Tutorial:

MongoDB and MongoDB integration with Java

(MongoDB Java driver and Spring Data for MongoDB)

Revisions:	
10.11.2017	First Edition

Table of Contents

About the tutorial

The tutorial is a combined collection of articles, examples and other published posts found on the web while trying to understand the DB architecture and the integration of the DB with the Java programming language. The tutorial contains material (mostly copy-pasted) about

- MongoDB
- MongoDB Java driver
- Spring Data MongoDB.

The tutorial covers the core ideas of the above issues alongside practical information relevant to our interests (e.g handling geolocation data, installing on Amazon Linux platform…)

Mongo DB Overview

General information

MongoDB is a free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with schemas. MongoDB is developed by MongoDB Inc., and is published under a combination of the GNU Affero General Public License and the Apache License.

Layers of data

- First layer – database: the database is spread out among several files and holds as a physical container for collections.
- Second layer - collection: Collection is the equivalent of table in the traditional RDBMS, but opposed to that it does not enforce a fix schema. Documents within a collection can have different fields (useful?).
- Third layer - Document: a set of key-value pairs, equivalent of RDBMS' row.

The data is saved in JSON format and documents in the same collection do not need to have the same structure.
For example, a collection of apartments:

```
{
         _id: APARTMENT_ID
         author: USER_1
         picture: 'house.jpg'
          price: 1500
         address:
           {
                  city: 'HAIFA',
                  street: 'TRUMPELDOR'
                  number: 10
           }
         description:  'looking for another friendly roommate for the best
apartment in town!'
         likes: 2
         tags: [AIR_CONDITION, NO_SMOKING, NO_PETS]
         comments: [
           {
                  User: USER_2
                  Message: 'how far is it from the closest market?'
           },
           {
```

```
                User: USER_1,
                Message: '2 minutes by foot'
            }
        ]
}
```

Key Features

- High Performance - MongoDB provides high performance data persistence. In particular, Support for embedded data models reduces I/O activity on database system.Indexes support faster queries and can include keys from embedded documents and arrays.
- Rich Query Language - MongoDB supports a rich query language to support read and write operations (CRUD) as well as: Data Aggregation and Text Search and Geospatial Queries.
- High Availability - MongoDB's replication facility, called replica set, provides: *automatic* failover and data redundancy. A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability.
- Horizontal Scalability - MongoDB provides horizontal scalability as part of its *core* functionality: Sharding distributes data across a cluster of machines. MongoDB 3.4 supports creating zones of data based on the shard key. In a balanced cluster, MongoDB directs reads and writes covered by a zone only to those shards inside the zone. See the Zones manual page for more information.
- Support for Multiple Storage Engines: MongoDB supports multiple storage engines, such as: WiredTiger Storage Engine and MMAPv1 Storage Engine.

Getting started with MongoDB

The packages of MongoDB

Package Name	Description
mongodb-org	A metapackage that will automatically install the four component packages listed below.
mongodb-org-server	Contains the mongod daemon and associated configuration and init scripts. The mongodb-org-server package provides an initialization script that starts mongod with the /etc/mongod.conf configuration file.
mongodb-org-mongos	Contains the mongos daemon.
mongodb-org-shell	Contains the mongo shell.
mongodb-org-tools	Contains the following MongoDB tools: mongoimport bsondump, mongodump, mongoexport, m

	ongofiles, mongooplog, mongoperf, mongorestore, mongostat, and mongotop.

Installation

Amazon Linux 64bit

1. Configure the package management system:
   Create a /etc/yum.repos.d/mongodb-org-3.4.repo file so that you can install MongoDB directly, using yum e.g. for the latest stable release use:
   [mongodb-org-3.4]
   name=MongoDB Repository
   baseurl=https://repo.mongodb.org/yum/amazon/2013.03/mongodb-org/3.4/x86_64/
   gpgcheck=1
   enabled=1
   gpgkey=https://www.mongodb.org/static/pgp/server-3.4.asc

2. Install the packages and associated tools:
   sudo yum install -y mongodb-org

Service Management and Monitoring

Description	Command/Action
Start/stop/restart the mongod service.	sudo service mongod start/stop/restart
verify that the mongod process has started successfully.	check the contents of the log file at /var/log/mongodb/mongod.log
Ensure that MongoDB will start following a system reboot.	sudo chkconfig mongod on

Integration with Java

Direct mode – Working with the Java Driver

Download

Maven

```
<dependencies>

        <dependency>

                <groupId>org.mongodb</groupId>

                <artifactId>mongodb-driver</artifactId> <version>3.5.0</version>

        </dependency>

</dependencies>
```

Gradle

```
dependencies { compile 'org.mongodb:mongodb-driver:3.5.0' }
```

Connecting the MongoDB

MongoClient mongoClient = new MongoClient(new MongoClientURI("mongodb://localhost:27017"))

If you're connecting to a local instance on the default port, you can simply use:

MongoClient mongoClient = new MongoClient();

Note! it is important to limit the number of MongoClient instances in your application, hence we suggest a singleton. Using a single MongoClient (and optionally configuring its settings) will allow the driver to correctly manage your connections to the server.

This MongoClient singleton is safe to be used by multiple threads.

One final thing you need to be aware of: you want your application to shut down the connections to MongoDB when it finishes running. Always make sure your application or web server calls MongoClient.close() when it shuts down.

Creating/Getting a DB

Creating and getting a database or collection is extremely easy in MongoDB:

DB database = mongoClient.getDB("TheDatabaseName");

If the database doesn't already exist, it will be created automatically the first time you insert anything into it, so there's no need for null checks or exception handling on the off-chance the database doesn't exist.

Getting a Collection

Getting the collection you want from the database is simple too:

DBCollection collection = database.getCollection("TheCollectionName");

Creating a Document and Saving It in a Collection

```
DBObject person = new BasicDBObject("_id", "jo").append("name", "Jo Bloggs")
                                                .append("address", new BasicDBObject("street", "123 Fake St")
                                                .append("city", "Faketon")
                                                .append("state", "MA")
                                                .append("zip", 12345))
                                                .append("books", books);
```

At this point, it's really easy to save it into your database:

MongoClient mongoClient = new MongoClient();

DB database = mongoClient.getDB("Examples");

DBCollection collection = database.getCollection("people");

collection.insert(person);

Will result with the following doc:

```
{
    "_id" : "jo",
    "name" : "Jo Bloggs",
    "age": 34,
    "address" : {
        "street" : "123 Fake St",
        "city" : "Faketon",
        "state" : "MA",
        "zip" : "12345"
    },
    "books" : [
        27464,
        747854
    ]
}
```

Convert Domain Object into a DBObject Design Pattern

you can see the similarities between the Document that's stored in MongoDB, and your domain object. In your code, that person would probably be a Person class, with simple primitive fields, an array field, and an Address field.

So rather than building your DBObject manually like the above example, you're more likely to be converting your domain object into a DBObject. It's best not to have the MongoDB-specific DBObject class in your domain objects, so you might want to create a PersonAdaptor that converts your Person domain object to a DBObject:

Public static final DBObject toDBObject(Person person){

return new BasicDBObject("_id", person.getId()).append("name", person.getName())

.append("address", new BasicDBObject("street", person.getAddress().getStreet())

.append("city",person.getAddress().getTown())

.append("phone", person.getAddress().getPhone()))

.append("books", person.getBookIds());

}

Querying

Retrieving a Document

By the fact that MongoDB is a document database that we're not going to be using SQL to query. Instead, we query by example, building up a document that looks like the document we're looking for. So, if we wanted to look for the person we saved into the database, "Jo Bloggs", we remember that the _id field had the value of "jo", and we create a document that matches this shape:

DBObject query = new BasicDBObject("_id", "jo");

DBCursor cursor = collection.find(query);

the find method returns a cursor for the results. Since _id needs to be unique, we know that if we look for a document with this ID, we will find only one document, and it will be the one we want:

```
DBObject jo = cursor.one();
```

If we wanted to look at the fields of the document we got back from the database, we can get them with:

```
(String)cursor.one().get("name");
```

Note that you'll need to cast the value to a String, as the compiler only knows that it's an Object.

Find documents by a not unique field –

```
DBCursor results = collection.find(new BasicDBObject("name", "The name I want to find"));
```

And than we can iterate over the results with a for loop: `for (DBObject result : results)`.

passing a second parameter into the find method that's another DBObject defining the fields you want to return.

DBCursor results = collection.find(new BasicDBObject("name", "SomeName"), new BasicDBObject("name", 1));

You can also use this method to exclude fields from the results:

DBCursor results = collection.find(new BasicDBObject("name", "SomeName"), new BasicDBObject("name", 0));

Query Operators

Numeric greater than operator:

DBCursor results = collection.find(new BasicDBObject("numberOfOrders", new BasicDBObject("$gt", 10)));

Available operators (examples):

Name	Description
$eq	Matches values that are equal to a specified value.
$gt	Matches values that are greater than a specified value.
$gte	Matches values that are greater than or equal to a specified value.
$in	Matches any of the values specified in an array.

$lt	Matches values that are less than a specified value.
$lte	Matches values that are less than a specified value.
$ne	Matches all values that are not equal to a specified value.
$nin	Matches none of the values specified in an array.

Full list can be found here: https://docs.mongodb.com/manual/reference/operator/query/

Geolocation Querying

To support geospatial queries, MongoDB provides various geospatial indexes as well as geospatial query operators (documented in the list above).

Creating 2dsphare index

To create a 2dsphere index, use the Indexes.geo2dsphere helper to create a specification for the 2dsphere index and pass to MongoCollection.createIndex() method.

The following example creates a 2dsphere index on the "contact.location" field for the restaurantscollection.

```
MongoCollection<Document> collection = database.getCollection("restaurants");

collection.createIndex(Indexes.geo2dsphere("contact.location"));
```

Querying for Locations Near a GeoJSON Point

MongoDB provides various geospatial query operators. To facilitate the creation of geospatial queries filters, the Java driver provides the Filters class and the com.mongodb.client.model.geojsonpackage.

The following example returns documents that are at least 1000 meters from and at most 5000 meters from the specified GeoJSON point com.mongodb.client.model.geojson.Point, sorted from nearest to farthest:

```
Point refPoint = new Point(new Position(-73.9667, 40.78));

collection.find(Filters.near("contact.location", refPoint, 5000.0, 1000.0)).forEach(printBlock);
```

Querying Subdocuments

we might want to query for values in a subdocument - for example, with our person document, we might want to find everyone who lives in the same city. We can use dot notation like this:

```
DBObject findLondoners = new DBObject("address.city", "London");

collection.find(findLondoners));
```

Updating Values

Firstly, by default only the first document that matches the query criteria is updated.

Secondly, if you pass in a document as the value to update to, this new document will replace the whole existing document.

DBObject jo = …// get the document representing jo

jo.put("name", "Jo In Disguise"); // replace the old name with the new one

collection.update(new BasicDBObject("_id", "jo"), // find jo by ID jo); // set the document in the DB to the new document for Jo

But sometimes you won't have the whole document to replace the old one, sometimes you just want to update a single field in whichever document matched your criteria.

Let's imagine that we only want to change Jo's phone number, and we don't have a DBObject with all of Jo's details but we do have the ID of the document. If we use the $set operator, we'll replace only the field we want to change:

collection.update(new BasicDBObject("_id", "jo"), new BasicDBObject("$set", new BasicDBObject("phone", "5559874321")));

Updating Multiple Documents

As I mentioned earlier, by default the update operation updates the first document it finds and no more. You can, however, set the multi flag on update to update everything.

collection.update(new BasicDBObject(), new BasicDBObject("$set", new BasicDBObject("country", "UK")), false, true);

The query parameter is an empty document which finds everything; the second boolean (set to true) is the flag that says to update all the values which were found.

Update-or-Insert

This will search for a document matching the criteria and either: update it if it's there; or insert it into the database if it wasn't.

collection.update(query, personDocument, true, false);

Remove

you pass a document that represents your selection criteria into the remove method. So, if we wanted to delete jo from our database, we'd do:

collection.remove(new BasicDBObject("_id", "jo"));

Unlike update, if the query matches more than one document, all those documents will be deleted

Indexes

We can programmatically create indexes via the Java driver, using createIndexes. For example:

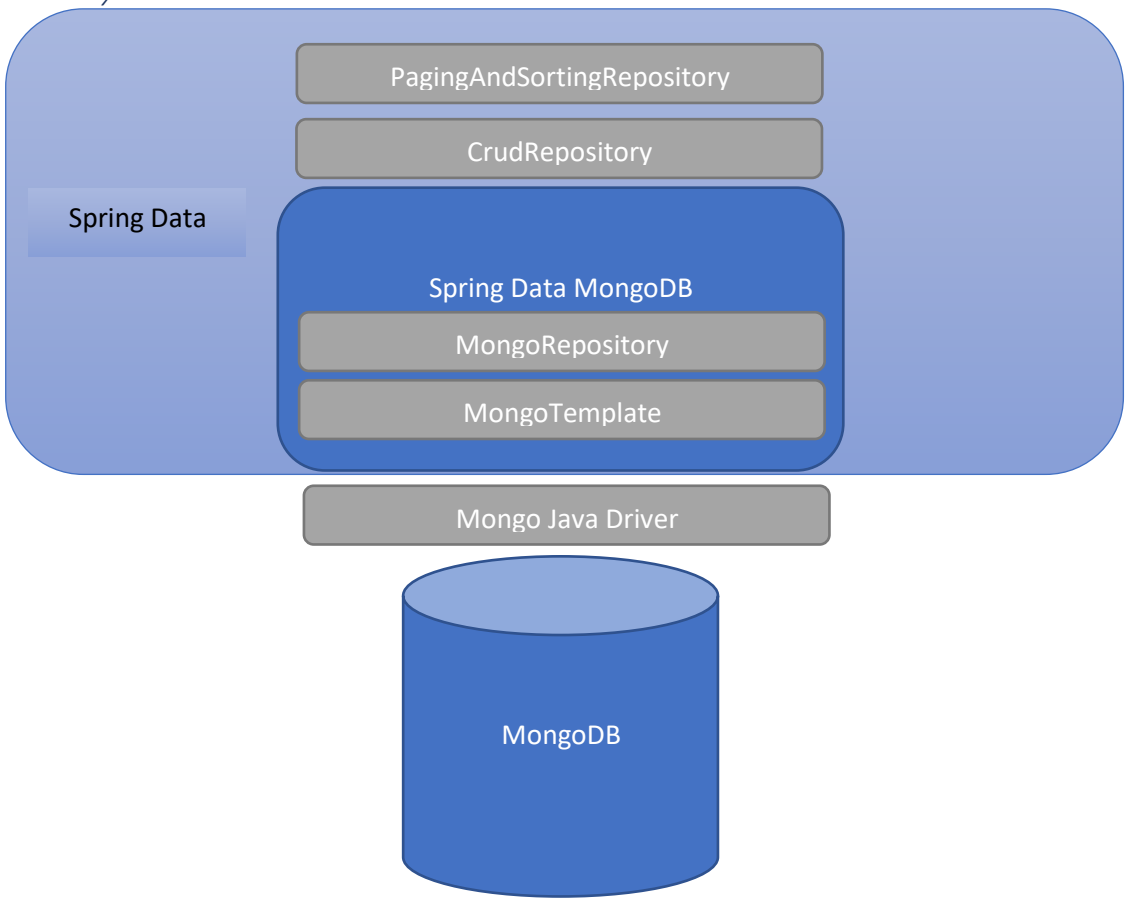
collection.createIndex(new BasicDBObject("fieldToIndex", 1));

Abstract mode – Working with Spring Data MongoDB

General

Spring Data is a high level SpringSource project whose purpose is to unify and ease the access to different kinds of persistence stores, both relational database systems and NoSQL data stores. Spring Data provides generic interfaces (CrudRepository, PagingAndSortingRepository) for the aspects of: CRUD (Create-Read-Update-Delete) operations on single domain objects, finder methods, sorting and pagination. Spring Data MongoDB comes with tons of features for the Java developers working with MongoDB: database and collection management, lightweight object-document mapping, and dynamic repositories are some of these features.

Layers



- MongoDB – the MongoDB itself
- MongoDB Java Driver – the official Java Driver of MongoDB (the one discussed in the previous part of the tutorial).

- MongoTemplate - A template interface called MongoTemplate is a high-level abstraction for storing and querying documents and its super interface called MongoOperations. You will find this approach familiar if you have been using the JDBC support in the Spring framework. MongoTemplate is the central support class for Mongo database operations. It provides:
    - Basic POJO mapping support to and from BSON
    - Convenience methods to interact with the store (insert object, update objects) and MongoDB specific ones (geo-spatial operations, upserts, map-reduce etc.)
    - Connection affinity callback
    - Exception translation into Spring's technology agnostic DAO exception hierarchy.

- MongoRepository - interface acts as a marker place to capture the document model, thus, providing a convenient way to derive DB statements directly from the field name of your documents. To simplify the creation of data repositories Spring Data MongoDB provides a generic repository programming model. It will automatically create a repository proxy for you that adds implementations of finder methods you specify on an interface.
- CrudRepository - The central interface in Spring Data repository abstraction is Repository. It is typeable to the domain class to manage as well as the id type of the domain class. Beyond that there's CrudRepository which provides some sophisticated functionality around CRUD for the entity being managed.

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    T save(T entity);

    T findOne(ID primaryKey);

    Iterable<T> findAll();

    Long count();

    void delete(T entity);

    boolean exists(ID primaryKey);

    // … more functionality omitted.
}
```

- PagingAndSortingRepository - On top of the CrudRepository there is a PagingAndSortingRepository abstraction that adds additional methods to ease paginated access to entities.

```
public interface PagingAndSortingRepository<T, ID extends Serializable> extends
CrudRepository<T, ID> {

    Iterable<T> findAll(Sort sort);

    Page<T> findAll(Pageable pageable);
```

```
}
```

Download

Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-mongodb</artifactId>
        <version>2.0.1.RELEASE</version>
    </dependency>
</dependencies><repositories>
    <repository>
        <id>spring-libs-release</id>
        <name>Spring Releases</name>
        <url>https://repo.spring.io/libs-release</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
```

Gradle

```
dependencies {
    compile 'org.springframework.data:spring-data-mongodb:2.0.1.RELEASE'
}repositories {
    maven {
        url 'https://repo.spring.io/libs-release'
    }
}
```

Querying

Next to standard CRUD functionality repositories are usually queries on the underlying datastore. With Spring Data declaring those queries becomes a four-step process:

1. *Defining Repository Interface*

   Declare an interface extending Repository or one of its sub-interfaces and type it to the domain class it shall handle.

   ```
   public interface PersonRepository extends Repository<User, Long> { … }
   ```

   As a very first step you define a domain class specific repository interface. It's got to extend Repository and be typed to the domain class and an ID type. If you want to expose CRUD methods for that domain type, extend CrudRepository instead of Repository. Usually you will have your repository interface

extend Repository, CrudRepository or PagingAndSortingRepository. If you don't like extending Spring Data interfaces at all you can also annotate your repository interface with @RepositoryDefinition. Extending CrudRepository will expose a complete set of methods to manipulate your entities. If you would rather be selective about the methods being exposed, simply copy the ones you want to expose from CrudRepository into your domain repository.

2. *Defining Query Methods*

Declare query methods on the interface.

```
List<Person> findByLastname(String lastname);
```

With Spring Data's repositories, you need only to write an interface with finder methods defined according to a given set of conventions (which may vary depending on the kind of persistence store you are using). Spring Data will provide an appropriate implementation of that interface at runtime.

The query builder mechanism built into Spring Data repository infrastructure is useful to build constraining queries over entities of the repository. We will strip the prefixes findBy, find, readBy, read, getBy as well as get from the method and start parsing the rest of it. At a very basic level you can define conditions on entity properties and concatenate them with AND and OR e.g:

List<Person> findByEmailAddressAndLastName(EmailAddress ea, String lastname);

The actual result of parsing that method will of course depend on the persistence store we create the query for, however, there are some general things to notice. The expressions are usually property traversals combined with operators that can be concatenated. As you can see in the example you can combine property expressions with And and Or. Beyond that you also get support for various operators like Between, LessThan, GreaterThan, Like for the property expressions. As the operators supported can vary from datastore to datastore please consult the according part of the reference documentation.

Property expressions can just refer to a direct property of the managed entity (as you just saw in the example above). On query creation time we already make sure that the parsed property is at a property of the managed domain class. However, you can also define constraints by traversing nested properties. Assume Persons have Addresses with ZipCodes. In that case a method name of

List<Person> findByAddressZipCode(ZipCode zipCode);

will create the property traversal x.address.zipCode. The resolution algorithm starts with interpreting the entire part (AddressZipCode) as property and checks the domain class for a property with that name (uncapitalized). If it succeeds it just uses that. If not it starts splitting up the source at the camel case parts from the right side into a head and a tail and tries to find the according property, e.g. AddressZip and Code. If we find a property with that head we take the tail and

continue building the tree down from there. As in our case the first split does not match we move the split point to the left (Address, ZipCode).

Although this should work for most cases, there might be cases where the algorithm could select the wrong property. Suppose our Person class has an addressZip property as well. Then our algorithm would match in the first split round already and essentially choose the wrong property and finally fail (as the type of addressZip probably has no code property). To resolve this ambiguity you can use _ inside your method name to manually define traversal points. So our method name would end up like so:

List<Person> findByAddress_ZipCode(ZipCode zipCode);

Special parameters –

To hand parameters to your query you simply define method parameters as already seen in the examples above. Besides that we will recognize certain specific types to apply pagination and sorting to your queries dynamically.

e.g:

```
Page<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);
```

The first method allows you to pass a Pageable instance to the query method to dynamically add paging to your statically defined query. Sorting options are handed via the Pageable instance too. If you only need sorting, simply add a Sort parameter to your method. As you also can see, simply returning a List is possible as well. We will then not retrieve the additional metadata required to build the actual Page instance but rather simply restrict the query to lookup only the given range of entities (To find out how many pages you get for a query entirely we have to trigger an additional count query. This will be derived from the query you actually trigger by default.).

3. *Setup Spring to create proxy instances for those interfaces.*

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

  <repositories base-package="com.acme.repositories" />
</beans>
```

4. *Get the repository instance injected and use it.*

```
public class SomeClient {

  @Autowired
  private PersonRepository repository;

  public void doSomething() {
    List<Person> persons = repository.findByLastname("Matthews");
```

Custom Querying

Often it is necessary to provide a custom implementation for a few repository methods. Spring Data repositories easily allow you to provide custom repository code and integrate it with generic CRUD abstraction and query method functionality. To enrich a repository with custom functionality you have to define an interface and an implementation for that functionality first and let the repository interface you provided so far extend that custom interface. E.g:

1. Define a Custom Interface

```
interface UserRepositoryCustom {

  public void someCustomMethod(User user);
}
```

2. Implement Your Method

```
class UserRepositoryImpl implements UserRepositoryCustom {

  public void someCustomMethod(User user) {
    // Your custom implementation
  }
}
```

Note that the implementation itself does not depend on Spring Data and can be a regular Spring bean. So you can use standard dependency injection behavior to inject references to other beans, take part in aspects and so on.

3. Let your standard repository interface extend the custom one

```
public interface UserRepository extends CrudRepository<User, Long>,
UserRepositoryCustom {

  // Declare query methods here
}
```

Let your standard repository interface extend the custom one. This makes CRUD and custom functionality available to clients.

4. Configuration

If you use namespace configuration the repository infrastructure tries to autodetect custom implementations by looking up classes in the package we found a repository using the naming conventions appending the namespace element's attribute repository-impl-postfix to the classname. This suffix defaults to Impl.

```
<repositories base-package="com.acme.repository">
  <repository id="userRepository" />
</repositories>

<repositories base-package="com.acme.repository" repository-impl-postfix="FooBar">
  <repository id="userRepository" />
</repositories>
```

The first configuration example will try to lookup a class com.acme.repository.UserRepositoryImpl to act as custom repository implementation, where the second example will try to lookup com.acme.repository.UserRepositoryFooBar.

Query Methods Operators

Keyword	Sample	Logical result
GreaterThan	findByAgeGreaterThan(int age)	{"age" : {"$gt" : age}}
LessThan	findByAgeLessThan(int age)	{"age" : {"$lt" : age}}
Between	findByAgeBetween(int from, int to)	{"age" : {"$gt" : from, "$lt" : to}}
IsNotNull, NotNull	findByFirstnameNotNull()	{"age" : {"$ne" : null}}
IsNull, Null	findByFirstnameNull()	{"age" : null}
Like	findByFirstnameLike(String name)	{"age" : age} (age as regex)

Keyword	Sample	Logical result
(No keyword)	findByFirstname(String name)	{"age" : name}
Not	findByFirstnameNot(String name)	{"age" : {"$ne" : name}}
Near	findByLocationNear(Point point)	{"location" : {"$near" : [x,y]}}
Within	findByLocationWithin(Circle circle)	{"location" : {"$within" : {"$center" : [[x, y], distance]}}}
Within	findByLocationWithin(Box box)	{"location" : {"$within" : {"$box" : [[x1, y1], x2, y2]}}}

Querying Geolocation Data

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:

```
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.

```
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.

```
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);

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

Usage Example

We will demonstrate the usage with an example:

```
public class Person {


  @Id private String id;
  // not annotated as it is assumed that they will be mapped
  // onto db fields that have the same name as the properties
  private String firstName;
  private String secondName;
  private LocalDateTime dateOfBirth;
  private String profession;
  private int salary;


  public Person(
      final String firstName,
      final String secondName,
      final LocalDateTime dateOfBirth,
      final String profession,
      final int salary) {
    this.firstName = firstName;
    this.secondName = secondName;
    this.dateOfBirth = dateOfBirth;
    this.profession = profession;
    this.salary = salary;
  }
```

The only actual Spring Data annotation in this class is the @Id annotation that represents the unique Id of the object which maps to _id and is generated when it is persisted to the database. The annotation can also be left off if the field is named id or _id, therefore in the example above the annotation is not necessary. If the annotation or a correctly named property is not included when persisted an _id field will be created when saved as MongoDB requires the field to be populated.

Available Annotations

- @Id – applied at the field level to mark the field used for identiy 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 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 geoindex the field.
- @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 described 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 other properties are left without annotations and when persisting or saving to the database it is assumed that they will map to fields that share the same name within the database.

The next step is creating a repository that will perform all the database operations to do with the Person object.

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

        List<Person> findBySalary(final int salary);

}
```

Normally we would need to create a implementation of the interface that was just created but instead Spring will create this for us when the application is started. Ok, but what about the method that we just defined on the interface surely that needs to know what it's doing? By using the name of the definition Spring infers the implementation,
therefore findBySalary will find Personobjects stored in the database
by salary, getBySalary could also be used. To execute these queries Spring Data uses the MongoTemplate.

No need implementation, just one interface, and you have CRUD, thanks Spring Data.

Extends MongoRepository, you have CRUD function automatically.

Now to tie all the code together and to show it in action we need to create the main application that has the @SpringBootApplication annotation.

@SpringBootApplication

// needed because the repository is not in the same package or a sub package of the SpringBootApplication

@EnableMongoRepositories(basePackageClasses = PersonRepository.class)

Important thing to notice is the @EnableMongoRepositories annotation, which is required as the PersonRepository is not found in the same package or sub-package as the @SpringBootApplication class. Therefore the annotation is necessary to specify that the repository should be injected into the application.

More information:

https://docs.spring.io/spring-data/data-document/docs/current/reference/html/#preface