(/)

Introduction to Spring Data MongoDB

Last modified: November 6, 2018

by baeldung (https://www.baeldung.com/author/baeldung/)

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

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

If you have a few years of experience in the Java ecosystem, and you're interested in sharing that experience with the community (and getting paid for your work of course), have a look at the "Write for Us" page (/contribution-guidelines). Cheers. Eugen

1. Overview

This article will be a quick and practical **introduction to Spring Data MongoDB**.

We'll go over the basics using both the *MongoTemplate* as well as *MongoRepository* using practical tests to illustrate each operation.

2. *MongoTemplate and MongoRepository*

The *MongoTemplate* follows the standard template pattern in Spring and provides a ready to go, basic API to the underlying persistence engine.

The repository follows the Spring Data-centric approach and comes with more flexible and complex API operations, based on the well-known access patterns in all Spring Data projects.

For both, we need to start by defining the dependency – for example, in the *pom.xml*, with Maven:

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>2.1.0.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-releasetrain</artifactId>
    <version>Lovelace-M3</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

Note that we need to add the milestone repository to our *pom.xml* as well:

```
<repositories>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone (https://repo.spring.io/milestone)</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
```

To check if any new version of the library has been released – track the releases here (https://search.maven.org/search?q=g:org.springframework.data%20AND%20a:spring-data-mongodb).

3. Configuration for *MongoTemplate*

3.1. XML Configuration

Let's start with the simple XML configuration for the Mongo template:

```
<mongo:mongo-client id="mongoClient" host="localhost" />
<mongo:db-factory id="mongoDbFactory" dbname="test" mongo-ref="mongoClient" />
```

First, we need to define the factory bean responsible for creating Mongo instances.

Next – we need to actually define (and configure) the template bean:

```
<bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg ref="mongoDbFactory"/>
</bean>
```

And finally we need to define a post processor to translate any *MongoExceptions* thrown in *@Repository* annotated classes:

```
<bean class=
  "org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>
```

3.2. Java Configuration

Let's now create a similar configuration using Java config by extending the base class for MongoDB configuration *AbstractMongoConfiguration*:

```
@Configuration
public class MongoConfig extends AbstractMongoConfiguration {

    @Override
    protected String getDatabaseName() {
        return "test";
    }

    @Override
    public MongoClient mongoClient() {
        return new MongoClient("127.0.0.1", 27017);
    }

    @Override
    protected String getMappingBasePackage() {
        return "org.baeldung";
    }
}
```

Note: We didn't need to define *MongoTemplate* bean in the previous configuration as it's already defined in *AbstractMongoConfiguration*

We can also use our configuration from scratch without extending *AbstractMongoConfiguration* – as follows:

```
@Configuration
public class SimpleMongoConfig {

    @Bean
    public MongoClient mongo() {
        return new MongoClient("localhost");
    }

    @Bean
    public MongoTemplate mongoTemplate() throws Exception {
        return new MongoTemplate(mongo(), "test");
    }
}
```

4. Configuration for *MongoRepository*

4.1. XML Configuration

To make use of custom repositories (extending the *MongoRepository*) – we need to continue the configuration from section 3.1 and set up the repositories:

```
<mongo:repositories
  base-package="org.baeldung.repository" mongo-template-ref="mongoTemplate"/>
```

4.2. Java Configuration

Similarly, we'll build on the configuration we already created in section 3.2 and add a new annotation into the mix:

```
@EnableMongoRepositories(basePackages = "org.baeldung.repository")
```

4.3. Create the Repository

Now, after the configuration, we need to create a repository – extending the existing *MongoRepository* interface:

```
public interface UserRepository extends MongoRepository<User, String> {
    //
}
```

Now we can auto-wire this *UserRepository* and use operations from *MongoRepository* or add custom operations.

5. Using *MongoTemplate*

5.1. *Insert*

Let's start with the insert operation; let's also start with a empty database:

```
{
}
```

Now if we insert a new user:

```
User user = new User();
user.setName("Jon");
mongoTemplate.insert(user, "user");
```

The database will look like this:

```
{
    "_id" : ObjectId("55b4fda5830b550a8c2ca25a"),
    "_class" : "org.baeldung.model.User",
    "name" : "Jon"
}
```

5.2. *Save – Insert*

The *save* operation has save-or-update semantics: if an id is present, it performs an update, if not – it does an insert.

Let's look at the first semantic – the insert; here's the initial state of the database*:*

```
{
}
```

When we now *save* a new user:

```
User user = new User();
user.setName("Albert");
mongoTemplate.save(user, "user");
```

The entity will be inserted in the database:

```
{
    "_id" : ObjectId("55b52bb7830b8c9b544b6ad5"),
    "_class" : "org.baeldung.model.User",
    "name" : "Albert"
}
```

Next, we'll look at the same operation – *save* – with update semantics.

5.3. *Save – Update*

Let's now look at *save* with update semantics, operating on an existing entity:

```
{
    "_id" : ObjectId("55b52bb7830b8c9b544b6ad5"),
    "_class" : "org.baeldung.model.User",
    "name" : "Jack"
}
```

Now, when we *save* the existing user – we will update it:

```
user = mongoTemplate.findOne(
  Query.query(Criteria.where("name").is("Jack")), User.class);
user.setName("Jim");
mongoTemplate.save(user, "user");
```

The database will look like this:

```
{
    "_id" : ObjectId("55b52bb7830b8c9b544b6ad5"),
    "_class" : "org.baeldung.model.User",
    "name" : "Jim"
}
```

As you can see, in this particular example, *save* uses the semantics of *update*, because we use an object with given *_id*.

5.4. *UpdateFirst*

updateFirst updates the very first document that matches the query.

Let's start with the initial state of the database:

```
[
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
        "_class" : "org.baeldung.model.User",
        "name" : "Alex"
    },
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614c"),
        "_class" : "org.baeldung.model.User",
        "name" : "Alex"
    }
]
```

When we now run the *updateFirst*:

```
Query query = new Query();
query.addCriteria(Criteria.where("name").is("Alex"));
Update update = new Update();
update.set("name", "James");
mongoTemplate.updateFirst(query, update, User.class);
```

Only the first entry will be updated:

```
[
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
        "_class" : "org.baeldung.model.User",
        "name" : "James"
    },
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614c"),
        "_class" : "org.baeldung.model.User",
        "name" : "Alex"
    }
]
```

5.5. *UpdateMulti*

UpdateMulti **updates all document that matches the given query**.

First – here's the state of the database before doing the *updateMulti*:

```
[
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
        "_class" : "org.baeldung.model.User",
        "name" : "Eugen"
    },
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614c"),
        "_class" : "org.baeldung.model.User",
        "name" : "Eugen"
    }
]
```

Now, let's now run the *updateMulti* operation:

```
Query query = new Query();
query.addCriteria(Criteria.where("name").is("Eugen"));
Update update = new Update();
update.set("name", "Victor");
mongoTemplate.updateMulti(query, update, User.class);
```

Both existing objects will be updated in the database:

```
[
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
        "_class" : "org.baeldung.model.User",
        "name" : "Victor"
    },
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614c"),
        "_class" : "org.baeldung.model.User",
        "name" : "Victor"
    }
]
```

5.6. *FindAndModify*

This operation works like *updateMulti*, but it **returns the object before it was modified**.

First – the state of the database before calling *findAndModify*:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "org.baeldung.model.User",
    "name" : "Markus"
}
```

Let's look at actual operation code:

```
Query query = new Query();
query.addCriteria(Criteria.where("name").is("Markus"));
Update update = new Update();
update.set("name", "Nick");
User user = mongoTemplate.findAndModify(query, update, User.class);
```

The returned *user object* has the same values as the initial state in the database.

However, the new state in the database is:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "org.baeldung.model.User",
    "name" : "Nick"
}
```

5.7. *Upsert*

The *upsert* works operate on the **find and modify else create semantics**: if the document is matched, update it, else create a new document by combining the query and update object.

Let's start with the initial state of the database:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "org.baeldung.model.User",
    "name" : "Markus"
}
```

Now – let's run the *upsert*:

```
Query query = new Query();
query.addCriteria(Criteria.where("name").is("Markus"));
Update update = new Update();
update.set("name", "Nick");
mongoTemplate.upsert(query, update, User.class);
```

Here's the state of the database after the operation:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "org.baeldung.model.User",
    "name" : "Nick"
}
```

5.8. *Remove*

The state of the database before calling *remove*:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "org.baeldung.model.User",
    "name" : "Benn"
}
```

Let's now run *remove*:

```
mongoTemplate.remove(user, "user");
```

The result will be as expected:

```
{
}
```

6. Using *MongoRepository*

6.1. *Insert*

First – the state of the database before running the *insert:*

```
{
}
```

Now, when we insert a new user:

```
User user = new User();
user.setName("Jon");
userRepository.insert(user);
```

Here's the end state of the database:

```
{
    "_id" : ObjectId("55b4fda5830b550a8c2ca25a"),
    "_class" : "org.baeldung.model.User",
    "name" : "Jon"
}
```

Note how the operation works the same as the *insert* in the *MongoTemplate* API.

6.2. *Save – Insert*

Similarly – *save* works the same as the *save* operation in the *MongoTemplate* API.

Let's start by looking at **the insert semantics** of the operation; here's the initial state of the database:

```
{
}
```

Now – we execute the *save* operation:

```
User user = new User();
user.setName("Aaron");
userRepository.save(user);
```

This results in the user being added to the database:

```
{
    "_id" : ObjectId("55b52bb7830b8c9b544b6ad5"),
    "_class" : "org.baeldung.model.User",
    "name" : "Aaron"
}
```

Note again how, in this example, *save* works with *insert* semantics, because we are inserting a new object.

6.3. *Save – Update*

Let's now look at the same operation but with **update semantics**.

First – here's the state of the database before running the new *save:*

```
{
    "_id" : ObjectId("55b52bb7830b8c9b544b6ad5"),
    "_class" : "org.baeldung.model.User",
    "name" : "Jack"81*6
}
```

Now – we execute the operation:

```
user = mongoTemplate.findOne(
  Query.query(Criteria.where("name").is("Jack")), User.class);
user.setName("Jim");
userRepository.save(user);
```

Finally, here is the state of the database:

```
{
    "_id" : ObjectId("55b52bb7830b8c9b544b6ad5"),
    "_class" : "org.baeldung.model.User",
    "name" : "Jim"
}
```

Note again how, in this example, *save* works with *update* semantics, because we are using an existing object.

6.4. *Delete*

The state of the database before calling *delete*:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "org.baeldung.model.User",
    "name" : "Benn"
}
```

Let's run *delete*:

```
userRepository.delete(user);
```

The result will simply be:

```
{
}
```

6.5. *FindOne*

The state of the database when *findOne* is called:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "org.baeldung.model.User",
    "name" : "Chris"
}
```

Let's now execute the *findOne*:

```
userRepository.findOne(user.getId())
```

The result which will return the existing data:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "org.baeldung.model.User",
    "name" : "Chris"
}
```

6.6. *Exists*

The state of the database before calling *exists*:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "org.baeldung.model.User",
    "name" : "Harris"
}
```

Now, let's run *exists*:

```
boolean isExists = userRepository.exists(user.getId());
```

Which of course will return *true*.

6.7. *FindAll* with *Sort*

The state of the database before calling *findAll*:

```
[
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
        "_class" : "org.baeldung.model.User",
        "name" : "Brendan"
    },
    {
       "_id" : ObjectId("67b5ffa5511fee0e45ed614b"),
       "_class" : "org.baeldung.model.User",
       "name" : "Adam"
    }
]
```

Let's now run *findAll* with *Sort*:

```
List<User> users = userRepository.findAll(new Sort(Sort.Direction.ASC, "name"));
```

The result will be **sorted by name in ascending order**:

```
[
    {
        "_id" : ObjectId("67b5ffa5511fee0e45ed614b"),
        "_class" : "org.baeldung.model.User",
        "name" : "Adam"
    },
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
        "_class" : "org.baeldung.model.User",
        "name" : "Brendan"
    }
]
```

6.8. *FindAll* with *Pageable*

The state of the database before calling *findAll*:

```
[
    {
        "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
        "_class" : "org.baeldung.model.User",
        "name" : "Brendan"
    },
    {
        "_id" : ObjectId("67b5ffa5511fee0e45ed614b"),
        "_class" : "org.baeldung.model.User",
        "name" : "Adam"
    }
]
```

Let's now execute *findAll* with a pagination request:

```
Pageable pageableRequest = PageRequest.of(0, 1);
Page<User> page = userRepository.findAll(pageableRequest);
List<User> users = pages.getContent();
```

The result in *users* list will be only one user:

```
{
    "_id" : ObjectId("55b5ffa5511fee0e45ed614b"),
    "_class" : "org.baeldung.model.User",
    "name" : "Brendan"
}
```

7. Annotations

Finally, let's also go over the simple annotations that Spring Data uses to drive these API operations.

```
@Id
private String id;
```

The field level *@Id* annotation can decorate any type, including *long* and *string*.

If the value of the *@Id* field is not null, it's stored in the database as-is; otherwise, the converter will assume you want to store an *ObjectId* in the database (either *ObjectId, String* or *BigInteger* work).

Next – *@Document*:

```
@Document
public class User {
    //
}
```

This annotation simply **marks a class as being a domain object** that needs to be persisted to the database, along with allowing us to choose the name of the collection to be used.

8. Conclusion

This article was a quick but comprehensive introduction to using MongoDB with Spring Data, both via the *MongoTemplate* API as well as making use of *MongoRepository*.

The implementation of all these examples and code snippets **can be found over on Github** (https://github.com/eugenp/tutorials/tree/master/persistence-modules/spring-data-mongodb) – this is a Maven-based project, so it should be easy to import and run as it is.

▲ newest ▲ oldest ▲ most voted

Guest

Jun Victorio

I need 2 mongo database instead of just one, how to do setup and configure that and how to you tell a spring repository which database/mongotemplate to use and persist?

Thank
Jun Victorio

✚ 0 ━ 3 years ago

Guest

Eugen Paraschiv (http://www.baeldung.com/)

That's an interesting usecase but I'm afraid Spring Data is rarely that configurable outside of JPA. So – there's no clean way of simply pointing a repository to a specific DB, but there are of course workarounds.
For example, you can use child contexts and basically instantiate each context to use one of the DBs.
There are other solutions as well – such as having two sets of repos, or configuring the repos with a more fine-grained approach.

But ultimately you'll have to take care of it manually.
Hope that helps. Cheers,
Eugen.

+ 0 −

3 years ago

Guest

Bheem Reddy

Jun ,

I have added a sample configuration which worked for me , please find the above comments added by me.

Cheers,
Bheem

+ 0 −

2 years ago

Guest

Eugen Paraschiv (http://www.baeldung.com/)

Hey Bheem – can you provide a quick link to them instead of a full code sample? The comments here don't render code well (especially if you don't use the proper code element). Cheers,
Eugen.

+ 0 −

2 years ago

Guest

Bheem Reddy

Please refer the below link
https://docs.google.com/document/d/1XltsJam-0q6TmiMc8iK192UxS5rw04kTjO22ycN4vvc/edit
(https://docs.google.com/document/d/1XltsJam-0q6TmiMc8iK192UxS5rw04kTjO22ycN4vvc/edit)

+ 0 −

2 years ago

Guest

Cherukuri

Hi Eugen, thats a great article but one small correction in the above post “findAll with Pegeable” section in second line it should be “pages” 🙂

+ 0 −

2 years ago

Guest

Eugen Paraschiv (http://www.baeldung.com/)

Nice catch Cherukuri – fixed. Cheers,
Eugen.

+ 0 −

2 years ago

Guest

Bheem Reddy

Is this possible to create connect from Context to multiple Data bases like below

Each Repository to one template

+ 0 −

2 years ago

Guest

Eugen Paraschiv (http://www.baeldung.com/)

Hey Bheem,
I haven't tried to do that before, so I'm not sure. That's an interesting usecase though – let me know if you get it working and implemented. Cheers,
Eugen.

+ 0 −

2 years ago

Guest

Marian Vasile Caraiman

Hi Bheem,
I don't see any problems doing what you want. Spring is searching by default a bean named template, but if you override it with mongo-template-ref it will happily wire it. Anyway, my suggestion is to "play" with various repositories using Maven support for profiles, i.e different settings based on properties set per profile.

+ 0 −

2 years ago

Guest

Todd

this is almost a great tutorial, but I have a couple of issue…
1) from step 4 to step 5 you jump to using mongoTemplate, what is mongoTemplate? there is no discussion of it or page setup or anything
2) I can't download the git repo, eclipse is giving me an error saying can't connect, I've been able to download guides from other sources…can you check that this is still working?

+ 0 −

2 years ago

Guest

Grzegorz Piwowarek

MongoTemplate creation is shown in the XML section explicitly but when it comes to the Java config, the bean definition can be found in AbstractMongoConfiguration. We will update an article.

Link in the description points straight to the one particular folder. If you want to point to the main repo, use this one: https://github.com/eugenp/tutorials (https://github.com/eugenp/tutorials)

+ 0 −

2 years ago

Guest

Todd

thanks for the prompt reply…got the code checked out just fine, I appreciate your help…

+ 0 −

2 years ago

Guest

Grzegorz Piwowarek

You are welcome 🙂

+ 0 −

2 years ago

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)

SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)

SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE COURSES (HTTPS://COURSES.BAELDUNG.COM)

CONSULTING WORK (/CONSULTING)

META BAELDUNG (HTTP://META.BAELDUNG.COM/)

THE FULL ARCHIVE (/FULL_ARCHIVE)

WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)

EDITORS (/EDITORS)

OUR PARTNERS (/PARTNERS)

ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)