

Spring Data MongoDB - Reference Documentation

Mark Pollack, Thomas Risberg, Oliver Gierke, Costin Leau, Jon Brisbin, Thomas Darimont, Christoph Strobl

Version 1.7.0.RC1
2015-03-05

Table of Contents

Preface	1
1. Knowing Spring	2
2. Knowing NoSQL and Document databases	3
3. Requirements	4
4. Additional Help Resources	5
4.1. Support	5
4.1.1. Community Forum	5
4.1.2. Professional Support	5
4.2. Following Development	5
Reference Documentation	5
5. Introduction	6
5.1. Document Structure	6
6. MongoDB support	7
6.1. Getting Started	7
6.2. Examples Repository	11
6.3. Connecting to MongoDB with Spring	11
6.3.1. Registering a Mongo instance using Java based metadata	11
6.3.2. Registering a Mongo instance using XML based metadata	13
6.3.3. The MongoClientFactory interface	15
6.3.4. Registering a MongoClientFactory instance using Java based metadata	16
6.3.5. Registering a MongoClientFactory instance using XML based metadata	17
6.4. Introduction to MongoTemplate	18
6.4.1. Instantiating MongoTemplate	19
6.4.2. WriteResultChecking Policy	21
6.4.3. WriteConcern	21
6.4.4. WriteConcernResolver	21
6.5. Saving, Updating, and Removing Documents	22
6.5.1. How the '_id' field is handled in the mapping layer	25
6.5.2. Type mapping	26
6.5.3. Methods for saving and inserting documents	29
6.5.4. Updating documents in a collection	30
6.5.5. Upserting documents in a collection	32
6.5.6. Finding and Upserting documents in a collection	32
6.5.7. Methods for removing documents	33
6.6. Querying Documents	33
6.6.1. Querying documents in a collection	34
6.6.2. Methods for querying for documents	36
6.6.3. GeoSpatial Queries	37
6.6.4. Full Text Queries	40
6.7. Map-Reduce Operations	41
6.7.1. Example Usage	41
6.8. Script Operations	44

6.8.1. Example Usage	44
6.9. Group Operations	44
6.9.1. Example Usage	45
6.10. Aggregation Framework Support	47
6.10.1. Basic Concepts	47
6.10.2. Supported Aggregation Operations	48
6.10.3. Projection Expressions	49
6.11. Overriding default mapping with custom converters	56
6.11.1. Saving using a registered Spring Converter	57
6.11.2. Reading using a Spring Converter	57
6.11.3. Registering Spring Converters with the MongoConverter	57
6.11.4. Converter disambiguation	58
6.12. Index and Collection management	59
6.12.1. Methods for creating an Index	59
6.12.2. Accessing index information	60
6.12.3. Methods for working with a Collection	60
6.13. Executing Commands	60
6.13.1. Methods for executing commands	61
6.14. Lifecycle Events	61
6.15. Exception Translation	62
6.16. Execution callbacks	63
6.17. GridFS support	64
7. MongoDB repositories	67
7.1. Introduction	67
7.2. Usage	67
7.3. Query methods	70
7.3.1. Repository delete queries	71
7.3.2. Geo-spatial repository queries	72
7.3.3. MongoDB JSON based query methods and field restriction	73
7.3.4. Type-safe Query methods	74
7.3.5. Full-text search queries	75
7.4. Miscellaneous	76
7.4.1. CDI Integration	76
7.5. General auditing configuration	77
8. Mapping	79
8.1. Convention based Mapping	79
8.1.1. How the '_id' field is handled in the mapping layer	79
8.2. Mapping Configuration	80
8.3. Metadata based Mapping	84
8.3.1. Mapping annotation overview	84
8.3.2. Customized Object Construction	87
8.3.3. Compound Indexes	88
8.3.4. Text Indexes	88
8.3.5. Using DBRefs	89

- 8.3.6. Mapping Framework Events 90
- 8.3.7. Overriding Mapping with explicit Converters 90
- 9. Cross Store support 92
 - 9.1. Cross Store Configuration 92
 - 9.2. Writing the Cross Store Application 98
- 10. Logging support 101
 - 10.1. MongoDB Log4j Configuration 101
- 11. JMX support 102
 - 11.1. MongoDB JMX Configuration 102
- 12. MongoDB 3.0 Support 105
 - 12.1. Using Spring Data MongoDB with MongoDB 3.0 105
 - 12.1.1. Configuration Options 105
 - 12.1.2. WriteConcern and WriteConcernChecking 106
 - 12.1.3. Authentication 106
 - 12.1.4. Other things to be aware of 107
- Appendix 107

© 2008-2014 The original authors.

NOTE

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Preface

The Spring Data MongoDB project applies core Spring concepts to the development of solutions using the MongoDB document style data store. We provide a "template" as a high-level abstraction for storing and querying documents. You will notice similarities to the JDBC support in the Spring Framework.

This document is the reference guide for Spring Data - Document Support. It explains Document module concepts and semantics and the syntax for various stores namespaces.

This section provides some basic introduction to Spring and Document database. The rest of the document refers only to Spring Data Document features and assumes the user is familiar with document databases such as MongoDB and CouchDB as well as Spring concepts.

Chapter 1. Knowing Spring

Spring Data uses Spring framework's [core](#) functionality, such as the [IoC](#) container, [type conversion system](#), [expression language](#), [JMX integration](#), and portable [DAO exception hierarchy](#). While it is not important to know the Spring APIs, understanding the concepts behind them is. At a minimum, the idea behind IoC should be familiar for whatever IoC container you choose to use.

The core functionality of the MongoDB and CouchDB support can be used directly, with no need to invoke the IoC services of the Spring Container. This is much like [JdbcTemplate](#) which can be used 'standalone' without any other services of the Spring container. To leverage all the features of Spring Data document, such as the repository support, you will need to configure some parts of the library using Spring.

To learn more about Spring, you can refer to the comprehensive (and sometimes disarming) documentation that explains in detail the Spring Framework. There are a lot of articles, blog entries and books on the matter - take a look at the Spring framework [home page](#) for more information.

Chapter 2. Knowing NoSQL and Document databases

NoSQL stores have taken the storage world by storm. It is a vast domain with a plethora of solutions, terms and patterns (to make things worth even the term itself has multiple [meanings](#)). While some of the principles are common, it is crucial that the user is familiar to some degree with the stores supported by DATADOC. The best way to get acquainted to this solutions is to read their documentation and follow their examples - it usually doesn't take more then 5-10 minutes to go through them and if you are coming from an RDMBS-only background many times these exercises can be an eye opener.

The jumping off ground for learning about MongoDB is www.mongodb.org. Here is a list of other useful resources:

- The [manual](#) introduces MongoDB and contains links to getting started guides, reference documentation and tutorials.
- The [online shell](#) provides a convenient way to interact with a MongoDB instance in combination with the online [tutorial](#).
- MongoDB [Java Language Center](#)
- Several [books](#) available for purchase
- Karl Seguin's online book: [The Little MongoDB Book](#)

Chapter 3. Requirements

Spring Data MongoDB 1.x binaries requires JDK level 6.0 and above, and [Spring Framework](#) 3.2.x and above.

In terms of document stores, [MongoDB](#) at least 2.4, preferably version 2.6.

Chapter 4. Additional Help Resources

Learning a new framework is not always straight forward. In this section, we try to provide what we think is an easy to follow guide for starting with Spring Data Document module. However, if you encounter issues or you are just looking for an advice, feel free to use one of the links below:

4.1. Support

There are a few support options available:

4.1.1. Community Forum

Spring Data on Stackoverflow [Stackoverflow](#) is a tag for all Spring Data (not just Document) users to share information and help each other. Note that registration is needed **only** for posting.

4.1.2. Professional Support

Professional, from-the-source support, with guaranteed response time, is available from [Pivotal Software, Inc.](#), the company behind Spring Data and Spring.

4.2. Following Development

For information on the Spring Data Mongo source code repository, nightly builds and snapshot artifacts please see the [Spring Data Mongo homepage](#). You can help make Spring Data best serve the needs of the Spring community by interacting with developers through the Community on [Stackoverflow](#). To follow developer activity look for the mailing list information on the Spring Data Mongo homepage. If you encounter a bug or want to suggest an improvement, please create a ticket on the Spring Data issue [tracker](#). To stay up to date with the latest news and announcements in the Spring eco system, subscribe to the Spring Community [Portal](#). Lastly, you can follow the SpringSource Data [blog](#) or the project team on Twitter ([SpringData](#)).

Reference Documentation

Chapter 5. Introduction

5.1. Document Structure

This part of the reference documentation explains the core functionality offered by Spring Data Document.

[MongoDB support](#) introduces the MongoDB module feature set.

[MongoDB repositories](#) introduces the repository support for MongoDB.

Chapter 6. MongoDB support

The MongoDB support contains a wide range of features which are summarized below.

- Spring configuration support using Java based `@Configuration` classes or an XML namespace for a Mongo driver instance and replica sets
- `MongoTemplate` helper class that increases productivity performing common Mongo operations. Includes integrated object mapping between documents and POJOs.
- Exception translation into Spring's portable Data Access Exception hierarchy
- Feature Rich Object Mapping integrated with Spring's Conversion Service
- Annotation based mapping metadata but extensible to support other metadata formats
- Persistence and mapping lifecycle events
- Java based Query, Criteria, and Update DSLs
- Automatic implementation of Repository interfaces including support for custom finder methods.
- QueryDSL integration to support type-safe queries.
- Cross-store persistence - support for JPA Entities with fields transparently persisted/retrieved using MongoDB
- Log4j log appender
- GeoSpatial integration

For most tasks you will find yourself using `MongoTemplate` or the Repository support that both leverage the rich mapping functionality. `MongoTemplate` is the place to look for accessing functionality such as incrementing counters or ad-hoc CRUD operations. `MongoTemplate` also provides callback methods so that it is easy for you to get a hold of the low level API artifacts such as `org.mongo.DB` to communicate directly with MongoDB. The goal with naming conventions on various API artifacts is to copy those in the base MongoDB Java driver so you can easily map your existing knowledge onto the Spring APIs.

6.1. Getting Started

Spring MongoDB support requires MongoDB 1.4 or higher and Java SE 5 or higher. The latest production release (2.4.9 as of this writing) is recommended. An easy way to bootstrap setting up a working environment is to create a Spring based project in [STS](#).

First you need to set up a running MongoDB server. Refer to the [MongoDB Quick Start guide](#) for an explanation on how to startup a MongoDB instance. Once installed starting MongoDB is typically a matter of executing the following command: `MONGO_HOME/bin/mongod`

To create a Spring project in STS go to File → New → Spring Template Project → Simple Spring Utility Project → press Yes when prompted. Then enter a project and a package name such as org.springframework.mongodb.example.

Then add the following to pom.xml dependencies section.

```
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>1.4.1.RELEASE</version>
  </dependency>

</dependencies>
```

Also change the version of Spring in the pom.xml to be

```
<spring.framework.version>3.2.8.RELEASE</spring.framework.version>
```

You will also need to add the location of the Spring Milestone repository for maven to your pom.xml which is at the same level of your <dependencies/> element

```
<repositories>
  <repository>
    <id>spring-milestone</id>
    <name>Spring Maven MILESTONE Repository</name>
    <url>http://repo.spring.io/libs-milestone</url>
  </repository>
</repositories>
```

The repository is also [browseable here](#).

You may also want to set the logging level to **DEBUG** to see some additional information, edit the log4j.properties file to have

```
log4j.category.org.springframework.data.document.mongodb=DEBUG
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %40.40c:%4L - %m%n
```

Create a simple Person class to persist:

```
package org.springframework.example;

public class Person {

    private String id;
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person [id=" + id + ", name=" + name + ", age=" + age + "];"
    }
}
```

And a main application to run

```

package org.springframework.mongodb.example;

import static org.springframework.data.mongodb.core.query.Criteria.where;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Query;

import com.mongodb.Mongo;

public class MongoApp {

    private static final Log log = LogFactory.getLog(MongoApp.class);

    public static void main(String[] args) throws Exception {

        MongoOperations mongoOps = new MongoTemplate(new Mongo(), "database");
        mongoOps.insert(new Person("Joe", 34));

        log.info(mongoOps.findOne(new Query(where("name").is("Joe")), Person.class));

        mongoOps.dropCollection("person");
    }
}

```

This will produce the following output

```

10:01:32,062 DEBUG apping.MongoPersistentEntityIndexCreator: 80 - Analyzing class class
org.springframework.example.Person for index information.
10:01:32,265 DEBUG ramework.data.mongodb.core.MongoTemplate: 631 - insertDBObject
containing fields: [_class, age, name] in collection: Person
10:01:32,765 DEBUG ramework.data.mongodb.core.MongoTemplate:1243 - findOne using query: {
"name" : "Joe"} in db.collection: database.Person
10:01:32,953 INFO      org.springframework.mongodb.example.MongoApp: 25 - Person
[id=4ddbba3c0be56b7e1b210166, name=Joe, age=34]
10:01:32,984 DEBUG ramework.data.mongodb.core.MongoTemplate: 375 - Dropped collection
[database.person]

```

Even in this simple example, there are few things to take notice of

- You can instantiate the central helper class of Spring Mongo, `MongoTemplate`, using the standard `com.mongodb.Mongo` object and the name of the database to use.

- The mapper works against standard POJO objects without the need for any additional metadata (though you can optionally provide that information. See [here](#)).
- Conventions are used for handling the id field, converting it to be a ObjectId when stored in the database.
- Mapping conventions can use field access. Notice the Person class has only getters.
- If the constructor argument names match the field names of the stored document, they will be used to instantiate the object

6.2. Examples Repository

There is an [github repository with several examples](#) that you can download and play around with to get a feel for how the library works.

6.3. Connecting to MongoDB with Spring

One of the first tasks when using MongoDB and Spring is to create a `com.mongodb.Mongo` object using the IoC container. There are two main ways to do this, either using Java based bean metadata or XML based bean metadata. These are discussed in the following sections.

NOTE

For those not familiar with how to configure the Spring container using Java based bean metadata instead of XML based metadata see the high level introduction in the reference docs [here](#) as well as the detailed documentation <http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-java-instantiating-container> [here].

6.3.1. Registering a Mongo instance using Java based metadata

An example of using Java based bean metadata to register an instance of a `com.mongodb.Mongo` is shown below

Example 1. Registering a com.mongodb.Mongo object using Java based bean metadata

```
@Configuration
public class AppConfig {

    /*
     * Use the standard Mongo driver API to create a com.mongodb.Mongo instance.
     */
    public @Bean Mongo mongo() throws UnknownHostException {
        return new Mongo("localhost");
    }
}
```

This approach allows you to use the standard `com.mongodb.Mongo` API that you may already be used to using but also pollutes the code with the `UnknownHostException` checked exception. The use of the checked exception is not desirable as Java based bean metadata uses methods as a means to set object dependencies, making the calling code cluttered.

An alternative is to register an instance of `com.mongodb.Mongo` instance with the container using Spring's `MongoFactoryBean`. As compared to instantiating a `com.mongodb.Mongo` instance directly, the `FactoryBean` approach does not throw a checked exception and has the added advantage of also providing the container with an `ExceptionHandler` implementation that translates MongoDB exceptions to exceptions in Spring's portable `DataAccessException` hierarchy for data access classes annotated with the `@Repository` annotation. This hierarchy and use of `@Repository` is described in [Spring's DAO support features](#).

An example of a Java based bean metadata that supports exception translation on `@Repository` annotated classes is shown below:

Example 2. Registering a `com.mongodb.Mongo` object using Spring's `MongoFactoryBean` and enabling Spring's exception translation support

```
@Configuration
public class AppConfig {

    /*
     * Factory bean that creates the com.mongodb.Mongo instance
     */
    public @Bean MongoFactoryBean mongo() {
        MongoFactoryBean mongo = new MongoFactoryBean();
        mongo.setHost("localhost");
        return mongo;
    }
}
```

To access the `com.mongodb.Mongo` object created by the `MongoFactoryBean` in other `@Configuration` or your own classes, use a “private `@Autowired Mongo mongo;`” field.

6.3.2. Registering a Mongo instance using XML based metadata

While you can use Spring's traditional `<beans/>` XML namespace to register an instance of `com.mongodb.Mongo` with the container, the XML can be quite verbose as it is general purpose. XML namespaces are a better alternative to configuring commonly used objects such as the Mongo instance. The mongo namespace allows you to create a Mongo instance server location, replica-sets, and options.

To use the Mongo namespace elements you will need to reference the Mongo schema:

Example 3. XML schema to configure MongoDB

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mongo="http://www.springframework.org/schema/data/mongo"
       xsi:schemaLocation=
         "http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context-3.0.xsd
         *http://www.springframework.org/schema/data/mongo
         http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd*
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <!-- Default bean name is 'mongo' -->
  *<mongo:mongo host="localhost" port="27017"/>*

</beans>
```

A more advanced configuration with `MongoOptions` is shown below (note these are not recommended values)

Example 4. XML schema to configure a `com.mongodb.Mongo` object with `MongoOptions`

```
<beans>

  <mongo:mongo host="localhost" port="27017">
    <mongo:options connections-per-host="8"
                  threads-allowed-to-block-for-connection-multiplier="4"
                  connect-timeout="1000"
                  max-wait-time="1500}"
                  auto-connect-retry="true"
                  socket-keep-alive="true"
                  socket-timeout="1500"
                  slave-ok="true"
                  write-number="1"
                  write-timeout="0"
                  write-fsync="true"/>
  </mongo:mongo/>

</beans>
```

A configuration using replica sets is shown below.

Example 5. XML schema to configure com.mongodb.Mongo object with Replica Sets

```
<mongo:mongo id="replicaSetMongo" replica-set="127.0.0.1:27017,localhost:27018"/>
```

6.3.3. The MongoClientFactory interface

While `com.mongodb.Mongo` is the entry point to the MongoDB driver API, connecting to a specific MongoDB database instance requires additional information such as the database name and an optional username and password. With that information you can obtain a `com.mongodb.DB` object and access all the functionality of a specific MongoDB database instance. Spring provides the `org.springframework.data.mongodb.core.MongoClientFactory` interface shown below to bootstrap connectivity to the database.

```
public interface MongoClientFactory {  
  
    DB getDb() throws DataAccessException;  
  
    DB getDb(String dbName) throws DataAccessException;  
}
```

The following sections show how you can use the container with either Java or the XML based metadata to configure an instance of the `MongoClientFactory` interface. In turn, you can use the `MongoClientFactory` instance to configure `MongoTemplate`.

The class `org.springframework.data.mongodb.core.SimpleMongoClientFactory` provides implements the `MongoClientFactory` interface and is created with a standard `com.mongodb.Mongo` instance, the database name and an optional `org.springframework.data.authentication.UserCredentials` constructor argument.

Instead of using the IoC container to create an instance of `MongoTemplate`, you can just use them in standard Java code as shown below.

```

public class MongoApp {

    private static final Log log = LoggerFactory.getLog(MongoApp.class);

    public static void main(String[] args) throws Exception {

        MongoOperations mongoOps = new MongoTemplate(*new SimpleMongoDbFactory(new Mongo(),
"database")*);

        mongoOps.insert(new Person("Joe", 34));

        log.info(mongoOps.findOne(new Query(where("name").is("Joe")), Person.class));

        mongoOps.dropCollection("person");
    }
}

```

The code in bold highlights the use of `SimpleMongoDbFactory` and is the only difference between the listing shown in the [getting started section](#).

6.3.4. Registering a `MongoDbFactory` instance using Java based metadata

To register a `MongoDbFactory` instance with the container, you write code much like what was highlighted in the previous code listing. A simple example is shown below

```

@Configuration
public class MongoConfiguration {

    public @Bean MongoDbFactory mongoDbFactory() throws Exception {
        return new SimpleMongoDbFactory(new Mongo(), "database");
    }
}

```

To define the username and password create an instance of `org.springframework.data.authentication.UserCredentials` and pass it into the constructor as shown below. This listing also shows using `MongoDbFactory` register an instance of `MongoTemplate` with the container.

```

@Configuration
public class MongoConfiguration {

    public @Bean MongoClientFactory mongoDbFactory() throws Exception {
        UserCredentials userCredentials = new UserCredentials("joe", "secret");
        return new SimpleMongoDbFactory(new MongoClient(), "database", userCredentials);
    }

    public @Bean MongoClient mongoTemplate() throws Exception {
        return new MongoClient(mongoDbFactory());
    }
}

```

6.3.5. Registering a MongoClientFactory instance using XML based metadata

The mongo namespace provides a convenient way to create a `SimpleMongoDbFactory` as compared to using the `<beans/>` namespace. Simple usage is shown below

```
<mongo:db-factory dbname="database">
```

In the above example a `com.mongodb.Mongo` instance is created using the default host and port number. The `SimpleMongoDbFactory` registered with the container is identified by the id 'mongoDbFactory' unless a value for the id attribute is specified.

You can also provide the host and port for the underlying `com.mongodb.Mongo` instance as shown below, in addition to username and password for the database.

```

<mongo:db-factory id="anotherMongoDbFactory"
    host="localhost"
    port="27017"
    dbname="database"
    username="joe"
    password="secret"/>

```

If you need to configure additional options on the `com.mongodb.Mongo` instance that is used to create a `SimpleMongoDbFactory` you can refer to an existing bean using the `mongo-ref` attribute as shown below. To show another common usage pattern, this listing shows the use of a property placeholder to parameterise the configuration and creating `MongoTemplate`.

```

<context:property-placeholder location=
"classpath:/com/myapp/mongodb/config/mongo.properties"/>

<mongo:mongo host="${mongo.host}" port="${mongo.port}">
  <mongo:options
    connections-per-host="${mongo.connectionsPerHost}"
    threads-allowed-to-block-for-connection-multiplier=
"${mongo.threadsAllowedToBlockForConnectionMultiplier}"
    connect-timeout="${mongo.connectTimeout}"
    max-wait-time="${mongo.maxWaitTime}"
    auto-connect-retry="${mongo.autoConnectRetry}"
    socket-keep-alive="${mongo.socketKeepAlive}"
    socket-timeout="${mongo.socketTimeout}"
    slave-ok="${mongo.slaveOk}"
    write-number="1"
    write-timeout="0"
    write-fsync="true"/>
</mongo:mongo>

<mongo:db-factory dbname="database" mongo-ref="mongo"/>

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

```

6.4. Introduction to MongoTemplate

The class `MongoTemplate`, located in the package `org.springframework.data.document.mongodb`, is the central class of the Spring's MongoDB support providing a rich feature set to interact with the database. The template offers convenience operations to create, update, delete and query for MongoDB documents and provides a mapping between your domain objects and MongoDB documents.

NOTE

Once configured, `MongoTemplate` is thread-safe and can be reused across multiple instances.

The mapping between MongoDB documents and domain classes is done by delegating to an implementation of the interface `MongoConverter`. Spring provides two implementations, `SimpleMappingConverter` and `MongoMappingConverter`, but you can also write your own converter. Please refer to the section on `MongoConverters` for more detailed information.

The `MongoTemplate` class implements the interface `MongoOperations`. In as much as possible, the methods on `MongoOperations` are named after methods available on the MongoDB driver `Collection` object as to make the API familiar to existing MongoDB developers who are used to the driver API. For example, you will find methods such as "find", "findAndModify", "findOne", "insert", "remove", "save", "update"

and "updateMulti". The design goal was to make it as easy as possible to transition between the use of the base MongoDB driver and `MongoOperations`. A major difference in between the two APIs is that `MongoOperations` can be passed domain objects instead of `DBObject` and there are fluent APIs for `Query`, `Criteria`, and `Update` operations instead of populating a `DBObject` to specify the parameters for those operations.

NOTE | The preferred way to reference the operations on `MongoTemplate` instance is via its interface `MongoOperations`.

The default converter implementation used by `MongoTemplate` is `MongoMappingConverter`. While the `MongoMappingConverter` can make use of additional metadata to specify the mapping of objects to documents it is also capable of converting objects that contain no additional metadata by using some conventions for the mapping of IDs and collection names. These conventions as well as the use of mapping annotations is explained in the [Mapping chapter](#).

NOTE | In the M2 release `SimpleMappingConverter`, was the default and this class is now deprecated as its functionality has been subsumed by the `MongoMappingConverter`.

Another central feature of `MongoTemplate` is exception translation of exceptions thrown in the MongoDB Java driver into Spring's portable Data Access Exception hierarchy. Refer to the section on [exception translation](#) for more information.

While there are many convenience methods on `MongoTemplate` to help you easily perform common tasks if you should need to access the MongoDB driver API directly to access functionality not explicitly exposed by the `MongoTemplate` you can use one of several Execute callback methods to access underlying driver APIs. The execute callbacks will give you a reference to either a `com.mongodb.Collection` or a `com.mongodb.DB` object. Please see the section `mongo.executioncallback[Execution Callbacks]` for more information.

Now let's look at a examples of how to work with the `MongoTemplate` in the context of the Spring container.

6.4.1. Instantiating `MongoTemplate`

You can use Java to create and register an instance of `MongoTemplate` as shown below.

Example 6. Registering a `com.mongodb.Mongo` object and enabling Spring's exception translation support

```
@Configuration
public class AppConfig {

    public @Bean Mongo mongo() throws Exception {
        return new Mongo("localhost");
    }

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

There are several overloaded constructors of `MongoTemplate`. These are

- `MongoTemplate(Mongo mongo, String dbName)` - takes the `com.mongodb.Mongo` object and the default database name to operate against.
- `MongoTemplate(Mongo mongo, String dbName, UserCredentials userCredentials)` - adds the username and password for authenticating with the database.
- `MongoTemplate(MongoDbFactory mongoDbFactory)` - takes a `MongoDbFactory` object that encapsulated the `com.mongodb.Mongo` object, database name, and username and password.
- `MongoTemplate(MongoDbFactory mongoDbFactory, MongoConverter mongoConverter)` - adds a `MongoConverter` to use for mapping.

You can also configure a `MongoTemplate` using Spring's XML `<beans/>` schema.

```
<mongo:mongo host="localhost" port="27017"/>

<bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg ref="mongo"/>
    <constructor-arg name="dbName" value="geospatial"/>
</bean>
```

Other optional properties that you might like to set when creating a `MongoTemplate` are the default `WriteResultCheckingPolicy`, `WriteConcern`, and `ReadPreference`.

NOTE

The preferred way to reference the operations on `MongoTemplate` instance is via its interface `MongoOperations`.

6.4.2. WriteResultChecking Policy

When in development it is very handy to either log or throw an exception if the `com.mongodb.WriteResult` returned from any MongoDB operation contains an error. It is quite common to forget to do this during development and then end up with an application that looks like it runs successfully but in fact the database was not modified according to your expectations. Set `MongoTemplate`'s `writeResultChecking` property to an enum with the following values, `LOG`, `EXCEPTION`, or `NONE` to either log the error, throw an exception or do nothing. The default is to use a `WriteResultChecking` value of `NONE`.

6.4.3. WriteConcern

You can set the `com.mongodb.WriteConcern` property that the `MongoTemplate` will use for write operations if it has not yet been specified via the driver at a higher level such as `com.mongodb.Mongo`. If `MongoTemplate`'s `writeConcern` property is not set it will default to the one set in the MongoDB driver's `DB` or `Collection` setting.

6.4.4. WriteConcernResolver

For more advanced cases where you want to set different `WriteConcern` values on a per-operation basis (for remove, update, insert and save operations), a strategy interface called `WriteConcernResolver` can be configured on `MongoTemplate`. Since `MongoTemplate` is used to persist POJOs, the `WriteConcernResolver` lets you create a policy that can map a specific POJO class to a `WriteConcern` value. The `WriteConcernResolver` interface is shown below.

```
public interface WriteConcernResolver {
    WriteConcern resolve(MongoAction action);
}
```

The passed in argument, `MongoAction`, is what you use to determine the `WriteConcern` value to be used or to use the value of the `Template` itself as a default. `MongoAction` contains the collection name being written to, the `java.lang.Class` of the POJO, the converted `DBObject`, as well as the operation as an enumeration (`MongoActionOperation`: `REMOVE`, `UPDATE`, `INSERT`, `INSERT_LIST`, `SAVE`) and a few other pieces of contextual information. For example,

```
private class MyAppWriteConcernResolver implements WriteConcernResolver {  
  
    public WriteConcern resolve(MongoAction action) {  
        if (action.getEntityClass().getSimpleName().contains("Audit")) {  
            return WriteConcern.NONE;  
        } else if (action.getEntityClass().getSimpleName().contains("Metadata")) {  
            return WriteConcern.JOURNAL_SAFE;  
        }  
        return action.getDefaultWriteConcern();  
    }  
}
```

6.5. Saving, Updating, and Removing Documents

MongoTemplate provides a simple way for you to save, update, and delete your domain objects and map those objects to documents stored in MongoDB.

Given a simple class such as Person

```

public class Person {

    private String id;
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person [id=" + id + ", name=" + name + ", age=" + age + "];"
    }

}

```

You can save, update and delete the object as shown below.

NOTE | `MongoOperations` is the interface that `MongoTemplate` implements.

```

package org.spring.example;

import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Update.update;
import static org.springframework.data.mongodb.core.query.Query.query;

import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.SimpleMongoDbFactory;

import com.mongodb.Mongo;

public class MongoApp {

    private static final Log log = LogFactory.getLog(MongoApp.class);

    public static void main(String[] args) throws Exception {

        MongoOperations mongoOps = new MongoTemplate(new SimpleMongoDbFactory(new Mongo(),
"database"));

        Person p = new Person("Joe", 34);

        // Insert is used to initially store the object into the database.
        mongoOps.insert(p);
        log.info("Insert: " + p);

        // Find
        p = mongoOps.findById(p.getId(), Person.class);
        log.info("Found: " + p);

        // Update
        mongoOps.updateFirst(query(where("name").is("Joe")), update("age", 35), Person.class
);
        p = mongoOps.findOne(query(where("name").is("Joe")), Person.class);
        log.info("Updated: " + p);

        // Delete
        mongoOps.remove(p);

        // Check that deletion worked
        List<Person> people = mongoOps.findAll(Person.class);
        log.info("Number of people = : " + people.size());
    }
}

```

```
mongoOps.dropCollection(Person.class);
}
}
```

This would produce the following log output (including debug messages from `MongoTemplate` itself)

```
DEBUG apping.MongoPersistentEntityIndexCreator: 80 - Analyzing class class
org.springframework.example.Person for index information.
DEBUG work.data.mongodb.core.MongoTemplate: 632 - insertDBObject containing fields:
[_class, age, name] in collection: person
INFO org.springframework.example.MongoApp: 30 - Insert: Person
[id=4ddc6e784ce5b1eba3ceaf5c, name=Joe, age=34]
DEBUG work.data.mongodb.core.MongoTemplate:1246 - findOne using query: { "_id" : { "$oid"
: "4ddc6e784ce5b1eba3ceaf5c"} } in db.collection: database.person
INFO org.springframework.example.MongoApp: 34 - Found: Person
[id=4ddc6e784ce5b1eba3ceaf5c, name=Joe, age=34]
DEBUG work.data.mongodb.core.MongoTemplate: 778 - calling update using query: { "name" :
"Joe"} and update: { "$set" : { "age" : 35}} in collection: person
DEBUG work.data.mongodb.core.MongoTemplate:1246 - findOne using query: { "name" : "Joe"}
in db.collection: database.person
INFO org.springframework.example.MongoApp: 39 - Updated: Person
[id=4ddc6e784ce5b1eba3ceaf5c, name=Joe, age=35]
DEBUG work.data.mongodb.core.MongoTemplate: 823 - remove using query: { "id" :
"4ddc6e784ce5b1eba3ceaf5c"} in collection: person
INFO org.springframework.example.MongoApp: 46 - Number of people = : 0
DEBUG work.data.mongodb.core.MongoTemplate: 376 - Dropped collection [database.person]
```

There was implicit conversion using the `MongoConverter` between a `String` and `ObjectId` as stored in the database and recognizing a convention of the property "Id" name.

NOTE | This example is meant to show the use of save, update and remove operations on `MongoTemplate` and not to show complex mapping functionality

The query syntax used in the example is explained in more detail in the section [Querying Documents](#).

6.5.1. How the '_id' field is handled in the mapping layer

MongoDB requires that you have an '_id' field for all documents. If you don't provide one the driver will assign a `ObjectId` with a generated value. When using the `MongoMappingConverter` there are certain rules that govern how properties from the Java class is mapped to this '_id' field.

The following outlines what property will be mapped to the '_id' document field:

- A property or field annotated with `@Id` (`org.springframework.data.annotation.Id`) will be mapped to

the '_id' field.

- A property or field without an annotation but named `id` will be mapped to the '_id' field.

The following outlines what type conversion, if any, will be done on the property mapped to the `_id` document field when using the `MappingMongoConverter`, the default for `MongoTemplate`.

- An id property or field declared as a `String` in the Java class will be converted to and stored as an `ObjectId` if possible using a Spring `Converter<String, ObjectId>`. Valid conversion rules are delegated to the MongoDB Java driver. If it cannot be converted to an `ObjectId`, then the value will be stored as a string in the database.
- An id property or field declared as `BigInteger` in the Java class will be converted to and stored as an `ObjectId` using a Spring `Converter<BigInteger, ObjectId>`.

If no field or property specified above is present in the Java class then an implicit '_id' field will be generated by the driver but not mapped to a property or field of the Java class.

When querying and updating `MongoTemplate` will use the converter to handle conversions of the `Query` and `Update` objects that correspond to the above rules for saving documents so field names and types used in your queries will be able to match what is in your domain classes.

6.5.2. Type mapping

As MongoDB collections can contain documents that represent instances of a variety of types. A great example here is if you store a hierarchy of classes or simply have a class with a property of type `Object`. In the latter case the values held inside that property have to be read in correctly when retrieving the object. Thus we need a mechanism to store type information alongside the actual document.

To achieve that the `MappingMongoConverter` uses a `MongoTypeMapper` abstraction with `DefaultMongoTypeMapper` as its main implementation. Its default behaviour is storing the fully qualified classname under `_class` inside the document for the top-level document as well as for every value if it's a complex type and a subtype of the property type declared.

Example 7. Type mapping

```
public class Sample {
    Contact value;
}

public abstract class Contact { }

public class Person extends Contact { }

Sample sample = new Sample();
sample.value = new Person();

mongoTemplate.save(sample);

{ "_class" : "com.acme.Sample",
  "value" : { "_class" : "com.acme.Person" }
}
```

As you can see we store the type information for the actual root class persistent as well as for the nested type as it is complex and a subtype of `Contact`. So if you're now using `mongoTemplate.findAll(Object.class, "sample")` we are able to find out that the document stored shall be a `Sample` instance. We are also able to find out that the value property shall be a `Person` actually.

Customizing type mapping

In case you want to avoid writing the entire Java class name as type information but rather like to use some key you can use the `@TypeAlias` annotation at the entity class being persisted. If you need to customize the mapping even more have a look at the `TypeInformationMapper` interface. An instance of that interface can be configured at the `DefaultMongoTypeMapper` which can be configured in turn on `MappingMongoConverter`.

Example 8. Defining a TypeAlias for an Entity

```
@TypeAlias("pers")
class Person {

}
```

Note that the resulting document will contain `"pers"` as the value in the `_class` Field.

Configuring custom type mapping

The following example demonstrates how to configure a custom `MongoTypeMapper` in `MappingMongoConverter`.

Example 9. Configuring a custom `MongoTypeMapper` via Spring Java Config

```
class CustomMongoTypeMapper extends DefaultMongoTypeMapper {
    //implement custom type mapping here
}
```

```
@Configuration
class SampleMongoConfiguration extends AbstractMongoConfiguration {

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

    @Override
    public Mongo mongo() throws Exception {
        return new Mongo();
    }

    @Bean
    @Override
    public MappingMongoConverter mappingMongoConverter() throws Exception {
        MappingMongoConverter mmc = super.mappingMongoConverter();
        mmc.setTypeMapper(customTypeMapper());
        return mmc;
    }

    @Bean
    public MongoTypeMapper customTypeMapper() {
        return new CustomMongoTypeMapper();
    }
}
```

Note that we are extending the `AbstractMongoConfiguration` class and override the bean definition of the `MappingMongoConverter` where we configure our custom `MongoTypeMapper`.

Example 10. Configuring a custom MongoTypeMapper via XML

```
<mongo:mapping-converter type-mapper-ref="customMongoTypeMapper"/>
<bean name="customMongoTypeMapper" class="com.bubu.mongo.CustomMongoTypeMapper"/>
```

6.5.3. Methods for saving and inserting documents

There are several convenient methods on `MongoTemplate` for saving and inserting your objects. To have more fine grained control over the conversion process you can register Spring converters with the `MappingMongoConverter`, for example `Converter<Person,DBObject>` and `Converter<DBObject, Person>`.

NOTE The difference between insert and save operations is that a save operation will perform an insert if the object is not already present.

The simple case of using the save operation is to save a POJO. In this case the collection name will be determined by name (not fully qualified) of the class. You may also call the save operation with a specific collection name. The collection to store the object can be overridden using mapping metadata.

When inserting or saving, if the Id property is not set, the assumption is that its value will be auto-generated by the database. As such, for auto-generation of an `ObjectId` to succeed the type of the Id property/field in your class must be either a `String`, `ObjectId`, or `BigInteger`.

Here is a basic example of using the save operation and retrieving its contents.

Example 11. Inserting and retrieving documents using the MongoTemplate

```
import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Criteria.query;

Person p = new Person("Bob", 33);
mongoTemplate.insert(p);

Person qp = mongoTemplate.findOne(query(where("age").is(33)), Person.class);
```

The insert/save operations available to you are listed below.

- `void save (Object objectToSave)` Save the object to the default collection.
- `void save (Object objectToSave, String collectionName)` Save the object to the specified collection.

A similar set of insert operations is listed below

- `void insert (Object objectToSave)` Insert the object to the default collection.
- `void insert (Object objectToSave, String collectionName)` Insert the object to the specified collection.

Which collection will my documents be saved into?

There are two ways to manage the collection name that is used for operating on the documents. The default collection name that is used is the class name changed to start with a lower-case letter. So a `com.test.Person` class would be stored in the "person" collection. You can customize this by providing a different collection name using the `@Document` annotation. You can also override the collection name by providing your own collection name as the last parameter for the selected `MongoTemplate` method calls.

Inserting or saving individual objects

The MongoDB driver supports inserting a collection of documents in one operation. The methods in the `MongoOperations` interface that support this functionality are listed below

- `insert` inserts an object. If there is an existing document with the same id then an error is generated.
- `insertAll` takes a `Collection` of objects as the first parameter. This method inspects each object and inserts it to the appropriate collection based on the rules specified above.
- `save` saves the object overwriting any object that might exist with the same id.

Inserting several objects in a batch

The MongoDB driver supports inserting a collection of documents in one operation. The methods in the `MongoOperations` interface that support this functionality are listed below

- `insert` methods that take a `Collection` as the first argument. This inserts a list of objects in a single batch write to the database.

6.5.4. Updating documents in a collection

For updates we can elect to update the first document found using `MongoOperations` method `updateFirst` or we can update all documents that were found to match the query using the method `updateMulti`. Here is an example of an update of all SAVINGS accounts where we are adding a one time \$50.00 bonus to the balance using the `$inc` operator.

Example 12. Updating documents using the MongoTemplate

```
import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Query;
import static org.springframework.data.mongodb.core.query.Update;

...

WriteResult wr = mongoTemplate.updateMulti(new Query(where("accounts.accountType").
is(Account.Type.SAVINGS)),
    new Update().inc("accounts.$balance", 50.00), Account.class);
```

In addition to the `Query` discussed above we provide the update definition using an `Update` object. The `Update` class has methods that match the update modifiers available for MongoDB.

As you can see most methods return the `Update` object to provide a fluent style for the API.

Methods for executing updates for documents

- **updateFirst** Updates the first document that matches the query document criteria with the provided updated document.
- **updateMulti** Updates all objects that match the query document criteria with the provided updated document.

Methods for the Update class

The `Update` class can be used with a little 'syntax sugar' as its methods are meant to be chained together and you can kick-start the creation of a new `Update` instance via the static method `public static Update update(String key, Object value)` and using static imports.

Here is a listing of methods on the `Update` class

- `Update addToSet (String key, Object value)` ` Update using the `\$addToSet` update modifier
- `Update inc (String key, Number inc)` Update using the `\$inc` update modifier
- `Update pop (String key, Update.Position pos)` Update using the `\$pop` update modifier
- `Update pull (String key, Object value)` Update using the `\$pull` update modifier
- `Update pullAll (String key, Object[] values)` Update using the `\$pullAll` update modifier
- `Update push (String key, Object value)` ` Update using the `\$push` update modifier
- `Update pushAll (String key, Object[] values)` Update using the `\$pushAll` update modifier

- Update **rename** (`String oldName, String newName`) Update using the `$rename` update modifier
- Update **set** (`String key, Object value`) Update using the `$set` update modifier
- Update **unset** (`String key`) Update using the `$unset` update modifier

6.5.5. Upserting documents in a collection

Related to performing an `updateFirst` operations, you can also perform an upsert operation which will perform an insert if no document is found that matches the query. The document that is inserted is a combination of the query document and the update document. Here is an example

```
template.upsert(query(where("ssn").is(1111).and("firstName").is("Joe").and("Fraizer").is("Update")), update("address", addr), Person.class);
```

6.5.6. Finding and Upserting documents in a collection

The `findAndModify()` method on `DBCollection` can update a document and return either the old or newly updated document in a single operation. `MongoTemplate` provides a `findAndModify` method that takes `Query` and `Update` classes and converts from `DBObject` to your POJOs. Here are the methods

```
<T> T findAndModify(Query query, Update update, Class<T> entityClass);

<T> T findAndModify(Query query, Update update, Class<T> entityClass, String
collectionName);

<T> T findAndModify(Query query, Update update, FindAndModifyOptions options, Class<T>
entityClass);

<T> T findAndModify(Query query, Update update, FindAndModifyOptions options, Class<T>
entityClass, String collectionName);
```

As an example usage, we will insert of few `Person` objects into the container and perform a simple `findAndUpdate` operation

```

mongoTemplate.insert(new Person("Tom", 21));
mongoTemplate.insert(new Person("Dick", 22));
mongoTemplate.insert(new Person("Harry", 23));

Query query = new Query(Criteria.where("firstName").is("Harry"));
Update update = new Update().inc("age", 1);
Person p = mongoTemplate.findAndModify(query, update, Person.class); // return's old
person object

assertThat(p.getFirstName(), is("Harry"));
assertThat(p.getAge(), is(23));
p = mongoTemplate.findOne(query, Person.class);
assertThat(p.getAge(), is(24));

// Now return the newly updated document when updating
p = template.findAndModify(query, update, new FindAndModifyOptions().returnNew(true),
Person.class);
assertThat(p.getAge(), is(25));

```

The `FindAndModifyOptions` lets you set the options of `returnNew`, `upsert`, and `remove`. An example extending off the previous code snippet is shown below

```

Query query2 = new Query(Criteria.where("firstName").is("Mary"));
p = mongoTemplate.findAndModify(query2, update, new FindAndModifyOptions().returnNew(
true).upsert(true), Person.class);
assertThat(p.getFirstName(), is("Mary"));
assertThat(p.getAge(), is(1));

```

6.5.7. Methods for removing documents

You can use several overloaded methods to remove an object from the database.

- **remove** Remove the given document based on one of the following: a specific object instance, a query document criteria combined with a class or a query document criteria combined with a specific collection name.

6.6. Querying Documents

You can express your queries using the `Query` and `Criteria` classes which have method names that mirror the native MongoDB operator names such as `lt`, `lte`, `is`, and others. The `Query` and `Criteria` classes follow a fluent API style so that you can easily chain together multiple method criteria and queries while having easy to understand code. Static imports in Java are used to help remove the need to see the 'new' keyword for creating `Query` and `Criteria` instances so as to improve readability. If you like to create `Query` instances from a plain JSON String use `BasicQuery`.

Example 13. Creating a Query instance from a plain JSON String

```
BasicQuery query = new BasicQuery("{ age : { $lt : 50 }, accounts.balance : { $gt : 1000.00 }}");
List<Person> result = mongoTemplate.find(query, Person.class);
```

GeoSpatial queries are also supported and are described more in the section [GeoSpatial Queries](#).

Map-Reduce operations are also supported and are described more in the section [Map-Reduce](#).

6.6.1. Querying documents in a collection

We saw how to retrieve a single document using the `findOne` and `findById` methods on `MongoTemplate` in previous sections which return a single domain object. We can also query for a collection of documents to be returned as a list of domain objects. Assuming that we have a number of `Person` objects with name and age stored as documents in a collection and that each person has an embedded account document with a balance. We can now run a query using the following code.

Example 14. Querying for documents using the MongoTemplate

```
import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Query.query;

List<Person> result = mongoTemplate.find(query(where("age").lt(50)
    .and("accounts.balance").gt(1000.00d)), Person.class);
```

All find methods take a `Query` object as a parameter. This object defines the criteria and options used to perform the query. The criteria is specified using a `Criteria` object that has a static factory method named `where` used to instantiate a new `Criteria` object. We recommend using a static import for `org.springframework.data.mongodb.core.query.Criteria.where` and `Query.query` to make the query more readable.

This query should return a list of `Person` objects that meet the specified criteria. The `Criteria` class has the following methods that correspond to the operators provided in MongoDB.

As you can see most methods return the `Criteria` object to provide a fluent style for the API.

Methods for the Criteria class

- `Criteria all (Object o)` Creates a criterion using the `$all` operator

- **Criteria and** (String key) Adds a chained **Criteria** with the specified **key** to the current **Criteria** and returns the newly created one
- **Criteria andOperator** (Criteria criteria) Creates an and query using the **\$and** operator for all of the provided criteria (requires MongoDB 2.0 or later)
- **Criteria elemMatch** (Criteria c) Creates a criterion using the **\$elemMatch** operator
- **Criteria exists** (boolean b) Creates a criterion using the **\$exists** operator
- **Criteria gt** (Object o) Creates a criterion using the **\$gt** operator
- **Criteria gte** (Object o) Creates a criterion using the **\$gte** operator
- **Criteria in** (Object o) Creates a criterion using the **\$in** operator for a varargs argument.
- **Criteria in** (Collection<?> collection) Creates a criterion using the **\$in** operator using a collection
- **Criteria is** (Object o) Creates a criterion using the **\$is** operator
- **Criteria lt** (Object o) Creates a criterion using the **\$lt** operator
- **Criteria lte** (Object o) Creates a criterion using the **\$lte** operator
- **Criteria mod** (Number value, Number remainder) Creates a criterion using the **\$mod** operator
- **Criteria ne** (Object o) Creates a criterion using the **\$ne** operator
- **Criteria nin** (Object o) Creates a criterion using the **\$nin** operator
- **Criteria norOperator** (Criteria criteria) Creates an nor query using the **\$nor** operator for all of the provided criteria
- **Criteria not** () Creates a criterion using the **\$not** meta operator which affects the clause directly following
- **Criteria orOperator** (Criteria criteria) Creates an or query using the **\$or** operator for all of the provided criteria
- **Criteria regex** (String re) Creates a criterion using a **\$regex**
- **Criteria size** (int s) Creates a criterion using the **\$size** operator
- **Criteria type** (int t) Creates a criterion using the **\$type** operator

There are also methods on the **Criteria** class for geospatial queries. Here is a listing but look at the section on [GeoSpatial Queries](#) to see them in action.

- **Criteria within** (Circle circle) Creates a geospatial criterion using **\$geoWithin** **\$center** operators.

- **Criteria within** (`Box box`) Creates a geospatial criterion using a `$geoWithin $box` operation.
- **Criteria withinSphere** (`Circle circle`) Creates a geospatial criterion using `$geoWithin $center` operators.
- **Criteria near** (`Point point`) Creates a geospatial criterion using a ``$near`` operation
- **Criteria nearSphere** (`Point point`) Creates a geospatial criterion using `$nearSphere$center` operations. This is only available for MongoDB 1.7 and higher.
- **Criteria minDistance** (`double minDistance`) Creates a geospatial criterion using the `$minDistance` operation, for use with `$near`.
- **Criteria maxDistance** (`double maxDistance`) Creates a geospatial criterion using the `$maxDistance` operation, for use with `$near`.

The `Query` class has some additional methods used to provide options for the query.

Methods for the Query class

- **Query addCriteria** (`Criteria criteria`) used to add additional criteria to the query
- **Field fields** (`()`) used to define fields to be included in the query results
- **Query limit** (`int limit`) used to limit the size of the returned results to the provided limit (used for paging)
- **Query skip** (`int skip`) used to skip the provided number of documents in the results (used for paging)
- **Sort sort** (`()`) used to provide sort definition for the results

6.6.2. Methods for querying for documents

The query methods need to specify the target type `T` that will be returned and they are also overloaded with an explicit collection name for queries that should operate on a collection other than the one indicated by the return type.

- **findAll** Query for a list of objects of type `T` from the collection.
- **findOne** Map the results of an ad-hoc query on the collection to a single instance of an object of the specified type.
- **findById** Return an object of the given id and target class.
- **find** Map the results of an ad-hoc query on the collection to a List of the specified type.
- **findAndRemove** Map the results of an ad-hoc query on the collection to a single instance of an object of the specified type. The first document that matches the query is returned and also

removed from the collection in the database.

6.6.3. GeoSpatial Queries

MongoDB supports GeoSpatial queries through the use of operators such as `$near`, `$within`, `geoWithin` and `$nearSphere`. Methods specific to geospatial queries are available on the `Criteria` class. There are also a few shape classes, `Box`, `Circle`, and `Point` that are used in conjunction with geospatial related `Criteria` methods.

To understand how to perform GeoSpatial queries we will use the following Venue class taken from the integration tests. which relies on using the rich `MappingMongoConverter`.

```

@Document(collection="newyork")
public class Venue {

    @Id
    private String id;
    private String name;
    private double[] location;

    @PersistenceConstructor
    Venue(String name, double[] location) {
        super();
        this.name = name;
        this.location = location;
    }

    public Venue(String name, double x, double y) {
        super();
        this.name = name;
        this.location = new double[] { x, y };
    }

    public String getName() {
        return name;
    }

    public double[] getLocation() {
        return location;
    }

    @Override
    public String toString() {
        return "Venue [id=" + id + ", name=" + name + ", location="
            + Arrays.toString(location) + "];"
    }
}

```

To find locations within a **Circle**, the following query can be used.

```

Circle circle = new Circle(-73.99171, 40.738868, 0.01);
List<Venue> venues =
    template.find(new Query(Criteria.where("location").within(circle)), Venue.class);

```

To find venues within a **Circle** using spherical coordinates the following query can be used

```
Circle circle = new Circle(-73.99171, 40.738868, 0.003712240453784);
List<Venue> venues =
    template.find(new Query(Criteria.where("location").withinSphere(circle)), Venue.
class);
```

To find venues within a **Box** the following query can be used

```
//lower-left then upper-right
Box box = new Box(new Point(-73.99756, 40.73083), new Point(-73.988135, 40.741404));
List<Venue> venues =
    template.find(new Query(Criteria.where("location").within(box)), Venue.class);
```

To find venues near a **Point**, the following queries can be used

```
Point point = new Point(-73.99171, 40.738868);
List<Venue> venues =
    template.find(new Query(Criteria.where("location").near(point).maxDistance(0.01)),
Venue.class);
```

```
Point point = new Point(-73.99171, 40.738868);
List<Venue> venues =
    template.find(new Query(Criteria.where("location").near(point).minDistance(0.01)
.maxDistance(100)), Venue.class);
```

To find venues near a **Point** using spherical coordinates the following query can be used

```
Point point = new Point(-73.99171, 40.738868);
List<Venue> venues =
    template.find(new Query(
Criteria.where("location").nearSphere(point).maxDistance(0.003712240453784)),
Venue.class);
```

Geo near queries

MongoDB supports querying the database for geo locations and calculation the distance from a given origin at the very same time. With geo-near queries it's possible to express queries like: "find all restaurants in the surrounding 10 miles". To do so **MongoOperations** provides **geoNear()** methods taking a **NearQuery** as argument as well as the already familiar entity type and collection

```
Point location = new Point(-73.99171, 40.738868);
NearQuery query = NearQuery.near(location).maxDistance(new Distance(10, Metrics.MILES));

GeoResults<Restaurant> = operations.geoNear(query, Restaurant.class);
```

As you can see we use the `NearQuery` builder API to set up a query to return all `Restaurant` instances surrounding the given `Point` by 10 miles maximum. The `Metrics` enum used here actually implements an interface so that other metrics could be plugged into a distance as well. A `Metric` is backed by a multiplier to transform the distance value of the given metric into native distances. The sample shown here would consider the 10 to be miles. Using one of the pre-built in metrics (miles and kilometers) will automatically trigger the spherical flag to be set on the query. If you want to avoid that, simply hand in plain `double` values into `maxDistance()`. For more information see the JavaDoc of `NearQuery` and `Distance`.

The geo near operations return a `GeoResults` wrapper object that encapsulates `GeoResult` instances. The wrapping `GeoResults` allows to access the average distance of all results. A single `GeoResult` object simply carries the entity found plus its distance from the origin.

6.6.4. Full Text Queries

Since MongoDB 2.6 full text queries can be executed using the `$text` operator. Methods and operations specific for full text queries are available in `TextQuery` and `TextCriteria`. When doing full text search please refer to the [MongoDB reference](#) for its behavior and limitations.

Full Text Search

Before we are actually able to use full text search we have to ensure to set up the search index correctly. Please refer to section [Text Index](#) for creating index structures.

```
db.foo.createIndex(
{
  title : "text",
  content : "text"
},
{
  weights : {
    title : 3
  }
}
)
```

A query searching for `coffee cake`, sorted by relevance according to the `weights` can be defined and executed as:

```
Query query = TextQuery.searching(new TextCriteria().matchingAny("coffee", "cake"))
    .sortByScore();
List<Document> page = template.find(query, Document.class);
```

Exclusion of search terms can directly be done by prefixing the term with `-` or using `notMatching`

```
// search for 'coffee' and not 'cake'
TextQuery.searching(new TextCriteria().matching("coffee").matching("-cake"));
TextQuery.searching(new TextCriteria().matching("coffee").notMatching("cake"));
```

As `TextCriteria.matching` takes the provided term as is. Therefore phrases can be defined by putting them between double quotes (eg. `"coffee cake"`) or using `TextCriteria.phrase`.

```
// search for phrase 'coffee cake'
TextQuery.searching(new TextCriteria().matching("\"coffee cake\""));
TextQuery.searching(new TextCriteria().phrase("coffee cake"));
```

6.7. Map-Reduce Operations

You can query MongoDB using Map-Reduce which is useful for batch processing, data aggregation, and for when the query language doesn't fulfill your needs.

Spring provides integration with MongoDB's map reduce by providing methods on `MongoOperations` to simplify the creation and execution of Map-Reduce operations. It can convert the results of a Map-Reduce operation to a POJO also integrates with Spring's [Resource abstraction](#) abstraction. This will let you place your JavaScript files on the file system, classpath, http server or any other Spring Resource implementation and then reference the JavaScript resources via an easy URI style syntax, e.g. `'classpath:reduce.js'`. Externalizing JavaScript code in files is often preferable to embedding them as Java strings in your code. Note that you can still pass JavaScript code as Java strings if you prefer.

6.7.1. Example Usage

To understand how to perform Map-Reduce operations an example from the book 'MongoDB - The definitive guide' is used. In this example we will create three documents that have the values [a,b], [b,c], and [c,d] respectfully. The values in each document are associated with the key 'x' as shown below. For this example assume these documents are in the collection named "jmr1".

```
{ "_id" : ObjectId("4e5ff893c0277826074ec533"), "x" : [ "a", "b" ] }
{ "_id" : ObjectId("4e5ff893c0277826074ec534"), "x" : [ "b", "c" ] }
{ "_id" : ObjectId("4e5ff893c0277826074ec535"), "x" : [ "c", "d" ] }
```

A map function that will count the occurrence of each letter in the array for each document is shown below

```
function () {
    for (var i = 0; i < this.x.length; i++) {
        emit(this.x[i], 1);
    }
}
```

The reduce function that will sum up the occurrence of each letter across all the documents is shown below

```
function (key, values) {
    var sum = 0;
    for (var i = 0; i < values.length; i++)
        sum += values[i];
    return sum;
}
```

Executing this will result in a collection as shown below.

```
{ "_id" : "a", "value" : 1 }
{ "_id" : "b", "value" : 2 }
{ "_id" : "c", "value" : 2 }
{ "_id" : "d", "value" : 1 }
```

Assuming that the map and reduce functions are located in map.js and reduce.js and bundled in your jar so they are available on the classpath, you can execute a map-reduce operation and obtain the results as shown below

```
MapReduceResults<ValueObject> results = mongoOperations.mapReduce("jmr1",
    "classpath:map.js", "classpath:reduce.js", ValueObject.class);
for (ValueObject valueObject : results) {
    System.out.println(valueObject);
}
```

The output of the above code is

```
ValueObject [id=a, value=1.0]
ValueObject [id=b, value=2.0]
ValueObject [id=c, value=2.0]
ValueObject [id=d, value=1.0]
```

The `MapReduceResults` class implements `Iterable` and provides access to the raw output, as well as timing and count statistics. The `ValueObject` class is simply

```
public class ValueObject {

    private String id;
    private float value;

    public String getId() {
        return id;
    }

    public float getValue() {
        return value;
    }

    public void setValue(float value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return "ValueObject [id=" + id + ", value=" + value + "];"
    }
}
```

By default the output type of `INLINE` is used so you don't have to specify an output collection. To specify additional map-reduce options use an overloaded method that takes an additional `MapReduceOptions` argument. The class `MapReduceOptions` has a fluent API so adding additional options can be done in a very compact syntax. Here an example that sets the output collection to "jmr1_out". Note that setting only the output collection assumes a default output type of `REPLACE`.

```
MapReduceResults<ValueObject> results = mongoOperations.mapReduce("jmr1",
    "classpath:map.js", "classpath:reduce.js",
    new
    MapReduceOptions().outputCollection("jmr1_out"), ValueObject.class);
```

There is also a static import `import static`

`org.springframework.data.mongodb.core.mapreduce.MapReduceOptions.options`; that can be used to make the syntax slightly more compact

```
MapReduceResults<ValueObject> results = mongoOperations.mapReduce("jmr1",
    "classpath:map.js", "classpath:reduce.js",
    options().outputCollection("jmr1_out"), ValueObject.class);
```

You can also specify a query to reduce the set of data that will be used to feed into the map-reduce operation. This will remove the document that contains [a,b] from consideration for map-reduce operations.

```
Query query = new Query(where("x").ne(new String[] { "a", "b" }));
MapReduceResults<ValueObject> results = mongoOperations.mapReduce(query, "jmr1",
    "classpath:map.js", "classpath:reduce.js",
    options().outputCollection("jmr1_out"), ValueObject.class);
```

Note that you can specify additional limit and sort values as well on the query but not skip values.

6.8. Script Operations

MongoDB allows to execute JavaScript functions on the server by either directly sending the script or calling a stored one. `ScriptOperations` can be accessed via `MongoTemplate` and provides basic abstraction for `JavaScript` usage.

6.8.1. Example Usage

```
ScriptOperations scriptOps = template.scriptOps();

ServerSideJavaScript echoScript = new ExecutableMongoScript("function(x) { return x; }");
scriptOps.execute(echoScript, "directly execute script");

scriptOps.register(new CallableMongoScript("echo", echoScript));
scriptOps.call("echo", "execute script via name");
```

6.9. Group Operations

As an alternative to using Map-Reduce to perform data aggregation, you can use the `group operation` which feels similar to using SQL's group by query style, so it may feel more approachable vs. using Map-Reduce. Using the group operations does have some limitations, for example it is not supported in a shared environment and it returns the full result set in a single BSON object, so the result should

be small, less than 10,000 keys.

Spring provides integration with MongoDB's group operation by providing methods on `MongoOperations` to simplify the creation and execution of group operations. It can convert the results of the group operation to a POJO and also integrates with Spring's [Resource abstraction](#) abstraction. This will let you place your JavaScript files on the file system, classpath, http server or any other Spring Resource implementation and then reference the JavaScript resources via an easy URI style syntax, e.g. `'classpath:reduce.js'`. Externalizing JavaScript code in files is often preferable to embedding them as Java strings in your code. Note that you can still pass JavaScript code as Java strings if you prefer.

6.9.1. Example Usage

In order to understand how group operations work the following example is used, which is somewhat artificial. For a more realistic example consult the book 'MongoDB - The definitive guide'. A collection named `"group_test_collection"` created with the following rows.

```
{ "_id" : ObjectId("4ec1d25d41421e2015da64f1"), "x" : 1 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f2"), "x" : 1 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f3"), "x" : 2 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f4"), "x" : 3 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f5"), "x" : 3 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f6"), "x" : 3 }
```

We would like to group by the only field in each row, the 'x' field and aggregate the number of times each specific value of 'x' occurs. To do this we need to create an initial document that contains our count variable and also a reduce function which will increment it each time it is encountered. The Java code to execute the group operation is shown below

```
GroupByResults<XObject> results = mongoTemplate.group("group_test_collection",
                                                    GroupBy.key("x").initialDocument("{
count: 0 }").reduceFunction("function(doc, prev) { prev.count += 1 }"),
                                                    XObject.class);
```

The first argument is the name of the collection to run the group operation over, the second is a fluent API that specifies properties of the group operation via a `GroupBy` class. In this example we are using just the `initialDocument` and `reduceFunction` methods. You can also specify a key-function, as well as a finalizer as part of the fluent API. If you have multiple keys to group by, you can pass in a comma separated list of keys.

The raw results of the group operation is a JSON document that looks like this

```

{
  "retval" : [ { "x" : 1.0 , "count" : 2.0} ,
                { "x" : 2.0 , "count" : 1.0} ,
                { "x" : 3.0 , "count" : 3.0} ] ,
  "count" : 6.0 ,
  "keys" : 3 ,
  "ok" : 1.0
}

```

The document under the "retval" field is mapped onto the third argument in the group method, in this case XObject which is shown below.

```

public class XObject {

    private float x;

    private float count;

    public float getX() {
        return x;
    }

    public void setX(float x) {
        this.x = x;
    }

    public float getCount() {
        return count;
    }

    public void setCount(float count) {
        this.count = count;
    }

    @Override
    public String toString() {
        return "XObject [x=" + x + " count = " + count + "]";
    }
}

```

You can also obtain the raw result as a `DBObject` by calling the method `getRawResults` on the `GroupByResults` class.

There is an additional method overload of the group method on `MongoOperations` which lets you specify

a **Criteria** object for selecting a subset of the rows. An example which uses a **Criteria** object, with some syntax sugar using static imports, as well as referencing a key-function and reduce function javascript files via a Spring Resource string is shown below.

```
import static org.springframework.data.mongodb.core.mapreduce.GroupBy.keyFunction;
import static org.springframework.data.mongodb.core.query.Criteria.where;

GroupByResults<XObject> results = mongoTemplate.group(where("x").gt(0),
                                                    "group_test_collection",
                                                    keyFunction("classpath:keyFunction.js").initialDo
cument("{ count: 0 }").reduceFunction("classpath:groupReduce.js"), XObject.class);
```

6.10. Aggregation Framework Support

Spring Data MongoDB provides support for the Aggregation Framework introduced to MongoDB in version 2.2.

The MongoDB Documentation describes the [Aggregation Framework](#) as follows:

For further information see the full [reference documentation](#) of the aggregation framework and other data aggregation tools for MongoDB.

6.10.1. Basic Concepts

The Aggregation Framework support in Spring Data MongoDB is based on the following key abstractions **Aggregation**, **AggregationOperation** and **AggregationResults**.

- **Aggregation**

An **Aggregation** represents a MongoDB **aggregate** operation and holds the description of the aggregation pipeline instructions. Aggregations are created by inoking the appropriate **newAggregation()** static factory Method of the **Aggregation** class which takes the list of **AggregateOperation** as a parameter next to the optional input class.

The actual aggregate operation is executed by the **aggregate** method of the **MongoTemplate** which also takes the desired output class as parameter.

- **AggregationOperation**

An **AggregationOperation** represents a MongoDB aggregation pipeline operation and describes the processing that should be performed in this aggregation step. Although one could manually create an **AggregationOperation** the recommended way to construct an **AggregateOperation** is to use the static factory methods provided by the **Aggregate** class.

- **AggregationResults**

AggregationResults is the container for the result of an aggregate operation. It provides access to the

raw aggregation result in the form of an `DBObject`, to the mapped objects and information which performed the aggregation.

The canonical example for using the Spring Data MongoDB support for the MongoDB Aggregation Framework looks as follows:

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

Aggregation agg = newAggregation(
    pipelineOP1(),
    pipelineOP2(),
    pipelineOPn()
);

AggregationResults<OutputType> results = mongoTemplate.aggregate(agg,
    "INPUT_COLLECTION_NAME", OutputType.class);
List<OutputType> mappedResult = results.getMappedResults();
```

Note that if you provide an input class as the first parameter to the `newAggregation` method the `MongoTemplate` will derive the name of the input collection from this class. Otherwise if you don't not specify an input class you must provide the name of the input collection explicitly. If an input-class and an input-collection is provided the latter takes precedence.

6.10.2. Supported Aggregation Operations

The MongoDB Aggregation Framework provides the following types of Aggregation Operations:

- Pipeline Aggregation Operators
- Group Aggregation Operators
- Boolean Aggregation Operators
- Comparison Aggregation Operators
- Arithmetic Aggregation Operators
- String Aggregation Operators
- Date Aggregation Operators
- Conditional Aggregation Operators

At the time of this writing we provide support for the following Aggregation Operations in Spring Data MongoDB.

Table 1. Aggregation Operations currently supported by Spring Data MongoDB

Pipeline Aggregation Operators	project, skip, limit, unwind, group, sort, geoNear
Group Aggregation Operators	addToSet, first, last, max, min, avg, push, sum, (*count)
Arithmetic Aggregation Operators	add (*via plus), subtract (*via minus), multiply, divide, mod
Comparison Aggregation Operators	eq (*via: is), gt, gte, lt, lte, ne

Note that the aggregation operations not listed here are currently not supported by Spring Data MongoDB. Comparison aggregation operators are expressed as **Criteria** expressions.

*) The operation is mapped or added by Spring Data MongoDB.

6.10.3. Projection Expressions

Projection expressions are used to define the fields that are the outcome of a particular aggregation step. Projection expressions can be defined via the `project` method of the `Aggregate` class either by passing a list of `String`'s or an aggregation framework `Fields` object. The projection can be extended with additional fields through a fluent API via the `and(String)` method and aliased via the `as(String)` method. Note that one can also define fields with aliases via the static factory method `Fields.field` of the aggregation framework that can then be used to construct a new `Fields` instance.

Example 15. Projection expression examples

```
project("name", "netPrice") // will generate {$project: {name: 1, netPrice: 1}}
project().and("foo").as("bar") // will generate {$project: {bar: $foo}}
project("a", "b").and("foo").as("bar") // will generate {$project: {a: 1, b: 1, bar: $foo}}
```

Note that more examples for project operations can be found in the `AggregationTests` class.

Note that further details regarding the projection expressions can be found in the [corresponding section](#) of the MongoDB Aggregation Framework reference documentation.

Spring Expression Support in Projection Expressions

As of Version 1.4.0 we support the use of SpEL expression in projection expressions via the `andExpression` method of the `ProjectionOperation` class. This allows you to define the desired expression as a SpEL expression which is translated into a corresponding MongoDB projection expression part on query execution. This makes it much easier to express complex calculations.

Complex calculations with SpEL expressions

The following SpEL expression:

$$1 + (q + 1) / (q - 1)$$

will be translated into the following projection expression part:

```
{ "$add" : [ 1, {  
  "$divide" : [ {  
    "$add":["$q", 1]}, {  
    "$subtract":["$q", 1]}  
  ]  
}]}
```

Have a look at an example in more context in [Aggregation Framework Example 5](#) and [Aggregation Framework Example 6](#). You can find more usage examples for supported SpEL expression constructs in [SpELExpressionTransformerUnitTests](#).

Aggregation Framework Examples

The following examples demonstrate the usage patterns for the MongoDB Aggregation Framework with Spring Data MongoDB.

In this introductory example we want to aggregate a list of tags to get the occurrence count of a particular tag from a MongoDB collection called "`tags`" sorted by the occurrence count in descending order. This example demonstrates the usage of grouping, sorting, projections (selection) and unwinding (result splitting).

```
class TagCount {  
  String tag;  
  int n;  
}
```

```

import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

Aggregation agg = newAggregation(
    project("tags"),
    unwind("tags"),
    group("tags").count().as("n"),
    project("n").and("tag").previousOperation(),
    sort(DESC, "n")
);

AggregationResults<TagCount> results = mongoTemplate.aggregate(agg, "tags", TagCount
.class);
List<TagCount> tagCount = results.getMappedResults();

```

- In order to do this we first create a new aggregation via the `newAggregation` static factory method to which we pass a list of aggregation operations. These aggregate operations define the aggregation pipeline of our `Aggregation`.
- As a second step we select the `"tags"` field (which is an array of strings) from the input collection with the `project` operation.
- In a third step we use the `unwind` operation to generate a new document for each tag within the `"tags"` array.
- In the forth step we use the `group` operation to define a group for each `"tags"`-value for which we aggregate the occurrence count via the `count` aggregation operator and collect the result in a new field called `"n"`.
- As a fifth step we select the field `"n"` and create an alias for the id-field generated from the previous group operation (hence the call to `previousOperation()`) with the name `"tag"`.
- As the sixth step we sort the resulting list of tags by their occurrence count in descending order via the `sort` operation.
- Finally we call the `aggregate` Method on the `MongoTemplate` in order to let MongoDB perform the acutal aggregation operation with the created `Aggregation` as an argument.

Note that the input collection is explicitly specified as the `"tags"` parameter to the `aggregate` Method. If the name of the input collection is not specified explicitly, it is derived from the input-class passed as first parameter to the `newAggregation` Method.

This example is based on the [Largest and Smallest Cities by State](#) example from the MongoDB Aggregation Framework documentation. We added additional sorting to produce stable results with different MongoDB versions. Here we want to return the smallest and largest cities by population for each state, using the aggregation framework. This example demonstrates the usage of grouping, sorting and projections (selection).

```

class ZipInfo {
    String id;
    String city;
    String state;
    @Field("pop") int population;
    @Field("loc") double[] location;
}

class City {
    String name;
    int population;
}

class ZipInfoStats {
    String id;
    String state;
    City biggestCity;
    City smallestCity;
}

```

```

import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<ZipInfo> aggregation = newAggregation(ZipInfo.class,
    group("state", "city")
        .sum("population").as("pop"),
    sort(ASC, "pop", "state", "city"),
    group("state")
        .last("city").as("biggestCity")
        .last("pop").as("biggestPop")
        .first("city").as("smallestCity")
        .first("pop").as("smallestPop"),
    project()
        .and("state").previousOperation()
        .and("biggestCity")
            .nested(bind("name", "biggestCity").and("population", "biggestPop"))
        .and("smallestCity")
            .nested(bind("name", "smallestCity").and("population", "smallestPop")),
    sort(ASC, "state")
);

AggregationResults<ZipInfoStats> result = mongoTemplate.aggregate(aggregation,
    ZipInfoStats.class);
ZipInfoStats firstZipInfoStats = result.getMappedResults().get(0);

```

- The class `ZipInfo` maps the structure of the given input-collection. The class `ZipInfoStats` defines the

structure in the desired output format.

- As a first step we use the `group` operation to define a group from the input-collection. The grouping criteria is the combination of the fields `"state"` and `"city"` which forms the id structure of the group. We aggregate the value of the `"population"` property from the grouped elements with by using the `sum` operator saving the result in the field `"pop"`.
- In a second step we use the `sort` operation to sort the intermediate-result by the fields `"pop"`, `"state"` and `"city"` in ascending order, such that the smallest city is at the top and the biggest city is at the bottom of the result. Note that the sorting on `"state"` and `"city"` is implicitly performed against the group id fields which Spring Data MongoDB took care of.
- In the third step we use a `group` operation again to group the intermediate result by `"state"`. Note that `"state"` again implicitly references an group-id field. We select the name and the population count of the biggest and smallest city with calls to the `last()` and `first()` operator respectively via the `project` operation.
- As the forth step we select the `"state"` field from the previous `group` operation. Note that `"state"` again implicitly references an group-id field. As we do not want an implicit generated id to appear, we exclude the id from the previous operation via `and(previousOperation()).exclude()`. As we want to populate the nested `City` structures in our output-class accordingly we have to emit appropriate sub-documents with the `nested` method.
- Finally as the fifth step we sort the resulting list of `StateStats` by their state name in ascending order via the `sort` operation.

Note that we derive the name of the input-collection from the `ZipInfo`-class passed as first parameter to the `newAggregation`-Method.

This example is based on the [States with Populations Over 10 Million](#) example from the MongoDB Aggregation Framework documentation. We added additional sorting to produce stable results with different MongoDB versions. Here we want to return all states with a population greater than 10 million, using the aggregation framework. This example demonstrates the usage of grouping, sorting and matching (filtering).

```
class StateStats {
    @Id String id;
    String state;
    @Field("totalPop") int totalPopulation;
}
```

```

import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<ZipInfo> agg = newAggregation(ZipInfo.class,
    group("state").sum("population").as("totalPop"),
    sort(ASC, previousOperation(), "totalPop"),
    match(where("totalPop").gte(10 * 1000 * 1000))
);

AggregationResults<StateStats> result = mongoTemplate.aggregate(agg, StateStats.class);
List<StateStats> stateStatsList = result.getMappedResults();

```

- As a first step we group the input collection by the "state" field and calculate the sum of the "population" field and store the result in the new field "totalPop".
- In the second step we sort the intermediate result by the id-reference of the previous group operation in addition to the "totalPop" field in ascending order.
- Finally in the third step we filter the intermediate result by using a `match` operation which accepts a `Criteria` query as an argument.

Note that we derive the name of the input-collection from the `ZipInfo`-class passed as first parameter to the `newAggregation`-Method.

This example demonstrates the use of simple arithmetic operations in the projection operation.

```

class Product {
    String id;
    String name;
    double netPrice;
    int spaceUnits;
}

```

```

import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<Product> agg = newAggregation(Product.class,
    project("name", "netPrice")
        .and("netPrice").plus(1).as("netPricePlus1")
        .and("netPrice").minus(1).as("netPriceMinus1")
        .and("netPrice").multiply(1.19).as("grossPrice")
        .and("netPrice").divide(2).as("netPriceDiv2")
        .and("spaceUnits").mod(2).as("spaceUnitsMod2")
);

AggregationResults<DBObject> result = mongoTemplate.aggregate(agg, DBObject.class);
List<DBObject> resultList = result.getMappedResults();

```

Note that we derive the name of the input-collection from the `Product`-class passed as first parameter to the `newAggregation`-Method.

This example demonstrates the use of simple arithmetic operations derived from SpEL Expressions in the projection operation.

```

class Product {
    String id;
    String name;
    double netPrice;
    int spaceUnits;
}

```

```

import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<Product> agg = newAggregation(Product.class,
    project("name", "netPrice")
        .andExpression("netPrice + 1").as("netPricePlus1")
        .andExpression("netPrice - 1").as("netPriceMinus1")
        .andExpression("netPrice / 2").as("netPriceDiv2")
        .andExpression("netPrice * 1.19").as("grossPrice")
        .andExpression("spaceUnits % 2").as("spaceUnitsMod2")
        .andExpression("(netPrice * 0.8 + 1.2) * 1.19").as(
"grossPriceIncludingDiscountAndCharge")
);

AggregationResults<DBObject> result = mongoTemplate.aggregate(agg, DBObject.class);
List<DBObject> resultList = result.getMappedResults();

```

This example demonstrates the use of complex arithmetic operations derived from SpEL Expressions in the projection operation.

Note: The additional parameters passed to the `addExpression` Method can be referenced via indexer expressions according to their position. In this example we reference the parameter which is the first parameter of the parameters array via `[0]`. External parameter expressions are replaced with their respective values when the SpEL expression is transformed into a MongoDB aggregation framework expression.

```
class Product {
    String id;
    String name;
    double netPrice;
    int spaceUnits;
}
```

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

double shippingCosts = 1.2;

TypedAggregation<Product> agg = newAggregation(Product.class,
    project("name", "netPrice")
        .andExpression("(netPrice * (1-discountRate) + [0]) * (1+taxRate)",
shippingCosts).as("salesPrice")
);

AggregationResults<DBObject> result = mongoTemplate.aggregate(agg, DBObject.class);
List<DBObject> resultList = result.getMappedResults();
```

Note that we can also refer to other fields of the document within the SpEL expression.

6.11. Overriding default mapping with custom converters

In order to have more fine grained control over the mapping process you can register Spring converters with the `MongoConverter` implementations such as the `MappingMongoConverter`.

The `MappingMongoConverter` checks to see if there are any Spring converters that can handle a specific class before attempting to map the object itself. To 'hijack' the normal mapping strategies of the `MappingMongoConverter`, perhaps for increased performance or other custom mapping needs, you first need to create an implementation of the Spring `Converter` interface and then register it with the `MappingConverter`.

NOTE

For more information on the Spring type conversion service see the reference docs [here](#).

6.11.1. Saving using a registered Spring Converter

An example implementation of the `Converter` that converts from a `Person` object to a `com.mongodb.DBObject` is shown below

```
import org.springframework.core.convert.converter.Converter;

import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;

public class PersonWriteConverter implements Converter<Person, DBObject> {

    public DBObject convert(Person source) {
        DBObject dbo = new BasicDBObject();
        dbo.put("_id", source.getId());
        dbo.put("name", source.getFirstName());
        dbo.put("age", source.getAge());
        return dbo;
    }
}
```

6.11.2. Reading using a Spring Converter

An example implementation of a `Converter` that converts from a `DBObject` of a `Person` object is shown below

```
public class PersonReadConverter implements Converter<DBObject, Person> {

    public Person convert(DBObject source) {
        Person p = new Person((ObjectId) source.get("_id"), (String) source.get("name"));
        p.setAge((Integer) source.get("age"));
        return p;
    }
}
```

6.11.3. Registering Spring Converters with the MongoConverter

The Mongo Spring namespace provides a convenience way to register Spring `Converter`'s with the `MappingMongoConverter`. The configuration snippet below shows how to manually register converter beans as well as configuring the wrapping `MappingMongoConverter` into a `MongoTemplate`.

```

<mongo:db-factory dbname="database"/>

<mongo:mapping-converter>
  <mongo:custom-converters>
    <mongo:converter ref="readConverter"/>
    <mongo:converter>
      <bean class="org.springframework.data.mongodb.test.PersonWriteConverter"/>
    </mongo:converter>
  </mongo:custom-converters>
</mongo:mapping-converter>

<bean id="readConverter" class="
org.springframework.data.mongodb.test.PersonReadConverter"/>

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

```

You can also use the base-package attribute of the custom-converters element to enable classpath scanning for all `Converter` and `GenericConverter` implementations below the given package.

```

<mongo:mapping-converter>
  <mongo:custom-converters base-package="com.acme.**.converters" />
</mongo:mapping-converter>

```

6.11.4. Converter disambiguation

Generally we inspect the `Converter` implementations for the source and target types they convert from and to. Depending on whether one of those is a type MongoDB can handle natively we will register the converter instance as reading or writing one. Have a look at the following samples:

```

// Write converter as only the target type is one Mongo can handle natively
class MyConverter implements Converter<Person, String> { }

// Read converter as only the source type is one Mongo can handle natively
class MyConverter implements Converter<String, Person> { }

```

In case you write a `Converter` whose source and target type are native Mongo types there's no way for us to determine whether we should consider it as reading or writing converter. Registering the converter instance as both might lead to unwanted results then. E.g. a `Converter<String, Long>` is ambiguous although it probably does not make sense to try to convert all `String`'s into `Long`'s when writing. To be generally able to force the infrastructure to register a converter for one way only

we provide `@ReadingConverter` as well as `@WritingConverter` to be used at the converter implementation.

6.12. Index and Collection management

`MongoTemplate` provides a few methods for managing indexes and collections. These are collected into a helper interface called `IndexOperations`. You access these operations by calling the method `indexOps` and pass in either the collection name or the `java.lang.Class` of your entity (the collection name will be derived from the `.class` either by name or via annotation metadata).

The `IndexOperations` interface is shown below

```
public interface IndexOperations {  
  
    void ensureIndex(IndexDefinition indexDefinition);  
  
    void dropIndex(String name);  
  
    void dropAllIndexes();  
  
    void resetIndexCache();  
  
    List<IndexInfo> getIndexInfo();  
}
```

6.12.1. Methods for creating an Index

We can create an index on a collection to improve query performance.

Creating an index using the `MongoTemplate`

```
mongoTemplate.indexOps(Person.class).ensureIndex(new Index().on("name", Order.ASCENDING));
```

- **ensureIndex** Ensure that an index for the provided `IndexDefinition` exists for the collection.

You can create standard, geospatial and text indexes using the classes `IndexDefinition`, `GeoSpatialIndex` and `TextIndexDefinition`. For example, given the `Venue` class defined in a previous section, you would declare a geospatial query as shown below.

```
mongoTemplate.indexOps(Venue.class).ensureIndex(new GeoSpatialIndex("location"));
```

6.12.2. Accessing index information

The `IndexOperations` interface has the method `getIndexInfo` that returns a list of `IndexInfo` objects. This contains all the indexes defined on the collection. Here is an example that defines an index on the `Person` class that has age property.

```
template.indexOps(Person.class).ensureIndex(new Index().on("age", Order.DESCENDING)
    .unique(Duplicates.DROP));

List<IndexInfo> indexInfoList = template.indexOps(Person.class).getIndexInfo();

// Contains
// [IndexInfo [fieldSpec={_id=ASCENDING}, name=_id_, unique=false, dropDuplicates=false,
// sparse=false],
// IndexInfo [fieldSpec={age=DESCENDING}, name=age_-1, unique=true, dropDuplicates=true,
// sparse=false]]
```

6.12.3. Methods for working with a Collection

It's time to look at some code examples showing how to use the `MongoTemplate`. First we look at creating our first collection.

Example 16. Working with collections using the `MongoTemplate`

```
DBCollection collection = null;
if (!mongoTemplate.getCollectionNames().contains("MyNewCollection")) {
    collection = mongoTemplate.createCollection("MyNewCollection");
}

mongoTemplate.dropCollection("MyNewCollection");
```

- **getCollectionNames** Returns a set of collection names.
- **collectionExists** Check to see if a collection with a given name exists.
- **createCollection** Create an uncapped collection
- **dropCollection** Drop the collection
- **getCollection** Get a collection by name, creating it if it doesn't exist.

6.13. Executing Commands

You can also get at the MongoDB driver's `DB.command()` method using the `executeCommand()` methods

on `MongoTemplate`. These will also perform exception translation into Spring's `DataAccessException` hierarchy.

6.13.1. Methods for executing commands

- `CommandResult executeCommand (DBObject command)` Execute a MongoDB command.
- `CommandResult executeCommand (String jsonCommand)` Execute the a MongoDB command expressed as a JSON string.

6.14. Lifecycle Events

Built into the MongoDB mapping framework are several `org.springframework.context.ApplicationEvent` events that your application can respond to by registering special beans in the `ApplicationContext`. By being based off Spring's `ApplicationContext` event infrastructure this enables other products, such as Spring Integration, to easily receive these events as they are a well known eventing mechanism in Spring based applications.

To intercept an object before it goes through the conversion process (which turns your domain object into a `com.mongodb.DBObject`), you'd register a subclass of `AbstractMongoEventListener` that overrides the `onBeforeConvert` method. When the event is dispatched, your listener will be called and passed the domain object before it goes into the converter.

```
public class BeforeConvertListener extends AbstractMongoEventListener<Person> {
    @Override
    public void onBeforeConvert(Person p) {
        ... does some auditing manipulation, set timestamps, whatever ...
    }
}
```

To intercept an object before it goes into the database, you'd register a subclass of `org.springframework.data.mongodb.core.mapping.event.AbstractMongoEventListener` that overrides the `onBeforeSave` method. When the event is dispatched, your listener will be called and passed the domain object and the converted `com.mongodb.DBObject`.

```

public class BeforeSaveListener extends AbstractMongoEventListener<Person> {
    @Override
    public void onBeforeSave(Person p, DBObject dbo) {
        change values, delete them, whatever
    }
}

```

Simply declaring these beans in your Spring ApplicationContext will cause them to be invoked whenever the event is dispatched.

The list of callback methods that are present in AbstractMappingEventListener are

- **onBeforeConvert** - called in MongoTemplate insert, insertList and save operations before the object is converted to a DBObject using a MongoConveter.
- **onBeforeSave** - called in MongoTemplate insert, insertList and save operations **before** inserting/saving the DBObject in the database.
- **onAfterSave** - called in MongoTemplate insert, insertList and save operations **after** inserting/saving the DBObject in the database.
- **onAfterLoad** - called in MongoTemplate find, findAndRemove, findOne and getCollection methods after the DBObject is retrieved from the database.
- **onAfterConvert** - called in MongoTemplate find, findAndRemove, findOne and getCollection methods after the DBObject retrieved from the database was converted to a POJO.

6.15. Exception Translation

The Spring framework provides exception translation for a wide variety of database and mapping technologies. This has traditionally been for JDBC and JPA. The Spring support for MongoDB extends this feature to the MongoDB Database by providing an implementation of the `org.springframework.dao.support.PersistenceExceptionTranslator` interface.

The motivation behind mapping to Spring's [consistent data access exception hierarchy](#) is that you are then able to write portable and descriptive exception handling code without resorting to coding against [MongoDB error codes](#). All of Spring's data access exceptions are inherited from the root `DataAccessException` class so you can be sure that you will be able to catch all database related exception within a single try-catch block. Note, that not all exceptions thrown by the MongoDB driver inherit from the `MongoException` class. The inner exception and message are preserved so no information is lost.

Some of the mappings performed by the `MongoExceptionTranslator` are: `com.mongodb.Network to DataAccessResourceFailureException` and `MongoException` error codes 1003, 12001, 12010, 12011, 12012

to `InvalidDataAccessApiUsageException`. Look into the implementation for more details on the mapping.

6.16. Execution callbacks

One common design feature of all Spring template classes is that all functionality is routed into one of the templates execute callback methods. This helps ensure that exceptions and any resource management that maybe required are performed consistency. While this was of much greater need in the case of JDBC and JMS than with MongoDB, it still offers a single spot for exception translation and logging to occur. As such, using these execute callback is the preferred way to access the MongoDB driver's `DB` and `DBCollection` objects to perform uncommon operations that were not exposed as methods on `MongoTemplate`.

Here is a list of execute callback methods.

- `<T> T execute (Class<?> entityClass, CollectionCallback<T> action)` Executes the given `CollectionCallback` for the entity collection of the specified class.
- `<T> T execute (String collectionName, CollectionCallback<T> action)` Executes the given `CollectionCallback` on the collection of the given name.
- `<T> T execute (DbCallback<T> action)` Spring Data MongoDB provides support for the `Aggregation Framework` introduced to MongoDB in version 2.2. Executes a `DbCallback` translating any exceptions as necessary.
- `<T> T execute (String collectionName, DbCallback<T> action)` Executes a `DbCallback` on the collection of the given name translating any exceptions as necessary.
- `<T> T executeInSession (DbCallback<T> action)` Executes the given `DbCallback` within the same connection to the database so as to ensure consistency in a write heavy environment where you may read the data that you wrote.

Here is an example that uses the `CollectionCallback` to return information about an index

```
boolean hasIndex = template.execute("geolocation", new CollectionCallbackBoolean<>() {
    public Boolean doInCollection(Venue.class, DBCollection collection) throws
MongoException, DataAccessException {
        List<DBObject> indexes = collection.getIndexInfo();
        for (DBObject dbo : indexes) {
            if ("location_2d".equals(dbo.get("name"))) {
                return true;
            }
        }
        return false;
    }
});
```

6.17. GridFS support

MongoDB supports storing binary files inside its filesystem GridFS. Spring Data MongoDB provides a `GridFsOperations` interface as well as the according implementation `GridFsTemplate` to easily interact with the filesystem. You can setup a `GridFsTemplate` instance by handing it a `MongoDbFactory` as well as a `MongoConverter`:

Example 17. JavaConfig setup for a GridFsTemplate

```
class GridFsConfiguration extends AbstractMongoConfiguration {  
  
    // further configuration omitted  
  
    @Bean  
    public GridFsTemplate gridFsTemplate() {  
        return new GridFsTemplate(mongoDbFactory(), mappingMongoConverter());  
    }  
}
```

An according XML configuration looks like this:

Example 18. XML configuration for a GridFsTemplate

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns:mongo="http://www.springframework.org/schema/data/mongo"  
    xsi:schemaLocation="http://www.springframework.org/schema/data/mongo  
        http://www.springframework.org/schema/data/mongo/spring-  
mongo.xsd  
        http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
    <mongo:db-factory id="mongoDbFactory" dbname="database" />  
    <mongo:mapping-converter id="converter" />  
  
    <bean class="org.springframework.data.mongodb.gridfs.GridFsTemplate">  
        <constructor-arg ref="mongoDbFactory" />  
        <constructor-arg ref="converter" />  
    </bean>  
  
</beans>
```

The template can now be injected and used to perform storage and retrieval operations.

Example 19. Using GridFsTemplate to store files

```
class GridFsClient {

    @Autowired
    GridFsOperations operations;

    @Test
    public void storeFileToGridFs {

        FileMetadata metadata = new FileMetadata();
        // populate metadata
        Resource file = // lookup File or Resource

        operations.store(file.getInputStream(), "filename.txt", metadata);
    }
}
```

The `store()` operations take an `InputStream`, a filename and optionally metadata information about the file to store. The metadata can be an arbitrary object which will be marshalled by the `MongoConverter` configured with the `GridFsTemplate`. Alternatively you can also provide a `DBObject` as well.

Reading files from the filesystem can either be achieved through the `find()` or `getResources()` methods. Let's have a look at the `find()` methods first. You can either find a single file matching a `Query` or multiple ones. To easily define file queries we provide the `GridFsCriteria` helper class. It provides static factory methods to encapsulate default metadata fields (e.g. `whereFilename()`, `whereContentType()`) or the custom one through `whereMetaData()`.

Example 20. Using GridFsTemplate to query for files

```
class GridFsClient {  
  
    @Autowired  
    GridFsOperations operations;  
  
    @Test  
    public void findFilesInGridFs {  
        List<GridFSDBFile> result = operations.find(query(whereFilename().is(  
"filename.txt")))  
    }  
}
```

NOTE

Currently MongoDB does not support defining sort criteria when retrieving files from GridFS. Thus any sort criteria defined on the `Query` instance handed into the `find()` method will be disregarded.

The other option to read files from the GridFs is using the methods introduced by the `ResourcePatternResolver` interface. They allow handing an Ant path into the method and thus retrieve files matching the given pattern.

Example 21. Using GridFsTemplate to read files

```
class GridFsClient {  
  
    @Autowired  
    GridFsOperations operations;  
  
    @Test  
    public void readFilesFromGridFs {  
        GridFsResources[] txtFiles = operations.getResources("*.txt");  
    }  
}
```

`GridFsOperations` extending `ResourcePatternResolver` allows the `GridFsTemplate` e.g. to be plugged into an `ApplicationContext` to read Spring Config files from a MongoDB.

Chapter 7. MongoDB repositories

7.1. Introduction

This chapter will point out the specialties for repository support for MongoDB. This builds on the core repository support explained in [\[repositories\]](#). So make sure you've got a sound understanding of the basic concepts explained there.

7.2. Usage

To access domain entities stored in a MongoDB you can leverage our sophisticated repository support that eases implementing those quite significantly. To do so, simply create an interface for your repository:

Example 22. Sample Person entity

```
public class Person {  
  
    @Id  
    private String id;  
    private String firstname;  
    private String lastname;  
    private Address address;  
  
    // getters and setters omitted  
}
```

We have a quite simple domain object here. Note that it has a property named `id` of type `ObjectId`. The default serialization mechanism used in `MongoTemplate` (which is backing the repository support) regards properties named `id` as document id. Currently we support `String`, `ObjectId` and `BigInteger` as id-types.

Example 23. Basic repository interface to persist Person entities

```
public interface PersonRepository extends PagingAndSortingRepository<Person, Long> {  
  
    // additional custom finder methods go here  
}
```

Right now this interface simply serves typing purposes but we will add additional methods to it later.

In your Spring configuration simply add

Example 24. General MongoDB repository Spring configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd">

  <mongo:mongo id="mongo" />

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

  <mongo:repositories base-package="com.acme.*.repositories" />

</beans>
```

This namespace element will cause the base packages to be scanned for interfaces extending `MongoRepository` and create Spring beans for each of them found. By default the repositories will get a `MongoTemplate` Spring bean wired that is called `mongoTemplate`, so you only need to configure `mongo-template-ref` explicitly if you deviate from this convention.

If you'd rather like to go with JavaConfig use the `@EnableMongoRepositories` annotation. The annotation carries the very same attributes like the namespace element. If no base package is configured the infrastructure will scan the package of the annotated configuration class.

Example 25. JavaConfig for repositories

```
@Configuration
@EnableMongoRepositories
class ApplicationConfig extends AbstractMongoConfiguration {

    @Override
    protected String getDatabaseName() {
        return "e-store";
    }

    @Override
    public Mongo mongo() throws Exception {
        return new Mongo();
    }

    @Override
    protected String getMappingBasePackage() {
        return "com.oreilly.springdata.mongodb"
    }
}
```

As our domain repository extends `PagingAndSortingRepository` it provides you with CRUD operations as well as methods for paginated and sorted access to the entities. Working with the repository instance is just a matter of dependency injecting it into a client. So accessing the second page of `Person`s at a page size of 10 would simply look something like this:

Example 26. Paging access to Person entities

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class PersonRepositoryTests {

    @Autowired PersonRepository repository;

    @Test
    public void readsFirstPageCorrectly() {

        Page<Person> persons = repository.findAll(new PageRequest(0, 10));
        assertThat(persons.isFirstPage(), is(true));
    }
}
```

The sample creates an application context with Spring's unit test support which will perform annotation based dependency injection into test cases. Inside the test method we simply use the repository to query the datastore. We hand the repository a `PageRequest` instance that requests the first page of persons at a page size of 10.

7.3. Query methods

Most of the data access operations you usually trigger on a repository result a query being executed against the MongoDB databases. Defining such a query is just a matter of declaring a method on the repository interface

Example 27. PersonRepository with query methods

```
public interface PersonRepository extends PagingAndSortingRepository<Person, String>
{
    List<Person> findByLastname(String lastname);

    Page<Person> findByFirstname(String firstname, Pageable pageable);

    Person findByShippingAddresses(Address address);
}
```

The first method shows a query for all people with the given lastname. The query will be derived parsing the method name for constraints which can be concatenated with `And` and `Or`. Thus the method name will result in a query expression of `{"lastname" : lastname}`. The second example shows how pagination is applied to a query. Just equip your method signature with a `Pageable` parameter and let the method return a `Page` instance and we will automatically page the query accordingly. The third examples shows that you can query based on properties which are not a primitive type.

NOTE | Note that for version 1.0 we currently don't support referring to parameters that are mapped as `DBRef` in the domain class.

Table 2. Supported keywords for query methods

Keyword	Sample	Logical result
GreaterThan	<code>findByAgeGreaterThan(int age)</code>	<code>{"age" : {"\$gt" : age}}</code>
GreaterThanEqual	<code>findByAgeGreaterThanEqual(int age)</code>	<code>{"age" : {"\$gte" : age}}</code>
LessThan	<code>findByAgeLessThan(int age)</code>	<code>{"age" : {"\$lt" : age}}</code>
LessThanEqual	<code>findByAgeLessThanEqual(int age)</code>	<code>{"age" : {"\$lte" : age}}</code>
Between	<code>findByAgeBetween(int from, int to)</code>	<code>{"age" : {"\$gt" : from, "\$lt" : to}}</code>

Keyword	Sample	Logical result
In	findByAgeIn(Collection ages)	{"age" : {"\$in" : [ages]}}
NotIn	findByAgeNotIn(Collection ages)	{"age" : {"\$nin" : [ages]}}
IsNull, NotNull	findByFirstnameNotNull()	{"firstname" : {"\$ne" : null}}
IsNull, Null	findByFirstnameNull()	{"firstname" : null}
Like	findByFirstnameLike(String name)	{"firstname" : name} (name as regex)
Regex	findByFirstnameRegex(String firstname)	{"firstname" : {"\$regex" : firstname }}
(No keyword)	findByFirstname(String name)	{"firstname" : name}
Not	findByFirstnameNot(String name)	{"firstname" : {"\$ne" : name}}
Near	findByLocationNear(Point point)	{"location" : {"\$near" : [x,y]}}
Near	findByLocationNear(Point point, Distance max)	{"location" : {"\$near" : [x,y], "\$maxDistance" : max}}
Near	findByLocationNear(Point point, Distance min, Distance max)	{"location" : {"\$near" : [x,y], "\$minDistance" : min, "\$maxDistance" : max}}
Within	findByLocationWithin(Circle circle)	{"location" : {"\$geoWithin" : {"\$center" : [[x, y], distance]}}
Within	findByLocationWithin(Box box)	{"location" : {"\$geoWithin" : {"\$box" : [[x1, y1], x2, y2]}}
IsTrue, True	findByActiveIsTrue()	{"active" : true}
IsFalse, False	findByActiveIsFalse()	{"active" : false}
Exists	findByLocationExists(boolean exists)	{"location" : {"\$exists" : exists }}

7.3.1. Repository delete queries

The above keywords can be used in conjunction with `delete By` or `remove By` to create queries deleting matching documents.

Example 28. Delete By Query

```
public interface PersonRepository extends MongoRepository<Person, String> {
    List <Person> deleteByLastname(String lastname);
    Long deletePersonByLastname(String lastname);
}
```

Using return type `List` will retrieve and return all matching documents before actually deleting them. A numeric return type directly removes the matching documents returning the total number of documents removed.

7.3.2. Geo-spatial repository queries

As you've just seen there are a few keywords triggering geo-spatial operations within a MongoDB query. The `Near` keyword allows some further modification. Let's have look at some examples:

Example 29. Advanced `Near` queries

```
public interface PersonRepository extends MongoRepository<Person, String>
{
    // { 'location' : { '$near' : [point.x, point.y], '$maxDistance' : distance}}
    List<Person> findByLocationNear(Point location, Distance distance);
}
```

Adding a `Distance` parameter to the query method allows restricting results to those within the given distance. If the `Distance` was set up containing a `Metric` we will transparently use `$nearSphere` instead of `$code`.

Example 30. Using `Distance` with `Metrics`

```
Point point = new Point(43.7, 48.8);
Distance distance = new Distance(200, Metrics.KILOMETERS);
= repository.findByLocationNear(point, distance);
// { 'location' : { '$nearSphere' : [43.7, 48.8], '$maxDistance' :
0.03135711885774796}}
```

As you can see using a `Distance` equipped with a `Metric` causes `$nearSphere` clause to be added instead of a plain `$near`. Beyond that the actual distance gets calculated according to the `Metrics` used.

NOTE Using `@GeoSpatialIndexed(type = GeoSpatialIndexType.GEO_2DSPHERE)` on the target property forces usage of `$nearSphere` operator.

Geo-near queries

```

public interface PersonRepository extends MongoRepository<Person, String>

// {'geoNear' : 'location', 'near' : [x, y] }
GeoResults<Person> findByLocationNear(Point location);

// No metric: {'geoNear' : 'person', 'near' : [x, y], maxDistance : distance }
// Metric: {'geoNear' : 'person', 'near' : [x, y], 'maxDistance' : distance,
//          'distanceMultiplier' : metric.multiplier, 'spherical' : true }
GeoResults<Person> findByLocationNear(Point location, Distance distance);

// Metric: {'geoNear' : 'person', 'near' : [x, y], 'minDistance' : min,
//          'maxDistance' : max, 'distanceMultiplier' : metric.multiplier,
//          'spherical' : true }
GeoResults<Person> findByLocationNear(Point location, Distance min, Distance max);

// {'geoNear' : 'location', 'near' : [x, y] }
GeoResults<Person> findByLocationNear(Point location);
}

```

7.3.3. MongoDB JSON based query methods and field restriction

By adding the annotation `org.springframework.data.mongodb.repository.Query` repository finder methods you can specify a MongoDB JSON query string to use instead of having the query derived from the method name. For example

```

public interface PersonRepository extends MongoRepository<Person, String>

@Query("{ 'firstname' : ?0 }")
List<Person> findByThePersonsFirstname(String firstname);

}

```

The placeholder `?0` lets you substitute the value from the method arguments into the JSON query string.

You can also use the `filter` property to restrict the set of properties that will be mapped into the Java object. For example,

```
public interface PersonRepository extends MongoRepository<Person, String>

    @Query(value="{ 'firstname' : ?0 }", fields="{ 'firstname' : 1, 'lastname' : 1}")
    List<Person> findByThePersonsFirstname(String firstname);

}
```

This will return only the firstname, lastname and Id properties of the Person objects. The age property, a java.lang.Integer, will not be set and its value will therefore be null.

7.3.4. Type-safe Query methods

MongoDB repository support integrates with the [QueryDSL](#) project which provides a means to perform type-safe queries in Java. To quote from the project description, "Instead of writing queries as inline strings or externalizing them into XML files they are constructed via a fluent API." It provides the following features

- Code completion in IDE (all properties, methods and operations can be expanded in your favorite Java IDE)
- Almost no syntactically invalid queries allowed (type-safe on all levels)
- Domain types and properties can be referenced safely (no Strings involved!)
- Adopts better to refactoring changes in domain types
- Incremental query definition is easier

Please refer to the QueryDSL documentation which describes how to bootstrap your environment for APT based code generation [using Maven](#) or [using Ant](#).

Using QueryDSL you will be able to write queries as shown below

```
QPerson person = new QPerson("person");
List<Person> result = repository.findAll(person.address.zipCode.eq("C0123"));

Page<Person> page = repository.findAll(person.lastname.contains("a"),
                                     new PageRequest(0, 2, Direction.ASC, "lastname"));
```

`QPerson` is a class that is generated (via the Java annotation post processing tool) which is a `Predicate` that allows you to write type safe queries. Notice that there are no strings in the query other than the value "C0123".

You can use the generated `Predicate` class via the interface `QueryDSLPredicateExecutor` which is shown below

```

public interface QueryDslPredicateExecutor<T> {

    T findOne(Predicate predicate);

    List<T> findAll(Predicate predicate);

    List<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);

    Page<T> findAll(Predicate predicate, Pageable pageable);

    Long count(Predicate predicate);
}

```

To use this in your repository implementation, simply inherit from it in addition to other repository interfaces. This is shown below

```

public interface PersonRepository extends MongoRepository<Person, String>,
QueryDslPredicateExecutor<Person> {

    // additional finder methods go here
}

```

We think you will find this an extremely powerful tool for writing MongoDB queries.

7.3.5. Full-text search queries

MongoDBs full text search feature is very store specific and therefore can rather be found on [MongoRepository](#) than on the more general [CrudRepository](#). What we need is a document with a full-text index defined for (Please see section [Text Indexes](#) for creating).

Additional methods on [MongoRepository](#) take [TextCriteria](#) as input parameter. In addition to those explicit methods, it is also possible to add a [TextCriteria](#) derived repository method. The criteria will be added as an additional **AND** criteria. Once the entity contains a [@TextScore](#) annotated property the document's full-text score will be retrieved. Furthermore the [@TextScore](#) annotated property will also make it possible to sort by the document's score.

```

@Document
class FullTextDocument {

    @Id String id;
    @TextIndexed String title;
    @TextIndexed String content;
    @TextScore Float score;
}

interface FullTextRepository extends Repository<FullTextDocument, String> {

    // Execute a full-text search and define sorting dynamically
    List<FullTextDocument> findAllBy(TextCriteria criteria, Sort sort);

    // Paginate over a full-text search result
    Page<FullTextDocument> findAllBy(TextCriteria criteria, Pageable pageable);

    // Combine a derived query with a full-text search
    List<FullTextDocument> findByTitleOrderByScoreDesc(String title, TextCriteria criteria
);
}

Sort sort = new Sort("score");
TextCriteria criteria = TextCriteria.forDefaultLanguage().matchingAny("spring", "data");
List<FullTextDocument> result = repository.findAllBy(criteria, sort);

criteria = TextCriteria.forDefaultLanguage().matching("film");
Page<FullTextDocument> page = repository.findAllBy(criteria, new PageRequest(1, 1, sort)
);
List<FullTextDocument> result = repository.findByTitleOrderByScoreDesc("mongodb",
criteria);

```

7.4. Miscellaneous

7.4.1. CDI Integration

Instances of the repository interfaces are usually created by a container, which Spring is the most natural choice when working with Spring Data. As of version 1.3.0 Spring Data MongoDB ships with a custom CDI extension that allows using the repository abstraction in CDI environments. The extension is part of the JAR so all you need to do to activate it is dropping the Spring Data MongoDB JAR into your classpath. You can now set up the infrastructure by implementing a CDI Producer for the `MongoTemplate`:


```

class MongoTemplateProducer {

    @Produces
    @ApplicationScoped
    public MongoOperations createMongoTemplate() throws UnknownHostException,
MongoException {

        MongoClientFactory factory = new SimpleMongoClientFactory(new MongoClient(), "database");
        return new MongoTemplate(factory);
    }
}

```

The Spring Data MongoDB CDI extension will pick up the `MongoTemplate` available as CDI bean and create a proxy for a Spring Data repository whenever an bean of a repository type is requested by the container. Thus obtaining an instance of a Spring Data repository is a matter of declaring an `@Inject`-ed property:

```

class RepositoryClient {

    @Inject
    PersonRepository repository;

    public void businessMethod() {
        List<Person> people = repository.findAll();
    }
}

```

7.5. General auditing configuration

Activating auditing functionality is just a matter of adding the Spring Data Mongo `auditing` namespace element to your configuration:

Example 31. Activating auditing using XML configuration

```

<mongo:auditing mapping-context-ref="customMappingContext" auditor-aware-ref=
"yourAuditorAwareImpl"/>

```

Since Spring Data MongoDB 1.4 auditing can be enabled by annotating a configuration class with the `@EnableMongoAuditing` annotation.

Example 32. Activating auditing using JavaConfig

```
@Configuration
@EnableMongoAuditing
class Config {

    @Bean
    public AuditorAware<AuditableUser> myAuditorProvider() {
        return new AuditorAwareImpl();
    }
}
```

If you expose a bean of type `AuditorAware` to the `ApplicationContext`, the auditing infrastructure will pick it up automatically and use it to determine the current user to be set on domain types. If you have multiple implementations registered in the `ApplicationContext`, you can select the one to be used by explicitly setting the `auditorAwareRef` attribute of `@EnableJpaAuditing`.

Chapter 8. Mapping

Rich mapping support is provided by the `MongoMappingConverter`. `MongoMappingConverter` has a rich metadata model that provides a full feature set of functionality to map domain objects to MongoDB documents. The mapping metadata model is populated using annotations on your domain objects. However, the infrastructure is not limited to using annotations as the only source of metadata information. The `MongoMappingConverter` also allows you to map objects to documents without providing any additional metadata, by following a set of conventions.

In this section we will describe the features of the `MongoMappingConverter`. How to use conventions for mapping objects to documents and how to override those conventions with annotation based mapping metadata.

NOTE `SimpleMongoConverter` has been deprecated in Spring Data MongoDB M3 as all of its functionality has been subsumed into `MappingMongoConverter`.

8.1. Convention based Mapping

`MongoMappingConverter` has a few conventions for mapping objects to documents when no additional mapping metadata is provided. The conventions are:

- The short Java class name is mapped to the collection name in the following manner. The class ‘com.bigbank.SavingsAccount’ maps to ‘savingsAccount’ collection name.
- All nested objects are stored as nested objects in the document and **not** as DBRefs
- The converter will use any Spring Converters registered with it to override the default mapping of object properties to document field/values.
- The fields of an object are used to convert to and from fields in the document. Public JavaBean properties are not used.
- You can have a single non-zero argument constructor whose constructor argument names match top level field names of document, that constructor will be used. Otherwise the zero arg constructor will be used. if there is more than one non-zero argument constructor an exception will be thrown.

8.1.1. How the '_id' field is handled in the mapping layer

MongoDB requires that you have an '_id' field for all documents. If you don't provide one the driver will assign a ObjectId with a generated value. The "_id" field can be of any type the, other than arrays, so long as it is unique. The driver naturally supports all primitive types and Dates. When using the `MongoMappingConverter` there are certain rules that govern how properties from the Java class is mapped to this '_id' field.

The following outlines what field will be mapped to the '_id' document field:

- A field annotated with `@Id` (`org.springframework.data.annotation.Id`) will be mapped to the `'_id'` field.
- A field without an annotation but named `'id'` will be mapped to the `'_id'` field.
- The default field name for identifiers is `'_id'` and can be customized via the `@Field` annotation.

Table 3. Examples for the translation of `'_id'`-field definitions

Field definition	Resulting Id-Fieldname in MongoDB
<code>String id</code>	<code>_id</code>
<code>@Field String id</code>	<code>_id</code>
<code>@Field('x') String id</code>	<code>x</code>
<code>@Id String x</code>	<code>_id</code>
<code>@Field('x') @Id String x</code>	<code>_id</code>

The following outlines what type conversion, if any, will be done on the property mapped to the `_id` document field.

- If a field named `'id'` is declared as a `String` or `BigInteger` in the Java class it will be converted to and stored as an `ObjectId` if possible. `ObjectId` as a field type is also valid. If you specify a value for `'id'` in your application, the conversion to an `ObjectId` is detected to the `MongoDBDriver`. If the specified `'id'` value cannot be converted to an `ObjectId`, then the value will be stored as is in the document's `_id` field.
- If a field named `'id'` id field is not declared as a `String`, `BigInteger`, or `ObjectID` in the Java class then you should assign it a value in your application so it can be stored 'as-is' in the document's `_id` field.
- If no field named `'id'` is present in the Java class then an implicit `'_id'` file will be generated by the driver but not mapped to a property or field of the Java class.

When querying and updating `MongoTemplate` will use the converter to handle conversions of the `Query` and `Update` objects that correspond to the above rules for saving documents so field names and types used in your queries will be able to match what is in your domain classes.

8.2. Mapping Configuration

Unless explicitly configured, an instance of `MongoMappingConverter` is created by default when creating a `MongoTemplate`. You can create your own instance of the `MappingMongoConverter` so as to tell it where to scan the classpath at startup your domain classes in order to extract metadata and construct indexes. Also, by creating your own instance you can register Spring converters to use for mapping specific classes to and from the database.

You can configure the `MongoMappingConverter` as well as `com.mongodb.Mongo` and `MongoTemplate` either using Java or XML based metadata. Here is an example using Spring's Java based configuration

Example 33. @Configuration class to configure MongoDB mapping support

```
@Configuration
public class GeoSpatialAppConfig extends AbstractMongoConfiguration {

    @Bean
    public Mongo mongo() throws Exception {
        return new Mongo("localhost");
    }

    @Override
    public String getDatabaseName() {
        return "database";
    }

    @Override
    public String getMappingBasePackage() {
        return "com.bigbank.domain";
    }

    // the following are optional

    @Bean
    @Override
    public CustomConversions customConversions() throws Exception {
        List<Converter<?, ?>> converterList = new ArrayList<Converter<?, ?>>();
        converterList.add(new org.springframework.data.mongodb.test.PersonReadConverter(
));
        converterList.add(new org.springframework.data.mongodb.test.PersonWriteConverter
());
        return new CustomConversions(converterList);
    }

    @Bean
    public LoggingEventListener<MongoMappingEvent> mappingEventsListener() {
        return new LoggingEventListener<MongoMappingEvent>();
    }
}
```

`AbstractMongoConfiguration` requires you to implement methods that define a `com.mongodb.Mongo` as well as provide a database name. `AbstractMongoConfiguration` also has a method you can override named 'getMappingBasePackage' which tells the converter where to scan for classes annotated with the `@org.springframework.data.mongodb.core.mapping.Document` annotation.

You can add additional converters to the converter by overriding the method `afterMappingMongoConverterCreation`. Also shown in the above example is a `LoggingEventListener` which logs `MongoMappingEvent`'s that are posted onto Spring's `ApplicationContextEvent` infrastructure.

NOTE

`AbstractMongoConfiguration` will create a `MongoTemplate` instance and registered with the container under the name 'mongoTemplate'.

You can also override the method `UserCredentials getUserCredentials()` to provide the username and password information to connect to the database.

Spring's MongoDB namespace enables you to easily enable mapping functionality in XML

Example 34. XML schema to configure MongoDB mapping support

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
  http://www.springframework.org/schema/data/mongo
http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd
  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <!-- Default bean name is 'mongo' -->
  <mongo:mongo host="localhost" port="27017"/>

  <mongo:db-factory dbname="database" mongo-ref="mongo"/>

  <!-- by default look for a Mongo object named 'mongo' - default name used for the
converter is 'mappingConverter' -->
  <mongo:mapping-converter base-package="com.bigbank.domain">
    <mongo:custom-converters>
      <mongo:converter ref="readConverter"/>
      <mongo:converter>
        <bean class="org.springframework.data.mongodb.test.PersonWriteConverter"/>
      </mongo:converter>
    </mongo:custom-converters>
  </mongo:mapping-converter>

  <bean id="readConverter" class=
"org.springframework.data.mongodb.test.PersonReadConverter"/>

  <!-- set the mapping converter to be used by the MongoTemplate -->
  <bean id="mongoTemplate" class="
org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
    <constructor-arg name="mappingConverter" ref="mappingConverter"/>
  </bean>

  <bean class=
"org.springframework.data.mongodb.core.mapping.event.LoggingEventListener"/>

</beans>
```

The `base-package` property tells it where to scan for classes annotated with the `@org.springframework.data.mongodb.core.mapping.Document` annotation.

8.3. Metadata based Mapping

To take full advantage of the object mapping functionality inside the Spring Data/MongoDB support, you should annotate your mapped objects with the `@org.springframework.data.mongodb.core.mapping.Document` annotation. Although it is not necessary for the mapping framework to have this annotation (your POJOs will be mapped correctly, even without any annotations), it allows the classpath scanner to find and pre-process your domain objects to extract the necessary metadata. If you don't use this annotation, your application will take a slight performance hit the first time you store a domain object because the mapping framework needs to build up its internal metadata model so it knows about the properties of your domain object and how to persist them.

Example 35. Example domain object

```
package com.mycompany.domain;

@Document
public class Person {

    @Id
    private ObjectId id;

    @Indexed
    private Integer ssn;

    private String firstName;

    @Indexed
    private String lastName;
}
```

IMPORTANT

The `@Id` annotation tells the mapper which property you want to use for the MongoDB `_id` property and the `@Indexed` annotation tells the mapping framework to call `createIndex()` on that property of your document, making searches faster.

IMPORTANT

Automatic index creation is only done for types annotated with `@Document`.

8.3.1. Mapping annotation overview

The `MappingMongoConverter` can use metadata to drive the mapping of objects to documents. An

overview of the annotations is provided below

- `@Id` - applied at the field level to mark the field used for identity purpose.
- `@Document` - applied at the class level to indicate this class is a candidate for mapping to the database. You can specify the name of the collection where the database will be stored.
- `@DBRef` - applied at the field level to indicate it is to be stored using a `com.mongodb.DBRef`.
- `@Indexed` - applied at the field level to describe how to index the field.
- `@CompoundIndex` - applied at the type level to declare Compound Indexes
- `@GeoSpatialIndexed` - applied at the field level to describe how to geospatially index the field.
- `@TextIndexed` - applied at the field level to mark the field to be included in the text index.
- `@Language` - applied at the field level to set the language override property for text index.
- `@Transient` - by default all private fields are mapped to the document, this annotation excludes the field where it is applied from being stored in the database
- `@PersistenceConstructor` - marks a given constructor - even a package protected one - to use when instantiating the object from the database. Constructor arguments are mapped by name to the key values in the retrieved `DBObject`.
- `@Value` - this annotation is part of the Spring Framework . Within the mapping framework it can be applied to constructor arguments. This lets you use a Spring Expression Language statement to transform a key's value retrieved in the database before it is used to construct a domain object. In order to reference a property of a given document one has to use expressions like: `@Value("#root.myProperty")` where `root` refers to the root of the given document.
- `@Field` - applied at the field level and describes the name of the field as it will be represented in the MongoDB BSON document thus allowing the name to be different than the fieldname of the class.

The mapping metadata infrastructure is defined in a separate spring-data-commons project that is technology agnostic. Specific subclasses are used in the MongoDB support to support annotation based metadata. Other strategies are also possible to put in place if there is demand.

Here is an example of a more complex mapping.

```

@Document
@CompoundIndexes({
    @CompoundIndex(name = "age_idx", def = "'lastName': 1, 'age': -1}")
})
public class Person<T extends Address> {

    @Id
    private String id;

    @Indexed(unique = true)
    private Integer ssn;

    @Field("fName")
    private String firstName;

    @Indexed
    private String lastName;

    private Integer age;

    @Transient
    private Integer accountTotal;

    @DBRef
    private List<Account> accounts;

    private T address;

    public Person(Integer ssn) {
        this.ssn = ssn;
    }

    @PersistenceConstructor
    public Person(Integer ssn, String firstName, String lastName, Integer age, T address) {
        this.ssn = ssn;
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.address = address;
    }

    public String getId() {
        return id;
    }

    // no setter for Id. (getter is only exposed for some unit testing)

```

```
public Integer getSsn() {
    return ssn;
}
```

```
// other getters/setters omitted
```

8.3.2. Customized Object Construction

The mapping subsystem allows the customization of the object construction by annotating a constructor with the `@PersistenceConstructor` annotation. The values to be used for the constructor parameters are resolved in the following way:

- If a parameter is annotated with the `@Value` annotation, the given expression is evaluated and the result is used as the parameter value.
- If the Java type has a property whose name matches the given field of the input document, then its property information is used to select the appropriate constructor parameter to pass the input field value to. This works only if the parameter name information is present in the java `.class` files which can be achieved by compiling the source with debug information or using the new `-parameters` command-line switch for javac in Java 8.
- Otherwise an `MappingException` will be thrown indicating that the given constructor parameter could not be bound.

```
class OrderItem {

    private @Id String id;
    private int quantity;
    private double unitPrice;

    OrderItem(String id, @Value("#root.qty ?: 0") int quantity, double unitPrice) {
        this.id = id;
        this.quantity = quantity;
        this.unitPrice = unitPrice;
    }

    // getters/setters omitted
}

DBObject input = new BasicDBObject("id", "4711");
input.put("unitPrice", 2.5);
input.put("qty", 5);
OrderItem item = converter.read(OrderItem.class, input);
```

NOTE | The SpEL expression in the `@Value` annotation of the `quantity` parameter falls back to the value `0` if the given property path cannot be resolved.

Additional examples for using the `@PersistenceConstructor` annotation can be found in the [MappingMongoConverterUnitTests](#) test suite.

8.3.3. Compound Indexes

Compound indexes are also supported. They are defined at the class level, rather than on individual properties.

NOTE | Compound indexes are very important to improve the performance of queries that involve criteria on multiple fields

Here's an example that creates a compound index of `lastName` in ascending order and `age` in descending order:

Example 36. Example Compound Index Usage

```
package com.mycompany.domain;

@Document
@CompoundIndexes({
    @CompoundIndex(name = "age_idx", def = "{ 'lastName': 1, 'age': -1 }")
})
public class Person {

    @Id
    private ObjectId id;
    private Integer age;
    private String firstName;
    private String lastName;

}
```

8.3.4. Text Indexes

NOTE | The text index feature is disabled by default for mongodb v.2.4.

Creating a text index allows to accumulate several fields into a searchable full text index. It is only possible to have one text index per collection so all fields marked with `@TextIndexed` are combined into this index. Properties can be weighted to influence document score for ranking results. The default language for the text index is english, to change the default language set `@Document(language="spanish")` to any language you want. Using a property called `language` or `@Language` allows to define a language override on a per document base.

Example 37. Example Text Index Usage

```
@Document(language = "spanish")
class SomeEntity {

    @TextIndexed String foo;

    @Language String lang;

    Nested nested;
}

class Nested {

    @TextIndexed(weight=5) String bar;
    String roo;
}
```

8.3.5. Using DBRefs

The mapping framework doesn't have to store child objects embedded within the document. You can also store them separately and use a DBRef to refer to that document. When the object is loaded from MongoDB, those references will be eagerly resolved and you will get back a mapped object that looks the same as if it had been stored embedded within your master document.

Here's an example of using a DBRef to refer to a specific document that exists independently of the object in which it is referenced (both classes are shown in-line for brevity's sake):

```

@Document
public class Account {

    @Id
    private ObjectId id;
    private Float total;
}

@Document
public class Person {

    @Id
    private ObjectId id;
    @Indexed
    private Integer ssn;
    @DBRef
    private List<Account> accounts;
}

```

There's no need to use something like `@OneToMany` because the mapping framework sees that you're wanting a one-to-many relationship because there is a `List` of objects. When the object is stored in MongoDB, there will be a list of DBRefs rather than the `Account` objects themselves.

IMPORTANT

The mapping framework does not handle cascading saves. If you change an `Account` object that is referenced by a `Person` object, you must save the `Account` object separately. Calling `save` on the `Person` object will not automatically save the `Account` objects in the property `accounts`.

8.3.6. Mapping Framework Events

Events are fired throughout the lifecycle of the mapping process. This is described in the [Lifecycle Events](#) section.

Simply declaring these beans in your Spring `ApplicationContext` will cause them to be invoked whenever the event is dispatched.

8.3.7. Overriding Mapping with explicit Converters

When storing and querying your objects it is convenient to have a `MongoConverter` instance handle the mapping of all Java types to `DBObject`s. However, sometimes you may want the `MongoConverter`'s do most of the work but allow you to selectively handle the conversion for a particular type or to optimize performance.

To selectively handle the conversion yourself, register one or more one or more

`org.springframework.core.convert.converter.Converter` instances with the `MongoConverter`.

NOTE Spring 3.0 introduced a `core.convert` package that provides a general type conversion system. This is described in detail in the Spring reference documentation section entitled [Spring 3 Type Conversion](#).

The method `customConversions` in `AbstractMongoConfiguration` can be used to configure Converters. The examples [here](#) at the beginning of this chapter show how to perform the configuration using Java and XML.

Below is an example of a Spring Converter implementation that converts from a `DBObject` to a `Person` POJO.

```
@ReadingConverter
public class PersonReadConverter implements Converter<DBObject, Person> {

    public Person convert(DBObject source) {
        Person p = new Person((ObjectId) source.get("_id"), (String) source.get("name"));
        p.setAge((Integer) source.get("age"));
        return p;
    }
}
```

Here is an example that converts from a `Person` to a `DBObject`.

```
@WritingConverter
public class PersonWriteConverter implements Converter<Person, DBObject> {

    public DBObject convert(Person source) {
        DBObject dbo = new BasicDBObject();
        dbo.put("_id", source.getId());
        dbo.put("name", source.getFirstName());
        dbo.put("age", source.getAge());
        return dbo;
    }
}
```

Chapter 9. Cross Store support

Sometimes you need to store data in multiple data stores and these data stores can be of different types. One might be relational while the other a document store. For this use case we have created a separate module in the MongoDB support that handles what we call cross-store support. The current implementation is based on JPA as the driver for the relational database and we allow select fields in the Entities to be stored in a Mongo database. In addition to allowing you to store your data in two stores we also coordinate persistence operations for the non-transactional MongoDB store with the transaction life-cycle for the relational database.

9.1. Cross Store Configuration

Assuming that you have a working JPA application and would like to add some cross-store persistence for MongoDB. What do you have to add to your configuration?

First of all you need to add a dependency on the module. Using Maven this is done by adding a dependency to your pom:

Example 38. Example Maven pom.xml with spring-data-mongodb-cross-store dependency

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  ...

  <!-- Spring Data -->
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb-cross-store</artifactId>
    <version>${spring.data.mongo.version}</version>
  </dependency>

  ...

</project>
```

Once this is done we need to enable AspectJ for the project. The cross-store support is implemented using AspectJ aspects so by enabling compile time AspectJ support the cross-store features will become available to your project. In Maven you would add an additional plugin to the <build> section of the pom:

Example 39. Example Maven pom.xml with AspectJ plugin enabled



```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  ...

  <build>
    <plugins>

      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>aspectj-maven-plugin</artifactId>
        <version>1.0</version>
        <dependencies>
          <!-- NB: You must use Maven 2.0.9 or above or these are ignored (see MNG-
2972) -->
          <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>${aspectj.version}</version>
          </dependency>
          <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>${aspectj.version}</version>
          </dependency>
        </dependencies>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>test-compile</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <outxml>>true</outxml>
          <aspectLibraries>
            <aspectLibrary>
              <groupId>org.springframework</groupId>
              <artifactId>spring-aspects</artifactId>
            </aspectLibrary>
          </aspectLibraries>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>

```

```
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-mongodb-cross-store</artifactId>
    </aspectLibrary>
</aspectLibraries>
    <source>1.6</source>
    <target>1.6</target>
</configuration>
</plugin>

...

</plugins>
</build>

...

</project>
```

Finally, you need to configure your project to use MongoDB and also configure the aspects that are used. The following XML snippet should be added to your application context:

Example 40. Example application context with MongoDB and cross-store aspect support

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd">
  ...

  <!-- Mongo config -->
  <mongo:mongo host="localhost" port="27017"/>

  <bean id="mongoTemplate" class="
org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg name="mongo" ref="mongo"/>
    <constructor-arg name="databaseName" value="test"/>
    <constructor-arg name="defaultCollectionName" value="cross-store"/>
  </bean>

  <bean class="org.springframework.data.mongodb.core.MongoExceptionTranslator"/>

  <!-- Mongo cross-store aspect config -->
  <bean class=
"org.springframework.data.persistence.document.mongo.MongoDocumentBacking"
    factory-method="aspectOf">
    <property name="changeSetPersister" ref="mongoChangeSetPersister"/>
  </bean>
  <bean id="mongoChangeSetPersister"
    class="org.springframework.data.persistence.document.mongo.MongoChangeSetPersis
ter">
    <property name="mongoTemplate" ref="mongoTemplate"/>
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>
  ...

</beans>
```

9.2. Writing the Cross Store Application

We are assuming that you have a working JPA application so we will only cover the additional steps needed to persist part of your Entity in your Mongo database. First you need to identify the field you want persisted. It should be a domain class and follow the general rules for the Mongo mapping support covered in previous chapters. The field you want persisted in MongoDB should be annotated using the `@RelatedDocument` annotation. That is really all you need to do!. The cross-store aspects take care of the rest. This includes marking the field with `@Transient` so it won't be persisted using JPA, keeping track of any changes made to the field value and writing them to the database on successful transaction completion, loading the document from MongoDB the first time the value is used in your application. Here is an example of a simple Entity that has a field annotated with `@RelatedEntity`.

Example 41. Example of Entity with @RelatedDocument

```
@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstName;

    private String lastName;

    @RelatedDocument
    private SurveyInfo surveyInfo;

    // getters and setters omitted
}
```

Example 42. Example of domain class to be stored as document

```
public class SurveyInfo {

    private Map<String, String> questionsAndAnswers;

    public SurveyInfo() {
        this.questionsAndAnswers = new HashMap<String, String>();
    }

    public SurveyInfo(Map<String, String> questionsAndAnswers) {
        this.questionsAndAnswers = questionsAndAnswers;
    }

    public Map<String, String> getQuestionsAndAnswers() {
        return questionsAndAnswers;
    }

    public void setQuestionsAndAnswers(Map<String, String> questionsAndAnswers) {
        this.questionsAndAnswers = questionsAndAnswers;
    }

    public SurveyInfo addQuestionAndAnswer(String question, String answer) {
        this.questionsAndAnswers.put(question, answer);
        return this;
    }
}
```

Once the SurveyInfo has been set on the Customer object above the MongoTemplate that was configured above is used to save the SurveyInfo along with some metadata about the JPA Entity is stored in a MongoDB collection named after the fully qualified name of the JPA Entity class. The following code:

Example 43. Example of code using the JPA Entity configured for cross-store persistence

```
Customer customer = new Customer();
customer.setFirstName("Sven");
customer.setLastName("Olafsen");
SurveyInfo surveyInfo = new SurveyInfo()
    .addQuestionAndAnswer("age", "22")
    .addQuestionAndAnswer("married", "Yes")
    .addQuestionAndAnswer("citizenship", "Norwegian");
customer.setSurveyInfo(surveyInfo);
customerRepository.save(customer);
```

Executing the code above results in the following JSON document stored in MongoDB.

Example 44. Example of JSON document stored in MongoDB

```
{ "_id" : ObjectId( "4d9e8b6e3c55287f87d4b79e" ),
  "_entity_id" : 1,
  "_entity_class" :
  "org.springframework.data.mongodb.examples.custsvc.domain.Customer",
  "_entity_field_name" : "surveyInfo",
  "questionsAndAnswers" : { "married" : "Yes",
    "age" : "22",
    "citizenship" : "Norwegian" },
  "_entity_field_class" :
  "org.springframework.data.mongodb.examples.custsvc.domain.SurveyInfo" }
```


Chapter 10. Logging support

An appender for Log4j is provided in the maven module "spring-data-mongodb-log4j". Note, there is no dependency on other Spring Mongo modules, only the MongoDB driver.

10.1. MongoDB Log4j Configuration

Here is an example configuration

```
log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.springframework.data.document.mongodb.log4j.MongoLog4jAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <%m>%n
log4j.appender.stdout.host = localhost
log4j.appender.stdout.port = 27017
log4j.appender.stdout.database = logs
log4j.appender.stdout.collectionPattern = %X{year}%X{month}
log4j.appender.stdout.applicationId = my.application
log4j.appender.stdout.warnOrHigherWriteConcern = FSYNC_SAFE

log4j.category.org.apache.activemq=ERROR
log4j.category.org.springframework.batch=DEBUG
log4j.category.org.springframework.data.document.mongodb=DEBUG
log4j.category.org.springframework.transaction=INFO
```

The important configuration to look at aside from host and port is the database and collectionPattern. The variables year, month, day and hour are available for you to use in forming a collection name. This is to support the common convention of grouping log information in a collection that corresponds to a specific time period, for example a collection per day.

There is also an applicationId which is put into the stored message. The document stored from logging as the following keys: level, name, applicationId, timestamp, properties, traceback, and message.

Chapter 11. JMX support

The JMX support for MongoDB exposes the results of executing the 'serverStatus' command on the admin database for a single MongoDB server instance. It also exposes an administrative MBean, MongoAdmin which will let you perform administrative operations such as drop or create a database. The JMX features build upon the JMX feature set available in the Spring Framework. See [here](#) for more details.

11.1. MongoDB JMX Configuration

Spring's Mongo namespace enables you to easily enable JMX functionality

Example 45. XML schema to configure MongoDB

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <!-- Default bean name is 'mongo' -->
  <mongo:mongo host="localhost" port="27017"/>

  <!--by default look for a Mongo object named 'mongo' -->
  <mongo:jmx/>

  <context:mbean-export/>

  <!-- To translate any MongoExceptions thrown in @Repository annotated classes -->
  <context:annotation-config/>

  <bean id="registry" class=
"org.springframework.remoting.rmi.RmiRegistryFactoryBean" p:port="1099" />

  <!-- Expose JMX over RMI -->
  <bean id="serverConnector" class=
"org.springframework.jmx.support.ConnectorServerFactoryBean"
    depends-on="registry"
    p:objectName="connector:name=rmi"
    p:serviceUrl="service:jmx:rmi:///localhost/jndi/rmi:///localhost:1099/myconnect
or" />

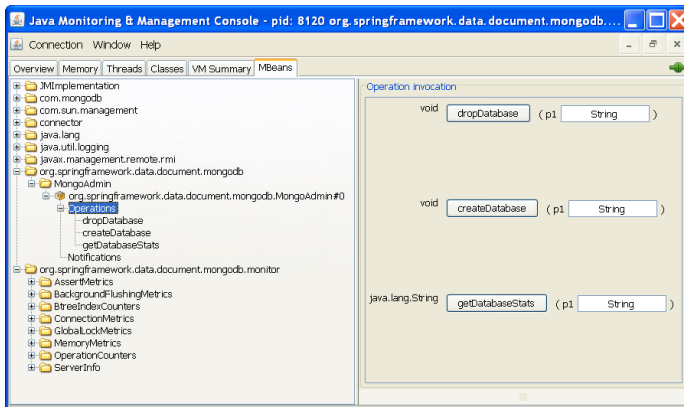
</beans>
```

This will expose several MBeans

- AssertMetrics
- BackgroundFlushingMetrics
- BtreeIndexCounters

- ConnectionMetrics
- GlobalLockMetrics
- MemoryMetrics
- OperationCounters
- ServerInfo
- MongoAdmin

This is shown below in a screenshot from JConsole



Chapter 12. MongoDB 3.0 Support

Spring Data MongoDB allows usage of both MongoDB Java driver generations 2 and 3 when connecting to a MongoDB 2.6/3.0 server running *MMap.v1* or a MongoDB server 3.0 using *MMap.v1* or the *WiredTiger* storage engine.

NOTE Please refer to the driver and database specific documentation for major differences between those.

NOTE Operations that are no longer valid using a 3.x MongoDB Java driver have been deprecated within Spring Data and will be removed in a subsequent release.

12.1. Using Spring Data MongoDB with MongoDB 3.0

12.1.1. Configuration Options

Some of the configuration options have been changed / removed for the *mongo-java-driver*. The following options will be ignored using the generation 3 driver:

- `autoConnectRetry`
- `maxAutoConnectRetryTime`
- `slaveOk`

Generally it is recommended to use the `<mongo:mongo-client />` and `<mongo:client-options />` elements instead of `<mongo:mongo />` when doing XML based configuration, since those elements will only provide you with attributes valid for the 3 generation java driver.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:mongo="http://www.springframework.org/schema/data/mongo"
xsi:schemaLocation="http://www.springframework.org/schema/data/mongo
http://www.springframework.org/schema/data/mongo/spring-mongo.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <mongo:mongo-client host="127.0.0.1" port="27017">
    <mongo:client-options write-concern="NORMAL" />
  </mongo:mongo-client>

</beans>
```

12.1.2. WriteConcern and WriteConcernChecking

The `WriteConcern.NONE`, which had been used as default by Spring Data MongoDB, was removed in 3.0. Therefore in a MongoDB 3 environment the `WriteConcern` will be defaulted to `WriteConcern.UNACKNOWLEDGED`. In case `WriteResultChecking.EXCEPTION` is enabled the `WriteConcern` will be altered to `WriteConcern.ACKNOWLEDGED` for write operations, as otherwise errors during execution would not be throw correctly, since simply not raised by the driver.

12.1.3. Authentication

MongoDB Server generation 3 changed the authentication model when connecting to the DB. Therefore some of the configuration options available for authentication are no longer valid. Please use the `MongoClient` specific options for setting credentials via `MongoCredential` to provide authentication data.

```
@Configuration
public class ApplicationContextEventTestsAppConfig extends AbstractMongoConfiguration {

    @Override
    public String getDatabaseName() {
        return "database";
    }

    @Override
    @Bean
    public Mongo mongo() throws Exception {
        return new MongoClient(singletonList(new ServerAddress("127.0.0.1", 27017)),
            singletonList(MongoCredential.createCredential("name", "db", "pwd".toCharArray())))
    };
}
```

In order to use authentication with XML configuration use the `credentials` attribute on `<mongo-client>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:mongo="http://www.springframework.org/schema/data/mongo"
xsi:schemaLocation="http://www.springframework.org/schema/data/mongo
http://www.springframework.org/schema/data/mongo/spring-mongo.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <mongo:mongo-client credentials="user:password@database" />

</beans>
```

12.1.4. Other things to be aware of

This section covers additional things to keep in mind when using the 3.0 driver.

- `IndexOperations.resetIndexCache()` is no longer supported.
- Any `MapReduceOptions.extraOption` is silently ignored.
- `WriteResult` does not longer hold error informations but throws an Exception.
- `MongoOperations.executeInSession()` no longer calls `requestStart / requestDone`.
- Index name generation has become a driver internal operations, still we use the 2.x schema to generate names.
- Some Exception messages differ between the generation 2 and 3 servers as well as between *MMap.v1* and *WiredTiger* storage engine.

Appendix