MongoTemplate

jim stafford

Fall 2024 v2022-07-24: Built: 2024-11-19 21:36 EST

Table of Contents

Chapter 1. Introduction

There are at least three (3) different APIs for interacting with MongoDB using Java — the last two from Spring are closely related.

MongoClient

is the core API from Mongo.

MongoOperations (interface)/MongoTemplate (implementation class)

is a command-based API around MongoClient from Spring and integrated into Spring Boot

Spring Data MongoDB Repository

is a repository-based API from Spring Data that is consistent with Spring Data JPA

This lecture covers implementing interactions with MongoDB using the MongoOperations API, implemented using MongoTemplate. Even if one intends to use the repository-based API, the MongoOperations API will still be necessary to implement various edge cases — like individual field changes versus whole document replacements.

1.1. Goals

The student will learn:

- to set up a MongoDB Maven project with references to embedded test and independent development and operational instances
- to map a POJO class to a MongoDB collection
- to implement MongoDB commands using a Spring command-level MongoOperations/MongoTemplate Java API

1.2. Objectives

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

1. declare project dependencies required for using Spring's MongoOperations/MongoTemplate API
2. implement basic unit testing using a (seemingly) embedded MongoDB
3. define a connection to a MongoDB
4. switch between the embedded test MongoDB and stand-alone MongoDB for interactive development inspection
5. define a `@Document` class to map to MongoDB collection
6. inject a MongoOperations/MongoTemplate instance to perform actions on a database
7. perform whole-document CRUD operations on a `@Document` class using the Java API
8. perform surgical field operations using the Java API
9. perform queries with paging properties

10. perform Aggregation pipeline operations using the Java API

Chapter 2. MongoDB Project

Except for the possibility of indexes and defining specialized collection features — there is much less schema rigor required to bootstrap a MongoDB project or collection before using. Our primary tasks will be to

- declare a few, required dependencies
- setup project for integration testing with an embedded MongoDB instance to be able to run tests with zero administration
- conveniently switch between an embedded and stand-alone MongoDB instance to be able to inspect the database using the MongoDB shell during development

2.1. MongoDB Project Dependencies

The following snippet shows a dependency declaration for MongoDB APIs.

MongoDB Project Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId> ①
</dependency>
```

① brings in all dependencies required to access the database using Spring Data MongoDB

That dependency primarily brings in dependencies that are general to Spring Data and specific to MongoDB.

MongoDB Starter Dependencies

```
[INFO] +- org.springframework.boot:spring-boot-starter-data-mongodb:jar:3.3.2:compile
[INFO] |  +- org.mongodb:mongodb-driver-sync:jar:5.0.1:compile
[INFO] |  |  +- org.mongodb:bson:jar:5.0.1:compile
[INFO] |  |  \- org.mongodb:mongodb-driver-core:jar:5.0.1:compile
[INFO] |  |     \- org.mongodb:bson-record-codec:jar:5.0.1:runtime
[INFO] |  \- org.springframework.data:spring-data-mongodb:jar:4.3.2:compile
[INFO] |     +- org.springframework:spring-tx:jar:6.1.11:compile
[INFO] |     \- org.springframework.data:spring-data-commons:jar:3.3.2:compile
```

That is enough to cover integration with an external MongoDB during operational end-to-end scenarios. Next, we need to address the integration test environment.

2.2. MongoDB Project Integration Testing Options

MongoDB, like Postgres, is not written in Java (MongoDB is written in C++). That means that we cannot simply instantiate MongoDB within our integration test JVM. We have at least four options:

- Mocks
- Fongo in-memory MongoDB implementation (dormant)
- Flapdoodle embedded MongoDB (alive) and Auto Configuration or referenced Maven plugins (dormant):
  - maven-mongodb-plugin
  - embedmongo-maven-plugin
- Testcontainers Docker wrapper (active)

Each should be able to do the job for what we want to do here. However,

- With nothing else in place, Mocks would be an option for trival cases. However, not an option to test out database client code.
- Although Fongo is an in-memory solution, it is not MongoDB and edge cases may not work the same as a real MongoDB instance. Most of the work on this option was developed in 2014, slowed by 2018, and the last commit to master was ~2020.
- Flapdoodle calls itself "embedded". However, the term embedded is meant to mean "within the scope of the test" and not "within the process itself". The download and management of the server is what is embedded. The Spring Boot 2.7 Documentation discusses Flapdoodle, and the Spring Boot 2.7 Embedded Mongo AutoConfiguration seamlessly integrates Flapdoodle with a few options. However, Spring Boot dropped direct support for Flapdoodle in 3.0. Flapdoodle is still actively maintained and can still be used with Spring Boot 3 (as I will show in the examples). Anything Spring Boot no longer supplies for the integration (e.g., AutoConfiguration, ConfigurationProperties) has now moved over to Flapdoodle.

  Full control of the configuration can be performed using the referenced Maven plugins or writing your own `@Configuration` beans that invoke the Flapdoodle API directly.

- Testcontainers provides full control over the versions and configuration of MongoDB instances using Docker. The following article points out some drawback to using Flapdoodle and how leveraging Testcontainers solved their issues. [1] Spring Boot dropped their direct support for Flapdoodle in version 3 in favor of using Testcontainers. I showed how to easily integrate Testcontainers into an integration test earlier in the lectures. This legacy example will continue to use Flapdoodle to show the still viable legacy option.

2.3. Flapdoodle Test Dependencies

This lecture will use the Flapdoodle Embedded Mongo Spring 3.x setup. The following Maven dependency will bring in Flapdoodle libraries and trigger the Spring Boot Embedded MongoDB Auto Configuration

Flapdoodle Test Dependencies

```
<dependency>
    <groupId>de.flapdoodle.embed</groupId>
    <artifactId>de.flapdoodle.embed.mongo.spring30x</artifactId>
    <scope>test</scope>
```

```
</dependency>
```

```
[INFO] +- de.flapdoodle.embed:de.flapdoodle.embed.mongo.spring30x:jar:4.11.0:test
[INFO] |  \- de.flapdoodle.embed:de.flapdoodle.embed.mongo:jar:4.11.1:test
...
```

A test instance of MongoDB is downloaded and managed through a test library called Flapdoodle Embedded Mongo. It is called "embedded", but unlike H2 and other embedded Java RDBMS implementations — the only thing embedded about this capability is the logical management feel. Technically, the library downloads a MongoDB instance (cached), starts, and stops the instance as part of running the test. A Flapdoodle AutoConfigure will activate Flapdoodle when running a unit integration test, and it detects the library on the classpath. We can bypass the use of Flapdoodle and use an externally managed MongoDB instance by turning off the Flapdoodle starter.

Same AutoConfigure, Different Artifact/Source

The Flapdoodle AutoConfiguration used to be included within Spring Boot (⇐ 2.7) and has now moved to Flapdoodle. It is basically the same class/purpose. It just now requires a direct dependency on a Flapdoodle artifact versus getting it through `spring-boot-starter-mongodb`.

2.4. Flapdoodle Properties

You must set the `de.flapdoodle.mongodb.embedded.version` property to a supported version in your test properties.

Required Version Property

```
de.flapdoodle.mongodb.embedded.version=4.4.0
```

Otherwise, you will likely get the following error when trying to start tests.

Missing Required Property Error

```
IllegalStateException: Set the de.flapdoodle.mongodb.embedded.version property or
define your own IFeatureAwareVersion bean to use embedded MongoDB
```

2.5. MongoDB Access Objects

There are two primary beans of interest when we connect and interact with MongoDB: MongoClient and MongoOperations/MongoTemplate.

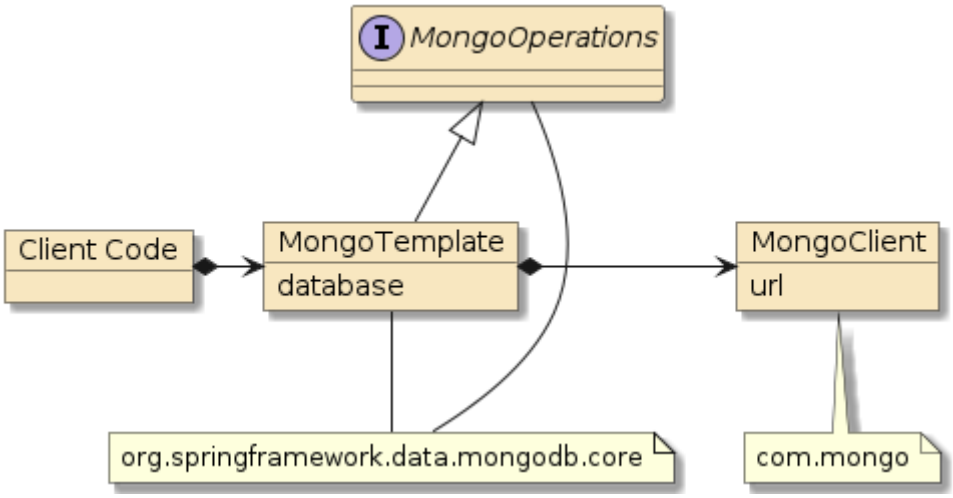

Figure 1. Injectable MongoDB Access Objects

- MongoClient is a client provided by Mongo that provides the direct connection to the database and mimics the behavior of the Mongo Shell using Java. AutoConfiguration will automatically instantiate this, but can be customized using `MongoClients` factory class.
- MongoOperations is an interface provided by Spring Data that defines a type-mapped way to use the client
- MongoTemplate is the implementation class for MongoOperations — also provided by Spring Data. AutoConfiguration will automatically instantiate this using the `MongoClient` and a specific database name.

Embedded MongoDB Auto Configuration Instantiates MongoClient to Reference Flapdoodle Instance

By default, the Embedded MongoDB Auto Configuration class will instantiate a MongoDB instance using Flapdoodle and instantiate a MongoClient that references that instance. The Embedded MongoDB Auto Configuration class must be explicitly disabled if you wish to use a different MongoDB target. This will be demonstrated in this lecture.

2.6. MongoDB Connection Properties

To communicate with an explicit MongoDB server, we need to supply various properties or combine them into a single `spring.data.mongodb.uri`

The following example property file lists the individual properties commented out and the combined properties expressed as a URL. These will be used to automatically instantiate an injectable `MongoClient` and `MongoTemplate` instance to a remote (non-Flapdoodle) instance.

application-mongodb.properties

```
#spring.data.mongodb.host=localhost
#spring.data.mongodb.port=27017
#spring.data.mongodb.database=test
#spring.data.mongodb.authentication-database=admin
#spring.data.mongodb.username=admin
```

```
#spring.data.mongodb.password=secret
spring.data.mongodb.uri=mongodb://admin:secret@localhost:27017/test?authSource=admin
```

2.7. Injecting MongoTemplate

The MongoDB starter takes care of declaring key `MongoClient` and `MongoTemplate` `@Bean` instances that can be injected into components. Generally, injection of the `MongoClient` will not be necessary.

MongoTemplate Class Injection

```
@Autowired
private MongoTemplate mongoTemplate; ①
```

① `MongoTemplate` defines a starting point to interface to MongoDB in a Spring application

Alternatively, we can inject using the interface of MongoTemplate.

Alternate Interface Injection

```
@Autowired
private MongoOperations mongoOps; ①
```

① `MongoOperations` is the interface for `MongoTemplate`

2.8. Disabling Embedded MongoDB

By default, our tests get configured to use the Embedded MongoDB, Flapdoodle test instance. For development, we may want to work against a live MongoDB instance so we can interactively inspect the database using the Mongo shell. The only way to prevent using Embedded MongoDB during testing — is to disable the starter.

The following snippet shows the command-line system property that will disable `EmbeddedMongoAutoConfiguration` from activating. That will leave only the standard `MongoAutoConfiguration` to execute and setup MongoClient using `spring.data.mongodb` properties.

Disable Embedded Mongo using Command Line System Property

```
-Dspring.autoconfigure.exclude=\
de.flapdoodle.embed.mongo.spring.autoconfigure.EmbeddedMongoAutoConfiguration
```

To make things simpler, I added a conditional `@Configuration` class that would automatically trigger the exclusion of the `EmbeddedMongoAutoConfiguration` when the `spring.data.mongodb.uri` was present.

Disable Embedded Mongo using Conditional @Configuration Class

```
import de.flapdoodle.embed.mongo.spring.autoconfigure.EmbeddedMongoAutoConfiguration;
...
@Configuration(proxyBeanMethods = false)
```

```
@ConditionalOnProperty(prefix="spring.data.mongodb",name="uri",matchIfMissing=false)①
@EnableAutoConfiguration(exclude = EmbeddedMongoAutoConfiguration.class) ②
public class DisableEmbeddedMongoConfiguration { }
```

① class is activated on the condition that property `spring.data.mongodb.uri` be present

② when activated, class definition will disable `EmbeddedMongoAutoConfiguration`

2.9. @ActiveProfiles

With the ability to turn on/off the EmbeddedMongo and MongoDB configurations, it would be nice to make this work seamlessly with profiles. We know that we can define an `@ActiveProfiles` for integration tests to use, but this is very static. It cannot be changed during normal build time using a command-line option.

Static Active Profiles Declaration

```
@SpringBootTest(classes= NTestConfiguration.class) ①
@ActiveProfiles(profiles="mongodb") ②
public class MongoOpsBooksNTest {
```

① defines various injectable instances for testing

② statically defines which profile will be currently active

To make this more flexible at runtime (from the Maven/Surefire command line), we can take advantage of the `resolver` option of `@ActiveProfiles`. Anything we list in `profiles` is the default. Anything that is returned from the `resolver` is what is used instead. The `resolver` is an instance of `ActiveProfilesResolver`.

Dynamic Active Profile Determination

```
@SpringBootTest(classes= NTestConfiguration.class)
@ActiveProfiles(profiles={ ... }, resolver = ...class) ① ②
public class MongoOpsBooksNTest {
```

① `profiles` list the default profile(s) to use

② `resolver` implements `ActiveProfilesResolver` and determines what profiles to use at runtime

2.10. TestProfileResolver

I implemented a simple class based on an example from the internet from Amit Kumar. [2] The class will inspect the `spring.profiles.active` if present and return an array of strings containing those profiles. If the property does not exist, then the default options of the test class are used.

Manual Specification of Active Profiles

```
-Dspring.profiles.active=mongodb,foo,bar
```

The following snippet shows how that is performed.

@ActiveProfile resolver class

```
import org.springframework.test.context.ActiveProfilesResolver;
import org.springframework.test.context.support.DefaultActiveProfilesResolver;
...
//Ref: https://www.allprogrammingtutorials.com/tutorials/overriding-active-profile-
boot-integration-tests.php
public class TestProfileResolver implements ActiveProfilesResolver {
    private final String PROFILE_KEY = "spring.profiles.active";
    private final ActiveProfilesResolver defaultResolver = new
DefaultActiveProfilesResolver();

    @Override
    public String[] resolve(Class<?> testClass) {
        return System.getProperties().containsKey(PROFILE_KEY) ?
                //return profiles expressed in property as array of strings
                System.getProperty(PROFILE_KEY).split("\\s*,\\s*") : ①
                //return profile(s) expressed in the class' annotation
                defaultResolver.resolve(testClass);
    }
}
```

① regexp splits string at the comma (',') character and an unlimited number of contiguous whitespace characters on either side

The `resolver` can get the expressed defaults from an instance of the `DefaultActiveProfilesResolver` class, which uses reflection to search for the `@ActiveProfile` annotation on the provided `testClass`.

2.11. Using TestProfileResolver

The following snippet shows how `TestProfileResolver` can be used by an integration test.

- The test uses no profile by default — activating Embedded MongoDB.
- If the `mongodb` profile is specified using a system property or temporarily inserted into the source — then that profile will be used.
- Since my mongodb profile declares `spring.data.mongodb.uri`, Embedded MongoDB is deactivated.

Example Use of TestProfileResolver

```
@SpringBootTest(classes= NTestConfiguration.class) ①
@ActiveProfiles(resolver = TestProfileResolver.class) ②
//@ActiveProfiles(profiles="mongodb", resolver = TestProfileResolver.class) ③
public class MongoOpsBooksNTest {
```

① defines various injectable instances for testing

② defines which profile will be currently active

③ defines which profile will be currently active, with `mongodb` being the default profile

2.12. Inject MongoTemplate

In case you got a bit lost in that testing detour, we are now at a point where we can begin interacting with our chosen MongoDB instance using an injected `MongoOperations` (the interface) or `MongoTemplate` (the implementation class).

Inject MongoTemplate

```
@AutoConfigure
private MongoTemplate mongoTemplate;
```

I wanted to show you how to use the running MongoDB when we write the integration tests using `MongoTemplate` so that you can inspect the live DB instance with the Mongo shell while the database is undergoing changes. Refer to the previous MongoDB lecture for information on how to connect the DB with the Mongo shell.

2.13. Testcontainers

I did not include coverage of Testcontainers/Mongo in this lecture. This is because I felt the setup and connection topic was covered good enough during the Testcontainers lecture , wanted to keep a demonstration of Flapdoodle, and to prevent topic/dependency overload. Testcontainers specifically replaces the Embedded MongoDB, Flapdoodle support and is just another way to launch the external MongoDB Docker container used within this lecture example.

[1] *"Fast and stable MongoDB-based tests in Spring"*, Piotr Kubowicz, Dec 2020

[2] *"Overriding Active Profiles in Spring Boot Integration Tests"*, Amit Kumar, 2018

Chapter 3. Example POJO

We will be using an example `Book` class to demonstrate some database mapping and interaction concepts. The class properties happen to be mutable, and the class provides an all-arg constructor to support a builder and adds `with()` modifiers to be able to chain modifications (using new instances). These are not specific requirements of Spring Data Mongo. Spring Data Mongo is designed to work with many different POJO designs.

MongoTemplate Works with Classes lacking No-Arg Constructor

This example happens to have no no-arg constructor. The default no-arg constructor is disabled by the all-arg constructor declared by both `@Builder` (package-friendly access) and `@AllArgsConstructor` (public access).

Book POJO being mapped to database

```
package info.ejava.examples.db.mongo.books.bo;
...
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;

@Document(collection = "books")
@Getter
@Setter
@Builder
@With
@AllArgsConstructor
public class Book {
    @Id @Setter(AccessLevel.NONE)
    private String id;
    @Field(name="title")
    private String title;
    private String author;
    private LocalDate published;
}
```

3.1. Property Mapping

3.1.1. Collection Mapping

Spring Data Mongo will map instances of the class to a collection

- by the same name as the class (e.g., `book`, by default)
- by the `collection` name supplied in the `@Document` annotation

Collection Mapping

```
@Document(collection = "books") ②
public class Book { ①
```

① instances are, by default, mapped to the "book" collection

② `@Documentation.collection` annotation property overrides default collection name

MongoTemplate also provides the ability to independently provide the collection name during the command — which makes the class mapping even less important.

3.1.2. Primary Key Mapping

The MongoDB `_id` field will be mapped to a field that either

- is called `id`
- is annotated with `@Id` (`@org.springframework.data.annotation.Id`; this is not the same as `@jakarta.persistence.Id` used with JPA)
- is mapped to field `_id` using `@Field` annotation

Primary Key Mapping

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

    @Id ①
    private String id; ① ②
```

① property is both named `id` and annotated with `@Id` to map to `_id` field

② `String` id type can be mapped to auto-generated MongoDB `_id` field

Only `_id` fields mapped to `String`, `BigInteger`, or `ObjectId` can have auto-generated `_id` fields mapped to them.

3.2. Field Mapping

Class properties will be mapped by default to a field of the same name. The `@Field` annotation can be used to customize that behavior.

Field Mapping

```
import org.springframework.data.mongodb.core.mapping.Field;

    @Field(name="title") ①
    private String titleXYZ;
```

① maps Java property `titleXYZ` to MongoDB document field `title`

We can annotate a property with `@Transient` to prevent a property from being stored in the

database.

Transient Mapping

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

    @Transient ①
    private String dontStoreMe;
```

① `@Transient` excludes the Java property from being mapped to the database

3.3. Instantiation

Spring Data Mongo leverages constructors in the following order

1. No argument constructor
2. Multiple argument constructor annotated with `@PersistenceConstructor`
3. Solo, multiple argument constructor (preferably an all-args constructor)

Given our example, the all-args constructor will be used.

3.4. Property Population

For properties not yet set by the constructor, Spring Data Mongo will set fields using the following order

1. use `setter()` if supplied
2. use `with()` if supplied, to construct a copy with the new value
3. directly modify the field using reflection

Chapter 4. Command Types

MongoTemplate offers different types of command interactions

Whole Document
: complete document passed in as argument and/or returned as a result

By Id
: command performed on document matching provided ID

Filter
: command performed on documents matching filter

Field Modifications
: command makes field level changes to database documents

Paging
: options to finder commands to limit results returned

Aggregation Pipeline
: sequential array of commands to be performed on the database

These are not the only categories of commands you could come up with describing the massive set, but it will be enough to work with for a while. Inpect the MongoTemplate Javadoc for more options and detail.

Chapter 5. Whole Document Operations

The `MongoTemplate` instance already contains a reference to a specific database and the `@Document` annotation of the POJO has the collection name — so the commands know exactly which collection to work with. Commands also offer options to express the collection as a string at command-time to add flexibility to mapping approaches.

5.1. insert()

`MongoTemplate` offers an explicit `insert()` that will always attempt to insert a new document without checking if the ID already exists. If the created document has a generated ID not yet assigned — then this should always successfully add a new document.

One thing to note about class mapping is that `MongoTemplate` adds an additional field to the document during insert. This field is added to support polymorphic instantiation of result classes.

MongoTemplate _class Field

```
{ "_id" : ObjectId("608b3021bd49095dd4994c9d"),
  "title" : "Vile Bodies",
  "author" : "Ernesto Rodriguez",
  "published" : ISODate("2015-03-10T04:00:00Z"),
  "_class" : "info.ejava.examples.db.mongo.books.bo.Book" } ①
```

① MongoTemplate adds extra `_class` field to help dynamically instantiate query results

This behavior can be turned off by configuring your own instance of MongoTemplate and following the example from Mkyong.com in this link.

5.1.1. insert() Successful

The following snippet shows an example of a transient book instance being successfully inserted into the database collection using the `insert` command.

MongoTemplate insert() Successful

```
//given an entity instance
Book book = ...
//when persisting
mongoTemplate.insert(book); ① ②
//then documented is persisted
then(book.getId()).isNotNull();
then(mongoTemplate.findById(book.getId(), Book.class)).isNotNull();
```

① transient document assigned an ID and inserted into database collection

② database referenced by `MongoTemplate` and collection identified in Book `@Document.collection` annotation

5.1.2. insert() Duplicate Fail

If the created document is given an assigned ID value, then the call will fail with a `DuplicateKeyException` exception if the ID already exists.

MongoTemplate create() with Duplicate Key Throws Exception

```
import org.springframework.dao.DuplicateKeyException;
...
//given a persisted instance
Book book = ...
mongoTemplate.insert(book);
//when persisting an instance by the same ID
Assertions.assertThrows(DuplicateKeyException.class,
        ()->mongoTemplate.insert(book)); ①
```

① document with ID matching database ID cannot be inserted

5.2. save()/Upsert

The `save()` command is an "upsert" (Update or Insert) command and likely the simplest form of "upsert" provided by MongoTemplate (there are more). It can be used to insert a document if new or replace if already exists - based only on the evaluation of the ID.

5.2.1. Save New

The following snippet shows a new transient document being saved to the database collection. We know that it is new because the ID is unassigned and generated at `save()` time.

Upsert Example - Save New

```
//given a document not yet saved to DB
Book transientBook = ...
assertThat(transientBook.getId()).isNull();
//when - updating
mongoTemplate.save(transientBook);
//then - db has new state
then(transientBook.getId()).isNotNull();
Book dbBook = mongoTemplate.findById(transientBook.getId());
then(dbBook.getTitle()).isEqualTo(transientBook.getTitle());
then(dbBook.getAuthor()).isEqualTo(transientBook.getAuthor());
then(dbBook.getPublished()).isEqualTo(transientBook.getPublished());
```

5.2.2. Replace Existing

The following snippet shows a new document instance with the same ID as a document in the database, but with different values. In this case, `save()` performs an update/(whole document replacement).

UpsertExample - Replace Existing

```
//given a persisted instance
Book originalBook = ...
mongoTemplate.insert(originalBook);
Book updatedBook = mapper.map(dtoFactory.make()).withId(originalBook.getId());
assertThat(updatedBook.getTitle()).isNotEqualTo(originalBook.getTitle());
//when - updating
mongoTemplate.save(updatedBook);
//then - db has new state
Book dbBook = mongoTemplate.findById(book.getId(), Book.class);
then(dbBook.getTitle()).isEqualTo(updatedBook.getTitle());
then(dbBook.getAuthor()).isEqualTo(updatedBook.getAuthor());
then(dbBook.getPublished()).isEqualTo(updatedBook.getPublished());
```

5.3. remove()

`remove()` is another command that accepts a document as its primary input. It returns some metrics about what was found and removed.

The snippet below shows the successful removal of an existing document. The `DeleteResult` response document provides feedback of what occurred.

Successful Remove Example

```
//given a persisted instance
Book book = ...
mongoTemplate.save(book);
//when - deleting
DeleteResult result = mongoTemplate.remove(book);
long count = result.getDeletedCount();
//then - no longer in DB
then(count).isEqualTo(1);
then(mongoTemplate.findById(book.getId(), Book.class)).isNotNull();
```

Chapter 6. Operations By ID

There are very few commands that operate on an explicit ID. `findById` is the only example. I wanted to highlight the fact that most commands use a flexible query filter, and we will show examples of that in after introducing byId.

6.1. findById()

`findById()` will return the complete document associated with the supplied ID.

The following snippet shows an example of the document being found.

findById() Found Example

```
//given a persisted instance
Book book = ...
//when finding
Book dbBook = mongoTemplate.findById(book.getId(), Book.class); ①
//then document is found
then(dbBook.getId()).isEqualTo(book.getId());
then(dbBook.getTitle()).isEqualTo(book.getTitle());
then(dbBook.getAuthor()).isEqualTo(book.getAuthor());
then(dbBook.getPublished()).isEqualTo(book.getPublished());
```

① `Book` class is supplied to identify the collection and the type of response object to populate

A missing document does not throw an exception — just returns a null object.

findById() Not Found Example

```
//given a persisted instance
String missingId = "12345";
//when finding
Book dbBook = mongoTemplate.findById(missingId, Book.class);
//then
then(dbBook).isNull();
```

Chapter 7. Operations By Query Filter

Many commands accept a `Query` object used to filter which documents in the collection the command applies to. The `Query` can express:

- criteria
- targeted types
- paging

We will stick to just simple the criteria here.

Example Criteria syntax

```
Criteria filter = Criteria.where("field1").is("value1")
                          .and("field2").not().is("value2");
```

If we specify the collection name (e.g., "books") in the command versus the type (e.g., `Book` class), we lack the field/type mapping information. That means we must explicitly name the field and use the type known by the MongoDB collection.

Collection Name versus Mapped Type ID Expressions

```
Query.query(Criteria.where("id").is(id));            //Book.class ①
Query.query(Criteria.where("_id").is(new ObjectId(id))); //"books" ②
```

① can use property values when supplying mapped class in full command

② must supply a field and explicit mapping type when supplying collection name in full command

7.1. exists() By Criteria

`exists()` accepts a `Query` and returns a simple true or false. The query can be as simple or complex as necessary.

The following snippet looks for documents with a matching ID.

exists() By Criteria

```
//given a persisted instance
Book book = ...
mongoTemplate.save(book);
//when testing exists
Query filter = Query.query(Criteria.where("id").is(id));
boolean exists = mongoTemplate.exists(filter,Book.class);
//then document exists
then(exists).isTrue();
```

MongoTemplate was smart enough to translate the "id" property to the `_id` field and the String

value to an `ObjectId` when building the criteria with a mapped class.

MongoTemplate Generated Criteria Document

```
{ "_id" : { "$oid" : "608ae2939f024c640c3b1d4b"}}
```

7.2. delete()

`delete()` is another command that can operate on a criteria filter.

```
//given a persisted instance
Book book = ...
mongoTemplate.save(book);
//when - deleting
Query filter = Query.query(Criteria.where("id").is(id));
DeleteResult result = mongoTemplate.remove(filter, Book.class);
//then - no long in DB
then(count).isEqualTo(1);
then(mongoTemplate.existsById(book.getId())).isFalse();
```

Chapter 8. Field Modification Operations

For cases with large documents — where it would be an unnecessary expense to retrieve the entire document and then to write it back with changes — MongoTemplate can issue individual field commands. This is also useful in concurrent modifications where one wants to upsert a document (and have only a single instance) but also update an existing document with fresh information (e.g., increment a counter, set a processing timestamp)

8.1. update() Field(s)

The `update()` command can be used to perform actions on individual fields. The following example changes the title of the first document that matches the provided criteria. Update commands can have a minor complexity to include incrementing, renaming, and moving fields — as well as manipulating arrays.

update() Fields Example

```
//given a persisted instance
Book originalBook = ...
mongoTemplate.save(originalBook);
String newTitle = "X" + originalBook.getTitle();
//when - updating
Query filter = Query.query(Criteria.where("_id").is(new ObjectId(id)));①
Update update = new Update(); ②
update.set("title", newTitle); ③
UpdateResult result = mongoTemplate.updateFirst(filter, update, "books"); ④
//{ "_id" : { "$oid" : "60858ca8a3b90c12d3bb15b2"}} ,
//{ "$set" : { "title" : "XTo Sail Beyond the Sunset"}}
long found = result.getMatchedCount();
//then - db has new state
then(found).isEqualTo(1);
Book dbBook = mongoTemplate.findById(originalBook.getId());
then(dbBook.getTitle()).isEqualTo(newTitle);
then(dbBook.getAuthor()).isEqualTo(originalBook.getAuthor());
then(dbBook.getPublished()).isEqualTo(originalBook.getPublished());
```

① identifies a criteria for update

② individual commands to apply to the database document

③ document found will have its `title` changed

④ must use explicit `_id` field and `ObjectId` value when using ("books") collection name versus `Book` class

8.2. upsert() Fields

If the document was not found and we want to be in a state where one will exist with the desired title, we could use an `upsert()` instead of an `update()`.

upsertFields() Example

```
UpdateResult result = mongoTemplate.upsert(filter, update, "books"); ①
```

① `upsert` guarantees us that we will have a document in the books collection with the intended modifications

Chapter 9. Paging

In conjunction with `find` commands, we need to soon look to add paging instructions to sort and slice up the results into page-sized bites. `RestTemplate` offers two primary ways to express paging

- Query configuration
- Pagable command parameter

9.1. skip()/limit()

We can express offset and limit on the `Query` object using `skip()` and `limit()` builder methods.

skip() and limit()

```
Query query = new Query().skip(offset).limit(limit);
```

In the example below, a `findOne()` with `skip()` is performed to locate a single, random document.

Find Random Document

```
private final SecureRandom random = new SecureRandom();
public Optional<Book> random() {
    Optional randomSong = Optional.empty();
    long count = mongoTemplate.count(new Query(), "books");

    if (count!=0) {
        int offset = random.nextInt((int)count);
        Book song = mongoTemplate.findOne(new Query().skip(offset), Book.class); ① ②
        randomSong = song==null ? Optional.empty() : Optional.of(song);
    }
    return randomSong;
}
```

① `skip()` is eliminating offset documents from the results

② `findOne()` is reducing the results to a single (first) document

We could have also expressed the command with `find()` and `limit(1)`.

find() with limit()

```
mongoTemplate.find(new Query().skip(offset).limit(1), Book.class);
```

9.2. Sort

With offset and limit, we often need to express sort — which can get complex. Spring Data defines a `Sort` class that can express a sequence of properties to sort in ascending and/or descending order. That too can be assigned to the `Query` instance.

Sort Example

```
public List<Book> find(List<String> order, int offset, int limit) {
    Query query = new Query();
    query.with( Sort.by(order.toArray(new String[0]))); ①
    query.skip(offset); ②
    query.limit(limit); ③
    return mongoTemplate.find(query, Book.class);
    }
```

① Query accepts a standard `Sort` type to implement ordering

② Query accepts a `skip` to perform an offset into the results

③ Query accepts a `limit` to restrict the number of returned results.

9.3. Pageable

Spring Data provides a `Pageable` type that can express sort, offset, and limit — using Sort, pageSize, and pageNumber. That too can be assigned to the `Query` instance.

```
int pageNo=1;
int pageSize=3;
Pageable pageable = PageRequest.of(pageNo, pageSize,
                            Sort.by(Sort.Direction.DESC, "published"));

public List<Book> find(Pageable pageable) {
    return mongoTemplate.find(new Query().with(pageable), Book.class); ①
}
```

① Query accepts a `Pageable` to permit flexible ordering, offset, and limit

Chapter 10. Aggregation

Most queries can be performed using the database `find()` commands. However, as we have seen in the MongoDB lecture — some complex queries require different stages and command types to handle selections, projections, grouping, etc. For those cases, Mongo provides the Aggregation Pipeline — which can be accessed through the `MongoTemplate`.

The following snippet shows a query that locates all documents that contain an `author` field and match a regular expression.

Example Aggregation Pipeline Call

```
//given
int minLength = ...
Set<String> ids = savedBooks.stream() ... //return IDs of docs matching criteria
String expression = String.format("^.{%d,}$", minLength);
//when pipeline executed
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
);
List<Book> foundSongs = result.getMappedResults();
//then expected IDs found
Set<String> foundIds = foundSongs.stream()
    .map(s->s.getId()).collect(Collectors.toSet());
then(foundIds).isEqualTo(ids);
```

Mongo BasicDocument Issue with $exists Command

Aggregation Pipeline was forced to be used in this case, because a normal collection `find()` command was not able to accept an `exists` command with another command for that same field.

```
Criteria.where("author").regex(expression).and("author").exists(true))
```

```
org.springframework.data.mongodb.InvalidMongoDbApiUsageException: Due
to limitations of the com.mongodb.BasicDocument, you can't add a second
'author' expression specified as 'author : Document{{$exists=true}}'.
Criteria already contains 'author : ^.{22,}$'.
```

This provides a good example of how to divide up the commands into independent queries using Aggregation Pipeline.

Chapter 11. ACID Transactions

Before we leave the accessing MongoDB through the `MongoTemplate` Java API topic, I wanted to lightly cover ACID transactions.

- Atomicity
- Consistency
- Isolation
- Durability

11.1. Atomicity

MongoDB has made a lot of great strides in scale and performance by providing flexible document structures. Individual caller commands to change a document represent separate, atomic transactions. Documents can be as large or small as one desires and should take document atomicity into account when forming document schema.

However, as of MongoDB 4.0, MongoDB supports multi-document atomic transactions if absolutely necessary. The following online resource provides some background on how to achieve this. [1]

MongoDB Multi-Document Transactions and not the Normal Path

Just because you can implement multi-document atomic transactions and references between documents, don't default to a RDBMS mindset when designing document schema. Try to make a single document represent state that is essential to be in a consistent state.

MongoDB documentation does warn against its use. So multi-document acid transactions should not be a first choice.

11.2. Consistency

Since MongoDB does not support a fixed schema or enforcement of foreign references between documents, there is very little for the database to keep consistent. The primary consistency rules the database must enforce are any unique indexes — requiring that specific fields be unique within the collection.

11.3. Isolation

Within the context of a single document change — MongoDB [2]

- will always prevent a reader from seeing partial changes to a document.
- will provide a reader a complete version of a document that may have been inserted/updated after a `find()` was initiated but before it was returned to the caller (i.e., can receive a document that no longer matches the original query)
- may miss including documents that satisfy a query after the query was initiated but before the

results are returned to the caller

11.4. Durability

The durability of a MongoDB transaction is a function of the number of nodes within a cluster that acknowledge a change before returning the call to the client. `UNACKNOWLEDGED` is fast but extremely unreliable. Other ranges, including `MAJORITY` at least guarantee that one or more nodes in the cluster have written the change. These are expressed using the MongoDB WriteConcern class.

MongoTemplate allows us to set the WriteConcern for follow-on `MongoTemplate` commands.

Durability is a more advanced topic and requires coverage of system administration and cluster setup — which is well beyond the scope of this lecture. My point of bringing this and other ACID topics up here is to only point out that the `MongoTemplate` offers access to these additional features.

[1] *"Spring Data MongoDB Transactions"*, baeldung, 2020

[2] *"Read Isolation, Consistency, and Recency"*, MongoDB Manual, Version 4.4

Chapter 12. Summary

In this module, we learned to:

- setup a MongoDB Maven project
- inject a MongoOperations/MongoTemplate instance to perform actions on a database
- instantiate a (seemingly) embedded MongoDB connection for integration tests
- instantiate a stand-alone MongoDB connection for interactive development and production deployment
- switch between the embedded test MongoDB and stand-alone MongoDB for interactive development inspection
- map a `@Document` class to a MongoDB collection
- implement MongoDB commands using a Spring command-level MongoOperations/MongoTemplate Java API
- perform whole-document CRUD operations on a `@Document` class using the Java API
- perform surgical field operations using the Java API
- perform queries with paging properties
- perform Aggregation pipeline operations using the Java API