



Spring RESTful API Pagination & Sorting, Query Creation

By

Pichet Limvajiranan

Spring Data REST: Pagination and Sorting

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

- **Page<T> findAll(Pageable pageable)**

returns a Page of entities meeting the paging restriction provided in the Pageable object.

```
Pageable firstPageTwoElements = PageRequest.of(0, 2); Pageable  
secondPageFiveElements = PageRequest.of(1, 5);
```

- **Iterable<T> findAll(Sort sort)**

returns all entities sorted by the given options. No paging is applied here.

```
Sort sortedByName = Sort.by("name");
```

- **Pagination & Sorting**

```
Pageable sortedByPriceDescNameAsc = PageRequest.of(0, 5,  
Sort.by("price").descending().and(Sort.by("name")));
```

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 `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.

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

Sort & Order object example

```
// 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")));
```

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

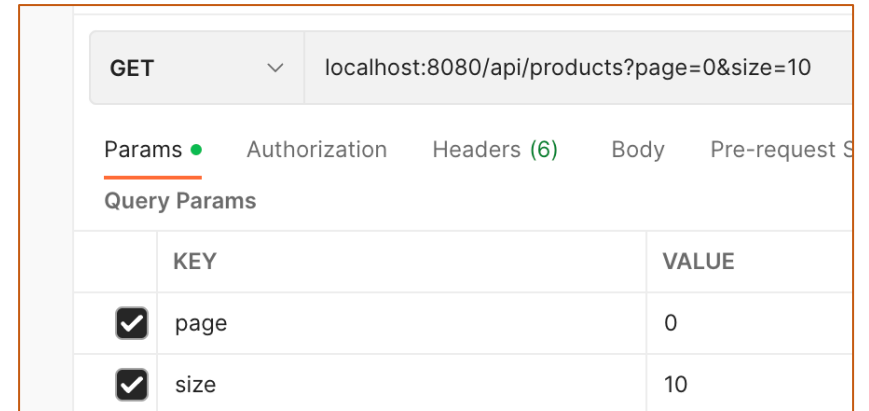
```
@Service
:
public Page<Customer> getAllCustomers(int page, int pageSize) {
    Pageable pageable = PageRequest.of(page, pageSize);
    return repository.findAll(pageable);
}
```

Accepting Page and Sort Parameters

- Generally, paging and sorting parameters are optional and thus part of the request URL as query parameters. If any API supports paging and sorting, ALWAYS provide default values to these parameters – to be used when the client does not choose to specify any paging or sorting preferences.

- Example:

```
@GetMapping("")
public List<Customer> getAllCustomers(
    @RequestParam(defaultValue = "id") String sortBy,
    @RequestParam(defaultValue = "0") Integer page,
    @RequestParam(defaultValue = "10") Integer pageSize) {
    Page<Customer> customers = service.findAll(sortBy, page, pageSize);
    return customers.getContent();
}
```



GET		localhost:8080/api/products?page=0&size=10
Params • Authorization Headers (6) Body Pre-request S		
Query Params		
	KEY	VALUE
<input checked="" type="checkbox"/>	page	0
<input checked="" type="checkbox"/>	size	10

Page<T> Object

```
{
  "content": [
    {
      "id": 323,
      "customerName": "Down Under Souvenirs, 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
}
```

Service - Paging & Sorting

```
@Service
public class CustomerService {
    @Autowired
    private CustomerRepository repository;

    public Page<Customer> getAllCustomers(
        String sortBy, int page, int pageSize) {
        Pageable pageble = PageRequest.of(page, pageSize);
        Page<Customer> customers = repository.findAll(pageble);
        return customers;
    }
}
```

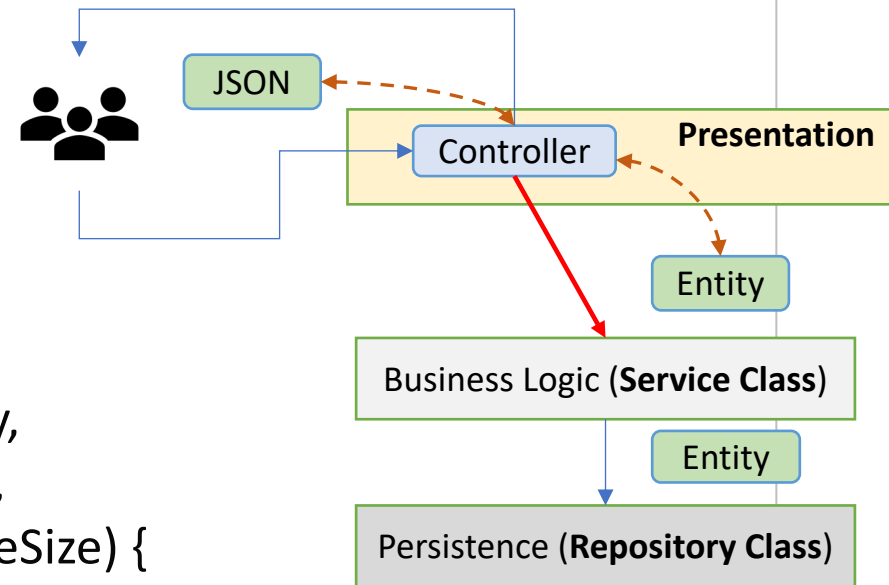
```
{
  "content " : {
    {},
    {},
    {}
  },
  "page": {
    "size": 20,
    "totalElements": 11,
    "totalPages": 1,
    "number": 0
  }
}
```


Controller - Paging & Sorting

```
@RestController
@RequestMapping("/api/customers")
public class CustomerController {
    @Autowired
    private CustomerService service;

    @GetMapping("")
    public List<Customer> getAllCustomers(
        @RequestParam(defaultValue = "id") String sortBy,
        @RequestParam(defaultValue = "0") Integer page,
        @RequestParam(defaultValue = "10") Integer pageSize) {
        return service.findAll(sortBy, page, pageSize).getContent();
    }
}
```

localhost:port/api/customers?sortBy=id&page=0&pageSize=10



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> findByEmailAndLastname(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,findByFirstnames,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	JSQL 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);  
}
```


Assignment (2)

- Create Services, Controllers for resources:

URI	HTTP verb	Description
api/products	GET	Get all products with pagination & sorting
api/products/10.0/250.0	GET	Get products for price between 10 to 250
api/products/ship	GET	Get products by product line sorting
api/products/S10_1234	PUT	Update product
api/products	POST	Add new product