Spring Data:

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data aims to significantly improve the implementation of data access layers by reducing the effort to the amount that’s actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

## Spring Data modules

* Spring Data Commons - Core Spring concepts underpinning every Spring Data project.
* Spring Data Gemfire - Provides easy configuration and access to GemFire from Spring applications.
* Spring Data JPA - Makes it easy to implement JPA-based repositories.
* Spring Data KeyValue - Map-based repositories and SPIs to easily build a Spring Data module for key-value stores.
* Spring Data LDAP - Provides Spring Data repository support for Spring LDAP.
* Spring Data MongoDB - Spring based, object-document support and repositories for MongoDB.
* Spring Data REST - Exports Spring Data repositories as hypermedia-driven RESTful resources.
* Spring Data Redis - Provides easy configuration and access to Redis from Spring applications.
* Spring Data for Apache Cassandra - Spring Data module for Apache Cassandra.
* Spring Data for Apache Solr - Spring Data module for Apache Solr.

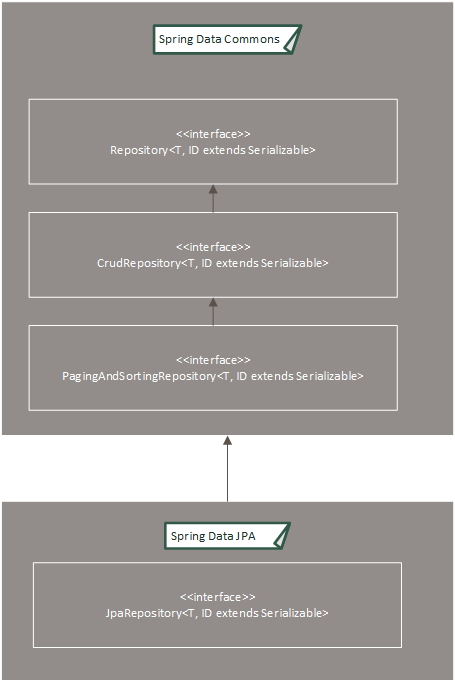
The following are the three base interfaces defined in the **spring data commons** project.

1. Repository
2. CrudRepository
3. PagingAndSortingRepository

Spring data JPA interface:

1. JPARepository.

The following is the hierarchy of spring data:



Example :

Here I am using spring data with spring boot

Step 1: Add required dependency in pom.xml

<dependency>

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

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

</dependency>

Step 2: Define a JPA entity

@Entity

public class Employee {

private @Id @GeneratedValue Long id;

private String firstName, lastName, description;

private Employee() {}

public Employee(String firstName, String lastName, String description) {

this.firstName = firstName;

this.lastName = lastName;

this.description = description;

}

}

Step3: create a repository interface extending CrudRepository

public interface EmployeeRepository extends CrudRepository<Employee, Long> {

Employee findByFirstName(String firstName);

List<Employee> findByLastName(String lastName);

}

This interface extends Spring Data’s CrudRepository and defines the type (Employee) and the id type (Long).

Spring Data (having been autoconfigured by Boot, [SQL](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-sql) or [NoSQL](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-nosql)) will automatically craft a concrete set of operations:

* save(Employee)
* delete(Employee)
* find(Employee)
* find(Long)
* findAll()

On top of the CRUD operations inherited from CrudRepository, the interface defines two query methods.

* findByFirstName(…) automatically writes a JPA query based on firstName and only return the first employee found.
* findByLastName(…) automatically writes a JPA query based on lastName and returns a collection.