Pagination, Sorting, and Projections

How the Pageable/Sort "REST API Design" resolves at the controller level actually becomes a real LIMIT/OFFSET plus count query at the repository level, sorting with findAll(Sort)/derived methods, and interface-based and record/constructor-expression projections that return only the fields a query needs instead of a whole entity -- using this project's own Topic/TopicTranslation classes. The 4th lesson in the Spring Data JPA category.

Intermediate 30 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/pagination-sorting-and-projections

---

"REST API Design" showed a controller resolving `Pageable` and `Sort` from query parameters, and returning a `Page<T>` — but its own example admitted, in a comment, that it built that `Page` over an already-fetched, in-memory list, "because a real repository does this in the database." This lesson is that other half: making paging, sorting, and result-shaping actually happen at the repository level, backed by real SQL.

What "REST API Design" Left Unfinished: Real Pagination

The controller-side story is already complete: a `@RestController` method parameter of type `Pageable` gets resolved from `?page=` / `?size=` / `?sort=` automatically, and `Sort.by(...).and(...)` builds a multi-field sort programmatically. None of that is repeated here. What's missing is what happens on the other end of that `Pageable` — the repository method that actually receives it and turns it into a real, efficient database query.

Declaring a Paged Repository Method

Making a repository method genuinely paged, instead of simulating it over a list already sitting in memory, needs surprisingly little.

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;

// "REST API Design"'s PaginationExample showed a controller resolving a
// Pageable from ?page=/?size= and returning a Page -- but it built the
// Page itself with PageImpl over an ALREADY-FETCHED, in-memory list,
// explicitly noting "a real repository does this in the database." This
// is that real repository method -- an extension to this project's own
// TopicRepository.
interface TopicPagingRepositoryExample extends JpaRepository<TopicExample, Long> {

    // Returning Page<TopicExample> (instead of List<TopicExample>) from a
    // method that takes a Pageable is all that's needed -- Spring Data JPA
    // generates a query with a real LIMIT/OFFSET, PLUS a second query that
    // counts the total matching rows, and packages both into the Page it
    // hands back.
    Page<TopicExample> findByCategoryId(Long categoryId, Pageable pageable);
}

class TopicExample {
}

class PagedRepositoryMethodExample {
    public static void main(String[] args) {
        // In a real application, this Pageable would already have been
        // resolved from ?page=0&size=2 by Spring MVC, exactly as "REST API
        // Design" covered -- constructed by hand here only to call the
        // repository method directly, outside of a running application.
        Pageable firstPage = PageRequest.of(0, 2);

        System.out.println(firstPage);
        // repository.findByCategoryId(5L, firstPage) would now run TWO real
        // queries against PostgreSQL: one SELECT ... LIMIT 2 OFFSET 0, and
        // one SELECT COUNT(*) -- see "What Actually Happens Underneath."
    }
}
```

`findByCategoryId(Long categoryId, Pageable pageable)` looks almost identical to the derived query methods from "Query Methods and JPQL with @Query" — the only change is the return type, `Page<TopicExample>` instead of `List<TopicExample>`. That single change is enough: Spring Data JPA generates a query with a real `LIMIT` / `OFFSET`, plus a second query counting the total matching rows, and packages both into the `Page` it returns.

What Actually Happens Underneath: LIMIT, OFFSET, and a Count Query

A single call to a `Page`-returning method quietly runs TWO queries, not one:

```
findByCategoryId(5L, PageRequest.of(0, 2))
        |
        +--> SELECT * FROM topic WHERE category_id = 5 LIMIT 2 OFFSET 0
        |         (the actual page of rows)
        |
        +--> SELECT COUNT(*) FROM topic WHERE category_id = 5
                  (how many rows exist in total, across every page)
```

The first query fetches only the current page's rows — never the whole table. The second is what makes `page.getTotalElements()` / `getTotalPages()` (already used at the controller level in "REST API Design") possible at all; without it, there'd be no way to know how many pages remain. Both queries share the same `WHERE` condition, generated once from the method's own derived name or `@Query`.

Sorting at the Repository Level

Sorting a paged (or unpaged) result doesn't need a new mechanism — it reuses pieces already covered.

```
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

interface TopicSortingRepositoryExample extends JpaRepository<TopicSortExample, Long> {

    // findAll(Sort sort) isn't written here at all -- it comes directly
    // from PagingAndSortingRepository, exactly as covered in "Entities and
    // the Repository Abstraction." No new method needed to sort every
    // Topic by an arbitrary field.

    // A derived method can ALSO accept a Sort parameter, combining a
    // filter condition with caller-supplied ordering -- CategoryId narrows
    // the rows, Sort decides what order they come back in.
    List<TopicSortExample> findByCategoryId(Long categoryId, Sort sort);
}

class TopicSortExample {
}

class SortAtRepositoryLevelExample {
    public static void main(String[] args) {
        // Sort.by(...).and(...) construction itself was already covered in
        // "REST API Design" -- this is the exact same Sort object, now
        // handed to a repository method instead of just resolved from
        // query parameters.
        Sort bySortOrder = Sort.by(Sort.Direction.ASC, "sortOrder");

        System.out.println(bySortOrder);
        // repository.findAll(bySortOrder) -- every Topic, ordered by sortOrder
        // repository.findByCategoryId(5L, bySortOrder) -- only category 5's
        //     Topics, in the same order
    }
}
```

`findAll(Sort sort)` isn't written anywhere in this interface at all — it's inherited directly from `PagingAndSortingRepository`, exactly as "Entities and the Repository Abstraction" already covered; no new method is needed just to sort every row by an arbitrary field. A derived method can also accept a `Sort` parameter directly, alongside its usual conditions — `findByCategoryId(Long categoryId, Sort sort)` filters by category AND orders the result, with the caller supplying the ordering. The `Sort` object itself — `Sort.by(...).and(...)` — is unchanged from "REST API Design"; only where it's handed to (a repository method, not just resolved at the controller) is new here.

Combining Paging, Sorting, and a Filter Condition

`Pageable` and `Sort` aren't actually two separate concerns to juggle — a `Pageable` already carries its own embedded `Sort`.

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.repository.JpaRepository;

interface TopicDifficultyPagingRepositoryExample extends
JpaRepository<TopicDifficultyExample, Long> {

    // A single Pageable already carries BOTH paging (page/size) AND
    // sorting information (Pageable has its own embedded Sort) -- no
    // separate Sort parameter is needed here at all.
    Page<TopicDifficultyExample> findByDifficulty(String difficulty, Pageable pageable);
}

class TopicDifficultyExample {
}

class PagedAndFilteredQueryExample {
    public static void main(String[] args) {
        // A Pageable built WITH a Sort attached -- PageRequest.of(page,
        // size, sort) -- carries paging and ordering together in one object.
        Pageable secondPageOrderedBySlug = PageRequest.of(1, 5, Sort.by("slug"));

        System.out.println(secondPageOrderedBySlug);
        // repository.findByDifficulty("INTERMEDIATE", secondPageOrderedBySlug)
        // generates one query filtering by difficulty, ordering by slug,
        // and applying LIMIT 5 OFFSET 5 -- all three concerns, from one
        // method call.
    }
}
```

`findByDifficulty(String difficulty, Pageable pageable)` needs no separate `Sort` parameter at all, because `PageRequest.of(page, size, sort)` already bundles paging and ordering into one `Pageable`. One method call, one `Pageable` argument, and the generated query filters, orders, and pages the result all at once — three concerns handled by a single, unremarkable method signature.

Why Project Instead of Fetching a Whole Entity?

Every query so far has returned a full entity — every mapped field, whether the caller needed it or not. A PROJECTION returns only the fields a specific query actually needs, both narrowing the SQL itself (fewer columns selected) and avoiding the overhead of managing a full entity for data that's only ever going to be read, never modified through it.

Interface-Based Projections

The simplest projection is just an interface with getters matching a subset of an entity's properties.

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

import java.util.List;

// An INTERFACE PROJECTION: instead of the full Topic entity (id, category,
// slug, difficulty, estimatedMinutes, sortOrder, and any relationships it
// carries), only THREE fields are actually needed here. Declaring an
// interface with getters matching a subset of the entity's property names
// is enough -- no implementation is written for it.
interface TopicSummary {
    String getSlug();
    String getDifficulty();
    Integer getEstimatedMinutes();
}

interface TopicProjectionRepositoryExample extends JpaRepository<TopicProjectionExample,
Long> {

    // Spring Data JPA generates a proxy implementing TopicSummary at
    // runtime, and -- this is the real benefit, not just less Java code --
    // generates a SQL SELECT that names only slug/difficulty/
    // estimated_minutes, not every column the full entity would require.
    List<TopicSummary> findByCategoryId(Long categoryId);
}

class TopicProjectionExample {
}
```

`TopicSummary` declares three getters — `getSlug()`, `getDifficulty()`, `getEstimatedMinutes()` — and nothing implements it by hand. Spring Data JPA generates a proxy implementing it at runtime, and — the actual benefit, not just less Java to write — generates a SQL `SELECT` naming only those three columns, not every column the full `Topic` entity would require.

DTO/Record Projections

A record projection asks for exactly the same narrowing, but the query has to say precisely how to build the result, rather than Spring Data JPA inferring it from getter names.

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

import java.util.List;

// A DTO/record projection -- as "Record" already covers, using a record
// as a JPQL "constructor expression" target. Unlike TopicSummary's
// interface projection (Spring Data JPA proxies it automatically), a
// record projection requires the query to say EXACTLY how to build one,
// via "select new fully.qualified.RecordName(...)".
record TopicTitleView(String slug, String title) {
}

interface TopicTranslationProjectionRepositoryExample extends org.springframework.data.jpa.
repository.JpaRepository<TopicTranslationProjectionExample, Long> {

    // This project's Topic entity carries no title itself -- title lives
    // on TopicTranslation, one per language (see "JPA, Hibernate, and
    // Spring Data JPA"'s and "Entities and the Repository Abstraction"'s
    // real Topic/TopicTranslation classes). This query joins across that
    // relationship and constructs a TopicTitleView directly -- pulling
    // back exactly two columns, from two tables, with no intermediate
    // entity ever loaded.
    @Query("select new com.cdurgun.learning.example.TopicTitleView(tt.topic.slug, tt.title)
" +
            "from TopicTranslationProjectionExample tt " +
            "where tt.language = :language and tt.published = true")
    List<TopicTitleView> findAllPublishedTitles(String language);
}

class TopicTranslationProjectionExample {
}
```

`TopicTitleView` — a `record`, exactly the kind of type "Record" recommends for this — is constructed directly inside JPQL with `select new ...TopicTitleView(tt.topic.slug, tt.title)`, a "constructor expression." This query joins across `TopicTranslation`'s relationship to `Topic` and pulls back exactly two columns from two tables, with no intermediate `Topic` or `TopicTranslation` entity ever fully loaded into memory.

> Prefer an interface projection when a query needs a straightforward subset of ONE entity's own fields — it needs no query changes at all. Reach for a record/constructor-expression projection once a projection needs to pull fields from ACROSS a relationship, the way `TopicTitleView` does here, or needs any computed value a plain getter can't express.

Common Misconceptions

"`Page<T>` and `List<T>` are basically the same thing, just with extra metadata." They come from genuinely different queries — a `List<T>`-returning method runs one query; a `Page<T>`-returning method runs two (the page's rows, and a separate count). **"A projection**

is just about writing less Java." The real benefit is a narrower SQL `SELECT` — fewer columns fetched from the database, not merely a smaller Java type to hold the result. **"`Pageable` and `Sort` are two separate things to pass around."** A `Pageable` already carries its own `Sort` internally — building one with `PageRequest.of(page, size, sort)` is usually enough, with no separate `Sort` parameter needed alongside it.

What Comes Next

Every query in this lesson filtered on a fixed, known condition — `categoryId`, `difficulty` — decided at compile time by the method's own name or `@Query`. "Dynamic Queries with Specifications," next in this category, covers what "REST API Design"'s filtering section only named in passing: building a query's conditions at RUNTIME, when the set of active filters isn't known until a request actually arrives.

Best Practices

- Return `Page<T>` (not `List<T>`) from any repository method a paginated API endpoint will call — the extra count query is what makes `totalElements` / `totalPages` possible at all.
- Reach for a projection — interface or record — whenever a query's caller only needs a subset of an entity's fields, rather than fetching (and paying for) the whole thing.
- Build a `Pageable` with `PageRequest.of(page, size, sort)` rather than juggling a separate `Sort` parameter alongside it.
- Use an interface projection for a straightforward subset of one entity's fields; reach for a record/constructor-expression projection once a relationship or a computed value is involved.

Common Mistakes

- Simulating pagination over an already-fully-fetched `List` (exactly what "REST API Design"'s own example deliberately avoided doing for real) instead of declaring a genuinely `Page`-returning repository method.
- Fetching a full entity and manually copying a few fields onto a DTO afterward, instead of letting a projection narrow the query itself.
- Passing a separate `Sort` parameter to a method that also takes a `Pageable`, not realizing the `Pageable` can already carry the ordering.
- Expecting an interface projection to work for anything beyond a straightforward subset of one entity's own getters — a relationship-spanning or computed result needs a constructor-expression projection instead.

Summary, Cheat Sheet, and Glossary

Summary

- A repository method returning `Page<T>` (instead of `List<T>`) generates a real `LIMIT`/`OFFSET` query plus a separate count query, packaged together.
- `findAll(Sort)` is inherited for free from `PagingAndSortingRepository`; a derived method can also take a `Sort` parameter directly.
- A `Pageable` already carries its own embedded `Sort` — `PageRequest.of(page, size, sort)` handles paging and ordering together.
- A projection returns only the fields a query actually needs, narrowing the generated SQL itself, not just the Java type holding the result.
- An interface projection needs only getters matching a subset of an entity's properties; a record/constructor-expression projection is needed once a query spans a relationship or computes a value.

Cheat Sheet

```
// A genuinely paged repository method
Page<Topic> findByCategoryId(Long categoryId, Pageable pageable);

// Sorting: inherited for free, or as a derived-method parameter
List<Topic> findAll(Sort sort); // from PagingAndSortingRepository
List<Topic> findByCategoryId(Long categoryId, Sort sort);

// Paging + sorting + filtering together
Page<Topic> findByDifficulty(String difficulty, Pageable pageable);
Pageable pageable = PageRequest.of(1, 5, Sort.by("slug"));

// Interface projection
interface TopicSummary {
    String getSlug();
    Integer getEstimatedMinutes();
}
List<TopicSummary> findByCategoryId(Long categoryId);

// Record / constructor-expression projection
record TopicTitleView(String slug, String title) {}

@Query("select new com.example.TopicTitleView(tt.topic.slug, tt.title) " +
       "from TopicTranslation tt where tt.language = :language")
List<TopicTitleView> findAllTitles(String language);
```

Glossary

- **Page<T>**: a Spring Data type representing one page of results plus metadata (total elements, total pages), backed by two real queries.
- **Pageable**: an object describing which page and size to fetch, along with its own embedded `Sort`.
- **Projection**: a query result narrowed to a subset of an entity's fields, reducing the generated SQL's own `SELECT` list.
- **Interface projection**: a projection defined as an interface whose getters match a subset of an entity's properties, implemented automatically by Spring Data JPA.

- **Constructor-expression (record/DTO) projection**: a projection built explicitly inside JPQL with `select new ...SomeType(...)`, needed when fields span a relationship or are computed.