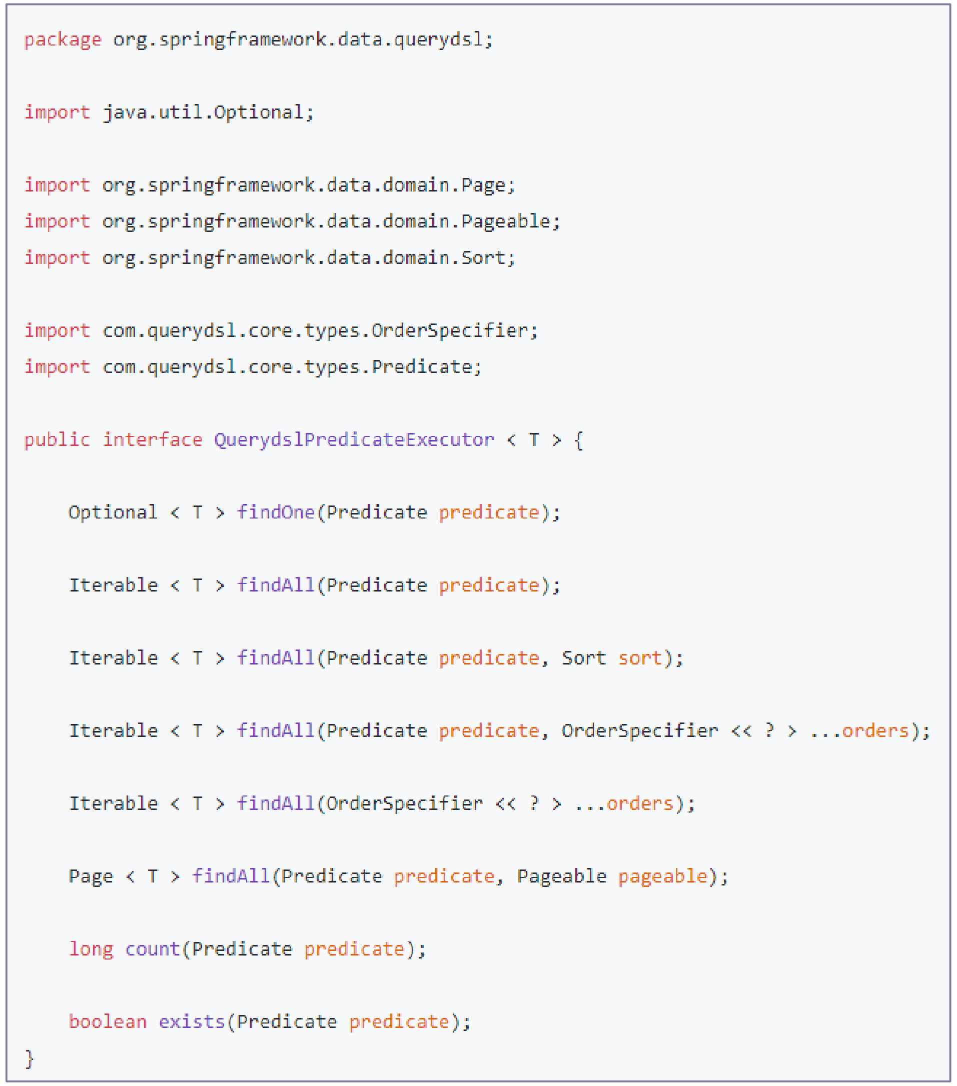
```java
Page<Product> allProductsSortedByName = productRepository.findAll(Sort.by("name"));
```

- What if we want to **both sort and page our data?**

```java
Pageable sortedByName = PageRequest.of(0, 3, Sort.by("name"));

Pageable sortedByPriceDesc = PageRequest.of(0, 3, Sort.by("price").descending());

Pageable sortedByPriceDescNameAsc = PageRequest.of(0, 5, Sort.by("price").descending()
                                                            .and(Sort.by("name")));
```

QueryDslPredicateExecutor interface

- The *QueryDslPredicateExecutor* interface is not a “repository interface”.
- It declares the methods that are used to retrieve entities from the database by using *QueryDsl* Predicate objects.



```
package org.springframework.data.querydsl;

import java.util.Optional;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;

import com.querydsl.core.types.OrderSpecifier;
import com.querydsl.core.types.Predicate;

public interface QuerydslPredicateExecutor < T > {

    Optional < T > findOne(Predicate predicate);

    Iterable < T > findAll(Predicate predicate);

    Iterable < T > findAll(Predicate predicate, Sort sort);

    Iterable < T > findAll(Predicate predicate, OrderSpecifier << ? > ...orders);

    Iterable < T > findAll(OrderSpecifier << ? > ...orders);

    Page < T > findAll(Predicate predicate, Pageable pageable);

    long count(Predicate predicate);

    boolean exists(Predicate predicate);
}
```

Spring Data JPA Interfaces

- *Spring Data JPA* module deals with enhanced support for JPA based data access layers.
- Spring Data JPA project provides the following interfaces:
  - ✓ `JpaRepository<T, ID extends Serializable> interface`
  - ✓ `JpaSpecificationExecutor interface`

JpaRepository Interface

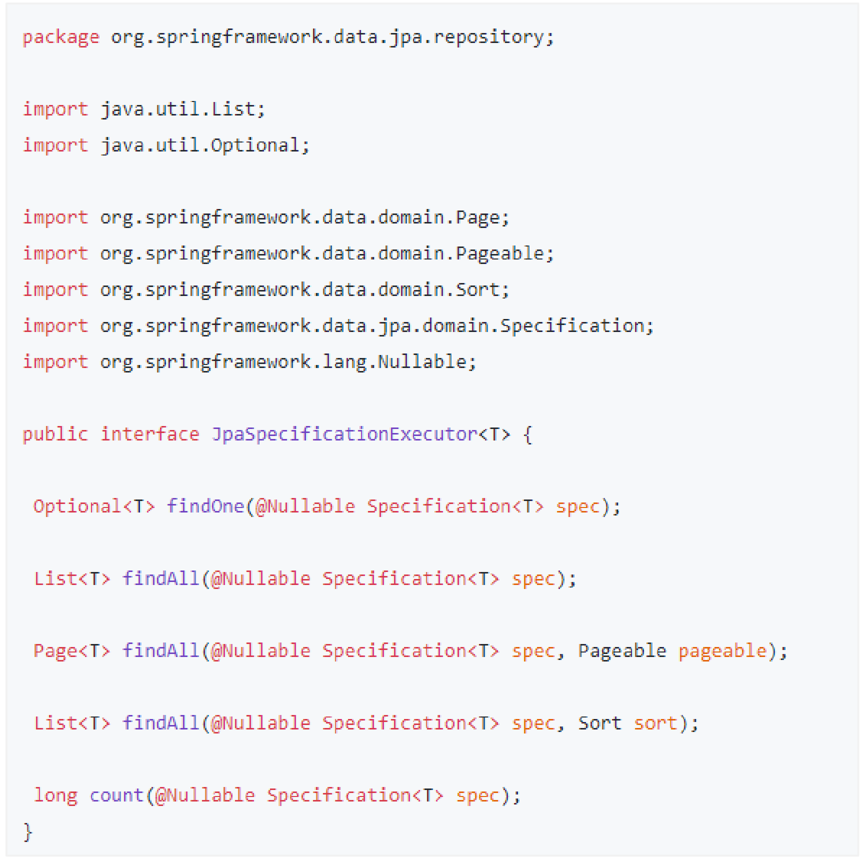
- The *JpaRepository<T, ID extends Serializable> interface* is a JPA specific repository interface that combines the methods declared by the common repository interfaces behind a single interface.

```
package org.springframework.data.jpa.repository;

import java.util.List;

import javax.persistence.EntityManager;

import org.springframework.data.domain.Example;
import org.springframework.data.domain.Sort;
import org.springframework.data.repository.NoRepositoryBean;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.repository.query.QueryByExampleExecutor;

@NoRepositoryBean
public interface JpaRepository < T, ID > extends
PagingAndSortingRepository < T, ID > ,  QueryByExampleExecutor < T > {


    List < T > findAll();

    List < T > findAll(Sort sort);

    List < T > findAllById(Iterable < ID > ids);
```

```
    <S extends T > List < S > saveAll(Iterable < S > entities);

    void flush();

     <S extends T > List < S > saveAll(Iterable < S > entities);

     void flush();

     <S extends T > S saveAndFlush(S entity);

     void deleteInBatch(Iterable < T > entities);

     void deleteAllInBatch();

     T getOne(ID id);

     @Override

     <S extends T > List < S > findAll(Example < S > example);

     @Override

     <S extends T > List < S > findAll(Example < S > example, Sort sort);

 }
```

JpaSpecificationExecutor interface

- The *JpaSpecificationExecutor* interface is not a “repository interface”.
- It declares the methods that are used to retrieve entities from the database by using Specification objects that use the JPA criteria API.

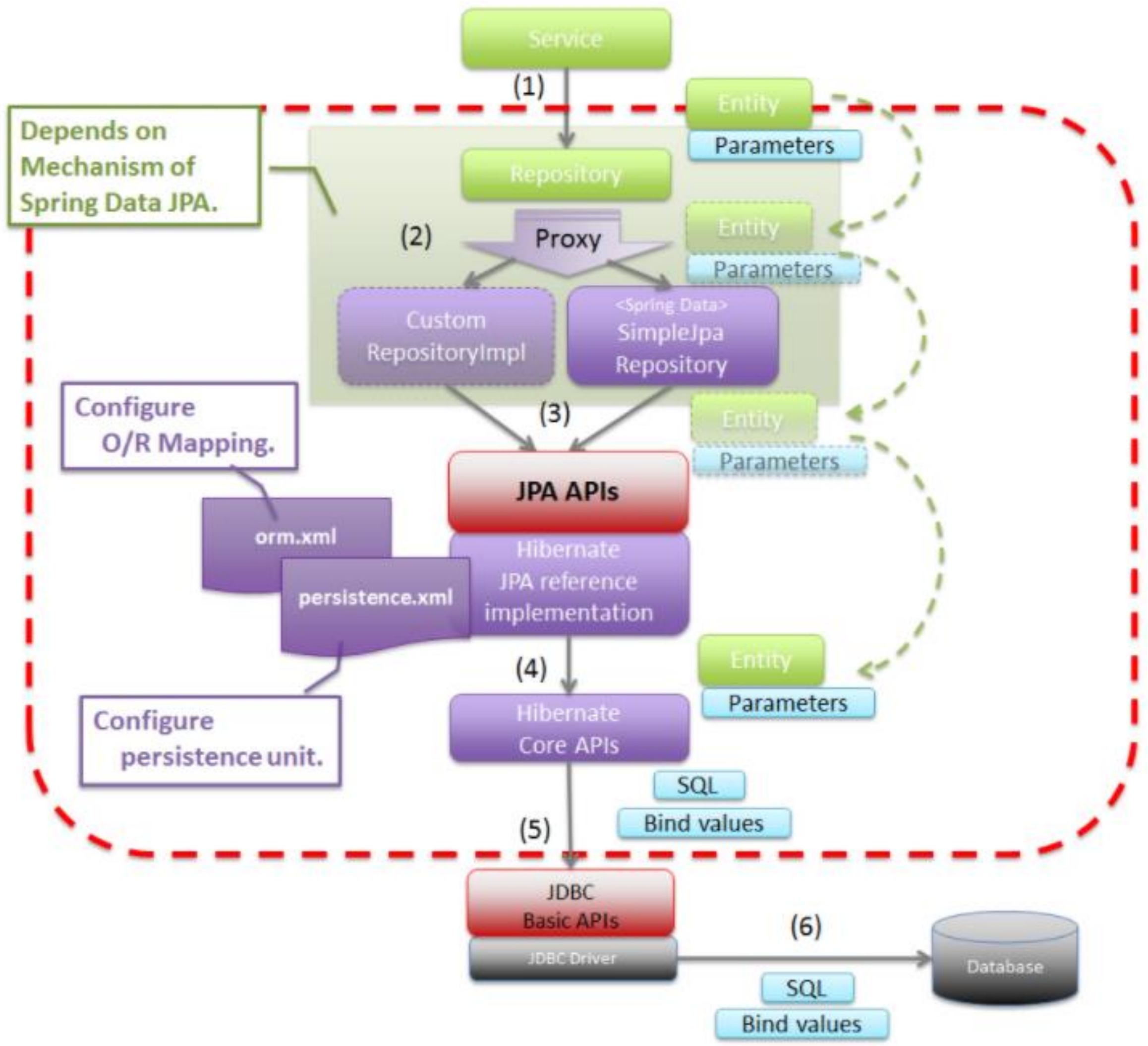


```
package org.springframework.data.jpa.repository;

import java.util.List;
import java.util.Optional;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.domain.Specification;
import org.springframework.lang.Nullable;

public interface JpaSpecificationExecutor<T> {

 Optional<T> findOne(@Nullable Specification<T> spec);

 List<T> findAll(@Nullable Specification<T> spec);

 Page<T> findAll(@Nullable Specification<T> spec, Pageable pageable);

 List<T> findAll(@Nullable Specification<T> spec, Sort sort);

 long count(@Nullable Specification<T> spec);
}
```

Section 3

How to Use Spring Data JPA Interfaces

Basic Spring Data JPA Flow



How to Use Spring Data JPA Interfaces

- *(1) Create a repository interface and extend one of the repository interfaces provided by Spring Data.*

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

}
```

- *(2) Add custom query methods to the created repository interface (if we need them that is).*

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

    long deleteByLastname(String lastname);

    List<User> removeByLastname(String lastname);

    long countByLastname(String lastname);
}
```

How to Use Spring Data JPA Interfaces

- *(3) Set up Spring to create proxy instances for those interfaces, either with JavaConfig or with XML configuration.*
  - ✓ To use Java configuration, create a class similar to the following:

```java
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
public class Config {}
```

  - ✓ To use XML configuration, define a bean similar to the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/data/jpa
     http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

   <jpa:repositories base-package="com.acme.repositories"/>

</beans>
```

***Note**:*

- ✓ *In Spring Boot is auto-configuration so without this.*

How to Use Spring Data JPA Interfaces

- *(4) Inject the repository interface to another component and use the implementation that is provided automatically by Spring.*

```java
@Service
public class CustomerServiceImpl implements CustomerService {

    @Autowired
    private CustomerRepository customerRepository;

    @Override
    @Transactional
    public List < Customer > getCustomers() {
        return customerRepository.findAll();
    }

    @Override
    @Transactional
    public void saveCustomer(Customer theCustomer) {
        customerRepository.save(theCustomer);
    }

    @Override
    @Transactional
    public Customer getCustomer(int id) throws ResourceNotFoundException {
        return customerRepository.findById(id).orElseThrow(
            () - > new ResourceNotFoundException(id));
    }

    @Override
    @Transactional
    public void deleteCustomer(int theId) {
        customerRepository.deleteById(theId);
    }
}
```

Section 4

Query Methods

Query Methods

The JPA module supports defining a *query manually as String* or have it being derived from the method name.

- ✓ *Declare an interface extending Repository or one of its subinterfaces and type it to the domain class and ID type that it should handle:*

```java
public interface PersonRepository extends Repository<Person, Long> { //…

}
```

- ✓ *Declare query methods on the interface.*

```java
public interface PersonRepository extends Repository<Person, Long> {
        List<Person> findByLastname(String lastname);
}
```

- ✓ *Set up Spring to create proxy instances for those interfaces, either with JavaConfig or with XML configuration.*

- ✓ *Inject the repository instance and use it*

Query Methods

- The repository proxy has two ways to derive a store-specific query from the method name:
  - ✓ By deriving the query from the **method name directly**.
  - ✓ By using a **manually defined query**.

Query lookup strategies

- **Query Creation:**
  - ✓ The query builder mechanism built into Spring Data repository infrastructure is useful for **building constraining queries over entities** of the repository.
  - ✓ The mechanism strips the prefixes *find…By*, *read…By*, *query…By*, *count…By*, and *get…By* from the method and starts parsing the rest of it.
  - ✓ You can define **conditions** on entity properties and concatenate them with And and Or.
- Examples: **Query creation from method names**

```
public interface PersonRepository extends Repository<User, Long> {

    List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);

    // Enables the distinct flag for the query
    List<Person> findDistinctByLastnameOrFirstname(String lastname, String firstname);
```

Query Creation

- Examples: **Query creation from method names**

```
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

    // Enabling ignoring case for an individual property
    List<Person> findByLastnameIgnoreCase(String lastname);

    // Enabling ignoring case for all suitable properties
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

    // Enabling static ORDER BY for a query
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);

    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

Query Creation

- **Special parameter handling:**
  - ✓ Besides that the infrastructure will recognize certain specific types like **Pageable** and **Sort** to apply pagination and sorting to your queries dynamically.

```
Page<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);
```

Query Creation

- **Query generated**:
  - ✓ Query creation from method names

```
public interface UserRepository extends Repository<User, Long> {

        List<User> findByEmailAddressAndLastname(String emailAddress,
                                                  String lastname);
    }
```

  - ✓ We will create a query using the JPA criteria API from this but essentially this translates into the following query:

```
select u from User u where u.emailAddress = ?1 and u.lastname = ?2
```

Limiting Query Results

- You can **limit the results** of query methods by using the **first** or **top** keywords, which you can use interchangeably.
  - ✓ You can **append an optional numeric value to top or first** to specify the maximum result size to be returned.
  - ✓ If the number is left out, a result size of 1 is assumed.
- The following example shows how to limit the query size:

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

Query Creation

- **Supported keywords inside method names:**

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
…	…	…

Using JPA NamedQueries

- **Annotation configuration**
  - ✓ Annotation configuration has the advantage of not needing another configuration file to be edited, probably lowering maintenance costs.
  - ✓ You pay for that benefit by the need to recompile your domain class for every new query declaration.
  - ✓ **Annotation based named query configuration**

```
@Entity
@Table(name = "USERS")
@NamedQuery(name = "User.findByEmailAddress",
  query = "select u from User u where u.emailAddress = ?1")
public class User {

}
```

Using JPA NamedQueries

- **Annotation configuration**
  - ✓ **Declaring interfaces:**
  - ✓ *To allow execution of these named queries all you need to do is to specify the UserRepository as follows:*

```
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByLastname(String lastname);

    User findByEmailAddress(String emailAddress);

}
```

Using @Query

- Using **named queries** to declare queries for entities is a valid approach and works fine for a **small number of queries**.
- As the queries themselves are tied to the Java method that executes them you actually can bind them directly using the Spring Data JPA **@Query** annotation rather than annotating them to the domain class.
- This will free the domain class from *persistence specific information* and *co-locate the query* to the repository interface.
- **Declare query at the query method using @Query**

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("SELECT u FROM User u WHERE u.emailAddress = ?1")
    User findByEmailAddress(String emailAddress);

}
```

Using @Query - JOIN Example

- **Table – department**

id	name	Description
1	IT	Information Technology
2	TelComm	Telecommunication
3	Ins	Insurance
4	HR	Human Resources

- **Table - employee**

id	name	email	dept_id
1	Soumitra	soumitra@gmail.com	1
2	Suman	suman@gmail.com	2
3	Avisek	avisek@gmail.com	3

Using @Query - JOIN Example

- Create a **DeptEmpDto** class:
  - ✓ A data transfer object (DTO) is an object that carries data between processes: represent data or send data to the remote call.
  - ✓ It is not a good idea to return the entity object to the client side or remote call.
  - ✓ A DTO does not have any behavior except for storage, retrieval, serialization and deserialization of its own data.

```java
package fa.training.dto;
public class DeptEmpDto {
    private String empDept;
    private String empName;
    private String empEmail;

    // setter, getter and constructor methods
}
```

Using @Query - JOIN Example

- **Or you can use Record in Java 14:**

```
package fa.training.records;

public record DeptEmpDto(String empDept, String empName, String empEmail) {

}
```

- *Records transfer this responsibility to the Java compiler, which generates the **constructor**, field **getters**, **hashCode**() and **equals**() as well **toString**() methods.*

*A record is also a **special class type** in Java. Records are intended to be used in **places where a class is created only to act as a plain data carrier**.*

Using @Query - JOIN Example

- **Create a Repository Interface:** you need to write your JOIN queries using @Query annotation.

```
public interface DepartmentRepository extends JpaRepository<Department, Integer> {

@Query("SELECT new fa.training.dto.DeptEmpDto(d.name, e.name, e.email) "
                      + "FROM Department d LEFT JOIN d.employees e")
List<DeptEmpDto> fetchEmpDeptDataLeftJoin();

@Query("SELECT new fa.training.DeptEmpDto(d.name, e.name, e.email, e.address) "
                      + "FROM Department d RIGHT JOIN d.employees e")
List<DeptEmpDto> fetchEmpDeptDataRightJoin();

}
```

Summary

- Introduction
- Spring Data Repositories Interfaces
- How to Use Spring Data Repositories Interfaces
- Query Methods

Software

CODE YOUR LIFE

THANK YOU!

CODE < YOUR LIFE >