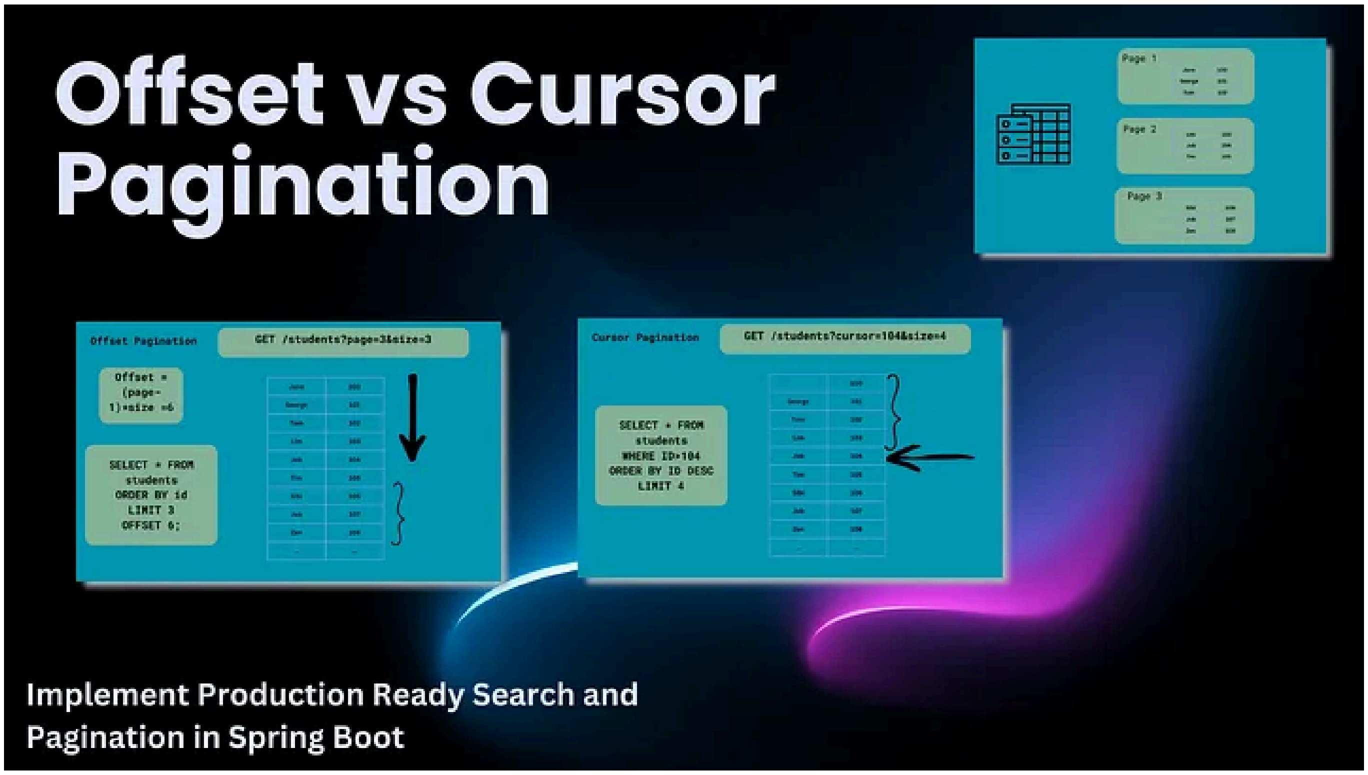
Offset vs Cursor Pagination : Implement production ready search and pagination in Spring Boot

Code Wiz Follow

13 min read · Apr 11, 2025

2 1


Offset vs Cursor Pagination
Page 1
Page 2
Page 3
Offset Pagination
Cursor Pagination
Implement Production Ready Search and Pagination in Spring Boot

```
GET /students?page=3&size=3
```

```
Offset =
(page-
1)*size =6
```

```
SELECT * FROM
students
ORDER BY id
LIMIT 3
OFFSET 6;
```

```
GET /students?cursor=104&size=4
```

```
SELECT * FROM
students
WHERE ID<104
ORDER BY ID DESC
LIMIT 4
```

Pagination is a crucial aspect of API design when dealing with large datasets. Instead of returning all records at once, pagination breaks the results into manageable chunks or "pages." In this blog post, we'll explore two popular pagination techniques, offset and cursor based and implement both in a Spring Boot application with search capabilities.

You can also read the same blog here

Understanding Pagination Techniques

Offset Pagination

Offset pagination, also known as page-based pagination, is the traditional approach where you specify:

- **Page Number:** Which page you want to retrieve
- **Page Size:** How many items per page

The database then calculates which records to return based on these parameters.

How Offset Pagination Works

First offset is calculated using the formula:

```
offset = pageNumber × pageSize  // when page number starts from 0

or

offset = (pageNumber - 1) × pageSize // when page number starts from 1
```

Then we will use a query to skip records till the offset and the return the next set of records.

Let's see how this works with an example:

Student Table

```
-----------------------------------------------------
| StudentID | Name      | Age | Course      | ... |
-----------------------------------------------------
| 1         | Alice     | 20  | Computer Sci| ... |  <-- Row 1
| 2         | Bob       | 21  | Physics     | ... |  <-- Row 2
| 3         | Charlie   | 19  | Math        | ... |  <-- Row 3
| 4         | David     | 22  | Chemistry   | ... |  <-- Row 4
| 5         | Eve       | 20  | Biology     | ... |  <-- Row 5
| 6         | Frank     | 21  | History     | ... |  <-- Row 6
| 7         | Grace     | 19  | English     | ... |  <-- Row 7
| 8         | Henry     | 22  | Economics   | ... |  <-- Row 8
| 9         | Ivy       | 20  | Sociology   | ... |  <-- Row 9
| 10        | Jack      | 21  | Psychology  | ... |  <-- Row 10
| 11        | Kelly     | 19  | Art         | ... |  <-- Row 11
| 12        | Liam      | 22  | Music       | ... |  <-- Row 12
| 13        | Mia       | 20  | Geography   | ... |  <-- Row 13
| 14        | Noah      | 21  | Philosophy  | ... |  <-- Row 14
| 15        | Olivia    | 19  | Political Sc| ... |  <-- Row 15
| ...       | ...       | ... | ...         | ... |
-----------------------------------------------------
```

Calculation of OFFSET:

For Page 3 with a Page Size of 5:

```
OFFSET = (Page Number - 1) * Page Size
```

```
   = (3 - 1) * 5 = 10
```

Skipping Records:

```
 -----------------------------------------------------
 | StudentID | Name       | Age | Course       | ... |  (Row Number)
 -----------------------------------------------------
 | 1         | Alice      | 20  | Computer Sci| ... |  (1)   <-- Skipped
 | 2         | Bob        | 21  | Physics      | ... |  (2)   <-- Skipped
 | 3         | Charlie    | 19  | Math         | ... |  (3)   <-- Skipped
 | 4         | David      | 22  | Chemistry    | ... |  (4)   <-- Skipped
 | 5         | Eve        | 20  | Biology      | ... |  (5)   <-- Skipped
 | 6         | Frank      | 21  | History      | ... |  (6)   <-- Skipped
 | 7         | Grace      | 19  | English      | ... |  (7)   <-- Skipped
 | 8         | Henry      | 22  | Economics    | ... |  (8)   <-- Skipped
 | 9         | Ivy        | 20  | Sociology    | ... |  (9)   <-- Skipped
 | 10        | Jack       | 21  | Psychology   | ... |  (10)  <-- Skipped
 | 11        | Kelly      | 19  | Art          | ... |  (11)  <-- Selected (Page 3
 | 12        | Liam       | 22  | Music        | ... |  (12)  <-- Selected (Page 3
 | 13        | Mia        | 20  | Geography    | ... |  (13)  <-- Selected (Page 3
 | 14        | Noah       | 21  | Philosophy   | ... |  (14)  <-- Selected (Page 3
 | 15        | Olivia     | 19  | Political Sc| ... |  (15)  <-- Selected (Page 3
 | ...       | ...        | ... | ...          | ... |
 -----------------------------------------------------
                                          ^
                                          |
                                    **OFFSET = 10 Records Skipped**
```

Selecting Records with LIMIT:

After skipping the 10 records, the LIMIT clause (equal to the Page Size of 5 in this case) is applied to select the next 5 records.

Result: Rows 11 through 15 are retrieved and represent Page 3 of the student data.

This translates to the SQL query:

```
SELECT * FROM students
ORDER BY id
LIMIT 5
OFFSET 10;
```

Cursor Pagination

Cursor pagination uses a “pointer” or “cursor” to track the user’s position in the result set. Instead of using page numbers, the client uses a reference value from the last item they’ve seen to fetch the next set of results.

How Cursor Pagination Works

Cursor pagination uses a “pointer” or “cursor” to track the user’s position in the result set. Let’s see how this works with an example:

Student Table

```
-----------------------------------------------------
| StudentID | Name      | Age | Course       | ... |
-----------------------------------------------------
| 1         | Alice     | 20  | Computer Sci| ... |
| 2         | Bob       | 21  | Physics      | ... |
| 3         | Charlie   | 19  | Math         | ... |
| 4         | David     | 22  | Chemistry    | ... |
| 5         | Eve       | 20  | Biology      | ... |
| 6         | Frank     | 21  | History      | ... |
| 7         | Grace     | 19  | English      | ... |
| 8         | Henry     | 22  | Economics    | ... |
| 9         | Ivy       | 20  | Sociology    | ... |
| 10        | Jack      | 21  | Psychology   | ... |
| 11        | Kelly     | 19  | Art          | ... |
| 12        | Liam      | 22  | Music        | ... |
| 13        | Mia       | 20  | Geography   | ... |
| 14        | Noah      | 21  | Philosophy  | ... |
| 15        | Olivia    | 19  | Political Sc| ... |
| ...       | ...       | ... | ...         | ... |
-----------------------------------------------------
```

First Page Request (No Cursor passed)

For the first request, we typically use a default "high" cursor value to get the first set of records:

```
Cursor = 999999 (Very high value to get the first page)
Limit = 5
```

The query becomes:

```
SELECT * FROM students
WHERE StudentID < 999999
ORDER BY StudentID DESC
LIMIT 5;
```

Result:

```
-----------------------------------------------------
| StudentID | Name      | Age | Course      | ... |
-----------------------------------------------------
| 15        | Olivia    | 19  | Political Sc| ... |  <-- Selected
| 14        | Noah      | 21  | Philosophy  | ... |  <-- Selected
| 13        | Mia       | 20  | Geography   | ... |  <-- Selected
| 12        | Liam      | 22  | Music       | ... |  <-- Selected
| 11        | Kelly     | 19  | Art         | ... |  <-- Selected
-----------------------------------------------------
                        ^
                        |
           Last record's ID becomes the next cursor (11)
```

1. **Second Page Request (Using Cursor)**

For the next page, we use the ID of the last record we received as the cursor:

```
Cursor = 11 (Last ID from previous result)
Limit = 5
```

The query becomes:

```
SELECT * FROM students
WHERE StudentID < 11
ORDER BY StudentID DESC
LIMIT 5;
```

Result:

```
-----------------------------------------------------
| StudentID | Name      | Age | Course      | ... |
-----------------------------------------------------
| 10        | Jack      | 21  | Psychology  | ... |  <-- Selected
| 9         | Ivy       | 20  | Sociology   | ... |  <-- Selected
| 8         | Henry     | 22  | Economics   | ... |  <-- Selected
| 7          | Grace     | 19  | English     | ... |  <-- Selected
| 6          | Frank     | 21  | History     | ... |  <-- Selected
-----------------------------------------------------
                        ^
                        |
              Last record's ID becomes the next cursor (6)
```

Now let us compare the query plan and cost for both pagination methods. I am running this against a Student table in a postgres db running in my machine with 1,000,000 records.

Query Plan Comparison

Offset Pagination Query Plan:

```
EXPLAIN ANALYZE
select *
    from
        students s1_0
    where
        1=1
    order by
        s1_0.id desc
    offset
        500000 rows
    fetch
        first 1000 rows only;
```

```
Limit  (cost=24062.33..24110.45 rows=1000 width=142) (actual time=145.510..145.7
    ->  Index Scan Backward using students_pkey on students s1_0  (cost=0.42..48
     (actual time=0.126..135.611 rows=501000 loops=1)
 Planning Time: 0.336 ms
 Execution Time: 145.783 ms
```

- The database uses an index scan on the primary key (good!)
- However, it must scan through 501,000 rows (500,000 to skip + 1,000 to return)
- Each of those 500,000 skipped rows must be read and discarded
- This can be inefficient, especially with large datasets.
- The cost of the query is high due to the need to scan and discard many records.

Cursor Pagination Query Plan:

```
EXPLAIN ANALYZE
select *
from
    students s1_0
where
    s1_0.id < 500000
order by
    s1_0.id desc
fetch
    first 1000 rows only;
```

```
Limit  (cost=0.42..51.06 rows=1000 width=142) (actual time=0.251..0.676 rows=100
    ->  Index Scan Backward using students_pkey on students s1_0  (cost=0.42..20
    (actual time=0.250..0.619 rows=1000 loops=1)
    Index Cond: (id < 500000)
Planning Time: 0.511 ms
Execution Time: 0.771 ms
```

- The database applies the id < 500000 condition directly to the index

- It immediately jumps to the relevant position in the index (like using a bookmark)
- This is much more efficient as it can utilize the index directly.
- It only processes the 1,000 rows it needs to return
- The cost of the query is significantly lower, and the execution time is much faster.

Unlike offset pagination, which must process and discard the offset records, cursor pagination directly starts from the relevant position in the index.

> *If you take an analogy with a book, offset pagination is like flipping through the pages to find the right one, while cursor pagination is like using a bookmark to jump directly to the last read page.*

Implementing Pagination in Spring Boot

Now we will implement both offset and cursor pagination in a Spring Boot application using Spring Data JPA. We will create simple student management application with a REST API to demonstrate both pagination techniques.

Project Setup

Create a Spring Boot Project: Use Spring Initializr to create a new Spring Boot project with the following dependencies:

- Spring Web
- Spring Data JPA
- PostgreSQL Driver

- Lombok
- Docker compose support

Docker Compose: Create a `docker-compose.yml` file to run PostgreSQL locally and also pgadmin to query the database.

```
services:
  postgres:
    image: 'postgres:latest'
    environment:
      - 'POSTGRES_DB=student-management-app-db'
      - 'POSTGRES_PASSWORD=secret'
      - 'POSTGRES_USER=student-management-user'
    ports:
      - '5432:5432'
    restart: unless-stopped
  pgadmin:
    image: dpage/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: admin
    ports:
      - "8081:80"
    depends_on:
      - postgres
    restart: unless-stopped
```

Add the following configuration to your `application.properties` file:

```
# JPA/Hibernate properties to auto-create tables
spring.jpa.hibernate.ddl-auto=update

# Set default profile to dev
spring.profiles.active=dev

# Batch configuration to optimize bulk inserts
```

```
spring.jpa.properties.hibernate.jdbc.batch_size=50
spring.jpa.properties.hibernate.order_inserts=true

# Config for logging SQL queries
logging.level.org.hibernate.SQL=debug
logging.level.org.hibernate.orm.jdbc.bind=trace
spring.jpa.properties.hibernate.format_sql=true
```

Add Student Entity

Let us add our Student entity with the following fields:

```
@Entity
@Table(name = "students")
@Data
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private LocalDate dob;
    private String address;
    private String phoneNumber;
    private String department;
    private LocalDate joiningDate;
    private String email;
    private String enrollmentNumber;
}
```

We will also create a DTO `StudentSearchCriteria` to hold the search criteria and pagination parameters.

```
@Data
public class StudentSearchCriteria {
    private String name;
    private String department;
```

```
        private String address;
        private String email;
        private String phoneNumber;
```

```
    }
```

Add Student Repository

```
public interface StudentRepository  extends JpaRepository<Student, Long>, JpaSpe
}
```

Add Student Service

Let us add a service class which will handle student creation and listing with search criteria.

For search we will use the `JpaSpecificationExecutor` interface to create dynamic queries based on the search criteria. We build a `Specification` object based on search criteria in `StudentSpecification` that can be passed to the repository methods.

```
@Service
@RequiredArgsConstructor
public class StudentService {
    private final StudentRepository studentRepository;
public Student createStudent(Student student) {
        return studentRepository.save(student);
    }
    public Page<Student> findStudentsWithCriteria(StudentSearchCriteria criteria
        return studentRepository.findAll(
            StudentSpecification.withSearchCriteria(criteria)
        );
    }
}
public class StudentSpecification {
    public static Specification<Student> withSearchCriteria(StudentSearchCriteri
        return (root, query, criteriaBuilder) -> {
            List<Predicate> predicates = new ArrayList<>();
            addPredicateIfNotNull(predicates, criteria.getName(), "name", root,
            addPredicateIfNotNull(predicates, criteria.getDepartment(), "departm
            addPredicateIfNotNull(predicates, criteria.getAddress(), "address",
            addPredicateIfNotNull(predicates, criteria.getEmail(), "email", root
            addPredicateIfNotNull(predicates, criteria.getPhoneNumber(), "phoneN
            addPredicateIfNotNull(predicates, criteria.getEnrollmentNumber(), "e
            return criteriaBuilder.and(predicates.toArray(new Predicate[0]));
        };
    }
    private static void addPredicateIfNotNull(List<Predicate> predicates, String
                                              Root<Student> root, CriteriaBuilde
        if (value != null && !value.isEmpty()) {
            predicates.add(criteriaBuilder.like(
                    criteriaBuilder.lower(root.get(fieldName)),
                    "%" + value.toLowerCase() + "%"
            ));
        }
    }
}
```

Other option for implementing search is to use query by example (QBE) using `ExampleMatcher` and `Example` classes.

```
public Page<Student> findStudentsWithExample(StudentSearchCriteria criteria, Pag
    Student studentExample = new Student();
    studentExample.setName(criteria.getName());
    studentExample.setDepartment(criteria.getDepartment());
    studentExample.setAddress(criteria.getAddress());
    studentExample.setEmail(criteria.getEmail());
    studentExample.setPhoneNumber(criteria.getPhoneNumber());
    studentExample.setEnrollmentNumber(criteria.getEnrollmentNumber());
    ExampleMatcher matcher = ExampleMatcher.matchingAll()
            .withIgnoreCase()
            .withStringMatcher(ExampleMatcher.StringMatcher.CONTAINING)
            .withIgnoreNullValues();
    Example<Student> example = Example.of(studentExample, matcher);
    return studentRepository.findAll(example, pageable);
}
```

Now let us implement offset pagination

Adding Offset Pagination

We can add offset pagination to our controller by using the `Pageable` interface provided by Spring Data JPA. Let us change our service method to accept a `Pageable` parameter and return a `Page<Student>` object.

```
public Page<Student> findStudentsWithCriteria(StudentSearchCriteria studentSearc
    return studentRepository.findAll(StudentSpecification.withSearchCriteria(stu
}
```

Basic Offset Pagination Controller

```
@RestController
@RequestMapping("/api/students")
@AllArgsConstructor
public class StudentController {
private final StudentService studentService;
    @GetMapping
    public ResponseEntity<Page<Student>> getAllStudents(@ModelAttribute StudentS
        String sortField = criteria.getSort()[0];
        String sortDirection = criteria.getSort()[1];
        Sort.Direction direction = sortDirection.equalsIgnoreCase("asc") ? Sort.
        Sort sortOrder = Sort.by(direction, sortField);
        Pageable pageable = PageRequest.of(criteria.getPage(), criteria.getSize(
        Page<Student> studentsPage = studentService.findStudentsWithCriteria(cri
        return new ResponseEntity<>(studentsPage, HttpStatus.OK);
    }
}
```

To test this we need to load some data. I have created a `DataLoader` class to load some sample data into the database. You can find that in the github repository.

After loading the data, you can test the API using Postman or any other API testing tool. We will use httpie to test the API.

```
http GET "http://localhost:8080/api/students?page=1000&size=5"
```

```
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Thu, 10 Apr 2025 14:46:43 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
```

```
{
    "content": [
        {
            "address": "906 Nana Knoll, Windlerburgh, SD 86681-5909",
            "department": "Computer Science",
            "dob": "2019-09-14",
            "email": "arianna.bosco@hotmail.com",
            "enrollmentNumber": "CO24181",
            "id": 995000,
            "joiningDate": "2024-11-12",
            "name": "Ms. Maya Hahn",
            "phoneNumber": "(277) 617-0505"
        },
        {
            "address": "10071 Gulgowski Parks, Gibsonborough, MI 37851",
            "department": "Computer Science",
            "dob": "2021-12-27",
            "email": "mazie.bosco@yahoo.com",
            "enrollmentNumber": "CO23727",
            "id": 994999,
            "joiningDate": "2023-01-20",
            "name": "Johnny Mann",
            "phoneNumber": "840.122.3269"
        }
        // ... 3 more records
    ],
    "empty": false,
    "first": false,
    "last": false,
    "number": 1000,
    "numberOfElements": 5,
    "pageable": {
        "offset": 5000,
        "pageNumber": 1000,
        "pageSize": 5,
        "paged": true,
        "sort": {
            "empty": false,
            "sorted": true,
            "unsorted": false
        },
        "unpaged": false
    },
    "size": 5,
    "sort": {
        "empty": false,
        "sorted": true,
        "unsorted": false
    },
    "totalElements": 1000000,
    "totalPages": 200000
}
```

If you check the logs now you will see the SQL query generated by Spring Data JPA:

```
select
        s1_0.id,
        s1_0.address,
        s1_0.department,
        s1_0.dob,
        s1_0.email,
        s1_0.enrollment_number,
        s1_0.joining_date,
        s1_0.name,
        s1_0.phone_number
    from
        students s1_0
    where
        1=1
    order by
        s1_0.id desc
    offset
        ? rows
    fetch
        first ? rows only

    binding parameter (1:INTEGER) <- [5000]
    binding parameter (2:INTEGER) <- [5]

    select
        count(s1_0.id)
    from
        students s1_0
    where
        1=1
```

You can see the offset and limit values are passed as parameters to the query. Also Spring runs a count query to get the total number of records in the table. This is useful for pagination as it allows us to calculate the total number of pages.

With search criteria:

```
http GET "http://localhost:8080/api/students?page=0&size=5&sort=id,desc&name=Joh
```

Adding Cursor Pagination

First we will use repository method to implement cursor pagination.

We will still use `Pageable` to pass the size, but offset will be set as 0. Also we will add a condition for cursor. The cursor will be the last record's ID from the previous page.

Service Layer Implementation

```
public List<Student> findStudentsWithCursorAndCriteria(Long cursor, int size, St
    Pageable pageable = PageRequest.of(0, size, Sort.by("id").descending());
    Specification<Student> spec = StudentSpecification.withSearchCriteria(criter
            .and((root, query, criteriaBuilder) -> {
                return criteriaBuilder.lessThan(root.get("id"), cursor);
            });
    return studentRepository.findAll(spec, pageable).getContent();
}
```

An alternative approach using CriteriaAPI directly:

```
public List<Student> findStudentsWithCursorAndCriteria(Long cursor, int size, St
    CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
    CriteriaQuery<Student> query = criteriaBuilder.createQuery(Student.class);
    Root<Student> root = query.from(Student.class);

    Specification<Student> spec = StudentSpecification.withSearchCriteria(criter
    Predicate predicate = spec.toPredicate(root, query, criteriaBuilder);
    Predicate cursorPredicate = criteriaBuilder.lessThan(root.get("id"), cursor)
    query.where(criteriaBuilder.and(predicate, cursorPredicate));
    query.orderBy(criteriaBuilder.desc(root.get("id")));
    return entityManager.createQuery(query)
            .setMaxResults(size)
            .getResultList();
}
```

Controller Layer Changes

We will add a new endpoint to the controller to handle cursor pagination requests. The endpoint will accept a cursor value and size as query parameters.

```
@GetMapping("/cursor")
public ResponseEntity<List<Student>> getAllStudentsCursorPagination(
        @RequestParam(required = false, defaultValue = "99999999999") Long curso
        @RequestParam(defaultValue = "10") int size,
        @ModelAttribute StudentSearchCriteria criteria) {
    List<Student> students = studentService.findStudentsWithCursorAndCriteria(cu
    return new ResponseEntity<>(students, HttpStatus.OK);
}
```

Testing Cursor Pagination

You can test the cursor pagination API using httpie as follows:

```
http GET "http://localhost:8080/api/students/cursor?cursor=100&size=10"
```

```
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Thu, 10 Apr 2025 14:57:36 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers

[
    {
        "address": "Apt. 064 20143 Swift Expressway, Colbyville, TX 28454",
        "department": "Electrical Engineering",
        "dob": "2019-11-17",
        "email": "gertrud.windler@gmail.com",
        "enrollmentNumber": "EL24190",
        "id": 99,
        "joiningDate": "2024-06-23",
        "name": "Charise Ondricka",
        "phoneNumber": "(648) 009-9370"
    },
    {
        "address": "Suite 311 22250 Jerilyn Ridges, South Matt, DE 48976-4431",
        "department": "Civil Engineering",
        "dob": "2024-05-02",
        "email": "violet.hegmann@hotmail.com",
        "enrollmentNumber": "CI22730",
        "id": 98,
        "joiningDate": "2022-06-09",
        "name": "Reinaldo Sawayn",
        "phoneNumber": "084.327.5967"
    },
    // ... 8 more records
]
```

From the logs we can see the query getting executed and the parameters

```
select
        s1_0.id,
        s1_0.address,
        s1_0.department,
        s1_0.dob,
        s1_0.email,
        s1_0.enrollment_number,
        s1_0.joining_date,
        s1_0.name,
        s1_0.phone_number
    from
        students s1_0
    where
        1=1
        and s1_0.id<?
    order by
        s1_0.id desc
    fetch
        first ? rows only

binding parameter (1:BIGINT) <- [100]
 binding parameter (2:INTEGER) <- [10]
```

I have also built a simple UI using Next JS to demonstrate integration with the API. You can find the code in the GitHub Repo repository or watch the video below.

Comparing Offset vs Cursor Pagination

Offset Pagination Pros:

- Simple to implement and understand
- Built-in support in Spring Data JPA

- Easy navigation to any page
- Works well for smaller datasets
- Familiar for most developers

Offset Pagination Cons:

- Performance degrades with large offset values
- Inconsistent results with concurrent inserts/deletes
- Database must scan and discard rows for large offsets
- Inefficient for deep pagination

Cursor Pagination Pros:

- Consistent performance regardless of depth
- More efficient for large datasets
- Stable results with concurrent data modifications
- Better utilization of database indexes

Cursor Pagination Cons:

- More complex to implement
- Cannot jump to arbitrary pages
- Requires a stable ordering field (usually ID)

Use Cases for Each Pagination Type

When to Use Offset Pagination:

- Admin panels where total count is needed
- Smaller datasets (<10,000 records)
- UIs that need explicit page numbers
- When simplicity is more important than performance
- When users need to jump to specific pages

When to Use Cursor Pagination:

- Infinite scrolling interfaces
- Large datasets
- Real-time feeds (social media, news, etc.)
- APIs with high throughput requirements
- When performance is critical

Conclusion

Choosing the right pagination strategy depends on your application's requirements. Offset pagination is simpler but can be inefficient for large datasets, while cursor pagination offers consistent performance but requires more implementation effort.

Spring Boot and Spring Data JPA provide flexible tools to implement both strategies. By understanding the strengths and weaknesses of each approach, you can make informed decisions about which pagination method to use in your applications.

For most modern applications with large datasets, cursor pagination is becoming the preferred choice due to its performance advantages, especially when implementing infinite scrolling interfaces common in today's web and mobile apps.

You can find the source code here.

To stay updated with the latest updates in Java and Spring follow us on:

- 🔗 [Blog](https://codewiz.info)
- 🔗 YouTube
- 🔗 LinkedIn
- 🔗 Medium
- 🔗 Github

Java Spring Pagination Rest Jpa

Written by Code Wiz

231 followers · 28 following

Follow

Software engineering and programming tutorials and blogs. Website - https://codewiz.info/

Responses (1)

Write a response

What are your thoughts?

Sankranti
Oct 31, 2025

There is a mistake in the "Service Layer Implementation" section. Only alternative approach based on CriteriaAPI works correctly, but approach based on Pageable works in the same way as offset based approach, eg. makes SELECT COUNT to populate Page... more

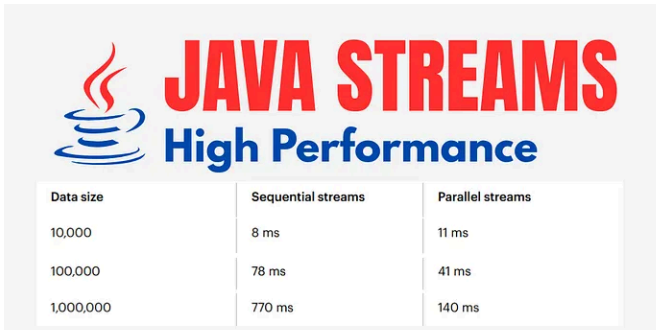
Reply

More from Code Wiz

Code Wiz

Structured Concurrency in Java 25: Complete Guide with Examples

Structured concurrency is a programming paradigm that treats groups of related tasks...


JAVA STREAMS
High Performance

Data size	Sequential streams	Parallel streams
10,000	8 ms	11 ms
100,000	78 ms	41 ms
1,000,000	770 ms	140 ms

Harry

High Performance Programming with Java Streams

Why most Java stream code is elegant... and still slow

Sep 19, 2025 12

Dec 22, 2025 184 2

Harry

1 Password, 700 Jobs Gone

The $20 Billion Mistake Your Company Is Probably Making Right Now

Nov 27, 2025 1.5K 24

Code Wiz

Webhook basics and implementing in Java and SpringBoot

Webhooks are a powerful mechanism for enabling real-time communication between...

Aug 15, 2025 122

See all from Code Wiz

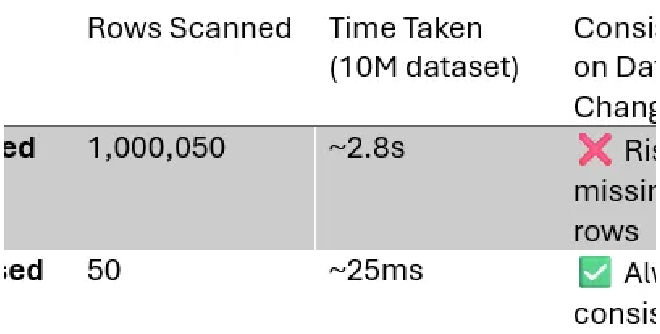
Recommended from Medium

Code With Sunil | Code Smarter, not harder

File Storage in Spring Boot: Database, Local, and Cloud…

If you are not a Member — Read for free here

✦ 3d ago 54



	Rows Scanned	Time Taken (10M dataset)	Consi on Da Chang
ed	1,000,050	~2.8s	❌ Ri missir rows
sed	50	~25ms	✅ Al consis

Puja Pandey

Scaling Database Queries in Java, Part 3: Cursor-Based Pagination f…

In Part 1, we solved the problem of large IN clauses with batching. In Part

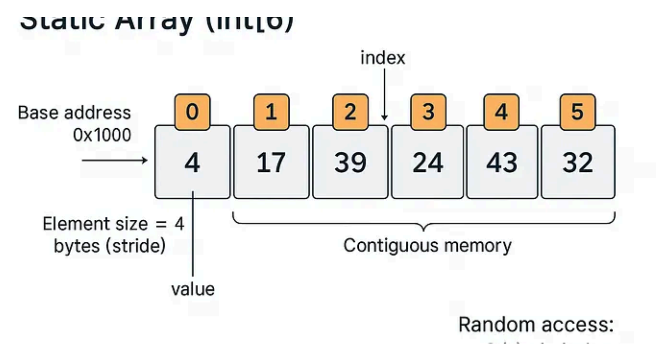
Sep 6, 2025 1

PraveenCodes

UnsatisfiedDependencyException in Spring Boot—A Real Productio…

if you are not a medium member then Click heree to read free

✦ 6d ago 2

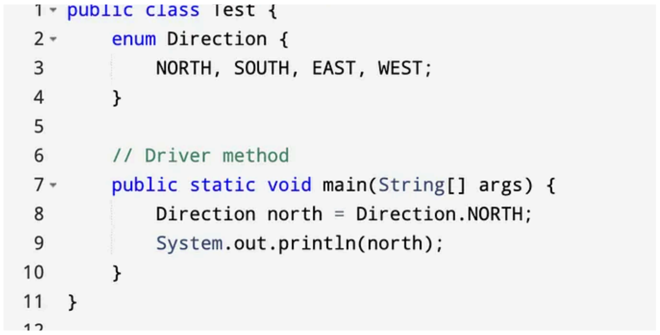


Noah Byteforge

The 12 Data Structures Every Developer Should Master Before …

Cracking coding interviews isn't just about how well you can write code—it's about how…

✦ Oct 9, 2025 394 6



```
public class Test {
    enum Direction {
        NORTH, SOUTH, EAST, WEST;
    }

    // Driver method
    public static void main(String[] args) {
        Direction north = Direction.NORTH;
        System.out.println(north);
    }
}
```

Sreehari Vasudevan

Spring Data JPA -One To Many and Join Query

In many projects, entity relationship mapping and join operations in Spring Data JPA can…

Aug 28, 2025 6

In tuanhdotnet by Anh Trần Tuấn

Use Enum in Entity Layers and How to Handle Getter Methods in Spri…

Aug 31, 2025 6

See more recommendations