7.4. Defining query methods

The repository proxy has two ways to derive a store-specific query from the method name. It can derive the query from the method name directly, or by using a manually defined query. Available options depend on the actual store. However, there’s got to be a strategy that decides what actual query is created. Let’s have a look at the available options.

7.4.1. Query lookup strategies

The following strategies are available for the repository infrastructure to resolve the query. You can configure the strategy at the namespace through the query-lookup-strategy attribute in case of XML configuration or via the queryLookupStrategyattribute of the Enable${store}Repositories annotation in case of Java config. Some strategies may not be supported for particular datastores.

* CREATE attempts to construct a store-specific query from the query method name. The general approach is to remove a given set of well-known prefixes from the method name and parse the rest of the method. Read more about query construction in [Query creation](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#repositories.query-methods.query-creation).
* USE\_DECLARED\_QUERY tries to find a declared query and will throw an exception in case it can’t find one. The query can be defined by an annotation somewhere or declared by other means. Consult the documentation of the specific store to find available options for that store. If the repository infrastructure does not find a declared query for the method at bootstrap time, it fails.
* CREATE\_IF\_NOT\_FOUND (default) combines CREATE and USE\_DECLARED\_QUERY. It looks up a declared query first, and if no declared query is found, it creates a custom method name-based query. This is the default lookup strategy and thus will be used if you do not configure anything explicitly. It allows quick query definition by method names but also custom-tuning of these queries by introducing declared queries as needed.

7.4.2. Query creation

The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository. The mechanism strips the prefixes find…By, read…By, query…By, count…By, and get…By from the method and starts parsing the rest of it. The introducing clause can contain further expressions such as a Distinct to set a distinct flag on the query to be created. However, the first By acts as delimiter to indicate the start of the actual criteria. At a very basic level you can define conditions on entity properties and concatenate them with And and Or.

*Example 16. Query creation from method names*

interface PersonRepository extends Repository<User, Long> {

List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

// Enables the distinct flag for the query

List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);

List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

// Enabling ignoring case for an individual property

List<Person> findByLastnameIgnoreCase(String lastname);

// Enabling ignoring case for all suitable properties

List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

// Enabling static ORDER BY for a query

List<Person> findByLastnameOrderByFirstnameAsc(String lastname);

List<Person> findByLastnameOrderByFirstnameDesc(String lastname);

}

The actual result of parsing the method depends on the persistence store for which you create the query. However, there are some general things to notice.

* The expressions are usually property traversals combined with operators that can be concatenated. You can combine property expressions with AND and OR. You also get support for operators such as Between, LessThan, GreaterThan, Likefor the property expressions. The supported operators can vary by datastore, so consult the appropriate part of your reference documentation.
* The method parser supports setting an IgnoreCase flag for individual properties (for example, findByLastnameIgnoreCase(…)) or for all properties of a type that support ignoring case (usually String instances, for example, findByLastnameAndFirstnameAllIgnoreCase(…)). Whether ignoring cases is supported may vary by store, so consult the relevant sections in the reference documentation for the store-specific query method.
* You can apply static ordering by appending an OrderBy clause to the query method that references a property and by providing a sorting direction (Asc or Desc). To create a query method that supports dynamic sorting, see [Special parameter handling](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#repositories.special-parameters).

7.4.3. Property expressions

Property expressions can refer only to a direct property of the managed entity, as shown in the preceding example. At query creation time you already make sure that the parsed property is a property of the managed domain class. However, you can also define constraints by traversing nested properties. Assume a Person has an Address with a ZipCode. In that case a method name of

List<Person> findByAddressZipCode(ZipCode zipCode);

creates the property traversal x.address.zipCode. The resolution algorithm starts with interpreting the entire part (AddressZipCode) as the property and checks the domain class for a property with that name (uncapitalized). If the algorithm succeeds it uses that property. If not, the algorithm splits up the source at the camel case parts from the right side into a head and a tail and tries to find the corresponding property, in our example, AddressZip and Code. If the algorithm finds a property with that head it takes the tail and continue building the tree down from there, splitting the tail up in the way just described. If the first split does not match, the algorithm move the split point to the left (Address, ZipCode) and continues.

Although this should work for most cases, it is possible for the algorithm to select the wrong property. Suppose the Personclass has an addressZip property as well. The 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);

As we treat underscore as a reserved character we strongly advise to follow standard Java naming conventions (i.e. **not** using underscores in property names but camel case instead).

7.4.4. Special parameter handling

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

*Example 17. Using Pageable, Slice and Sort in query methods*

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

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

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

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

The first method allows you to pass an org.springframework.data.domain.Pageable instance to the query method to dynamically add paging to your statically defined query. A Page knows about the total number of elements and pages available. It does so by the infrastructure triggering a count query to calculate the overall number. As this might be expensive depending on the store used, Slice can be used as return instead. A Slice only knows about whether there’s a next Slice available which might be just sufficient when walking through a larger result set.

Sorting options are handled through the Pageable instance too. If you only need sorting, simply add an org.springframework.data.domain.Sort parameter to your method. As you also can see, simply returning a List is possible as well. In this case the additional metadata required to build the actual Page instance will not be created (which in turn means that the additional count query that would have been necessary not being issued) but rather simply restricts the query to look up only the given range of entities.

|  |  |
| --- | --- |
|  | To find out how many pages you get for a query entirely you have to trigger an additional count query. By default this query will be derived from the query you actually trigger. |

7.4.5. Limiting query results

The results of query methods can be limited via the keywords first or top, which can be used interchangeably. An optional numeric value can be appended to top/first to specify the maximum result size to be returned. If the number is left out, a result size of 1 is assumed.

*Example 18. Limiting the result size of a query with Top and First*

User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);

The limiting expressions also support the Distinct keyword. Also, for the queries limiting the result set to one instance, wrapping the result into an Optional is supported.

If pagination or slicing is applied to a limiting query pagination (and the calculation of the number of pages available) then it is applied within the limited result.

|  |  |
| --- | --- |
|  | Note that limiting the results in combination with dynamic sorting via a Sort parameter allows to express query methods for the 'K' smallest as well as for the 'K' biggest elements. |

9.6. 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 the 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 61. 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](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mongo.geospatial).

Map-Reduce operations are also supported and are described more in the section [Map-Reduce](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mongo.mapreduce).

9.6.1. 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 62. 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 Criteriaobject. 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 field matching ({ key:value }). If the specified value is a document, the order of the fields and exact equality in the document matters.
* 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](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mongo.geospatial)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)
* Query **with** (Sort sort) used to provide sort definition for the results

9.6.2. 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.

9.6.3. 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 GeoResultsallows accessing the average distance of all results. A single GeoResult object simply carries the entity found plus its distance from the origin.

9.6.4. GeoJSON Support

MongoDB supports [GeoJSON](http://geojson.org/) and simple (legacy) coordinate pairs for geospatial data. Those formats can both be used for storing as well as querying data.

|  |  |
| --- | --- |
|  | Please refer to the [MongoDB manual on GeoJSON support](http://docs.mongodb.org/manual/core/2dsphere/#geospatial-indexes-store-geojson/) to learn about requirements and restrictions. |

GeoJSON types in domain classes

Usage of [GeoJSON](http://geojson.org/) types in domain classes is straight forward. The org.springframework.data.mongodb.core.geo package contains types like GeoJsonPoint, GeoJsonPolygon and others. Those are extensions to the existing org.springframework.data.geo types.

public class Store {

String id;

/\*\*

\* location is stored in GeoJSON format.

\* {

\* "type" : "Point",

\* "coordinates" : [ x, y ]

\* }

\*/

GeoJsonPoint location;

}

GeoJSON types in repository query methods

Using GeoJSON types as repository query parameters forces usage of the $geometry operator when creating the query.

public interface StoreRepository extends CrudRepository<Store, String> {

List<Store> findByLocationWithin(Polygon polygon);

}

/\*

\* {

\* "location": {

\* "$geoWithin": {

\* "$geometry": {

\* "type": "Polygon",

\* "coordinates": [

\* [

\* [-73.992514,40.758934],

\* [-73.961138,40.760348],

\* [-73.991658,40.730006],

\* [-73.992514,40.758934]

\* ]

\* ]

\* }

\* }

\* }

\* }

\*/

repo.findByLocationWithin(

new GeoJsonPolygon(

new Point(-73.992514, 40.758934),

new Point(-73.961138, 40.760348),

new Point(-73.991658, 40.730006),

new Point(-73.992514, 40.758934)));

/\*

\* {

\* "location" : {

\* "$geoWithin" : {

\* "$polygon" : [ [-73.992514,40.758934] , [-73.961138,40.760348] , [-73.991658,40.730006] ]

\* }

\* }

\* }

\*/

repo.findByLocationWithin(

new Polygon(

new Point(-73.992514, 40.758934),

new Point(-73.961138, 40.760348),

new Point(-73.991658, 40.730006));

|  |  |
| --- | --- |
|  | Repository method definition using the commons type allows calling it with both GeoJSON and legacy format. |
|  | Use GeoJSON type the make use of $geometry operator. |
|  | Plase note that GeoJSON polygons need the define a closed ring. |
|  | Use legacy format $polygon operator. |

9.7. Query by Example

9.7.1. Introduction

This chapter will give you an introduction to Query by Example and explain how to use Examples.

Query by Example (QBE) is a user-friendly querying technique with a simple interface. It allows dynamic query creation and does not require to write queries containing field names. In fact, Query by Example does not require to write queries using store-specific query languages at all.

9.7.2. Usage

The Query by Example API consists of three parts:

* Probe: That is the actual example of a domain object with populated fields.
* ExampleMatcher: The ExampleMatcher carries details on how to match particular fields. It can be reused across multiple Examples.
* Example: An Example consists of the probe and the ExampleMatcher. It is used to create the query.

Query by Example is suited for several use-cases but also comes with limitations:

**When to use**

* Querying your data store with a set of static or dynamic constraints
* Frequent refactoring of the domain objects without worrying about breaking existing queries
* Works independently from the underlying data store API

**Limitations**

* No support for nested/grouped property constraints like firstname = ?0 or (firstname = ?1 and lastname = ?2)
* Only supports starts/contains/ends/regex matching for strings and exact matching for other property types

Before getting started with Query by Example, you need to have a domain object. To get started, simply create an interface for your repository:

*Example 65. Sample Person object*

public class Person {

@Id

private String id;

private String firstname;

private String lastname;

private Address address;

// … getters and setters omitted

}

This is a simple domain object. You can use it to create an Example. By default, fields having null values are ignored, and strings are matched using the store specific defaults. Examples can be built by either using the of factory method or by using [ExampleMatcher](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#query-by-example.matchers). Example is immutable.

*Example 66. Simple Example*

Person person = new Person();

person.setFirstname("Dave");

Example<Person> example = Example.of(person);

|  |  |
| --- | --- |
|  | Create a new instance of the domain object |
|  | Set the properties to query |
|  | Create the Example |

Examples are ideally be executed with repositories. To do so, let your repository interface extend QueryByExampleExecutor<T>. Here’s an excerpt from the QueryByExampleExecutor interface:

*Example 67. The QueryByExampleExecutor*

public interface QueryByExampleExecutor<T> {

<S extends T> S findOne(Example<S> example);

<S extends T> Iterable<S> findAll(Example<S> example);

// … more functionality omitted.

}

9.7.3. Example matchers

Examples are not limited to default settings. You can specify own defaults for string matching, null handling and property-specific settings using the ExampleMatcher.

*Example 68. Example matcher with customized matching*

Person person = new Person();

person.setFirstname("Dave");

ExampleMatcher matcher = ExampleMatcher.matching()

.withIgnorePaths("lastname")

.withIncludeNullValues()

.withStringMatcherEnding();

Example<Person> example = Example.of(person, matcher);

|  |  |
| --- | --- |
|  | Create a new instance of the domain object. |
|  | Set properties. |
|  | Create an ExampleMatcher to expect all values to match. It’s usable at this stage even without further configuration. |
|  | Construct a new ExampleMatcher to ignore the property path lastname. |
|  | Construct a new ExampleMatcher to ignore the property path lastname and to include null values. |
|  | Construct a new ExampleMatcher to ignore the property path lastname, to include null values, and use perform suffix string matching. |
|  | Create a new Example based on the domain object and the configured ExampleMatcher. |

By default the ExampleMatcher will expect all values set on the probe to match. If you want to get results matching any of the predicates defined implicitly, use ExampleMatcher.matchingAny().

You can specify behavior for individual properties (e.g. "firstname" and "lastname", "address.city" for nested properties). You can tune it with matching options and case sensitivity.

*Example 69. Configuring matcher options*

ExampleMatcher matcher = ExampleMatcher.matching()

.withMatcher("firstname", endsWith())

.withMatcher("lastname", startsWith().ignoreCase());

}

Another style to configure matcher options is by using Java 8 lambdas. This approach is a callback that asks the implementor to modify the matcher. It’s not required to return the matcher because configuration options are held within the matcher instance.

*Example 70. Configuring matcher options with lambdas*

ExampleMatcher matcher = ExampleMatcher.matching()

.withMatcher("firstname", match -> match.endsWith())

.withMatcher("firstname", match -> match.startsWith());

}

Queries created by Example use a merged view of the configuration. Default matching settings can be set at ExampleMatcherlevel while individual settings can be applied to particular property paths. Settings that are set on ExampleMatcher are inherited by property path settings unless they are defined explicitly. Settings on a property patch have higher precedence than default settings.

| *Table 2. Scope of ExampleMatcher settings* | |
| --- | --- |
| **Setting** | **Scope** |
| Null-handling | ExampleMatcher |
| String matching | ExampleMatcher and property path |
| Ignoring properties | Property path |
| Case sensitivity | ExampleMatcher and property path |
| Value transformation | Property path |

9.7.4. Executing an example

*Example 71. Query by Example using a Repository*

public interface PersonRepository extends QueryByExampleExecutor<Person> {

}

public class PersonService {

@Autowired PersonRepository personRepository;

public List<Person> findPeople(Person probe) {

return personRepository.findAll(Example.of(probe));

}

}

An Example containing an untyped ExampleSpec uses the Repository type and its collection name. Typed ExampleSpec use their type as result type and the collection name from the Repository.

|  |  |
| --- | --- |
|  | When including null values in the ExampleSpec Spring Data Mongo uses embedded document matching instead of dot notation property matching. This forces exact document matching for all property values and the property order in the embedded document. |

Spring Data MongoDB provides support for the following matching options:

| *Table 3. StringMatcher options* | |
| --- | --- |
| **Matching** | **Logical result** |
| DEFAULT (case-sensitive) | {"firstname" : firstname} |
| DEFAULT (case-insensitive) | {"firstname" : { $regex: firstname, $options: 'i'}} |
| EXACT (case-sensitive) | {"firstname" : { $regex: /^firstname$/}} |
| EXACT (case-insensitive) | {"firstname" : { $regex: /^firstname$/, $options: 'i'}} |
| STARTING (case-sensitive) | {"firstname" : { $regex: /^firstname/}} |
| STARTING (case-insensitive) | {"firstname" : { $regex: /^firstname/, $options: 'i'}} |
| ENDING (case-sensitive) | {"firstname" : { $regex: /firstname$/}} |
| ENDING (case-insensitive) | {"firstname" : { $regex: /firstname$/, $options: 'i'}} |
| CONTAINING (case-sensitive) | {"firstname" : { $regex: /.\*firstname.\*/}} |
| CONTAINING (case-insensitive) | {"firstname" : { $regex: /.\*firstname.\*/, $options: 'i'}} |
| REGEX (case-sensitive) | {"firstname" : { $regex: /firstname/}} |
| REGEX (case-insensitive) | {"firstname" : { $regex: /firstname/, $options: 'i'}} |

9.7.5. Untyped Example

By default Example is strictly typed. This means the mapped query will have a type match included restricting it to probe assignable types. Eg. when sticking with the default type key \_class the query has restrictions like \_class : { $in : [ com.acme.Person] }.

By using the UntypedExampleMatcher it is possible bypasses the default behavior and skip the type restriction. So as long as field names match nearly any domain type can be used as the probe for creating the reference.

*Example 72. Untyped Example Query*

class JustAnArbitraryClassWithMatchingFieldName {

@Field("lastname") String value;

}

JustAnArbitraryClassWithMatchingFieldNames probe = new JustAnArbitraryClassWithMatchingFieldNames();

probe.value = "stark";

Example example = Example.of(probe, UntypedExampleMatcher.matching());

Query query = new Query(new Criteria().alike(example));

List<Person> result = template.find(query, Person.class);

### 9.11. 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](http://docs.mongodb.org/manual/core/aggregation/) as follows:

For further information see the full [reference documentation](http://docs.mongodb.org/manual/aggregation/) of the aggregation framework and other data aggregation tools for MongoDB.

#### 9.11.1. 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 invoking 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 Document, 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.

#### 9.11.2. 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
* Array Aggregation Operators
* Conditional Aggregation Operators
* Lookup Aggregation Operators

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

|  |  |
| --- | --- |
| *Table 4. Aggregation Operations currently supported by Spring Data MongoDB* | |
| Pipeline Aggregation Operators | bucket, bucketAuto, count, facet, geoNear, graphLookup, group, limit, lookup, match, project, replaceRoot, skip, sort, unwind |
| Set Aggregation Operators | setEquals, setIntersection, setUnion, setDifference, setIsSubset, anyElementTrue, allElementsTrue |
| Group Aggregation Operators | addToSet, first, last, max, min, avg, push, sum, (\*count), stdDevPop, stdDevSamp |
| Arithmetic Aggregation Operators | abs, add (\*via plus), ceil, divide, exp, floor, ln, log, log10, mod, multiply, pow, sqrt, subtract (\*via minus), trunc |
| String Aggregation Operators | concat, substr, toLower, toUpper, stcasecmp, indexOfBytes, indexOfCP, split, strLenBytes, strLenCP, substrCP, |
| Comparison Aggregation Operators | eq (\*via: is), gt, gte, lt, lte, ne |
| Array Aggregation Operators | arrayElementAt, concatArrays, filter, in, indexOfArray, isArray, range, reverseArray, reduce, size, slice, zip |
| Literal Operators | literal |
| Date Aggregation Operators | dayOfYear, dayOfMonth, dayOfWeek, year, month, week, hour, minute, second, millisecond, dateToString, isoDayOfWeek, isoWeek, isoWeekYear |
| Variable Operators | map |
| Conditional Aggregation Operators | cond, ifNull, switch |
| Type Aggregation Operators | type |

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.

#### 9.11.3. 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 Aggregation 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. References to projected fields in later aggregation stages are only valid by using the field name of included fields or their alias of aliased or newly defined fields. Fields not included in the projection cannot be referenced in later aggregation stages.

*Example 73. Projection expression examples*

// will generate {$project: {name: 1, netPrice: 1}}

project("name", "netPrice")

// will generate {$project: {bar: $foo}}

project().and("foo").as("bar")

// will generate {$project: {a: 1, b: 1, bar: $foo}}

project("a","b").and("foo").as("bar")

*Example 74. Multi-Stage Aggregation using Projection and Sorting*

// will generate {$project: {name: 1, netPrice: 1}}, {$sort: {name: 1}}

project("name", "netPrice"), sort(ASC, "name")

// will generate {$project: {bar: $foo}}, {$sort: {bar: 1}}

project().and("foo").as("bar"), sort(ASC, "bar")

// this will not work

project().and("foo").as("bar"), sort(ASC, "foo")

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](http://docs.mongodb.org/manual/reference/operator/aggregation/project/#pipe._S_project) of the MongoDB Aggregation Framework reference documentation.

#### 9.11.4. Faceted classification

MongoDB supports as of Version 3.4 faceted classification using the Aggregation Framework. A faceted classification uses semantic categories, either general or subject-specific, that are combined to create the full classification entry. Documents flowing through the aggregation pipeline are classificated into buckets. A multi-faceted classification enables various aggregations on the same set of input documents, without needing to retrieve the input documents multiple times.

##### Buckets

Bucket operations categorize incoming documents into groups, called buckets, based on a specified expression and bucket boundaries. Bucket operations require a grouping field or grouping expression. They can be defined via the bucket()/bucketAuto() methods of the Aggregate class. BucketOperation and BucketAutoOperation can expose accumulations based on aggregation expressions for input documents. The bucket operation can be extended with additional parameters through a fluent API via the with…() methods, the andOutput(String) method and aliased via the as(String)method. Each bucket is represented as a document in the output.

BucketOperation takes a defined set of boundaries to group incoming documents into these categories. Boundaries are required to be sorted.

*Example 75. Bucket operation examples*

// will generate {$bucket: {groupBy: $price, boundaries: [0, 100, 400]}}

bucket("price").withBoundaries(0, 100, 400);

// will generate {$bucket: {groupBy: $price, default: "Other" boundaries: [0, 100]}}

bucket("price").withBoundaries(0, 100).withDefault("Other");

// will generate {$bucket: {groupBy: $price, boundaries: [0, 100], output: { count: { $sum: 1}}}}

bucket("price").withBoundaries(0, 100).andOutputCount().as("count");

// will generate {$bucket: {groupBy: $price, boundaries: [0, 100], 5, output: { titles: { $push: "$title"}}}

bucket("price").withBoundaries(0, 100).andOutput("title").push().as("titles");

BucketAutoOperation determines boundaries itself in an attempt to evenly distribute documents into a specified number of buckets. BucketAutoOperation optionally takes a granularity specifies the [preferred number](https://en.wikipedia.org/wiki/Preferred_number) series to use to ensure that the calculated boundary edges end on preferred round numbers or their powers of 10.

*Example 76. Bucket operation examples*

// will generate {$bucketAuto: {groupBy: $price, buckets: 5}}

bucketAuto("price", 5)

// will generate {$bucketAuto: {groupBy: $price, buckets: 5, granularity: "E24"}}

bucketAuto("price", 5).withGranularity(Granularities.E24).withDefault("Other");

// will generate {$bucketAuto: {groupBy: $price, buckets: 5, output: { titles: { $push: "$title"}}}

bucketAuto("price", 5).andOutput("title").push().as("titles");

Bucket operations can use AggregationExpression via andOutput() and [SpEL expressions](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mongo.aggregation.projection.expressions) via andOutputExpression() to create output fields in buckets.

Note that further details regarding bucket expressions can be found in the [$bucket section](http://docs.mongodb.org/manual/reference/operator/aggregation/bucket/) and [$bucketAuto section](http://docs.mongodb.org/manual/reference/operator/aggregation/bucketAuto/) of the MongoDB Aggregation Framework reference documentation.

##### Multi-faceted aggregation

Multiple aggregation pipelines can be used to create multi-faceted aggregations which characterize data across multiple dimensions, or facets, within a single aggregation stage. Multi-faceted aggregations provide multiple filters and categorizations to guide data browsing and analysis. A common implementation of faceting is how many online retailers provide ways to narrow down search results by applying filters on product price, manufacturer, size, etc.

A FacetOperation can be defined via the facet() method of the Aggregation class. It can be customized with multiple aggregation pipelines via the and() method. Each sub-pipeline has its own field in the output document where its results are stored as an array of documents.

Sub-pipelines can project and filter input documents prior grouping. Common cases are extraction of date parts or calculations before categorization.

*Example 77. Facet operation examples*

// will generate {$facet: {categorizedByPrice: [ { $match: { price: {$exists : true}}}, { $bucketAuto: {groupBy: $price, buckets: 5}}]}}

facet(match(Criteria.where("price").exists(true)), bucketAuto("price", 5)).as("categorizedByPrice"))

// will generate {$facet: {categorizedByYear: [

// { $project: { title: 1, publicationYear: { $year: "publicationDate"}}},

// { $bucketAuto: {groupBy: $price, buckets: 5, output: { titles: {$push:"$title"}}}

// ]}}

facet(project("title").and("publicationDate").extractYear().as("publicationYear"),

bucketAuto("publicationYear", 5).andOutput("title").push().as("titles"))

.as("categorizedByYear"))

Note that further details regarding facet operation can be found in the [$facet section](http://docs.mongodb.org/manual/reference/operator/aggregation/facet/) of the MongoDB Aggregation Framework reference documentation.

##### Spring Expression Support in Projection Expressions

We support the use of SpEL expression in projection expressions via the andExpression method of the ProjectionOperationand BucketOperation classes. 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](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mongo.aggregation.examples.example5) and [Aggregation Framework Example 6](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mongo.aggregation.examples.example6). You can find more usage examples for supported SpEL expression constructs in SpelExpressionTransformerUnitTests.

|  |  |
| --- | --- |
| *Table 5. Supported SpEL transformations* | |
| a == b | { $eq : [$a, $b] } |
| a != b | { $ne : [$a , $b] } |
| a > b | { $gt : [$a, $b] } |
| a >= b | { $gte : [$a, $b] } |
| a < b | { $lt : [$a, $b] } |
| a ⇐ b | { $lte : [$a, $b] } |
| a + b | { $add : [$a, $b] } |
| a - b | { $subtract : [$a, $b] } |
| a \* b | { $multiply : [$a, $b] } |
| a / b | { $divide : [$a, $b] } |
| a^b | { $pow : [$a, $b] } |
| a % b | { $mod : [$a, $b] } |
| a && b | { $and : [$a, $b] } |
| a || b | { $or : [$a, $b] } |
| !a | { $not : [$a] } |

Next to the transformations shown in Supported SpEL transformations it is possible to use standard SpEL operations like new to eg. create arrays and reference expressions via their name followed by the arguments to use in brackets.

// { $setEquals : [$a, [5, 8, 13] ] }

.andExpression("setEquals(a, new int[]{5, 8, 13})");

##### Aggregation Framework Examples

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

*Aggregation Framework Example 1*

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 projectoperation.
* 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 actual 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 newAggreation Method.

*Aggregation Framework Example 2*

This example is based on the [Largest and Smallest Cities by State](http://docs.mongodb.org/manual/tutorial/aggregation-examples/#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 implicitly 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 sortoperation.

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

*Aggregation Framework Example 3*

This example is based on the [States with Populations Over 10 Million](http://docs.mongodb.org/manual/tutorial/aggregation-examples/#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.

*Aggregation Framework Example 4*

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<Document> result = mongoTemplate.aggregate(agg, Document.class);

List<Document> 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.

*Aggregation Framework Example 5*

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<Document> result = mongoTemplate.aggregate(agg, Document.class);

List<Document> resultList = result.getMappedResults();

*Aggregation Framework Example 6*

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<Document> result = mongoTemplate.aggregate(agg, Document.class);

List<Document> resultList = result.getMappedResults();

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

*Aggregation Framework Example 7*

This example uses conditional projection. It’s derived from the [$cond reference documentation](https://docs.mongodb.com/manual/reference/operator/aggregation/cond/).

public class InventoryItem {

@Id int id;

String item;

String description;

int qty;

}

public class InventoryItemProjection {

@Id int id;

String item;

String description;

int qty;

int discount

}

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

TypedAggregation<InventoryItem> agg = newAggregation(InventoryItem.class,

project("item").and("discount")

.applyCondition(ConditionalOperator.newBuilder().when(Criteria.where("qty").gte(250))

.then(30)

.otherwise(20))

.and(ifNull("description", "Unspecified")).as("description")

);

AggregationResults<InventoryItemProjection> result = mongoTemplate.aggregate(agg, "inventory", InventoryItemProjection.class);

List<InventoryItemProjection> stateStatsList = result.getMappedResults();

* This one-step aggregation uses a projection operation with the inventory collection. We project the discount field using a conditional operation for all inventory items that have a qty greater or equal to 250. A second conditional projection is performed for the description field. We apply the description Unspecified to all items that either do not have a description field of items that have a null description.

#### 9.13.1. 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"));

|  |  |
| --- | --- |
|  | Index and GeospatialIndex support configuration of [collations](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mongo.collation). |

#### 9.13.2. 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]]

#### 9.13.3. 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 78. 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.

|  |  |
| --- | --- |
|  | Collection creation allows customization via CollectionOptions and supports [collations](https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mongo.collation). |

### 9.14. Executing Commands

You can also get at the MongoDB driver’s MongoDatabase.runCommand( ) method using the executeCommand(…) methods on MongoTemplate. These will also perform exception translation into Spring’s DataAccessException hierarchy.

#### 9.14.1. Methods for executing commands

* Document **executeCommand** (Document command) Execute a MongoDB command.
* Document **executeCommand** (Document command, ReadPreference readPreference) Execute a MongoDB command using the given nullable MongoDB ReadPreference.
* Document **executeCommand** (String jsonCommand) Execute the a MongoDB command expressed as a JSON string.

### 9.15. 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 org.bson.Document), 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(BeforeConvertEvent<Person> event) {

... 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.Document.

public class BeforeSaveListener extends AbstractMongoEventListener<Person> {

@Override

public void onBeforeSave(BeforeSaveEvent<Person> event) {

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

|  |  |
| --- | --- |
|  | Lifecycle events are only emitted for root level types. Complex types used as properties within a document root are not subject of event publication unless they are document references annotated with @DBRef. |

# **Calculate Distance Using Spherical Geometry**

**On this page**

* [Distance Multiplier](https://docs.mongodb.com/manual/tutorial/calculate-distances-using-spherical-geometry-with-2d-geospatial-indexes/#distance-multiplier)

**NOTE**

While basic queries using spherical distance are supported by the 2d index, consider moving to a 2dsphere index if your data is primarily longitude and latitude.

The 2d index supports queries that calculate distances on a Euclidean plane (flat surface). The index also supports the following query operators and command that calculate distances using spherical geometry:

* [$nearSphere](https://docs.mongodb.com/manual/reference/operator/query/nearSphere/#op._S_nearSphere)
* [$centerSphere](https://docs.mongodb.com/manual/reference/operator/query/centerSphere/#op._S_centerSphere)
* [$near](https://docs.mongodb.com/manual/reference/operator/query/near/#op._S_near)
* [geoNear](https://docs.mongodb.com/manual/reference/command/geoNear/#dbcmd.geoNear) command with the { spherical: true } option.

**IMPORTANT**

These three queries use radians for distance. Other query types do not.

For spherical query operators to function properly, you must convert distances to radians, and convert from radians to the distances units used by your application.

To convert:

* distance to radians: divide the distance by the radius of the sphere (e.g. the Earth) in the same units as the distance measurement.
* radians to distance: multiply the radian measure by the radius of the sphere (e.g. the Earth) in the units system that you want to convert the distance to.

The equatorial radius of the Earth is approximately 3,963.2 miles or 6,378.1 kilometers.

The following query would return documents from the places collection within the circle described by the center [ -74, 40.74 ] with a radius of 100 miles:

db.places.find( { loc: { $geoWithin: { $centerSphere: [ [ -74, 40.74 ] ,

100 / 3963.2 ] } } } )

You may also use the distanceMultiplier option to the [geoNear](https://docs.mongodb.com/manual/reference/command/geoNear/#dbcmd.geoNear) to convert radians in the [mongod](https://docs.mongodb.com/manual/reference/program/mongod/#bin.mongod)process, rather than in your application code. See [distance multiplier](https://docs.mongodb.com/manual/tutorial/calculate-distances-using-spherical-geometry-with-2d-geospatial-indexes/#geospatial-indexes-distance-multiplier).

The following spherical query, returns all documents in the collection places within 100 miles from the point [ -74, 40.74 ].

db.runCommand( { geoNear: "places",

near: [ -74, 40.74 ],

spherical: **true**

} )

The output of the above command would be:

{

*// [ ... ]*

"results" : [

{

"dis" : 0.01853688938212826,

"obj" : {

"\_id" : ObjectId( ... )

"loc" : [

-73,

40

]

}

}

],

"stats" : {

*// [ ... ]*

"avgDistance" : 0.01853688938212826,

"maxDistance" : 0.01853714811400047

},

"ok" : 1

}

**WARNING**

For spherical queries, use the 2dsphere index result.

The use of 2d index for spherical queries may lead to incorrect results, such as the use of the 2d index for spherical queries that wrap around the poles.

**NOTE**

If specifying latitude and longitude coordinates, list the **longitude** first and then **latitude**:

* Valid longitude values are between -180 and 180, both inclusive.
* Valid latitude values are between -90 and 90 (both inclusive).

## **Distance Multiplier**

The distanceMultiplier option of the [geoNear](https://docs.mongodb.com/manual/reference/command/geoNear/#dbcmd.geoNear) command returns distances only after multiplying the results by an assigned value. This allows MongoDB to return converted values, and removes the requirement to convert units in application logic.

Using distanceMultiplier in spherical queries provides results from the [geoNear](https://docs.mongodb.com/manual/reference/command/geoNear/#dbcmd.geoNear) command that do not need radian-to-distance conversion. The following example uses distanceMultiplier in the [geoNear](https://docs.mongodb.com/manual/reference/command/geoNear/#dbcmd.geoNear)command with a [spherical](https://docs.mongodb.com/manual/tutorial/calculate-distances-using-spherical-geometry-with-2d-geospatial-indexes/) example:

db.runCommand( { geoNear: "places",

near: [ -74, 40.74 ],

spherical: **true**,

distanceMultiplier: 3963.2

} )

The output of the above operation would resemble the following:

{

*// [ ... ]*

"results" : [

{

"dis" : 73.46525170413567,

"obj" : {

"\_id" : ObjectId( ... )

"loc" : [

-73,

40

]

}

}

],

"stats" : {

*// [ ... ]*

"avgDistance" : 0.01853688938212826,

"maxDistance" : 0.01853714811400047

},

"ok" : 1

}

# **$near**

**On this page**

* [Definition](https://docs.mongodb.com/manual/reference/operator/query/near/#definition)
* [Behavior](https://docs.mongodb.com/manual/reference/operator/query/near/#behavior)
* [Examples](https://docs.mongodb.com/manual/reference/operator/query/near/#examples)

## **Definition**

$near

Specifies a point for which a [geospatial](https://docs.mongodb.com/manual/reference/glossary/#term-geospatial) query returns the documents from nearest to farthest. The $nearoperator can specify either a [GeoJSON](https://docs.mongodb.com/manual/reference/glossary/#term-geojson) point or legacy coordinate point.

$near requires a geospatial index:

* [2dsphere](https://docs.mongodb.com/manual/core/2dsphere/) index if specifying a [GeoJSON](https://docs.mongodb.com/manual/reference/glossary/#term-geojson) point,
* [2d](https://docs.mongodb.com/manual/core/2d/) index if specifying a point using legacy coordinates.

To specify a [GeoJSON](https://docs.mongodb.com/manual/reference/glossary/#term-geojson) point, $near operator requires a [2dsphere](https://docs.mongodb.com/manual/core/2dsphere/) index and has the following syntax:

{

<location field>: {

$near: {

$geometry: {

type: "Point" ,

coordinates: [ <longitude> , <latitude> ]

},

$maxDistance: <distance **in** meters>,

$minDistance: <distance **in** meters>

}

}

}

If specifying latitude and longitude coordinates, list the **longitude** first and then **latitude**:

* Valid longitude values are between -180 and 180, both inclusive.
* Valid latitude values are between -90 and 90 (both inclusive).

When specifying a [GeoJSON](https://docs.mongodb.com/manual/reference/glossary/#term-geojson) point, you can use the optional [$minDistance](https://docs.mongodb.com/manual/reference/operator/query/minDistance/#op._S_minDistance) and [$maxDistance](https://docs.mongodb.com/manual/reference/operator/query/maxDistance/#op._S_maxDistance)specifications to limit the $near results by distance in meters:

* [$minDistance](https://docs.mongodb.com/manual/reference/operator/query/minDistance/#op._S_minDistance) limits the results to those documents that are at least the specified distance from the center point. [$minDistance](https://docs.mongodb.com/manual/reference/operator/query/minDistance/#op._S_minDistance) is only available for use with [2dsphere](https://docs.mongodb.com/manual/core/2dsphere/) index.

*New in version 2.6.*

* [$maxDistance](https://docs.mongodb.com/manual/reference/operator/query/maxDistance/#op._S_maxDistance) limits the results to those documents that are at most the specified distance from the center point.

To specify a point using legacy coordinates, $near requires a [2d](https://docs.mongodb.com/manual/core/2d/) index and has the following syntax:

{

$near: [ <x>, <y> ],

$maxDistance: <distance **in** radians>

}

When specifying a legacy coordinate, you can use the optional [$maxDistance](https://docs.mongodb.com/manual/reference/operator/query/maxDistance/#op._S_maxDistance) specification to limit the $near results by distance in radians. [$maxDistance](https://docs.mongodb.com/manual/reference/operator/query/maxDistance/#op._S_maxDistance) limits the results to those documents that are at most the specified distance from the center point.

## **Behavior**

### Special Indexes Restriction

You cannot combine the $near operator, which requires a special [geospatial index](https://docs.mongodb.com/manual/geospatial-queries/#index-feature-geospatial), with a query operator or command that requires another special index. For example you cannot combine $near with the [$text](https://docs.mongodb.com/manual/reference/operator/query/text/#op._S_text) query.

### Sharded Collections Restrictions

For sharded collections, queries using $near are not supported. You can instead use either the [geoNear](https://docs.mongodb.com/manual/reference/command/geoNear/#dbcmd.geoNear)command or the [$geoNear](https://docs.mongodb.com/manual/reference/operator/aggregation/geoNear/#pipe._S_geoNear) aggregation stage.

### Sort Operation

$near sorts documents by distance. If you also include a [sort()](https://docs.mongodb.com/manual/reference/method/cursor.sort/#cursor.sort) for the query, [sort()](https://docs.mongodb.com/manual/reference/method/cursor.sort/#cursor.sort) re-orders the matching documents, effectively overriding the sort operation already performed by $near. When using[sort()](https://docs.mongodb.com/manual/reference/method/cursor.sort/#cursor.sort) with geospatial queries, consider using [$geoWithin](https://docs.mongodb.com/manual/reference/operator/query/geoWithin/#op._S_geoWithin) operator, which does not sort documents, instead of $near.

**SEE ALSO**

[2d Indexes and Geospatial Near Queries](https://docs.mongodb.com/manual/release-notes/3.0-compatibility/#geo-near-compatibility)

## **Examples**

### Query on GeoJSON Data

**IMPORTANT**

If specifying latitude and longitude coordinates, list the **longitude** first and then **latitude**:

* Valid longitude values are between -180 and 180, both inclusive.
* Valid latitude values are between -90 and 90 (both inclusive).

Consider a collection places that has a 2dsphere index.

The following example returns documents that are at least 1000 meters from and at most 5000 meters from the specified GeoJSON point, sorted from nearest to farthest:

db.places.find(

{

location:

{ $near :

{

$geometry: { type: "Point", coordinates: [ -73.9667, 40.78 ] },

$minDistance: 1000,

$maxDistance: 5000

}

}

}

)

### Query on Legacy Coordinates

**IMPORTANT**

If specifying latitude and longitude coordinates, list the **longitude** first and then **latitude**:

* Valid longitude values are between -180 and 180, both inclusive.
* Valid latitude values are between -90 and 90 (both inclusive).

Consider a collection legacy2d that has a 2d index.

The following example returns documents that are at most 0.10 radians from the specified legacy coordinate pair, sorted from nearest to farthest:

db.legacy2d.find(

{ location : { $near : [ -73.9667, 40.78 ], $maxDistance: 0.10 } }

)

Criteria criteria = Criteria.where("contentType").is("application/vnd.sometype");

List<Criteria> docCriterias = new ArrayList<Criteria>(docs.size());

for (Document doc: docs) {

docCriterias.add(Criteria.where("metadata.name").is(doc.getName())

.and("metadata.version").is(doc.getVersion()));

}

criteria = criteria.orOperator(docCriterias.toArray(new Criteria[docs.size()]));

Multiple criteria addition