**Pagination and Sorting using Spring Data JPA - PagingAndSortingRepository**

**author:**[**Ramesh Fadatare**](https://www.blogger.com/profile/14691512106162803120)

[**spring boot**](https://www.javaguides.net/search/label/Spring%20Boot?&max-results=10)[**spring data jpa tutorial**](https://www.javaguides.net/search/label/Spring%20Data%20JPA%20Tutorial?&max-results=10)

[**← Back to All Spring Data JPA Tutorials**](https://www.javaguides.net/p/spring-data-jpa-tutorial.html)

In this article, we will learn how to implement pagination and sorting using Spring Data JPA.

Learn everything about Spring Data JPA at [**https://www.javaguides.net/p/spring-data-jpa-tutorial.html**](https://www.javaguides.net/p/spring-data-jpa-tutorial.html)

As you know, pagination allows the users to see a small portion of data at a time (a page), and sorting allows the users to view the data in a more organized way. Both paging and sorting help the users consume information more easily and conveniently.

Let's first discuss pagination implementation using Spring Data JPA then we will move to sorting implementation.

**1. Pagination Implementation**

**Understand Spring Data JPA’s Pagination APIs**

To use paging and sorting APIs provided by Spring Data JPA, your repository interface must extend the PagingAndSortingRepository interface.

*PagingAndSortingRepository* is an extension of the *CrudRepository* to provide additional methods to retrieve entities using the pagination and sorting abstraction. It provides two methods :

* *Page findAll(Pageable pageable)* – returns a Page of entities meeting the paging restriction provided in the Pageable object.
* *Iterable findAll(Sort sort)* – returns all entities sorted by the given options. No paging is applied here.

Here is the internal source code of *PagingAndSortingRepository* interface:

@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);

}

JpaRepository interface extends the PagingAndSortingRepository interface so if your repository interface is of type JpaRepository, you don’t have to make a change to it.

Here is the internal source code of the *JpaRepository* interface which extends PagingAndSortingRepository interface:

@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);

}

Let's understand the usage of the *PagingAndSortingRepository* interface and it''s methods with an example.

Consider we have an *Employee* JPA entity:

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private long id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column(name = "email")

private String email;

Now, we use the following code to get the first page from the database, with 5 items per page:

int pageNumber = 1;

int pageSize = 5;

Pageable pageable = PageRequest.of(pageNumber, pageSize);

Page<Employee> page = employeeRepository.findAll(pageable);

Then you can get the actual content as follows:

List<Employee> listEmployees = page.getContent();

With a Page object you can know the total rows in the database and the total pages according to the given page size:

long totalItems = page.getTotalElements();

int totalPages = page.getTotalPages();

**Create EmployeeRepository**

Let's create an EmployeeRepository interface which extends *JpaRepository* which intern extends the *PagingAndSortingRepository* interface so we can leverage pagination API:

package net.javaguides.springboot.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import net.javaguides.springboot.model.Employee;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long>{

}

Let's see how to use *EmployeeRepository* in the service layer.

**Create EmployeeService interface**

In the service layer, let's create the *EmployeeService* interface and add the following code to it:

package net.javaguides.springboot.service;

import java.util.List;

import org.springframework.data.domain.Page;

import net.javaguides.springboot.model.Employee;

public interface EmployeeService {

Page < Employee > findPaginated(int pageNo, int pageSize);

}

**EmployeeServiceImpl Class**

In the service layer, let's create an *EmployeeServiceImpl* class which implements the *EmployeeService* interface and provides implementation for pagination:

@Override

public Page<Employee> findPaginated(int pageNo, int pageSize) {

Pageable pageable = PageRequest.of(pageNo - 1, pageSize);

return this.employeeRepository.findAll(pageable);

}

Check out the complete example at [**Pagination and Sorting with Spring Boot, ThymeLeaf, Spring Data JPA, Hibernate, MySQL**](https://www.javaguides.net/2020/06/pagination-and-sorting-with-spring-boot-thymeleaf-spring-data-jpa-hibernate-mysql.html)

**2. Sorting Implementation**

**Understand Spring Data JPA’s Sorting API**

The Spring Data JPA provides PagingAndSortingRepository interface which supports sorting and pagination with the following APIs:

@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);

}

First, create a **Sort**object like this:

Sort sort = Sort.by(“fieldName”).ascending();

This will sort the result by **fieldName**in ascending order. **fieldName**must match a field name declared in the entity class.

We can also sort by more than one field, for example:

Sort sort = Sort.by("fieldName1").ascending().and(Sort.by("fieldName2").descending());

Then we pass the **Sort**object to create a **Pageable**as follows:

Pageable pageable = PageRequest.of(pageNum - 1, pageSize, sort);

Finally, we pass the *pageable* object to the *findAll()* method:

this.employeeRepository.findAll(pageable);

**Create EmployeeRepository**

Let's create an EmployeeRepository interface which extends *JpaRepository* which intern extends the *PagingAndSortingRepository* interface so we can leverage sorting API:

package net.javaguides.springboot.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import net.javaguides.springboot.model.Employee;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long>{

}

Let's see how to use *EmployeeRepository* in the service layer.

**EmployeeService.java interface changes**

Let's add two fields to the existing method:

Page<Employee> findPaginated(int pageNo, int pageSize, String sortField, String sortDirection);

The complete code:

package net.javaguides.springboot.service;

import java.util.List;

import org.springframework.data.domain.Page;

import net.javaguides.springboot.model.Employee;

public interface EmployeeService {

Page < Employee > findPaginated(int pageNo, int pageSize, String sortField, String sortDirection);

}

**EmployeeServiceImpl.java class changes**

The sorting logic implemented in the below method:

@Override

public Page<Employee> findPaginated(int pageNo, int pageSize, String sortField, String sortDirection) {

Sort sort = sortDirection.equalsIgnoreCase(Sort.Direction.ASC.name()) ? Sort.by(sortField).ascending() :

Sort.by(sortField).descending();

Pageable pageable = PageRequest.of(pageNo - 1, pageSize, sort);

return this.employeeRepository.findAll(pageable);

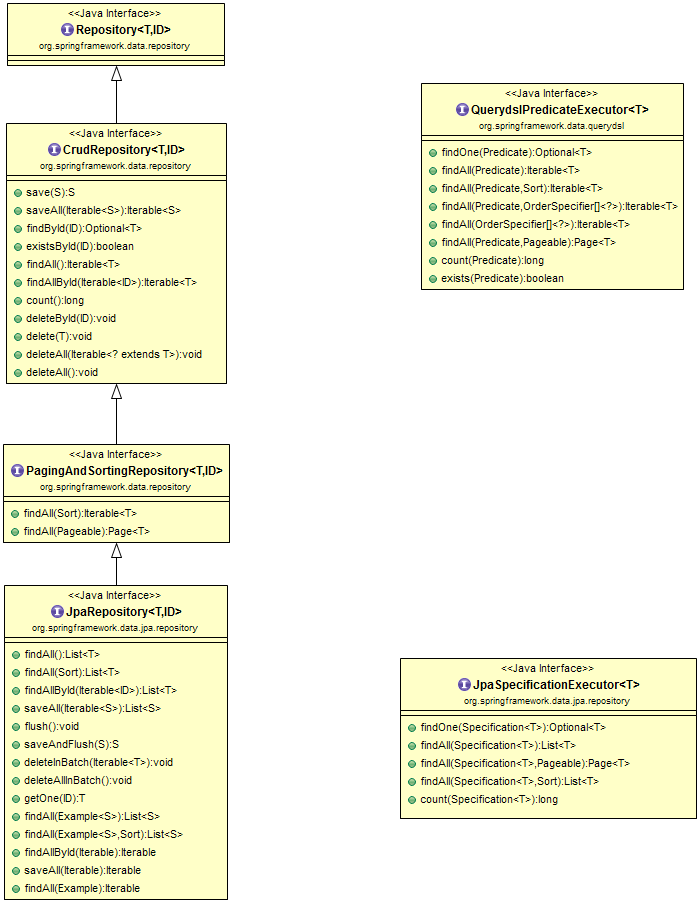
}

Note that we are using the same method to perform both pagination and sorting operations.

Check out the complete example at [**Pagination and Sorting with Spring Boot, ThymeLeaf, Spring Data JPA, Hibernate, MySQL**](https://www.javaguides.net/2020/06/pagination-and-sorting-with-spring-boot-thymeleaf-spring-data-jpa-hibernate-mysql.html)

**3. Spring Data JPA interfaces**

Check out the below diagram which shows the main interfaces of Spring Data JPA for your reference:

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

**Video Tutorial**

Watch this video on my YouTube channel to understand more about pagination and sorting implementations in Spring boot application using Spring Data JPA:

[**← Back to All Spring Data JPA Tutorials**](https://www.javaguides.net/p/spring-data-jpa-tutorial.html)

[**spring boot**](https://www.javaguides.net/search/label/Spring%20Boot?&max-results=10)[**spring data jpa tutorial**](https://www.javaguides.net/search/label/Spring%20Data%20JPA%20Tutorial?&max-results=10)

**Free Spring Boot Tutorial | Full In-depth Course | Learn Spring Boot in 10 Hours**

Watch this course on YouTube at [**Spring Boot Tutorial | Fee 10 Hours Full Course**](https://youtu.be/_thI-4AF7M8)

[SPRING BOOT](https://www.javaguides.net/search/label/Spring%20Boot)[SPRING DATA JPA TUTORIAL](https://www.javaguides.net/search/label/Spring%20Data%20JPA%20Tutorial)

**Comment**