Spring Data JPA

Pagination & Sorting

By

Pichet Limvajiranan

Spring Data Sort and Order

- The Sort class provides sorting options for database queries with more flexibility in choosing single/multiple sort columns and directions (ascending/descending).
  - we use by(), descending(), and() methods to create Sort object and pass it to Repository.findAll()
- You can sort results by Sort and Order object with one or more specified variables.
- Sorting can be done in ascending or descending order.

```
@GetMapping("")
public List<Customer> getAllCustomers(@RequestParam String sortBy) {
   return repository.findAll(Sort.Direction.DESC, Sort.by(sortBy));
}
```

Sort & Order object example

```java
// order by 'published' column - ascending
List<Tutorial> tutorials = tutorialRepository.findAll(Sort.by("published"));

// order by 'published' column, descending
tutorialRepository.findAll(Sort.by("published").descending());

// order by 'published' column - descending, then order by 'title' - ascending
tutorialRepository.findAll(Sort.by("published").descending().and(Sort.by("title")));
```

```java
List<Order> orders = new ArrayList<Order>();
Order order1 = new Order(Sort.Direction.DESC, "published");
orders.add(order1);
Order order2 = new Order(Sort.Direction.ASC, "title");
orders.add(order2);

List<Tutorial> tutorials = tutorialRepository.findAll(Sort.by(orders));
```

JpaRepository with Pagination

- findAll(Pageable pageable): returns a Page of entities meeting the paging condition provided by Pageable object.
- Pagination can be added by creation of PageRequest object which is implementation of Pageable interface.
- Similar to sorting adding pagination depends from type of Repository extended by our interface.

```
@GetMapping("")
 public Page<Customer> getAllCustomers(
        @RequestParam(defaultValue = "1") int page,
        @RequestParam(defaultValue = "10") int pageSize) {
        return repository.findAll( PageRequest.of(page, pageSize) );
 }
```

Page<T> Object

```
{
   "content": [
      {
         "id": 323,
         "customerName": "Down Under Souveniers, Inc",
         "contactLastName": "Graham",
         "contactFirstName": "Mike",
         "phone": "+64 9 312 5555",
         "addressLine1": "162-164 Grafton Road",
         "addressLine2": "Level 2",
         :
    ]
   "pageable": {
      "sort": {
         "empty": false,
         "sorted": true,
         "unsorted": false
      },
```

```
      "offset": 5,
      "pageSize": 5,
      "pageNumber": 1,
      "unpaged": false,
      "paged": true
   },
   "last": false,
   "totalPages": 25,
   "totalElements": 122,
   "size": 5,
   "number": 1,
   "sort": {
      "empty": false,
      "sorted": true,
      "unsorted": false
   },
   "numberOfElements": 5,
   "first": false,
   "empty": false
}
```

Pagination with Sorting example

```
tutorialRepository.findAll(PageRequest.of(3,10, Sort.by("published"));

tutorialRepository.findAll(PageRequest.of(1, 5, Sort.by("published").descending());
```

Query Creation

- Generally, the query creation mechanism for JPA works as described in “Query Methods”. The following example shows what a JPA query method translates into:
- Example: Query creation from method names

```
public interface UserRepository extends Repository<User, Long> {
  List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);
}
```

- We 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.
- Spring Data JPA does a property check and traverses nested properties, as described in “Property Expressions”.

Supported keywords inside method names

Keyword	**Sample**	**JPQL snippet**
Distinct	findDistinctByLastnameAndFirstname	select distinct … where x.lastname = ?1 and x.firstname = ?2
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

Supported keywords inside method names (2)

Keyword	**Sample**	**JPQL snippet**
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull, Null	findByAge(Is)Null	… where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1

Supported keywords inside method names (3)

Keyword	**Sample**	**JPQL snippet**
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false

Query Method Example

```
public interface CustomerRepository extends JpaRepository<Customer, Integer> {
   public List<Customer> findAllByCustomerNameContaining(String name);
   public List<Customer> findAllByCityContainsOrderByCountryDesc(String name);
   public List<Customer> findAllByCreditLimitBetween(BigDecimal lower, BigDecimal upper);
   public List<Customer> findAllByCustomerNameBetween(String lower, String upper);
}
```

JPA Named Queries

- 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 runs them, you can actually bind them directly by using the Spring Data JPA @Query annotation rather than annotating them to the domain class.
- This frees the domain class from persistence specific information and co-locates the query to the repository interface.

```
public interface UserRepository extends JpaRepository<User, Long> {
   @Query("select u from User u where u.emailAddress = ?1")
   User findByEmailAddress(String emailAddress);
}
```

Native Queries

- The @Query annotation allows for running native queries by setting the nativeQuery flag to true, as shown in the following example:

- Declare a native query at the query method using @Query

```
public interface UserRepository extends JpaRepository<User, Long> {
      @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery = true)
      User findByEmailAddress(String emailAddress);
}
```

Dynamic sorting with native query

- Spring Data JPA does not currently support dynamic sorting for native queries, because it would have to manipulate the actual query declared, which it cannot do reliably for native SQL. You can, however, use native queries for pagination by specifying the count query yourself, as shown in the following example:

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query(value = "SELECT * FROM USERS WHERE LASTNAME = ?1",
     countQuery = "SELECT count(*) FROM USERS WHERE LASTNAME = ?1", nativeQuery = true)
    Page<User> findByLastname(String lastname, Pageable pageable);
}
```