Spring Data MongoDB Repository

jim stafford

Fall 2024 v2022-07-25: Built: 2024-11-19 21:37 EST

Table of Contents

Chapter 1. Introduction

MongoTemplate provided a lot of capability to interface with the database, but with a significant amount of code required. Spring Data MongoDB Repository eliminates much of the boilerplate code for the most common operations and allows us access to MongoTemplate for the harder edge-cases.

Due to the common Spring Data framework between the two libraries and the resulting similarity between Spring Data JPA and Spring Data MongoDB repositories, this lecture is about 95% the same as the Spring Data JPA lecture. Although it is presumed that the Spring Data JPA lecture precedes this lecture — it was written so that was not a requirement. If you have already mastered Spring Data JPA Repositories, you should be able to quickly breeze through this material because of the significant similarities in concepts and APIs.

1.1. Goals

The student will learn:

- to manage objects in the database using the Spring Data MongoDB Repository
- to leverage different types of built-in repository features
- to extend the repository with custom features when necessary

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. declare a `MongoRepository` for an existing `@Document`
2. perform simple CRUD methods using provided repository methods
3. add paging and sorting to query methods
4. implement queries based on POJO examples and configured matchers
5. implement queries based on predicates derived from repository interface methods
6. implement a custom extension of the repository for complex or compound database access

Chapter 2. Spring Data MongoDB Repository

Spring Data MongoDB provides repository support for `@Document`-based mappings. [1] We start off by writing no mapping code — just interfaces associated with our `@Document` and primary key type — and have Spring Data MongoDB implement the desired code. The Spring Data MongoDB interfaces are layered — offering useful tools for interacting with the database. Our primary `@Document` types will have a repository interface declared that inherit from `MongoRepository` and any custom interfaces we optionally define.

Figure 1. Spring Data MongoDB Repository Interfaces

The extends path was modified some with the latest version of Spring Data Commons, but the `MongoRepository` ends up being mostly the same by the time the interfaces get merged at the bottom of the inheritance tree.

[1] *"Spring Data MongoDB - Reference Documentation"*

Chapter 3. Spring Data MongoDB Repository Interfaces

As we go through these interfaces and methods, please remember that all of the method implementations of these interfaces (except for custom) will be provided for us.

Repository<T, ID>	marker interface capturing the `@Document` class and primary key type. Everything extends from this type.
CrudRepository<T,ID>	depicts many of the CRUD capabilities we demonstrated with the MongoOps DAO in previous MongoTemplate lecture
PagingAndSortingRepository<T, ID>	Spring Data MongoDB provides some nice end-to-end support for sorting and paging. This interface adds some sorting and paging to the `findAll()` query method provided in `CrudRepository`.
ListPagingAndSortingRepository<T,ID>	overrides the PagingAndSorting-based `Iterable<T>` return type to be a `List<T>`
ListCrudRepository	overrides all CRUD-based `Iterable<T>` return types with `List<T>`
QueryByExampleExecutor<T>	provides query-by-example methods that use prototype `@Document` instances and configured matchers to locate matching results
MongoRepository<T, ID>	brings together the `CrudRepository`, `PagingAndSortingRepository`, and `QueryByExampleExecutor` interfaces and adds several methods of its own. The methods declared are mostly generic to working with repositories — only the `insert()` methods have any specific meaning to MongoDB.
BooksRepositoryCustom/ BooksRepositoryCustomImpl	we can write our own extensions for complex or compound calls — while taking advantage of an `MongoTemplate` and existing repository methods. This allows us to encapsulate details of `update()` methods and Aggregation Pipeline as well as other MongoTemplate interfaces like GridFS and Geolocation searches.
BooksRepository	our repository inherits from the repository hierarchy and adds additional methods that are automatically implemented by Spring Data MongoDB

@Document is not Technically Required

Technically, the `@Document` annotation is not required unless mapping to a non-default collection. However, `@Document` will continue to be referenced in this lecture to mean the "subject of the repository".

Chapter 4. BooksRepository

All we need to create a functional repository is a `@Document` class and a primary key type. The `@Document` annotation is optional and only required to specify a collection name different from the class name. From our work to date, we know that our `@Document` is the Book class and the primary key is the primitive `String` type. This type works well with MongoDB auto-generated IDs.

4.1. Book @Document

Book @Document Example

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
...
@Document(collection = "books")
public class Book {
    @Id
    private String id;
```

Multiple @Id Annotations, Use Spring Data's @Id Annotation

The `@Id` annotation looks the same as the JPA `@Id`, but instead comes from the Spring Data package

```
import org.springframework.data.annotation.Id;
```

4.2. BooksRepository

We declare our repository to extend `MongoRepository`.

```
public interface BooksRepository extends MongoRepository<Book, String> {}① ②
```

① Book is the repository type

② String is used for the primary key type

Consider Using Non-Primitive Primary Key Types

You will find that Spring Data MongoDB works easier with nullable object types.

Chapter 5. Configuration

Assuming your repository classes are in a package below the class annotated with `@SpringBootApplication` — not much is else is needed. Adding `@EnableMongoRepositories` is necessary when working with more complex classpaths.

Typical MongoDB Repository Support Declaration

```
@SpringBootApplication
@EnableMongoRepositories
public class MongoDBBooksApp {
```

If your repository is not located in the default packages scanned, their packages can be scanned with configuration options to the `@EnableMongoRepositories` annotation.

Configuring Repository Package Scanning

```
@EnableMongoRepositories(basePackageClasses = {BooksRepository.class}) ① ②
```

① the Java class provided here is used to identify the base Java package

② where to scan for repository interfaces

5.1. Injection

With the repository interface declared and the Mongo repository support enabled, we can then successfully inject the repository into our application.

BooksRepository Injection

```
@Autowired
private BooksRepository booksRepository;
```

Chapter 6. CrudRepository

Lets start looking at the capability of our repository — starting with the declared methods of the `CrudRepository` interface.

CrudRepository<T, ID> Interface

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
  <S extends T> S save(S);
  <S extends T> Iterable<S> saveAll(Iterable<S>);
  Optional<T> findById(ID);
  boolean existsById(ID);
  Iterable<T> findAll();
  Iterable<T> findAllById(Iterable<ID>);
  long count();
  void deleteById(ID);
  void delete(T);
  void deleteAllById(Iterable<? extends ID>);
  void deleteAll(Iterable<? extends T>);
  void deleteAll();
}

public interface ListCrudRepository<T, ID> extends CrudRepository<T, ID> {
  <S extends T> List<S> saveAll(Iterable<S>);
  List<T> findAll();
  List<T> findAllById(Iterable<ID>);
}
```

6.1. CrudRepository save() New

We can use the `CrudRepository.save()` method to either create or update our `@Document` instance in the database. It has a direct correlation to MongoTemplate's `save()` method so there is not much extra functionality added by the repository layer.

In this specific example, we call `save()` with an object with an unassigned primary key. The primary key will be generated by the database when inserted and assigned to the object by the time the command completes.

CrudRepository.save() New Example

```
//given a transient document instance
Book book = ...
assertThat(book.getId()).isNull(); ①
//when persisting
booksRepo.save(book);
//then document is persisted
then(book.getId()).isNotNull(); ②
```

① document not yet assigned a generated primary key

② primary key assigned by database

6.2. CrudRepository save() Update Existing

The `CrudRepository.save()` method is an "upsert" method.

- if the `@Document` is new it will be inserted
- if a `@Document` exists with the currently assigned primary key, the original contents will be replaced

CrudRepository.save() Update Existing Example

```
//given a persisted document instance
Book book = ...
booksRepo.save(book); ①
Book updatedBook = book.withTitle("new title"); ②
//when persisting update
booksRepo.save(updatedBook);
//then new document state is persisted
then(booksRepo.findOne(Example.of(updatedBook))).isPresent(); ③
```

① object inserted into database — resulting in primary key assigned

② a separate instance with the same ID has modified title

③ object's new state is found in database

6.3. CrudRepository save()/Update Resulting MongoDB Command

Watching the low-level MongoDB commands, we can see that Mongo's built-in `upsert` capability allows the client to perform the action without a separate query.

MongoDB Update Command Performed with Upsert

```
update{"q":{"_id":{"$oid":"606cbfc0932e084392422bb6"}}, ①
   "u":{"_id":{"$oid":"606cbfc0932e084392422bb6"},"title":"new title","author":...},
   "multi":false,
   "upsert":true} ②
```

① filter looks for ID

② insert if not exist, update if exists

6.4. CrudRepository existsById()

The repository adds a convenience method that can check whether the `@Document` exists in the database without already having an instance or writing a criteria query.

The following snippet demonstrates how we can check for the existence of a given ID.

CrudRepository existsById()

```
//given a persisted document instance
Book pojoBook = ...
booksRepo.save(pojoBook);
//when - determining if document exists
boolean exists = booksRepo.existsById(pojoBook.getId());
//then
then(exists).isTrue();
```

The resulting MongoDB command issued a query for the ID, limiting the results to a single result, and a projection with only the primary key contained.

CrudRepository existsById() SQL

```
query: { _id: ObjectId('606cc5d742931870e951e08e') }
 sort: {}
 projection: {} ①
 collation: { locale: \"simple\" }
 limit: 1
```

① `projection: {}` returns only the primary key

6.5. CrudRepository findById()

If we need the full object, we can always invoke the `findById()` method, which should be a thin wrapper above `MongoTemplate.find()`, except that the return type is a Java `Optional<T>` versus the `@Document` type (`T`).

CrudRepository.findById()

```
//given a persisted document instance
Book pojoBook = ...
booksRepo.save(pojoBook);
//when - finding the existing document
Optional<Book> result = booksRepo.findById(pojoBook.getId()); ①
//then
then(result.isPresent()).isTrue();
```

① `findById()` always returns a non-null `Optional<T>` object

6.5.1. CrudRepository findById() Found Example

The `Optional<T>` can be safely tested for existence using `isPresent()`. If `isPresent()` returns `true`, then `get()` can be called to obtain the targeted `@Document`.

Present Optional Example

```
//given
then(result).isPresent();
//when - obtaining the instance
Book dbBook = result.get();
//then - instance provided
then(dbBook).isNotNull();
//then - database copy matches initial POJO
then(dbBook.getAuthor()).isEqualTo(pojoBook.getAuthor());
then(dbBook.getTitle()).isEqualTo(pojoBook.getTitle());
then(pojoBook.getPublished()).isEqualTo(dbBook.getPublished());
```

6.5.2. CrudRepository findById() Not Found Example

If `isPresent()` returns `false`, then `get()` will throw a `NoSuchElementException` if called. This gives your code some flexibility for how you wish to handle a target `@Document` not being found.

Missing Optional Example

```
//then - the optional can be benignly tested
then(result).isNotPresent();
//then - the optional is asserted during the get()
assertThatThrownBy(() -> result.get())
        .isInstanceOf(NoSuchElementException.class);
```

6.6. CrudRepository delete()

The repository also offers a wrapper around `MongoTemplate.remove()` that accepts an instance. Whether the instance existed or not, a successful call will always result in the `@Document` no longer in the database.

CrudRepository delete() Example

```
//when - deleting an existing instance
booksRepo.delete(existingBook);
//then - instance will be removed from DB
then(booksRepo.existsById(existingBook.getId())).isFalse();
```

6.6.1. CrudRepository delete() Not Exist

If the instance did not exist, the `delete()` call silently returns.

CrudRepository delete() Does Not Exists Example

```
//when - deleting a non-existing instance
booksRepo.delete(doesNotExist);
```

6.7. CrudRepository deleteById()

The repository also offers a convenience `deleteById()` method taking only the primary key.

CrudRepository deleteById() Example

```
//when - deleting an existing instance
booksRepo.deleteById(existingBook.getId());
```

6.8. Other CrudRepository Methods

That was a quick tour of the `CrudRepository<T,ID>` interface methods. The following snippet shows the methods not covered. Most provide convenience methods around the entire repository.

Other CrudRepository Methods

```
//public interface CrudRepository<T, ID> extends Repository<T, ID> {
<S extends T> Iterable<S> saveAll(Iterable<S>);
Iterable<T> findAll();
Iterable<T> findAllById(Iterable<ID>);
long count();
void deleteAll(Iterable<? extends T>);
void deleteAll();

//public interface ListCrudRepository<T, ID> extends CrudRepository<T, ID> {
<S extends T> List<S> saveAll(Iterable<S>);
List<T> findAll();
List<T> findAllById(Iterable<ID>);
```

Chapter 7. PagingAndSortingRepository

Before we get too deep into queries, it is good to know that Spring Data MongoDB has first-class support for sorting and paging.

- **sorting** - determines the order which matching results are returned
- **paging** - breaks up results into chunks that are easier to handle than entire database collections

Here is a look at the declared methods of the `PagingAndSortingRepository<T,ID>` interface. This defines extra parameters for the `CrudRepository.findAll()` methods.

PagingAndSortingRepository<T,ID> Interface

```
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {
    Iterable<T> findAll(Sort var1);
    Page<T> findAll(Pageable var1);
}
```

We will see paging and sorting option come up in many other query types as well.

Use Paging and Sorting for Collection Queries

All queries that return a collection should seriously consider adding paging and sorting parameters. Small test databases can become significantly populated production databases over time and cause eventual failure if paging and sorting is not applied to unbounded collection query return methods.

7.1. Sorting

Sorting can be performed on one or more properties and in ascending and/or descending order.

The following snippet shows an example of calling the `findAll()` method and having it return

- `Book` entities in descending order according to `published` date
- `Book` entities in ascending order according to `id` value when `published` dates are equal

Sort.by() Example

```
//when
List<Book> byPublished = booksRepository.findAll(
        Sort.by("published").descending().and(Sort.by("id").ascending()));① ②
//then
LocalDate previous = null;
for (Book s: byPublished) {
    if (previous!=null) {
        then(previous).isAfterOrEqualTo(s.getPublished()); //DESC order
    }
    previous=s.getPublished();
```

```
}
```

① results can be sorted by one or more properties

② order of sorting can be ascending or descending

The following snippet shows how the MongoDB command was impacted by the `Sort.by()` parameter.

Sort.by() Example MongoDB Command

```
query: {}
sort: { published: -1, _id: 1 } ①
projection: {}
```

① Sort.by() added the extra sort parameters to MongoDB command

7.2. Paging

Paging permits the caller to designate how many instances are to be returned in a call and the offset to start that group (called a page or slice) of instances.

The snippet below shows an example of using one of the factory methods of `Pageable` to create a `PageRequest` definition using page size (limit), offset, and sorting criteria. If many pages will be traversed — it is advised to sort by a property that will produce a stable sort over time during table modifications.

Defining Initial Pageable

```
//given
int offset = 0;
int pageSize = 3;
Pageable pageable = PageRequest.of(offset/pageSize, pageSize, Sort.by("published"));①
//when
Page<Book> bookPage = booksRepository.findAll(pageable);
```

① using `PageRequest` factory method to create `Pageable` from provided page information

Use Stable Sort over Large Collections

Try to use a property for sort (at least by default) that will produce a stable sort when paging through a large collection to avoid repeated or missing objects from follow-on pages because of new changes to the table.

7.3. Page Result

The page result is represented by a container object of type `Page<T>`, which extends `Slice<T>`. I will describe the difference next, but the `PagingAndSortingRepository<T,ID>` interface always returns a `Page<T>`, which will provide:

- the sequential number of the page/slice
- the requested size of the page/slice
- the number of elements found
- the total number of elements available in the database

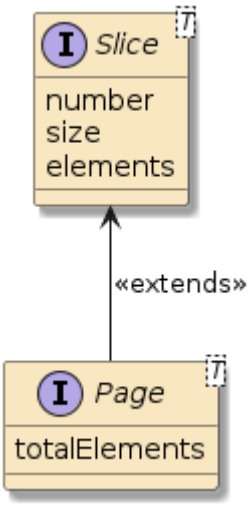

Figure 2. Page<T> Extends Slice<T>

7.4. Slice Properties

The `Slice<T>` base interface represents properties about the content returned.

Slice Properties

```
//then
Slice bookSlice = bookPage; ①
then(bookSlice).isNotNull();
then(bookSlice.isEmpty()).isFalse();
then(bookSlice.getNumber()).isEqualTo(0); ②
then(bookSlice.getSize()).isEqualTo(pageSize); ③
then(bookSlice.getNumberOfElements()).isEqualTo(pageSize); ④

List<Book> booksList = bookSlice.getContent();
then(booksList).hasSize(pageSize);
```

① `Page<T>` extends `Slice<T>`

② slice increment — first slice is 0

③ the number of elements requested for this slice

④ the number of elements returned in this slice

7.5. Page Properties

The `Page<T>` derived interface represents properties about the entire collection/table.

The snippet below shows an example of the total number of elements in the table being made available to the caller.

Page Properties

```
then(bookPage.getTotalElements()).isEqualTo(savedBooks.size());
```

7.6. Stateful Pageable Creation

In the above example, we created a `Pageable` from stateless parameters. We can also use the original `Pageable` to generate the next or other relative page specifications.

Relative Pageable Creation

```
Pageable pageable = PageRequest.of(offset / pageSize, pageSize, Sort.by("published"));
...
Pageable next = pageable.next();
Pageable previous = pageable.previousOrFirst();
Pageable first = pageable.first();
```

7.7. Page Iteration

The next `Pageable` can be used to advance through the complete set of query results, using the previous `Pageable` and testing the returned `Slice`.

Page Iteration

```
for (int i=1; bookSlice.hasNext(); i++) { ①
    pageable = pageable.next(); ②
    bookSlice = booksRepository.findAll(pageable);
    booksList = bookSlice.getContent();
    then(bookSlice).isNotNull();
    then(bookSlice.getNumber()).isEqualTo(i);
    then(bookSlice.getSize()).isEqualTo(pageSize);
    then(bookSlice.getNumberOfElements()).isLessThanOrEqualTo(pageSize);
    then(((Page)bookSlice).getTotalElements()).isEqualTo(savedBooks.size());//unique
to Page
}
then(bookSlice.hasNext()).isFalse();
then(bookSlice.getNumber()).isEqualTo(booksRepository.count() / pageSize);
```

① `Slice.hasNext()` will indicate when previous `Slice` represented the end of the results

② next `Pageable` obtained from previous `Pageable`

Chapter 8. Query By Example

Not all queries will be as simple as `findAll()`. We now need to start looking at queries that can return a subset of results based on them matching a set of predicates. The `QueryByExampleExecutor<T>` parent interface to `MongoRepository<T,ID>` provides a set of variants to the collection-based results that accepts an "example" to base a set of predicates off of.

QueryByExampleExecutor<T> Interface

```
public interface QueryByExampleExecutor<T> {
  <S extends T> Optional<S> findOne(Example<S>);
  <S extends T> Iterable<S> findAll(Example<S>);
  <S extends T> Iterable<S> findAll(Example<S>, Sort);
  <S extends T> Page<S> findAll(Example<S>, Pageable);
  <S extends T> long count(Example<S>);
  <S extends T> boolean exists(Example<S>);
  <S extends T, R> R findBy(Example<S>, Function<FluentQuery$FetchableFluentQuery<S>,
R>);
}
```

8.1. Example Object

An `Example` is an interface with the ability to hold onto a probe and matcher.

8.1.1. Probe Object

The probe is an instance of the repository `@Document` type.

The following snippet is an example of creating a probe that represents the fields we are looking to match.

Probe Example

```
//given
Book savedBook = savedBooks.get(0);
Book probe = Book.builder()
        .title(savedBook.getTitle())
        .author(savedBook.getAuthor())
        .build(); ①
```

① probe will carry values for `title` and `author` to match

8.1.2. ExampleMatcher Object

The matcher defaults to an exact match of all non-null properties in the probe. There are many definitions we can supply to customize the matcher.

- `ExampleMatcher.matchingAny()` - forms an OR relationship between all predicates

- `ExampleMatcher.matchingAll()` - forms an AND relationship between all predicates

The matcher can be broken down into specific fields, designing a fair number of options for String-based predicates but very limited options for non-String fields.

- exact match
- case insensitive match
- starts with, ends with
- contains
- regular expression
- include or ignore nulls

The following snippet shows an example of the default `ExampleMatcher`.

Default ExampleMatcher

```
ExampleMatcher matcher = ExampleMatcher.matching(); ①
```

① default matcher is `matchingAll`

8.2. findAll By Example

We can supply an `Example` instance to the `findAll()` method to conduct our query.

The following snippet shows an example of using a probe with a default matcher. It is intended to locate all books matching the `author` and `title` we specified in the probe.

```
//when
List<Book> foundBooks = booksRepository.findAll(
        Example.of(probe),//default matcher is matchingAll() and non-null
        Sort.by("id"));
```

The default matcher ends up working perfectly with our `@Document` class because a nullable primary key was used — keeping the primary key from being added to the criteria.

8.3. Ignoring Properties

If we encounter any built-in types that cannot be null — we can configure a match to explicitly ignore certain fields.

The following snippet shows an example matcher configured to ignore the primary key.

matchingAll ExampleMatcher with Ignored Property

```
ExampleMatcher ignoreId = ExampleMatcher.matchingAll().withIgnorePaths("id");①
//when
List<Book> foundBooks = booksRepository.findAll(
        Example.of(probe, ignoreId), ②
        Sort.by("id"));
//then
then(foundBooks).isNotEmpty();
```

```
then(foundBooks.get(0).getId()).isEqualTo(savedBook.getId());
```

① `id` primary key is being excluded from predicates

② non-null and non-id fields of probe are used for **AND** matching

8.4. Contains ExampleMatcher

We have some options on what we can do with the String matches.

The following snippet provides an example of testing whether `title` contains the text in the probe while performing an exact match of the `author` and ignoring the `id` field.

Contains ExampleMatcher

```
Book probe = Book.builder()
        .title(savedBook.getTitle().substring(2))
        .author(savedBook.getArtist())
        .build();
ExampleMatcher matcher = ExampleMatcher
        .matching()
        .withIgnorePaths("id")
        .withMatcher("title", ExampleMatcher.GenericPropertyMatchers.contains());
```

8.4.1. Using Contains ExampleMatcher

The following snippet shows that the `Example` successfully matched on the `Book` we were interested in.

Example is Found

```
//when
List<Book> foundBooks=booksRepository.findAll(Example.of(probe,matcher), Sort.by("id"
));
//then
then(foundBooks).isNotEmpty();
then(foundBooks.get(0).getId()).isEqualTo(savedBook.getId());
```

Chapter 9. Derived Queries

For fairly straight forward queries, Spring Data MongoDB can derive the required commands from a method signature declared in the repository interface. This provides a more self-documenting version of similar queries we could have formed with query-by-example.

The following snippet shows a few example queries added to our repository interface to address specific queries needed in our application.

Example Query Method Names

```
public interface BooksRepository extends MongoRepository<Book, String> {
    Optional<Book> getByTitle(String title); ①

    List<Book> findByTitleNullAndPublishedAfter(LocalDate date); ②

    List<Book> findByTitleStartingWith(String string, Sort sort); ③
    Slice<Book> findByTitleStartingWith(String string, Pageable pageable); ④
    Page<Book> findPageByTitleStartingWith(String string, Pageable pageable); ⑤
```

① query by an exact match of `title`

② query by a match of two fields (`title` and `released`)

③ query using sort

④ query with paging support

⑤ query with paging support and table total

Let's look at a complete example first.

9.1. Single Field Exact Match Example

In the following example, we have created a query method `getByTitle` that accepts the exact match title value and an `Optional` return value.

Interface Method Signature

```
Optional<Book> getByTitle(String title); ①
```

We use the declared interface method in a normal manner and Spring Data MongoDB takes care of the implementation.

Interface Method Usage

```
//when
Optional<Book> result = booksRepository.getByTitle(book.getTitle());
//then
then(result.isPresent()).isTrue();
```

The result is essentially the same as if we implemented it using query-by-example or more directly through the MongoTemplate.

9.2. Query Keywords

Spring Data has several keywords, followed by `By`, that it looks for starting the interface method name. Those with multiple terms can be used interchangeably.

Meaning	Keywords		
Query	• find • read	• get • query	• search • stream
Count	• count		
Exists	• exists		
Delete	• delete • remove		

9.3. Other Keywords

Other keywords are clearly documented in the JPA reference [1] [2]

- Distinct (e.g., `findDistinctByTitle`)
- Is, Equals (e.g., `findByTitle`, `findByTitleIs`, `findByTitleEquals`)
- Not (e.g., `findByTitleNot`, `findByTitleIsNot`, `findByTitleNotEquals`)
- IsNull, IsNotNull (e.g., `findByTitle(null)`, `findByTitleIsNull()`, `findByTitleIsNotNull()`)
- StartingWith, EndingWith, Containing (e.g., `findByTitleStartingWith`, `findByTitleEndingWith`, `findByTitleContaining`)
- LessThan, LessThanEqual, GreaterThan, GreaterThanEqual, Between (e.g., `findByIdLessThan`, `findByIdBetween(lo,hi)`)
- Before, After (e.g., `findByPublishedAfter`)
- In (e.g., `findByTitleIn(collection)`)
- OrderBy (e.g., `findByTitleContainingOrderByTitle`)

The list is significant, but not meant to be exhaustive. Perform a web search for your specific needs (e.g., "Spring Data Derived Query ...") if what is needed is not found here.

9.4. Multiple Fields

We can define queries using one or more fields using `And` and `Or`.

The following example defines an interface method that will test two fields: `title` and `published`. `title` will be tested for null and `published` must be after a certain date.

Multiple Fields Interface Method Declaration

```
List<Book> findByTitleNullAndPublishedAfter(LocalDate date);
```

The following snippet shows an example of how we can call/use the repository method. We are using a simple collection return without sorting or paging.

Multiple Fields Example Use

```
//when
List<Book> foundBooks = booksRepository.findByTitleNullAndPublishedAfter(firstBook
.getPublished());
//then
Set<String> foundIds = foundBooks.stream().map(s->s.getId()).collect(Collectors.toSet
());
then(foundIds).isEqualTo(expectedIds);
```

9.5. Collection Response Query Example

We can perform queries with various types of additional arguments and return types. The following shows an example of a query that accepts a sorting order and returns a simple collection with all objects found.

Collection Response Interface Method Declaration

```
List<Book> findByTitleStartingWith(String string, Sort sort);
```

The following snippet shows an example of how to form the `Sort` and call the query method derived from our interface declaration.

Collection Response Interface Method Use

```
//when
Sort sort = Sort.by("id").ascending();
List<Book> books = booksRepository.findByTitleStartingWith(startingWith, sort);
//then
then(books.size()).isEqualTo(expectedCount);
```

9.6. Slice Response Query Example

Derived queries can also be declared to accept a `Pageable` definition and return a `Slice`. The following example shows a similar interface method declaration to what we had prior — except we have wrapped the `Sort` within a `Pageable` and requested a `Slice`, which will contain only those items that match the predicate and comply with the paging constraints.

Slice Response Interface Method Declaration

```
Slice<Book> findByTitleStartingWith(String string, Pageable pageable);
```

The following snippet shows an example of forming the `PageRequest`, making the call, and inspecting the returned `Slice`.

Slice Response Interface Method Use

```
//when
PageRequest pageable=PageRequest.of(0, 1, Sort.by("id").ascending());① ②
Slice<Book> booksSlice=booksRepository.findByTitleStartingWith(startingWith,pageable);
//then
then(booksSlice.getNumberOfElements()).isEqualTo(pageable.getPageSize());
```

① `pageNumber` is 0

② `pageSize` is 1

9.7. Page Response Query Example

We can alternatively declare a `Page` return type if we also need to know information about all available matches in the table. The following shows an example of returning a `Page`. The only reason `Page` shows up in the method name is to form a different method signature than its sibling examples. `Page` is not required to be in the method name.

Page Response Interface Method Declaration

```
Page<Book> findPageByTitleStartingWith(String string, Pageable pageable); ①
```

① the `Page` return type (versus `Slice`) triggers an extra query performed to supply `totalElements` `Page` property

The following snippet shows how we can form a `PageRequest` to pass to the derived query method and accept a `Page` in reponse with additional table information.

Page Response Interface Method Use

```
//when
PageRequest pageable = PageRequest.of(0, 1, Sort.by("id").ascending());
Page<Book> booksPage = booksRepository.findPageByTitleStartingWith(startingWith,
pageable);
//then
then(booksPage.getNumberOfElements()).isEqualTo(pageable.getPageSize());
then(booksPage.getTotalElements()).isEqualTo(expectedCount); ①
```

① an extra property is available to tell us the total number of matches relative to the entire table — that may not have been reported on the current page

[1] *"Query Creation"*, Spring Data JPA - Reference Documentation

[2] *"Derived Query Methods in Spring Data JPA"*, Atta

Chapter 10. @Query Annotation Queries

Spring Data MongoDB provides an option for the query to be expressed on the repository method.

The following example will locate a book published between the provided dates — inclusive. The default derived query implemented it exclusive of the two dates. The `@Query` annotation takes precidence over the default derived query. This shows how easy it is to define a customized version of the query.

Example @Query

```
@Query("{ 'published': { $gte: ?0, $lte: ?1 } }") ①
List<Book> findByPublishedBetween(LocalDate starting, LocalDate ending);
```

① `?0` is the first parameter (`starting`) and `?1` is the second parameter (`ending`)

The following snippet shows an example of implementing a query using a regular expression completed by the input parameters. It locates all books with `titles` greater-than or equal to the `length` parameter. It also declares that only the `title` field of the `Book` instances need to be returned — making the result smaller.

Query Supplied on Repository Method

```
@Query(value="{ 'title': /^.{?0,}$/ }", fields="{'_id':0, 'title':1}") ① ②
List<Book> getTitlesGESizeAsBook(int length);
```

① `value` expresses which Books should match

② `fields` expresses which fields should be returned and populated in the instance

Named Queries can be supplied in property file

Named queries can also be expressed in a property file — versus being placed directly onto the method. Property files can provide a more convenient source for expressing more complex queries.

```
@EnableMongoRepositories(namedQueriesLocation="...")
```

The default location is `META-INF/mongo-named-queries.properties`

10.1. @Query Annotation Attributes

The matches in the query can be used for more than just `find`. We can alternately apply `count`, `exists`, or `delete` and include information for `fields` projected, `sort`, and `collation`.

Table 1. @Query Annotation Attributes

Attribute	Default	Description	Example
String fields	""	projected fields	fields = "{ title : 1 }"
boolean count	false	count() action performed on query matches	
boolean exists	false	exists() action performed on query matches	
boolean delete	false	delete() action performed on query matches	
String sort	""	sort expression for query results	sort = "{ published : -1 }"
String collation	""	location information	

10.2. @Query Sort and Paging

The `@Query` approach supports paging via `Pageable` parameter. Sort must be defined using the `@Query.sort` property.

@Query Sort and Paging

```
@Query
@Query(value="{ 'published': { $gte: ?0, $lte: ?1 } }", sort = "{ '_id':1 }")
Page<Book> findByPublishedBetween(LocalDate starting, LocalDate ending, Pageable
pageable);
```

Chapter 11. MongoRepository Methods

Many of the methods and capabilities of the `MongoRepository<T,ID>` are available at the higher level interfaces. The `MongoRepository<T,ID>` itself declares two types of additional methods

- insert/upsert state-specific optimizations
- return type extensions

MongoRepository<T, ID> Interface

```
<S extends T> S insert(S); ①
<S extends T> List<S> insert(Iterable<S>);

<S extends T> List<S> findAll(Example<S>); ②
<S extends T> List<S> findAll(Example<S>, Sort);
public default Iterable findAll(Example, Sort);
public default Iterable findAll(Example);
```

① `insert` is specific to MongoRepository and assumes the document is new

② `List<T>` is a sub-type of `Iterable<T>` and provides a richer set of inspection methods for the returned result from QueryByExample methods

Chapter 12. Custom Queries

Sooner or later, a repository action requires some complexity that is beyond the ability to leverage a single query-by-example or derived query. We may need to implement some custom logic or may want to encapsulate multiple calls within a single method.

12.1. Custom Query Interface

The following example shows how we can extend the repository interface to implement custom calls using the MongoTemplate and the other repository methods. Our custom implementation will return a random `Book` from the database.

Interface for Public Custom Query Methods

```
public interface BooksRepositoryCustom {
    Optional<Book> random();
}
```

12.2. Repository Extends Custom Query Interface

We then declare the repository to extend the additional custom query interface — making the new method(s) available to callers of the repository.

Repository Implements Custom Query Interface

```
public interface BooksRepository extends MongoRepository<Book, String>,
BooksRepositoryCustom { ①
    ...
```

① added additional `BookRepositoryCustom` interface for `BookRepository` to extend

12.3. Custom Query Method Implementation

Of course, the new interface will need an implementation. This will require at least two lower-level database calls

1. determine how many objects there are in the database
2. return an instance for one of those random values

The following snippet shows a portion of the custom method implementation. Note that two additional helper methods are required. We will address them in a moment. By default, this class must have the same name as the interface, followed by "Impl".

Custom Query Method Implementation

```
public class BookRepositoryCustomImpl implements BookRepositoryCustom {
    private final SecureRandom random = new SecureRandom();
```

```
...
    @Override
    public Optional<Book> random() {
        Optional randomBook = Optional.empty();
        int count = (int) booksRepository.count(); ①

        if (count!=0) {
            int offset = random.nextInt(count);
            List<Book> books = books(offset, 1); ②
            randomBook=books.isEmpty() ? Optional.empty() : Optional.of(books.get(0));
        }
        return randomBook;
    }
```

① leverages `CrudRepository.count()` helper method

② leverages a local, private helper method to access specific `Book`

12.4. Repository Implementation Postfix

If you have an alternate suffix pattern other than "Impl" in your application, you can set that value in an attribute of the `@EnableMongoRepositories` annotation.

The following shows a declaration that sets the suffix to its normal default value (i.e., we did not have to do this). If we changed this value from "Impl" to "Xxx", then we would need to change `BooksRepositoryCustomImpl` to `BooksRepositoryCustomXxx`.

Optional Custom Query Method Implementation Suffix

```
@EnableMongoRepositories(repositoryImplementationPostfix="Impl")①
```

① `Impl` is the default value. Configure this attribute to use non-Impl postfix

12.5. Helper Methods

The custom `random()` method makes use of two helper methods. One is in the `CrudRepository` interface and the other directly uses the `MongoTemplate` to issue a query.

CrudRepository.count() Used as Helper Method

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    long count();
```

EntityManager NamedQuery used as Helper Method

```
protected List<Book> books(int offset, int limit) {
    return mongoTemplate.find(new Query().skip(offset).limit(limit), Book.class);
}
```

We will need to inject some additional resources in order to make these calls:

- `BooksRepository`
- `MongoTemplate`

12.6. Naive Injections

Since we are not using sessions or transactions with Mongo, a simple/naive injection will work fine. We do not have to worry about injecting a specific instance. However, we will run into a circular dependency issue with the BooksRepository.

Naive Injections

```
@RequiredArgsConstructor
public class BooksRepositoryCustomImpl implements BooksRepositoryCustom {
    private final MongoTemplate mongoTemplate; ①
    private final BooksRepository booksRepository; ②
```

① any MongoTemplate instance referencing the correct database and collection is fine

② eager/mandatory injection of self needs to be delayed

12.7. Required Injections

We need to instead

- use `@Autowired @Lazy` and a non-final attribute for the `BooksRepository` injection to indicate that this instance can be initialized without access to the injected bean

Required Injections

```
import org.springframework.data.jpa.repository.MongoContext;
...
public class BooksRepositoryCustomImpl implements BooksRepositoryCustom {
    private final MongoTemplate mongoTemplate;
    @Autowired @Lazy ①
    private BooksRepository booksRepository;
```

① `BooksRepository` lazily injected to mitigate the recursive dependency between the `Impl` class and the full repository instance

12.8. Calling Custom Query

With all that in place, we can then call our custom `random()` method and obtain a sample `Book` to work with from the database.

Example Custom Query Client Call

```
//when
```

```
Optional<Book> randomBook = booksRepository.random();
//then
then(randomBook.isPresent()).isTrue();
```

12.9. Implementing Aggregation

MongoTemplate has more power in it than what can be expressed with MongoRepository. As seen with the `random()` implementation, we have the option of combining operations and dropping down the to `MongoTemplate` for a portion of the implementation. That can also include use of the Aggregation Pipeline, GridFS, Geolocation, etc.

The following custom implementation is declared in the Custom interface, extended by the BooksRepository.

Custom Query Interface Definition

```
public interface BookRepositoryCustom {
...
   List<Book> findByAuthorGESize(int length);
```

The snippet below shows the example leveraging the Aggregation Pipeline for its implementation and returning a normal `List<Book>` collection.

Custom Query Implementation Based On Aggregation Pipeline

```
@Override
public List<Book> findByAuthorGESize(int length) {
    String expression = String.format("^.{%d,}$", length);

    Aggregation pipeline = Aggregation.newAggregation(
            Aggregation.match(Criteria.where("author").regex(expression)),
            Aggregation.match(Criteria.where("author").exists(true))
    );
    AggregationResults<Book> result =
            mongoTemplate.aggregate(pipeline, "books", Book.class);
    return result.getMappedResults();
}
```

That allows us unlimited behavior in the data access layer and the ability to encapsulate the capability into a single data access component.

Chapter 13. Summary

In this module, we learned:

- that Spring Data MongoDB eliminates the need to write boilerplate MongoTemplate code
- to perform basic CRUD management for `@Document` classes using a repository
- to implement query-by-example
- that unbounded collections can grow over time and cause our applications to eventually fail
    - that paging and sorting can easily be used with repositories
- to implement query methods derived from a query DSL
- to implement custom repository extensions