(/)

Introduction to Spring Data MongoDB

Last updated: February 10, 2025

Written by: Hardik Singh Behl
(https://www.baeldung.com/author/hardiksinghbehl)

Reviewed by: Liam Williams
(https://www.baeldung.com/editor/liamwilliams)

NoSQL (https://www.baeldung.com/category/persistence/nosql)

Spring Data (https://www.baeldung.com/category/persistence/spring-persistence/spring-data)

MongoDB Basics (https://www.baeldung.com/tag/mongodb-basics)

Persistence Basics (https://www.baeldung.com/tag/persistence-basics)

reference

Spring Data MongoDB (https://www.baeldung.com/tag/spring-data-mongodb)

1. Overview

(/)

NoSQL databases (/cs/sql-vs-nosql#nosql-databases) have become a popular choice when building a flexible and scalable persistence layer. MongoDB (https://www.mongodb.com/) is a widely adopted document-oriented database that handles large volumes of unstructured and semi-structured data.

Spring Data MongoDB provides a high-level abstraction over the MongoDB Query API (https://www.mongodb.com/docs/manual/query-api/) and simplifies integration of MongoDB into our application.

In this tutorial, we'll explore integrating MongoDB into a Spring Boot application using Spring Data MongoDB. We'll walk through the necessary configuration, establish a connection, and perform basic CRUD operations.

2. Setting up the Project

Before we can start interacting with MongoDB, we'll need to include the necessary dependencies and configure our application correctly.

2.1. Dependencies

Let's start by adding the Spring Boot starter for Spring Data MongoDB (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-mongodb) to our project's *pom.xml* file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
    <version>3.4.1</version>
</dependency>
```

This dependency provides us with the necessary classes to connect and interact with MongoDB from our application.

2.2. Connecting to MongoDB

To facilitate local development and testing, we'll use Testcontainers (/docker-test-containers) to set up MongoDB.

First, let's add the necessary test dependencies to our *pom.xml*:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-testcontainers</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>mongodb</artifactId>
    <scope>test</scope>
</dependency>
```

We import the Spring Boot Testcontainers (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-testcontainers) dependency along with the MongoDB module (https://mvnrepository.com/artifact/org.testcontainers/mongodb) of Testcontainers.

Next, let's create a *@TestConfiguration* (/spring-boot-testing#test-configuration-withtestconfiguration) class that defines our Testcontainers beans:

```
@TestConfiguration(proxyBeanMethods = false)
class TestcontainersConfiguration {
    @Bean
    MongoDBContainer mongoDbContainer() {
        return new MongoDBContainer(DockerImageName.parse("mongo:8"));
    }

    @Bean
    DynamicPropertyRegistrar dynamicPropertyRegistrar(MongoDBContainer
mongoDbContainer) {
        return registry -> {
            registry.add("spring.data.mongodb.uri",
mongoDbContainer::getConnectionString);
            registry.add("spring.data.mongodb.database", () ->
"technical-content-management");
        };
    }
}
```

We specify the latest stable version of the MongoDB Docker image (https://hub.docker.com/_/mongo) when creating the *MongoDBContainer* bean.

Then, **we define a *DynamicPropertyRegistrar* bean to configure the MongoDB connection URI**, allowing our application to connect to the started container.

Additionally, we define the database name against which we want to perform CRUD operations against. **If the database with the configured name does not exist, MongoDB will create one for us**.

2.3. Defining the Domain Entity

Now, let's define our entity class:

```
@Document(collection = "authors")
class Author {
    @Id
    private UUID id;

    @Field(name = "full_name")
    private String name;

    @Indexed(unique = true)
    private String email;

    @Field(name = "article_count")
    private Integer articleCount;

    private Boolean active;

    // standard getters and setters
}
```

The *Author* class is the central entity in our example, and we'll be using it to learn how to perform CRUD operations in the upcoming sections.

We use the *@Document* annotation to mark our entity class as a MongoDB document, mapping it to the *authors* collection in the database. **Additionally, we use the *@Id* annotation to specify our primary key**.

By default, the name of the entity attribute is mapped to the document field name. **To map our attribute to a different field name, we use the *@Field* annotation**. Here, we map the *name* attribute in our *Author* class to the *full_name* field in the MongoDB document.

Finally, **to enforce that no two authors can have the same email address, we use the *@Indexed* annotation and set the *unique* parameter to *true***.

3. Extending the *MongoRepository* Interface

Spring Data MongoDB provides the *MongoRepository* interface, which, **when extended, gives us access to various CRUD operations without needing to write explicit implementations**.

Let's create a repository interface for our *Author* entity:

```
interface AuthorRepository extends MongoRepository<Author, UUID> {
}
```

We specify the entity type and primary key type as generic parameters.

The Spring Boot starter for Spring Data MongoDB scans for repository interfaces within the main class package and creates their corresponding beans automatically.

If our repositories are in a separate package, we can use the *@EnableMongoRepositories* annotation:

```
@EnableMongoRepositories(basePackages = "com.non.root.repositories")
```

We specify the non-root package using the *basePackages* parameter. The *@EnableMongoRepositories* annotation can be placed on any *@Configuration* class, preferably the main *@SpringBootApplication* class.

3.1. Performing Basic CRUD Operations

The *MongoRepository* interface extends the *CrudRepository* (https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) interface, providing several generic methods for performing basic CRUD operations on our entities.

Let's start by creating a new *Author* document:

```
Author author = Instancio.create(Author.class);
Author savedAuthor = authorRepository.save(author);

String updatedName = RandomString.make();
savedAuthor.setName(updatedName);
authorRepository.save(savedAuthor);

Optional<Author> updatedAuthor =
authorRepository.findById(savedAuthor.getId());
assertThat(updatedAuthor)
  .isPresent()
  .get()
  .extracting(Author::getName)
  .isEqualTo(updatedName);
```

We use Instancio (/java-test-data-instancio) to create a new *Author* object with random test data and persist the *author* using the *save()* method.

Then, we update the name attribute of our *savedAuthor* object and persist it again, using the same *save()* method. We verify the update by fetching our document by its *id* using the *findById()* method.

Alternatively, we can use the *insert()* method to save a new document:

```
Author author = Instancio.create(Author.class);
Author savedAuthor = authorRepository.insert(author);

assertThrows(DuplicateKeyException.class, () ->
    authorRepository.insert(savedAuthor)
);
```

Note that **the *insert()* method is MongoDB-specific and not part of the *CrudRepository* interface**.

When we attempt to save an existing document — one with the same primary key — with *insert()*, we encounter an exception of type *DuplicateKeyException*. We'll also get this exception when we try to save a new document with an existing *email*, due to the unique constraint we specified earlier.

Finally, let's see how we can delete a document from the *authors* collection:

```
Author author = Instancio.create(Author.class);
Author savedAuthor = authorRepository.save(author);

Boolean authorExists = authorRepository.existsById(savedAuthor.getId());
assertThat(authorExists)
  .isTrue();

authorRepository.delete(savedAuthor);

authorExists = authorRepository.existsById(savedAuthor.getId());
assertThat(authorExists)
  .isFalse();
```

Again, we start by creating a new *author* document and persist it in the *authors* collection. We assert its existence by calling the *existsById()* method using the document's primary key.

We then delete the document using the *delete()* method and confirm it no longer exists. **Alternatively, the repository exposes a *deleteById()* method that we can use to delete a document with its primary key**.

3.2. Implementing Pagination and Sorting

When working with large datasets, **it's often necessary to paginate and sort the results**.

The *MongoRepository* also extends the *PagingAndSortingRepository* (https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html) interface, which provides a convenient way to do so:

```
int authorCount = 10;                    (/)
List<Author> authors = Instancio.ofList(Author.class)
  .size(authorCount)
  .create();
authorRepository.saveAll(authors);

Sort sort = Sort.by("name").ascending();
PageRequest pageRequest = PageRequest.of(0, 5, sort);
List<Author> retrievedAuthors = authorRepository.findAll(pageRequest)
  .getContent();

assertThat(retrievedAuthors)
  .hasSize(5)
  .extracting(Author::getName)
  .isSorted();
```

Here, we create *10* sample *Author* documents and save them to our collection.

We define a *Sort* object to sort the results by the *name* field in ascending order. Then, we use it along with the page number and page size to create a *PageRequest* object. **It's important to note that the page number is 0-indexed**.

We pass the *pageRequest* to the *findAll()* method and assert that the results are limited to *5* and are sorted by *name*.

To retrieve the next page of results, we can simply increment the page number:

```
PageRequest nextPageRequest = PageRequest.of(1, 5, sort);
retrievedAuthors = authorRepository.findAll(nextPageRequest)
  .getContent();
```

Here, we create a new *PageRequest* object with page number *1* and use it to fetch the next *5 author* documents.

3.3. Declaring Derived Query Methods

Similar to other modules of Spring Data, Spring Data MongoDB also allows us to declare simple queries using the derived query method approach.

By following a naming convention, we can declare methods in our repository interface, and Spring Data will automatically generate the appropriate MongoDB queries for us.

Let's declare a simple method in our *AuthorRepository* class to find an *Author* document by its *email*:

```
Optional<Author> findByEmail(String email);
```

Here, we declare a *findByEmail()* method that takes an *email* parameter. Let's test it out to see if Spring Data MongoDB is actually able to implement the method:

```
Author author = Instancio.create(Author.class);
authorRepository.save(author);

Optional<Author> retrievedAuthor =
authorRepository.findByEmail(author.getEmail());

assertThat(retrievedAuthor)
  .isPresent()
  .get()
  .usingRecursiveComparison()
  .isEqualTo(author);
```

After creating and saving an *Author* document, we use our derived query method of *findByEmail()* to fetch it by its *email* and assert that the retrieved *author* document matches the one we saved.

Similarly, **we can create more complex derived queries by chaining multiple conditions**:

```
List<Author> findByActiveTrueAndArticleCountGreaterThanEqual(int
articleCount);
```

We declare the above to find all *active author* documents with at least a given number of articles.

Spring Data MongoDB will generate a query where the value of the *active* field is *true* and the *article_count* is greater than or equal to the provided *articleCount* parameter:

```
int articleCount = 10;                    (/)
Author author = Instancio.of(Author.class)
  .set(field(Author::isActive), true)
  .generate(field(Author::getArticleCount), gen ->
gen.ints().min(articleCount))
  .create();
authorRepository.save(author);

List<Author> retrievedAuthors = authorRepository
  .findByActiveTrueAndArticleCountGreaterThanEqual(articleCount);

assertThat(retrievedAuthors)
  .singleElement()
  .usingRecursiveComparison()
  .isEqualTo(author);
```

We create an active *Author* document with at least *10* articles and save it to our collection. Then, we call our derived query method and assert that the saved *author* is returned as expected.

As we can see, this approach provides a convenient way to declare queries without writing any explicit implementation logic. **Check out our dedicated guide (/spring-data-derived-queries) to learn more about the structure and conditions of the derived query methods in Spring Data**.

3.4. Executing Custom Queries with the *@Query* Annotation

While derived query methods are convenient for simple queries, sometimes we need more fine-grained control over the executed queries. **We can use the *@Query* annotation, which allows us to define custom queries using the MongoDB query language**.

Let's declare a method to find all *active Author* documents with an article count within a given range:

```
@Query("{ 'article_count': { $gte: ?0, $lte: ?1 }, 'active': true }")
List<Author> findActiveAuthorsInArticleRange(int minArticles, int
maxArticles);
```

We use the *@Query* annotation to specify our custom query. **The *?0* and *?1* placeholders refer to the method parameters *minArticles* and *maxArticles***, respectively. Spring Data MongoDB will substitute these placeholders with the actual parameter values when executing the query.

Let's try out our new method:

```
int minArticleCount = 20;
int maxArticleCount = 50;
Author author = Instancio.of(Author.class)
  .set(field(Author::isActive), true)
  .generate(
      field(Author::getArticleCount),
      gen -> gen.ints().range(minArticleCount, maxArticleCount)
  )
  .create();
authorRepository.save(author);

List<Author> retrievedAuthors = authorRepository
  .findActiveAuthorsInArticleRange(minArticleCount, maxArticleCount);

assertThat(retrievedAuthors)
  .singleElement()
  .usingRecursiveComparison()
  .isEqualTo(author);
```

We create and save an active *Author* with an article count between *20* and *50*. Then, we call our custom query method, providing the same article count range, and assert that the *author* document we saved is returned.

Additionally, **to improve query performance, we can use the *@Query* annotation to specify the fields we want to retrieve**.

For our demonstration, let's create a method to find the email addresses of all active authors:

```
@Query(value = "{ 'active': true }", fields = "{ 'email': 1 }")
List<Author> findActiveAuthorEmails();
```

The above will only return the *email* and *id* fields, **since primary keys are included by default**:

```
int authorCount = 10;
List<Author> authors = Instancio.ofList(Author.class)
  .size(authorCount)
  .set(field(Author::isActive), true)
  .create();
authorRepository.saveAll(authors);

List<Author> retrievedAuthors =
authorRepository.findActiveAuthorEmails();

assertThat(retrievedAuthors)
  .hasSize(authorCount)
  .allSatisfy(author -> assertThat(author)
    .hasAllNullFieldsOrPropertiesExcept("id", "email"));
```

We create and save *10 Author* documents with their *active* field set to *true*. Then, we call our custom query method and assert that the result contains the same number of documents, each having only the *id* and *email* fields populated.

The *@Query* annotation allows us to leverage the capabilities of the native MongoDB query language while still benefiting from the repository abstraction that Spring Data MongoDB provides.

4. Using the *MongoTemplate* Bean

In addition to extending the *MongoRepository* interface, Spring Data MongoDB provides the *MongoTemplate* class as another way to interact with our MongoDB collections.

While the repository abstraction is suitable for most use cases, **the *MongoTemplate* is best suited for executing complex queries and provides fine-grained control over the executed operations**.

On configuring a valid MongoDB connection URI, **the Spring Boot starter automatically creates a *MongoTemplate* bean that we can autowire in our classes**.

4.1. Performing Basic CRUD Operations

Let's see how we can perform basic CRUD operations using *MongoTemplate*.

We'll start by inserting a new *Author* document:

```
Author author = Instancio.create(Author.class);
Author savedAuthor = mongoTemplate.insert(author);

Author retrievedAuthor = mongoTemplate.findById(savedAuthor.getId(),
Author.class);
assertThat(retrievedAuthor)
  .usingRecursiveComparison()
  .isEqualTo(author);
```

We insert a new *Author* using the *insert()* method. Then, we retrieve the saved document by its *id* using the *findById()* method and assert that it matches the original *author* object.

Alternatively, **similar to *MongoRepository*, we can use the *save()* method to use the save-or-update semantic**:

```
Author author = Instancio.create(Author.class);
Author savedAuthor = mongoTemplate.save(author);

String updatedName = RandomString.make();
savedAuthor.setName(updatedName);
mongoTemplate.save(savedAuthor);

Author updatedAuthor = mongoTemplate.findById(savedAuthor.getId(),
Author.class);
assertThat(updatedAuthor)
  .extracting(Author::getName)
  .isEqualTo(updatedName);

mongoTemplate.remove(updatedAuthor);

retrievedAuthor = mongoTemplate.findById(savedAuthor.getId(),
Author.class);
assertThat(retrievedAuthor)
  .isNull();
```

After persisting an *Author* object, we update its *name* attribute and save it again using the *save()* method. We verify the update by retrieving the document and asserting its *name* attribute.

Finally, we remove the document using the *remove()* method and confirm that it no longer exists in the database:

```
(//)
```

4.2. Complex Queries With *Query* and *Criteria*

To build complex queries with *MongoTemplate*, we can use the *Query* and *Criteria* classes. **We create *Criteria* objects to define the conditions for our query and combine them using the *Query* class**.

Let's query inactive *Author* documents from the *@baeldung.com* domain name:

```
Author author = Instancio.of(Author.class)
  .set(field(Author::isActive), false)
  .generate(field(Author::getEmail), gen ->
gen.text().pattern("#a#a#a@baeldung.com"))
  .create();
mongoTemplate.save(author);

Criteria nonActive = Criteria.where("active").is(false);
Criteria baeldungEmail =
Criteria.where("email").regex("@baeldung\\.com$");
Query query = new Query();
query.addCriteria(nonActive);
query.addCriteria(baeldungEmail);

List<Author> retrievedAuthors = mongoTemplate.find(query, Author.class);
assertThat(retrievedAuthors)
  .singleElement()
  .usingRecursiveComparison()
  .isEqualTo(author);
```

We create two *Criteria* objects, one for the *active* field and another for the *email* field. Then, we add both *criteria* objects to our *Query* object using the *addCriteria()* method. **This acts as an *AND* operator, requiring both conditions to be met**.

Finally, we execute the *query* using the *find()* method and assert that the *author* document we saved earlier is returned. When expecting a single result, we can use the *findOne()* method instead, which returns a single document or *null* if it finds no match.

Additionally, **we can create a query to implement the *OR* condition**:

```
Criteria highArticleCount = Criteria.where("article_count").gte(10);
Criteria missingName = Criteria.where("full_name").exists(false);
Criteria criteria = new Criteria();
criteria.orOperator(highArticleCount, missingName);
Query query = new Query();
query.addCriteria(criteria);

List<Author> retrievedAuthors = mongoTemplate.find(query, Author.class);
```

Here, we define two *Criteria* objects, one for *article_count* greater than or equal to *10* and another for documents missing the *full_name* field. We combine these two conditions by creating a new *Criteria* object and calling the *orOperator()* method.

Alternatively, **we can also use the actual Java field names of *articleCount* and *name* when defining our *Criteria* objects**.

As we've seen, the *Query* and *Criteria* classes provide a flexible and expressive way to build complex queries. We can even mix and match *AND* and *OR* operators to create even more complex queries.

4.3. Executing Update Operations

In addition to querying, ***MongoTemplate* also provides methods for executing update operations on our documents**.

First, let's see how we can update a single document:

```
Query query = new Query(Criteria.where("name").is(name));
Update update = new Update();
update.set("active", false);

UpdateResult updateResult = mongoTemplate.updateFirst(query, update,
Author.class);
assertThat(updateResult.getModifiedCount())
  .isEqualTo(1);
```

We create a *Query* object to find the document we want to update and an *Update* object to specify the fields and values to update. Using both of these, **we call the *updateFirst()* method, which updates the first document that matches the query**.

We verify the number of modified documents using the *getModifiedCount()* method of the returned *UpdateResult*.

Alternatively, to update multiple documents that match the query:

```
int lowArticleCount = 10;
int authorCount = 20;
for (int i = 0; i < 20; i++) {
    Author lowArticleCountAuthor = Instancio.of(Author.class)
      .set(field(Author::isActive), true)
      .generate(field(Author::getArticleCount), gen ->
gen.ints().max(lowArticleCount))
      .create();
    mongoTemplate.save(lowArticleCountAuthor);
}

Query query = new
Query(Criteria.where("article_count").lte(lowArticleCount));
Update update = new Update();
update.set("active", false);
UpdateResult updateResult = mongoTemplate.updateMulti(query, update,
Author.class);

assertThat(updateResult.getModifiedCount())
  .isEqualTo(authorCount);
```

Here, we create and save *20 Author* documents for authors who've written very few articles. Then, **we use the *updateMulti()* method to update all documents matching the query, setting their *active* field to *false***. We assert that the number of modified documents is equal to the number of authors we'd saved.

Finally, **let's look at an *upsert* operation, which combines the actions of inserting and updating documents** using a defined query:

```
UUID authorId = UUID.randomUUID();
String email = RandomString.make() + "@baeldung.com";
String name = RandomString.make();

Query query = new Query(Criteria.where("email").is(email));
Update update = new Update()
  .set("name", name)
  .setOnInsert("id", authorId)
  .setOnInsert("active", true);

mongoTemplate.upsert(query, update, Author.class);

Author retrievedAuthor = mongoTemplate.findOne(query, Author.class);
assertThat(retrievedAuthor)
  .isNotNull();
```

Here, we create a *query* to find a document by an *email*. In the *update*, we specify the *name* field to be updated. Additionally, we use *setOnInsert()* to set the *id* and *active* fields, which will only be applied if a new document is inserted.

When executing the *upsert()* method, if a document with the specified *email* already exists, only the *name* field will be updated. However, if no document matches the *query*, a new one will be inserted with the provided *id*, *email*, *name*, and *active* fields.

5. Conclusion

In this article, we've explored integrating MongoDB into our Spring Boot application.

We walked through the necessary configurations and started an ephemeral Docker container for MongoDB using Testcontainers, creating a local test environment.

We discussed two options to interact with MongoDB in our application: one where we extend the *MongoRepository* interface and another where we use the *MongoTemplate* class for executing complex queries with fine-grained control.

In our applications, we can make the best of both worlds by creating a service layer that autowires both of the above options.

The code backing this article is available on GitHub. Once you're **logged in as a Baeldung Pro Member (/members/)**, start learning and coding on the project.

COURSES

ALL COURSES (/COURSES/ALL-COURSES)
BAELDUNG ALL ACCESS (/COURSES/ALL-ACCESS)
BAELDUNG ALL TEAM ACCESS (/COURSES/ALL-ACCESS-TEAM)
THE COURSES PLATFORM (HTTPS://WWW.BAELDUNG.COM/MEMBERS/ALL-COURSES)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
LEARN SPRING BOOT SERIES (/SPRING-BOOT)
SPRING TUTORIAL (/SPRING-TUTORIAL)
GET STARTED WITH JAVA (/GET-STARTED-WITH-JAVA-SERIES)
ALL ABOUT STRING IN JAVA (/JAVA-STRING)
SECURITY WITH SPRING (/SECURITY-SPRING)
JAVA COLLECTIONS (/JAVA-COLLECTIONS)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS/)
PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)
EBOOKS (/LIBRARY/)
FAQ (/LIBRARY/FAQ)