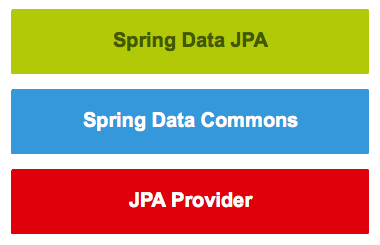
**Spring Data JPA Tutorial - Getting Started**

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 5, OpenJPA 2.4, and EclipseLink 2.6.1.  
  
**What Spring Data JPA?**

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

**[](https://3.bp.blogspot.com/-25dgXaEwfjw/W9skvXDdZcI/AAAAAAAAEjI/lHbqfkykUtw3FzzjfEeKxQHEGGxvXBVtgCLcBGAs/s1600/springdatajpalayers.png)**

1. *Spring Data JPA* provides support for creating JPA repositories by extending the Spring Data repository interfaces.
2. *Spring Data Commons* provides the infrastructure that is shared by the datastore-specific Spring Data projects.
3. *The JPA Provider* (like hibernate) implements the Java Persistence API.

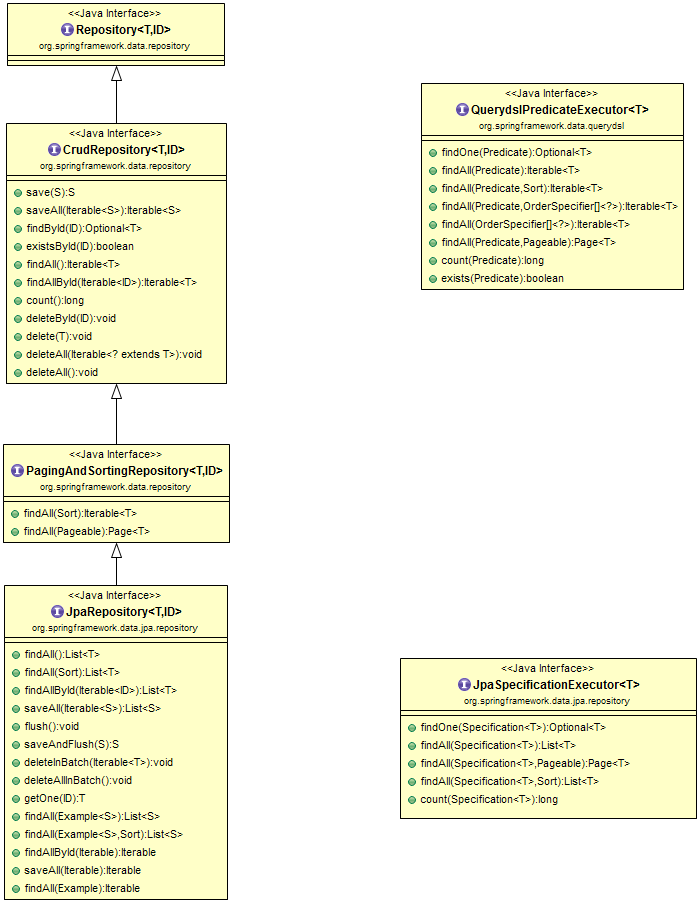
**Spring Data JPA Features**

* Sophisticated support to build repositories based on *Spring* and *JPA*
* Support for Querydsl predicates and thus type-safe JPA queries
* Transparent auditing of a domain class
* Pagination support, dynamic query execution, the ability to integrate custom data access code
* Validation of @Query annotated queries at a bootstrap time
* Support for XML based entity mapping
* JavaConfig based repository configuration by introducing *@EnableJpaRepositories*.

Let's quickly discuss important repositories or interfaces of *Spring Data Commons* and *Spring Data JPA*.

**Spring Data Commons and Spring Data JPA Repositories/interfaces**

Below diagram shows the main interfaces from Spring Data Commons and Spring Data JPA modules.

**[](https://4.bp.blogspot.com/-Xai15KaHfZQ/W9soAydfoJI/AAAAAAAAEjU/OfYzs-a_dUsfU5njdj5wP1iiP2yqzZd5ACLcBGAs/s1600/spring-data-jpa-diagram.png)**

*Spring Data Commons* is part of the umbrella Spring Data project that provides shared infrastructure across the Spring Data projects. It contains technology neutral repository interfaces as well as a metadata model for persisting Java classes. Spring Data Commons project provides the following interfaces:

1. Repository<T, ID extends Serializable> interface
2. CrudRepository<T, ID extends Serializable> interface
3. PagingAndSortingRepository<T, ID extends Serializable> interface
4. QueryDslPredicateExecutor interface

**The Repository<T, ID extends Serializable> interface**

The *Repository<T, ID extends Serializable> interface* is a marker interface that has two purposes:

1. It captures the type of the managed entity and the type of the entity’s id.
2. 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> {

}

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

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

Let's look at the methods/APIs that the *CrudRepository* interface provides:

package org.springframework.data.repository;

import java.util.Optional;

@NoRepositoryBean

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

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

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

Optional < T > findById(ID id);

boolean existsById(ID id);

Iterable < T > findAll();

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

long count();

void deleteById(ID id);

void delete(T entity);

void deleteAll();

}

Let's look at the usage of each method with description.

1. *long count()* - Returns the number of entities available.
2. *void delete(T entity)* - Deletes a given entity.
3. *void deleteAll()* - Deletes all entities managed by the repository.
4. *void deleteAll(Iterable<? extends T> entities)* - Deletes the given entities.
5. *void deleteById(ID id)* - Deletes the entity with the given id.
6. *boolean existsById(ID id)* - Returns whether an entity with the given id exists.
7. *Iterable findAll()* - Returns all instances of the type.
8. *Iterable findAllById(Iterable ids)* - Returns all instances of the type with the given IDs.
9. *Optional findById(ID id)* - Retrieves an entity by its id.
10. *save(S entity)*- Saves a given entity.
11. *Iterable saveAll(Iterable entities)*- Saves all given entities.

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

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.

Let's look at the methods/APIs that the *PagingAndSortingRepository* interface provides:

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.

\*

\* @param sort

\* @return 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.

\*

\* @param pageable

\* @return a page of entities

\*/

Page < T > findAll(Pageable pageable);

}

**The 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.

Let's look at the methods/APIs that the *QueryDslPredicateExecutor* interface provides:

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<T, ID extends Serializable> 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.

Let's look at the methods/APIs that the *JpaRepository* interface provides:

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 > 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.

Let's look at the methods/APIs that the *JpaSpecificationExecutor* interface provides:

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);

}

**How to Use Spring Data JPA interfaces**

So far, we have seen what is *Spring Data JPA* and it's main interfaces. Now the question is how to use them. so follow below three steps to use *Spring Data JPA* repositories/interfaces.

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

public interface CustomerRepository extends CrudRepository<Customer, Long> {

}

1. 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);

}

1. 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:

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

@EnableJpaRepositories

public class Config {}

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

<?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>

1. Inject the repository interface to another component and use the implementation that is provided automatically by Spring.

@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);

}

}