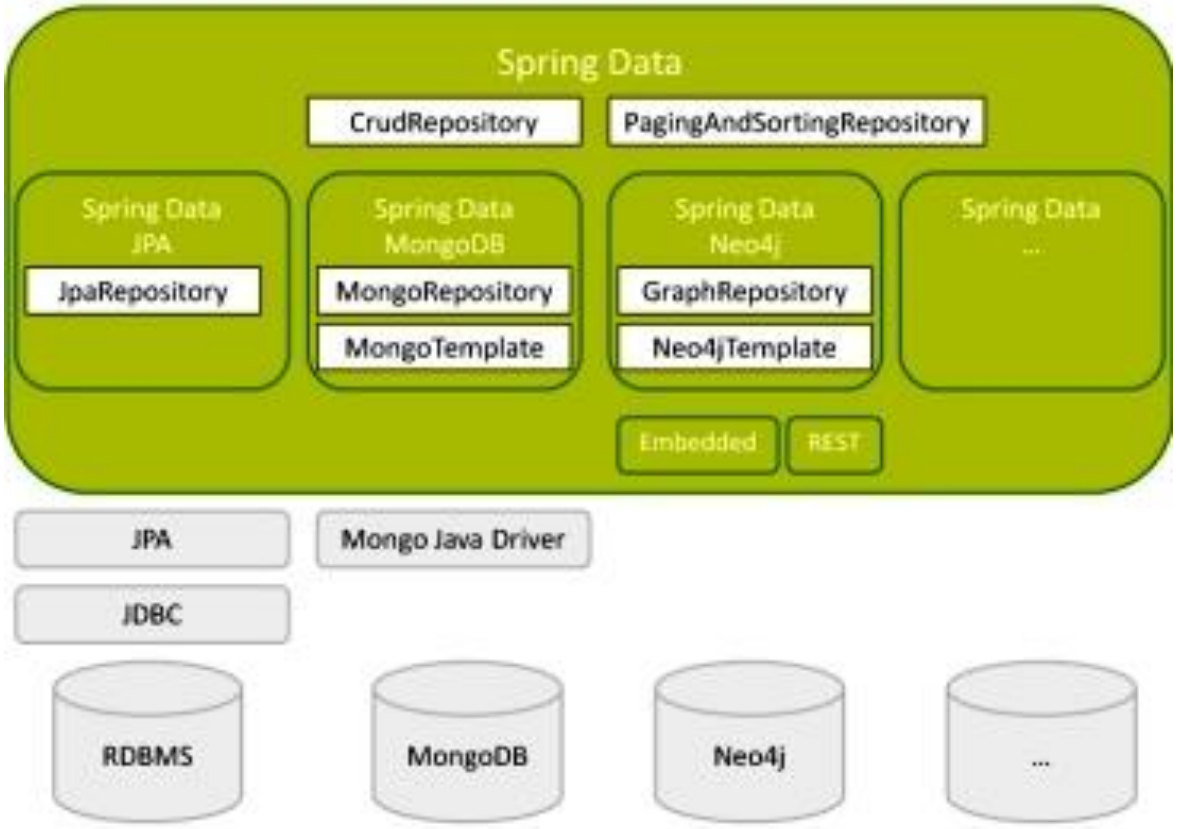
KLFSPRO
Learn & Grow

Spring Boot + spring data + jpa

Spring Data

- **Spring Data is a high level SpringSource project whose purpose is to unify and ease the access to different kinds of persistence stores, both relational database systems and NoSQL data stores.**



- JPA is part of the JEE stack and defines a unified API for accessing relational databases and performing O/R mapping.
- MongoDB is a scalable, high-performance, open source, document-oriented database.
- Neo4j is a graph database, a fully transactional database that stores data structured as graphs.

Main modules

- **Spring Data Commons - Core Spring concepts underpinning every Spring Data module.**
- **Spring Data JDBC - Spring Data repository support for JDBC.**
- **Spring Data JDBC Ext - Support for database specific extensions to standard JDBC including support for Oracle RAC fast connection failover, AQ JMS support and support for using advanced data types.**
- **Spring Data JPA - Spring Data repository support for JPA.**
- **Spring Data KeyValue - Map based repositories to easily build a Spring Data module for key-value stores.**
- **Spring Data LDAP - Spring Data repository support for Spring LDAP.**
- **Spring Data MongoDB - Spring based, object-document support and repositories for MongoDB.**
- **Spring Data Redis - Easy configuration and access to Redis from Spring apps**
- **Spring Data for Apache Cassandra - Easy configuration and access to Apache Cassandra or large scale, highly available, data oriented Spring applications.**

Spring Data JPA

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

1.Repository

It is the central interface in the spring data repository abstraction. This is a marker interface. If you are extending this interface, you have to declare your own methods and the implementations will be provided by the spring run-time. For this interface also we have to pass two parameters:type of the entity and type of the entity's id field.This is the super interface for CrudRepository.

CrudRepository

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

2.CrudRepository

CrudRepository provides methods for the CRUD operations. This interface extends the Repository interface. When you define CrudRepository, you have to pass the two parameters: **type of the entity and type of the entity's id field.** This interface has the following methods:

```
1.  S save(S entity)
2.  Iterable<S> save(Iterable<S> entities);
3.  T findOne(ID primaryKey)
4.  boolean exists(ID primaryKey)
5.  Iterable findAll()
6.  Iterable<T> findAll(Iterable<ID> ids);
7.  Long count()
8.  delete(ID id);
9.  void delete(T entity)
10. delete(Iterable<? extends T> entities);
11. deleteAll();
```

If you are extending the CrudRepository, there is no need for implementing your own methods. Just extend this interface and leave it as blank. Required implementations are provided at run time.

- Look at our example:

```
public interface BookRepository extends JpaRepository<Book,Long>
{
}
```

We have used **JpaRepository**, which is the special version specific to the JPA technology. Unless until you are using any of the JPA specific things in your applications, it is highly recommended to use the **CrudRepository**, because it will not tie your application with any specific store implementations.

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

3. PagingAndSortingRepository

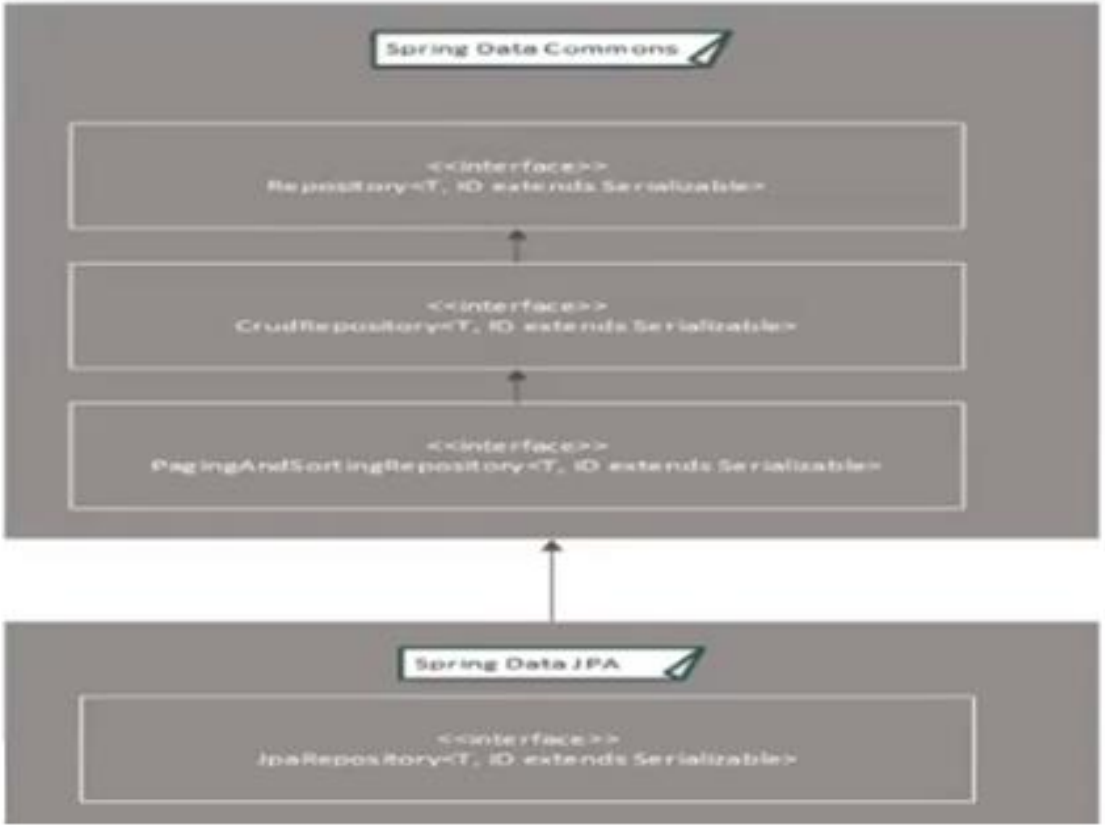
- This is extension of **CrudRepository**. It is specialized version for the paging operations.

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
  extends CrudRepository<T, ID> {
Iterable<T> findAll(Sort sort);
Page<T> findAll(Pageable pageable);
}
```

Using Interfaces

- **Two reasons to use these interfaces(CrudRepository,JpaRepository)**
- **1. If your interfaces are extending these interfaces then it helps spring boot to find your interfaces. And create proxy objects for them.**
- **2. It provides you some methods which performs some common operations for you without writing its implementations.**

The below diagram shows the repository module structure.



JpaRepository vs CrudRepository

Both, **JpaRepository** and **CrudRepository** are base interfaces in Spring Data. Application developer has to choose any of the base interfaces for your own repository. It has two purposes,

- One is to allow spring data to create proxy instance for your repository interface.
- Second one is to inherit the maximum default functionality from the base interfaces without declaring your own methods.

- If your dao layer is tightly coupled with data store then you should go for JPA repository
- findAll method in JPA repository returns List and in CrudRepository it returns Iterable

- So if we want to perform any batch operation then we have following methods in JpaRepository
- deleteInBatch
- deleteAllInBatch
- getOne to get one entity object

Steps

1. create a Maven SpringBoot application
2. Add an entity class
3. Add service interface and class
4. Add DAO layer interface extends CrudRepository/JpaRepository Interface

Entries I application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/springbootdb
spring.datasource.username=root
spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Demo

- **Step-1 : Create a simple Maven project.**

- Step-2: Bring in the following into pom.xml
  - Step 2.1 : update the Maven project

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>com.oracle</groupId>
        <artifactId>ojdbc6</artifactId>
        <version>11.2.0</version>
        <scope>system</scope>
        <systemPath>${basedir}/lib/ojdbc6.jar</systemPath>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
</dependencies>
```

Spring Data JPA dependency : not only adds Spring Data JPA packages but also transitively includes Hibernate as the JPA implementation

Demo

```java
@Entity
public class Coffeetable {

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="s2")
    @SequenceGenerator(name="s2", sequenceName="s2", allocationSize=1)
    int id;
    String tname, mfr;
    double price;

    //appropriate cons, getter and setters
```

entity

Demo

- **Step-4 : Declare the following interface to manage Coffeetable items**

```
import org.springframework.data.repository.CrudRepository;

public interface CoffeeTableDao extends CrudRepository<Coffeetable, Integer>{

}
```

- Spring Boot Data enables JPA repository support by default.
- CrudRepository provides generic CRUD operation on a repository for a specific type; contains methods such as save, findById, delete, count etc.
- CrudRepository is a Spring data interface and to use it we need to create our interface by extending CrudRepository.
- The CoffeeTableDao extends from the CrudRepository. It provides the type of the entity and of its primary key
- Spring provides CrudRepository implementation class automatically at runtime.
- It Spring boot automatically detects our repository if the package of that repository interface is the same or sub-package of the class annotated with @SpringBootApplication.

CrudRepository

- **The central interface in Spring Data repository abstraction is Repository**
  - It takes the domain class to manage as well as the id type of the domain class as type arguments.
  - The CrudRepository provides sophisticated CRUD functionality for the entity class that is being managed.
  - We need not to implement our interface, its implementation will be created automatically at runtime.

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

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

    T findOne(ID primaryKey);                ❷

    Iterable<T> findAll();                   ❸

    Long count();

    void delete(T entity);                   ❹

    boolean exists(ID primaryKey);           ❺

    // … more functionality omitted.         ❻
}
```

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.

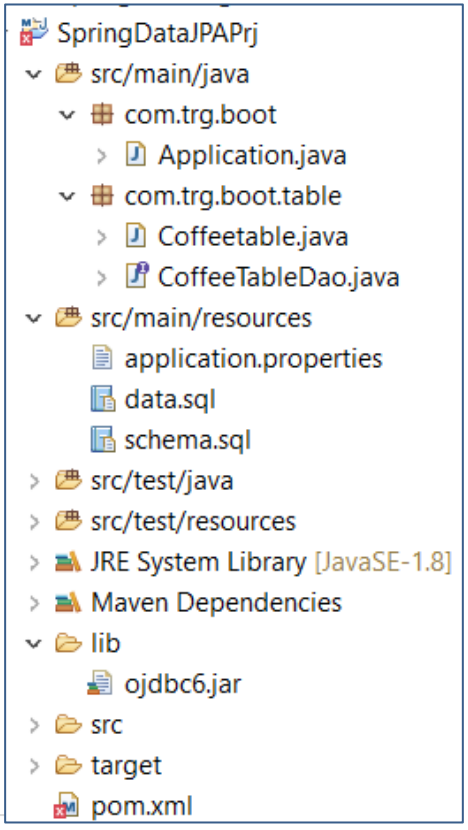
Demo

- **Step-5 : Create and put following entries in application.properties,**

```
/* schema.sql */
create table coffeetable (id number,
                          tname varchar(20),
                          mfr varchar(10),
                          price number(7,2),
                          CONSTRAINT CT_PK PRIMARY KEY (id));

CREATE SEQUENCE s2 INCREMENT BY 1 START WITH 1  MAXVALUE 10000 NOCYCLE;
```

```
/* data.sql */
insert into coffeetable values(101,'Kosmo table','Spacewood',7499);
insert into coffeetable values(102,'Kuro table', 'Mintwud',5000);
insert into coffeetable values(103,'Yuko table','Mintwud',7499);
insert into coffeetable values(104,'Arabia table','Urban L',4199);
insert into coffeetable values(105,'Ormond table','Urban L',13000);
```

```
#application.properties
spring.datasource.initialization-mode=always
spring.jpa.hibernate.ddl-auto=none

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=oracle
spring.datasource.password=oracle123
spring.datasource.driver-class=oracle.jdbc.driver.OracleDriver
```



Demo

- **Step-6 : The client application**

```
@SpringBootApplication
public class Application implements CommandLineRunner {

    @Autowired
    CoffeeTableDao coffeeTableDao;

    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println(coffeeTableDao);
        Coffeetable newtable = new Coffeetable(112,"Mario table","PepperFry", 10000);
        coffeeTableDao.save(newtable);
        Iterable<Coffeetable> itemData = coffeeTableDao.findAll();
        List<Coffeetable> itemList = new ArrayList<>();
        itemData.forEach(itemList::add);
        for (Coffeetable table : itemList)
            System.out.println(table);
    }
}
```

SpringDataJPAPrj

```
Coffeetable [id=101, tname=Kosmo table, mfr=Spacewood, price=7499.0]
Coffeetable [id=102, tname=Kuro table, mfr=Mintwud, price=5000.0]
Coffeetable [id=103, tname=Yuko table, mfr=Mintwud, price=7499.0]
Coffeetable [id=104, tname=Arabia table, mfr=Urban L, price=4199.0]
Coffeetable [id=105, tname=Ormond table, mfr=Urban L, price=13000.0]
Coffeetable [id=1, tname=Mario table, mfr=PepperFry, price=10000.0]
```

Spring Data JPA provides three different approaches for creating custom queries with query methods.

- **Creating Database Queries From Method Names**: describes how we can create database queries from the method names of our query methods.
- **Creating Database Queries With Named Queries** :describes how we can create database queries by using named queries.
- **Creating Database Queries With the @Query Annotation**: describes how we can create database queries by annotating our query methods with the ***@Query*** annotation.

Query Creation from Method Name

- Spring Data JPA has a built in query creation mechanism which can be used for parsing queries straight from the method name of a query method. This mechanism first removes common prefixes from the method name and parses the constraints of the query from the rest of the method name. The query builder mechanism is described with more details in **Defining Query Methods Subsection of Spring Data JPA reference documentation.**

- Using this approach is quite simple. All you have to do is to ensure that the method names of your repository interface are created by combining the property names of an entity object and the supported keywords. **The Query Creation Subsection of the Spring Data JPA reference documentation** has nice examples concerning the usage of supported keywords.

The source code of the repository

```
import org.springframework.data.repository.Repository;
/**
 * Specifies methods used to obtain and modify person related information
 * which is stored in the database.

 */
public interface PeopleManangementDao extends extends Repository<Person, Integer> {
    /**
    * Finds persons by using the last name as a search criteria.
    * @param lastName
    * @return  A list of persons which last name is an exact match with the given last name.
    *          If no persons is found, this method returns an empty list.
    */
   public List<Person> findByLastName(String lastName);

}
```

4.4.2. 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. The introducing clause can contain further expressions, such as a Distinct to set a distinct flag on the query to be created. However, the first By acts as delimiter to indicate the start of the actual criteria. At a very basic level, you can define conditions on entity properties and concatenate them with And and Or. The following example shows how to create a number of queries:**

Example 13. Query creation from method names

- interface PersonRepository extends Repository<Person, Long> {
- List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
- // Enables the distinct flag for the query
- List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
- 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);
- }

Table 3. 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
Before	findByStartDateBefore	… where x.startDate < ?1

JPA Named Queries

Spring Data JPA provides also support for the JPA Named Queries. You have got following alternatives for declaring the named queries:

- You can use either *named-query* XML element or ***@NamedQuery*** annotation to create named queries with the JPA query language.
- You can use either *named-native-query* XML element or ***@NamedNative*** query annotation to create queries with SQL if you are ready to tie your application with a specific database platform.

The only thing you have to do to use the created named queries is to name the query method of your repository interface to match with the name

The source code of the *Person* class is given in following:

```
@Entity
@Table(name="person_table")
@NamedQuery(name = "Person.findByName", query = "SELECT p FROM Person p WHERE  p.lastName = ?1")
public class Person {
public Person() {
}
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="person_id")
private int id;

@Column(name="first_name",length=60,nullable=false)
private String firstName;

@Column(name="last_name",length=60,nullable=false)
private String lastName;

@Column(name="email",unique=true)
private String email;

@Column(name="creation_date")
private Date creationDate;
Setter/getters methods and rest of the code.....

}
```

The relevant part of my PeopleManangementDao interface looks following:

```
import org.springframework.data.repository.Repository;
/**
 * Specifies methods used to obtain  person related information
 * which is stored in the database.

 */
public interface PeopleManangementDao extends Repository<Person, Integer>{
    /**
    * Finds person by using the last name as a search criteria.
    * @param lastName
    * @return  A list of persons whose last name is an exact match with the given last name.
    *          If no persons is found, this method returns null.
    */
  public List<Person> findByName(String lastName);
}
```

Drawback JPA Named Queries Approach

- Using named queries is valid option if your application is small or if you have to use native queries. If your application has a lot of custom queries, this approach will litter the code of your entity class with query declarations (You can of course use the XML configuration to avoid this but in my opinion this approach is even more horrible).

@Query Annotation

- The *@Query* annotation can be used to create queries by using the JPA query language and to bind these queries directly to the methods of your repository interface. When the query method is called, Spring Data JPA will execute the query specified by the *@Query* annotation (If there is a collision between the *@Query* annotation and the named queries, the query specified by using *@Query* annotation will be executed).

The source code of the repository method which is implemented by using this approach is given in following:

```
import org.springframework.data.repository.Repository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
 /**
 * Specifies methods used to obtain  person related information
 * which is stored in the database.

 */
public interface PeopleManangementDao  extends Repository<Person,  Integer>{
    /**
    * Finds person  by using the last name as a search criteria.
    * @param lastName
    * @return   A list of persons  whose last name is an exact match with the given last name.
    *           If no persons  is found,  this method returns  null.
    */
   @Query("SELECT  p FROM Person p WHERE LOWER(p.lastName)  = LOWER(:lastName)")
   public  List<Person>  find(@Param('lastName') String lastName);
}
```

:lastname is parameter you need to pass as parameter using @param

NOTE:This approach gives you access to the JPA query language and keeps your queries in the repository layer where they belong. On the other hand, you cannot use the *@Query* annotation if the JPA query language cannot be used to create the query you need.

4.4.4. Special parameter handling

To handle parameters in your query you simply define method parameters as already seen in the examples above. Besides that the infrastructure will recognize certain specific types like `Pageable` and `Sort` to apply pagination and sorting to your queries dynamically.

Example 14. Using Pageable, Slice and Sort in query methods

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

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

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

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

The first method allows you to pass an `org.springframework.data.domain.Pageable` instance to the query method to dynamically add paging to your statically defined query. A `Page` knows about the total number of elements and pages available. It does so by the infrastructure triggering a count query to calculate the overall number. As this might be expensive depending on the store used, `Slice` can be used as return instead. A `Slice` only knows about whether there's a next `Slice` available which might be just sufficient when walking through a larger result set.

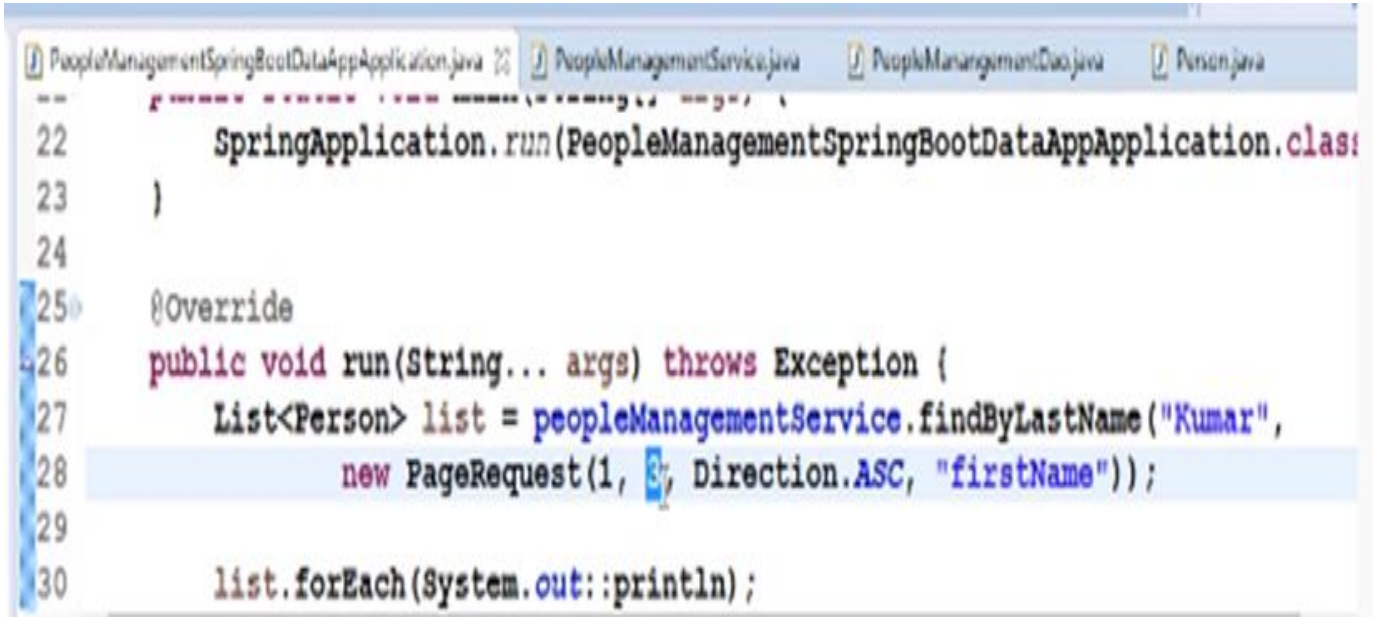
PeopleManagementSpringBootDataAppApplication.java | PeopleManagementService.java | PeopleManangementDao.java | Person.java

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort.Direction;

import com.infotech.people.manangement.app.entities.Person;
import com.infotech.people.manangement.app.service.PeopleManagementService;

@SpringBootApplication
public class PeopleManagementSpringBootDataAppApplication implements CommandLineRunner{

    @Autowired
    private PeopleManagementService peopleManagementService;

    public static void main(String[] args) {
        SpringApplication.run(PeopleManagementSpringBootDataAppApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        List<Person> list = peopleManagementService.findByLastName("Kumar",
                new PageRequest(0, 4, Direction.ASC, "firstName"));

        list.forEach(System.out::println);
```

Zeroth page first 4 records.

```
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.stereotype.Service;

import com.infotech.people.manangement.app.dao.PeopleManangementDao;
import com.infotech.people.manangement.app.entities.Person;

@Service
public class PeopleManagementService {

    @Autowired
    private PeopleManangementDao peopleManangementDao;

    public List<Person> findByLastName(String lastName, PageRequest pageRequest) {
        return peopleManangementDao.findByLastName(lastName,pageRequest);
    }
}
```

```
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.stereotype.Service;

import com.infotech.people.manangement.app.dao.PeopleManangementDao;
import com.infotech.people.manangement.app.entities.Person;

@Service
public class PeopleManagementService {

    @Autowired
    private PeopleManangementDao peopleManangementDao;

    public List<Person> findByLastName(String lastName, PageRequest pageRequest) {
        return peopleManangementDao.findByLastName(lastName,pageRequest);
    }
}
```



1,3 indicates 2 nd page next 4 records. Because page number starts with 0