About this guide

This guide covers:

- Querying documents with Monger
- Using Monger Query DSL
- Using query operators with Monger
- Working with database cursors
- Counting documents
- Database cursor options

What version of Monger does this guide cover?

This guide covers Monger 3.1 (including preview releases).

Overview

Monger provides two means of querying:

- Regular finder functions
- Query DSL

Regular finder functions are very much similar to those in the MongoDB shell and various MongoDB drivers. For regular finders, Monger does not invent its own query language or "syntax": you use the same document structure as you would in the shell. This makes finder functions a lot more predictable, easier to learn and follows the convention existing MongoDB drivers have.

Finder functions belong to the `monger.collection` namespace and always take collection name as their first argument. Some of them return database cursors (that are iterable Java objects and thus seqable) that produce `DBObjects`, other (much more commonly used) return lazy sequences of Clojure maps. This means that with Monger, you work with Clojure data structures (collections and maps). It is natural to express documents as maps in Clojure and Monger fully embraces this idea.

Monger Query DSL is expressive and composable (we will demonstrate what it means later in this guide). It was designed for cases when you may need to perform a sophisticated query that includes conditions, sorting, limit and/or offset and may benefit from paginating results or using advanced MongoDB features like cursor snapshotting.

Finding Documents

To find multiple documents, use `monger.collection/find`:

```
(ns my.service.server
  (:require [monger.core :as mg]
            [monger.collection :as mc]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "documents"]
  (mc/insert db coll {:first_name "John"  :last_name "Lennon"})
  (mc/insert db coll {:first_name "Ringo" :last_name "Starr"})

  (mc/find db coll {:first_name "Ringo"}))
```

`monger.collection/find` takes a database, collection name and query conditoins and returns a database cursor you can use clojure.core/seq on. Each element of the sequence is a `com.mongodb.DBObject` instance. They can be transformed into Clojure maps using `monger.conversion/from-db-object` fn:

```
(ns my.service.server
  (:require [monger.core :as mg]
            [monger.collection :as mc]
            [monger.conversion :refer [from-db-object]]))
```

```
(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "documents"]
  (mc/insert db coll {:first_name "John"  :last_name "Lennon"})
  (mc/insert db coll {:first_name "Ringo" :last_name "Starr"})

  (from-db-object (mc/find "documents" {:first_name "Ringo"}) true)
;= {:first_name "Ringo" :last_name "Starr"}
```

Turning the entire result set into Clojure maps is so common, however, that Monger provides a function that works exactly like `monger.collection/find` but returns a lazy sequence of maps, `monger.collection/find-maps`:

```
;; returns all documents as Clojure maps
(mc/find-maps db "documents")

;; returns documents with year field value of 1998, as Clojure maps
(mc/find-maps db "documents" { :year 1998 })
```

Normally you should prefer `monger.collection/find-maps` to `monger.collection/find`, which is considered to be part of the lower-level API.

Sorting, Limits, Offsets

To use sorting, limit, offset, pagination and so on, please use Monger's Query DSL (covered later in this guide).

Finding a Single Document

`monger.collection/find-one` finds one document and returns it as a `DBObject` instance:

```
;; find one document by id, as `com.mongodb.DBObject` instance
(mc/find-one db "documents" { :_id (ObjectId. "4ec2d1a6b55634a935ea4ac8") })
```

`monger.collection/find-one-as-map` is similar to `monger.collection/find-one` but converts `DBObject` instances to Clojure maps:

```
;; find one document by id, as a Clojure map
(mc/find-one-as-map db "documents" { :_id (ObjectId. "4ec2d1a6b55634a935ea4ac8") })
```

A more convenient way of finding a document by id as Clojure map is `monger.collection/find-map-by-id`:

```
(ns my.service.finders
  (:require [monger.core :as mg]
            [monger.collection :as mc]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "documents"
      oid  (ObjectId.)]
  (mc/insert db coll {:_id oid :first_name "John" :last_name "Lennon"})
  (mc/find-map-by-id db coll oid))
```

Normally you should prefer `monger.collection/find-one-as-map` and `monger.collection/find-map-by-id` to `monger.collection/find-one`.

Convert a String to a MongoDB (BSON) ObjectId

To convert a string in the object id form (for example, coming from a Web form) to an `ObjectId`, instantiate `ObjectId` with an argument:

```
(ns my.service.server
  (:import org.bson.types.ObjectId))

;; MongoDB: convert a string to an ObjectId:
(ObjectId. "4fea999c0364d8e880c05157") ;; => #<ObjectId 4fea999c0364d8e880c05157>
```

Document ids in MongoDB do not have to be of the object id type, they also can be strings, integers and any value you can store that MongoDB knows how to compare order (sort). However, using `ObjectId`s is usually a good idea.

To coerce an input to `ObjectId` (instantiate one from a string of the input is a string, or just return the input if it is an `ObjectId`), there is monger.conversion/to-object-id.

Convert a MongoDB (BSON) ObjectId to a String

To convert a BSON ObjectId (`org.bson.types.ObjectId` instance) to a string, just use clojure.core/str to it or call `org.bson.types.ObjectId#toString` on it.

Loading a Subset of Fields

Both `monger.collection/find` and `monger.collection/find-maps` take an argument that specifies what fields need to be retrieved:

```
(mc/find-one-as-map db "accounts" {:username "happyjoe"} ["email" "username"])
```

This is useful to excluding very large fields from loading when you won't operate on them.

Fields can be specified as a document (just like in the MongoDB shell) but it is more common to pass them as a vector of keywords. Monger will transform them into a document for you.

Reaching Into Nested Documents in Conditions

To "reach into" nested documents and use them in conditions, MongoDB uses the "dot syntax" for fields. For example, with a document that looks like this:

```
{:address {:country "United States of America" :city "New York City" :state "New York" :zip "10001"}}
```

it is possible to address the zip field in a condition as `"address.zip"`. This is exactly how you do it in Monger in conditions and arguments for operators like `$set`:

```
(mc/find-maps db "locations" {"address.zip" "10001"})
```

Keyword and String Field Names

Clojure maps commonly use keywords, however, BSON and many other programming languages do not have a data type like that. Intead, strings are used as keys. Several Monger finder functions are "low level", such as monger.collection/find, and return `com.mongodb.DBObject` instances. They can be thought of as regular Java maps with a little bit of MongoDB-specific metadata.

Other finders combine `monger.collection/find` with monger.conversion/from-db-object to return Clojure maps. Some of those functions take the extra `keywordize` argument that control if resulting map keys will be turned into keywords. An example of such finder is monger.collection/find-one-as-map. By default Monger will keywordize keys.

You can use monger.conversion/from-db-object and monger.conversion/to-db-object to convert maps to `DBObject` instances and back using a custom field name conversion strategy if you need to. Keep in mind that it likely will affect interoperability with other technologies (that may or may not use the same naming/encoding conversion), query capabilities for cases when exact field names are not known and performance for write-heavy workloads.

Using MongoDB Query Operators

Monger provides a convenient way to use MongoDB query operators. While operators can be used in queries with strings, for example:

```
;; with a query that uses operators as strings
(mc/find db "products" { :price_in_subunits { "$gt" 1200 "$lte" 4000 } })
```

there is a better way to do it with Clojure. By using `monger.operators` namespace, MongoDB $operators can be written as Clojure symbols. $operators are implemented as macros that expand at compile time. Here is what it looks like with operator macros:

```
(ns my.app
  (:require [monger.operators :refer :all]))

;; using MongoDB operators as symbols
(mc/find db "products" { :price_in_subunits { $gt 1200 $lte 4000 } })
```

Below are more examples that use various query operators (you can use any operator supported by the MongoDB shell with Monger):

<, <=, >, >=

```
(ns my.app
  (:require [monger.core :as mg]
            [monger.collection :as mc]
            [monger.operators :refer :all]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "libraries"]
    (mc/insert-batch db coll [{:language "Clojure", :name "monger"   :users 1}
                              {:language "Clojure", :name "langohr"  :users 5}
                              {:language "Clojure", :name "incanter" :users 15}
                              {:language "Scala",   :name "akka"     :users 150}])
    (mc/find db coll {:users { $gt 10 }}) ;= 2 documents
    (mc/find db coll {:users { $gte 5 }}) ;= 3 documents
    (mc/find db coll {:users { $lt 10 }}) ;= 2 documents
    (mc/find db coll {:users { $lte 5 }}) ;= 2 documents
```

```
    (mc/find db coll {:users { $gt 10 $lt 150 }})) ;= 1 document
```

The `$gt`, `$lt`, `$gte`, `$lte` operators work on dates and are very commonly used for time range queries. Date values can be `java.util.Date` instances or (highly recommended) Joda Time dates. If you want to use Joda Time in Clojure, clj-time is the most popular option.

$exists

```
(ns my.app
  (:require [monger.collection :as mc]
    [monger.operators :refer :all])

  (:import org.bson.types.ObjectId))

(let [coll "docs"
      doc1 {:_id (ObjectId.) :published-by "Jill The Blogger" :draft false :title "X announces another Y"}
      doc2 {:_id (ObjectId.) :draft true :title "Z announces a Y competitor"}
      _    (mc/insert-batch coll [doc1 doc2])
      result1 (mc/find-one-as-map coll {:published-by {$exists true}})
      result2 (mc/find-one-as-map coll {:published-by {$exists false}})]
  ;= true
  (= doc1 result1)
  ;= true
  (= doc2 result2))
```

$mod

```
(ns my.app
  (:require [monger.core :as mg]
            [monger.collection :as mc]
            [monger.operators :refer :all])
  (:import org.bson.types.ObjectId))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "docs"
      doc1 {:_id (ObjectId.) :counter 25}
      doc2 {:_id (ObjectId.) :counter 32}
      doc3 {:_id (ObjectId.) :counter 63}
      _    (mc/insert-batch coll [doc1 doc2 doc3])
```

```
      result1 (mc/find-one-as-map db coll {:counter {$mod [10 5]}})
      result2 (mc/find-one-as-map db coll {:counter {$mod [10 2]}})
      result3 (mc/find-one-as-map db coll {:counter {$mod [11 1]}})]
  ;= true
  (= doc1 result1)
  ;= true
  (= doc2 result2)
  ;= true
  (empty? result3))
```

$ne

```
(ns my.app
  (:require [monger.core :as mg]
            [monger.collection :as mc]
            [monger.operators :refer :all]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "libraries"]
    (mc/insert-batch db coll [{:language "Ruby"    :name "mongoid"  :users 1}
                              {:language "Clojure" :name "langohr"  :users 5}
                              {:language "Clojure" :name "incanter" :users 15}
                              {:language "Scala"   :name "akka"     :users 150}])
    ;= 2
    (mc/count db coll {$ne {:language "Clojure"}}))
```

$all, $in, $nin

```
(ns my.app
  (:require [monger.core :as mg]
            [monger.collection :as mc]
            [monger.operators :refer :all]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "libraries"]
    (mc/insert-batch db coll [{:language "Clojure" :tags ["functional"]}
                              {:language "Scala"   :tags ["functional" "object-oriented"]}
                              {:language "Ruby"    :tags ["object-oriented" "dynamic"]}])
```

```
    ; = "Scala"
    (:language (first (mc/find-maps db coll {:tags {$all ["functional" "object-oriented"]}})))
    ;= 3
    (mc/count db coll {:tags {$in ["functional" "object-oriented"]}})
    ;= 2
    (mc/count db coll {:language {$in ["Scala" "Ruby"]}})
    ;= 1
    (mc/count db coll {:tags {$nin ["dynamic", "object-oriented"]}})
    ;= 3
    (mc/count db coll {:language {$nin ["C#"]}}))
```

$and, $or, $nor

```
(ns my.app
  (:require [monger.collection :as mc]
    [monger.operators :refer :all]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "libraries"]
    (mc/insert-batch db coll [{ :language "Ruby",    :name "mongoid"  :users 1}
                              { :language "Clojure", :name "langohr"  :users 5}
                              { :language "Clojure", :name "incanter" :users 15}
                              { :language "Scala",   :name "akka"     :users 150}])
    ;= 1
    (mc/count db coll {$and [{:language "Clojure"}
                             {:users {$gt 10}}]})
    ;= 3
    (mc/count db coll {$or [{:language "Clojure"}
                            {:users {$gt 10}}]})
    ;= 1
    (mc/count db coll {$nor [{:language "Clojure"}
                             {:users {$gt 10}} ]}))
```

$regex (regular expression matches)

```
(ns my.app
  (:require [monger.core :as mg]
            [monger.collection :as mc]
            [monger.operators :refer :all]))
```

```
(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "libraries"]
    (mc/insert-batch db coll [{:language "Ruby"    :name "Mongoid"  :users 1}
                              {:language "Clojure" :name "Langohr"  :users 5}
                              {:language "Clojure" :name "Incanter" :users 15}
                              {:language "Scala"   :name "Akka"     :users 150}])
    ;= 2
    (mc/count db coll {:language {$regex "Clo.*"}})
    ;= 2
    (mc/count db coll {:language {$regex "clo.*" $options "i"}})
    ;= 1
    (mc/count db coll {:language {$regex "aK.*" $options "i"}})
    ;= 1
    (mc/count db coll {:language {$regex ".*by"}})
    ;= 1
    (mc/count db coll {:language {$regex ".*ala.*"}}))
```

$elemMatch

```
(ns my.app
  (:require [monger.core :as mg]
            [monger.collection :as mc]
            [monger.operators :refer :all]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "people"]
    (mc/insert-batch db coll [{:name "Bob" :comments [{:text "Nice!" :rating 1}
                                                      {:text "Love it" :rating 4}
                                                      {:text "What?":rating -5} ]}
                              {:name "Alice" :comments [{:text "Yeah" :rating 2}
                                                        {:text "Doh" :rating 1}
                                                        {:text "Agreed" :rating 3}]}])
    (mc/find db coll {:comments {$elemMatch {:text "Nice!" :rating {$gte 1}}}}))
```

$group, $project, etc (MongoDB 2.2 Aggregation Framework support)

Monger supports a new feature in MongoDB 2.2, the Aggregation Framework. It is a vast topic that is out of scope

of this guide and will be covered in a separate one when MongoDB 2.2 reaches release candidate stages.

More Examples

These and other examples of Monger finders in one gist:

```
(ns my.service.finders
  (:require [monger.core :as mg]
            [monger.collection :as mc]
            [monger.operators :refer :all])
  (:import org.bson.types.ObjectId))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "documents"]
  ;; find one document by id, as Clojure map
  (mc/find-map-by-id db coll (ObjectId. "4ec2d1a6b55634a935ea4ac8"))

  ;; find one document by id, as `com.mongodb.DBObject` instance
  (mc/find-by-id db coll (ObjectId. "4ec2d1a6b55634a935ea4ac8"))

  ;; find one document as Clojure map
  (mc/find-one-as-map db coll { :_id (ObjectId. "4ec2d1a6b55634a935ea4ac8") })

  ;; find one document by id, as `com.mongodb.DBObject` instance
  (mc/find-one db coll { :_id (ObjectId. "4ec2d1a6b55634a935ea4ac8") })

  ;; all documents  as Clojure maps
  (mc/find-maps db coll)

  ;; all documents  as `com.mongodb.DBObject` instances
  (mc/find db coll)

  ;; with a query, as Clojure maps
  (mc/find-maps db coll { :year 1998 })

  ;; with a query, as `com.mongodb.DBObject` instances
  (mc/find db coll { :year 1998 })

  ;; with a query that uses operators
```

```
  (mc/find db "products" { :price_in_subunits { $gt 4000 $lte 1200 } })

  ;; with a query that uses operators as strings
  (mc/find db "products" { :price_in_subunits { "$gt" 4000 "$lte" 1200 } }))
```

Getting Distinct Documents

To get a collection of distinct documents by field or query, use the `monger.collection/distinct` function that returns a lazy sequence of documents. There is currently no `monger.collection/distinct-maps` or similar function so to produce a sequence of Clojure maps, it is necessary to map (`clojure.core/map`) with `monger.conversion/from-db-object` over the results.

```
(:require [monger.core :as mg]
          [monger.collection :as mc]
          [monger.conversion :refer [from-db-object]])

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "posts"]
  ;; get distinct values from the posts collection for the field category.
  (mc/distinct db coll "category")

  ;; get distinct values from the posts collection for the field category with a query.
  (mc/distinct db coll "category" {:limit 5})

  ;; convert values to maps using an anonymous function
  (map
    (fn [cat] (from-db-object cat false))
    (mc/distinct db coll "category")))
```

Monger Query DSL

Queries that need sorting (and with it, commonly skip/limit/pagination) use Monger's Query DSL. It is composed of functions and macros in the `monger.query` namespace, with additional convenience operator macros from the `monger.operators` namespace. Monger's Query DSL is heavily inspired by SQL Korma, is composable and easy to extend if necessary.

Queries performed via Query DSL always return sequences of Clojure maps, like `monger.collection/find-maps` does.

Lets take a look at its core features first.

Sorting, Skip and Limit

Sorting documents are specified exactly as they are in the MongoDB shell (1 for ascending, -1 for descending ordering):

```
(ns my.service.server
  (:refer-clojure :exclude [sort find])
  (:require [monger.core :as mg]
            [monger.query :refer :all]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "scores"]
  ;; find top 10 scores that will be returned as Clojure maps
  (with-collection db coll
    (find {})
    (fields [:score :name])
    ;; it is VERY IMPORTANT to use array maps with sort
    (sort (array-map :score -1 :name 1))
    (limit 10))

  ;; find scores 10 to 20
  (with-collection db coll
    (find {})
    (fields [:score :name])
    ;; it is VERY IMPORTANT to use array maps with sort
    (sort (array-map :score -1 :name 1))
    (limit 10)
    (skip 10)))
```

This example also demonstrates query conditions and fetching a subset of fields. Note that because the order of keys matters for sorting, you should always use array maps with `(sort ...)`. Regular Clojure maps do not have ordering guarantees and this may lead to incorrect sorting of results.

Using Pagination

Using `skip` and `limit` to do pagination in the query DSL is so common that Monger provides a DSL extension for that:

```
(ns my.service.server
  (:refer-clojure :exclude [sort find])
  (:require [monger.core :as mg]
            [monger.query :refer :all]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "scores"]
  ;; find top 10 scores
  (with-collection db coll
    (find {})
    (fields [:score :name])
    (sort {:score -1})
    (paginate :page 1 :per-page 10))

  ;; find scores 10 to 20
  (with-collection db coll
    (find {})
    (fields [:score :name])
    (sort {:score -1})
    (paginate :page 2 :per-page 10)))
```

Read Preference

Read preference lets MongoDB clients specify whether a query should go to the master/primary node (thus guaranteeing consistency but also putting extra load on primaries) or it's OK to read from slaves (and thus get eventual consistency, which ocassionally may result in slightly out of date data to be returned):

```
(ns my.service.server
  (:refer-clojure :exclude [sort find])
  (:require [monger.core :as mg]
            [monger.query :refer :all])
  (:import com.mongodb.ReadPreference))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "scores"]
  ;; reads from primary (master) to guarantee consistency
  ;; (at the cost of putting extra load on the primary)
  (with-collection db coll
```

```
    (find {:email "joe@example.com"})
    (read-preference (ReadPreference/primary))))
```

Possible read preference values are returned by the following static methods:

- `com.mongodb.ReadPreference/primary` (read only from master, throw an error if it is not available)
- `com.mongodb.ReadPreference/primaryPreferred` (read from master if available, a slave otherwise)
- `com.mongodb.ReadPreference/secondary` (read from a slave if available, otherwise throw an error)
- `com.mongodb.ReadPreference/secondaryPreferred` (read from a slave if available, from master otherwise)
- `com.mongodb.ReadPreference/nearest` (read from the nearest node)

Snapshotting Cursors

A MongoDB query returns data as well as a cursor ID for additional lookups, should more data exist. Drivers lazily perform a "get more" operation as needed on the cursor to get more data. Cursors may have latent getMore accesses that occurs after an intervening write operation on the database collection (i.e., an insert, update, or delete).

Conceptually, a cursor has a current position. If you delete the item at the current position, the cursor automatically skips its current position forward to the next item. Snapshotting a cursor assures that objects which update during the lifetime of a query are returned once and only once. This is most important when doing a find-and-update loop that changes.

Here is how to snapshot a cursor with Monger query DSL:

```
(ns my.service.server
  (:refer-clojure :exclude [sort find])
  (:require [monger.core :as mg]
            [monger.query :refer :all]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "documents"]
  ;; performs a snapshotted query
  (with-collection db coll
                   (find {:email "joe@example.com"})
                   (snapshot)))
```

Index Hints

While not necessary in most cases, it is possible to force query to use the given index:

```
(ns my.service.server
  (:refer-clojure :exclude [sort find])
  (:require [monger.core :as mg]
            [monger.query :refer :all]
            [monger.operators :refer [$gt $lt]])))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")]
  (with-collection db coll
    (find {:age_in_days {$gt 365} :number_of_signins {$lt 3}})
    (sort {:age_in_days -1 :number_of_signins 1})
    (hint {:age_in_days -1 :number_of_signins 1})))
```

Hinting only makes sense in the presence of multiple compound indexes that may be used by the optimizer and sorting by one or both indexed fields.

Setting batch size

Cursors fetch documents from the server in batches. It is possible to specify the size of the batch to improve performance or limit the number of documents returned by the server:

```
(ns my.service.server
  (:refer-clojure :exclude [sort find])
  (:require [monger.core :as mg]
            [monger.query :refer :all]
            [monger.operators :refer [$gt]])))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")]
  (with-collection db coll
     (find {:age_in_days {$gt 365}})
     (sort {:age_in_days -1})
     (batch-size 5000))
```

If batch size is negative, it will limit of number objects returned, that fit within the max batch size limit (usually 4MB), and cursor will be closed. For example if batch-size is -10, then the server will return a maximum of 10 documents and as many as can fit in 4MB, then close the cursor. Note that this feature is different from limit in that documents must fit within a maximum size, and it removes the need to send a request to close the cursor server-

side.

Counting Documents

Use `monger.collection/count`, `monger.collection/empty?` and `monger.collection/any?`:

(let conn (mg/connect) db (mg/get-db conn "monger-test") coll "documents" ;= false

(mc/empty? db coll) ;= true

(mc/insert db coll {:first_name "John" :last_name "Lennon"}) (mc/insert db coll {:first_name "Ringo" :last_name "Starr"})

(mc/any? db coll) ;= true

(mc/empty? db coll) ;= false

(mc/count db coll {:first_name "Ringo"}) ;= 1

(mc/count db coll {:first_name "Adam"}) ;= 0

(mc/any? db coll {:first_name "Ringo"}) ;= true

(mc/any? db coll {:first_name "Joe"}) ;= false) ```

Tweaking Query Options

There're special cases when you need to tweak the settings of query - for example you have a long running query that needs to run more than 10minutes without getting a timeout exception. You can tweak the options two ways: using the DSL `options` specifier or low-level helpers from `monger.cursor` namespace.

Query DSL

When you are using the query DSL, then it's easy to tweak the options, just add the option specifier to your query. The option specifier accepts Clojure map, list, keyword or constant value from class `com.mongodb.Bytes` as argument.

```
(require '[monger.query :refer [with-collection find options]])
```

```
(with-collection db "products"
  (find {:language "Clojure"})
  (options {:notimeout true, :slaveok false}) ;;`false` turns option off
  (options [:notimeout :slaveok])             ;; activate these 2 options
  (options :notimeout)                        ;; only timeout
  (options com.mongodb.Bytes/QUERYOPTION_NOTIMEOUT))
```

Fine-tuning Cursors

You can use helpers from `monger.cursor` namespace, when you are working with a low-level finder as `monger.collection/find`, returns the database cursor object.

Here's an example usage of cursor helper

```
(require '[monger.collection :as coll]
         '[monger.cursor :as cur])

(let [db-cur (coll/find db :languages {:language "Clojure"})]
    (cur/reset-options db-cur)              ;; cleans previous settings
    (cur/add-option! db-cur :notimeout)     ;; adds only one options
    (cur/remove-option! db-cur :notimeout)  ;;removes specific option, keep other untouched
    (cur/add-options db-cur {:notimeout true :slaveok false})
    (cur/add-options db-cur [:notimeout :slaveok])
    (cur/add-options db-cur :notimeout)
    (cur/add-options db-cur com.mongodb.Bytes/QUERYOPTION_NOTIMEOUT)
    (cur/get-options db-cur)                ;; returns map of settings, where values show current state of option
    (cur/format-as db-cur :map)             ;; turns lazy-seq of clojure map
)
```

You cannot tweak the query settings for `find-map` or `find-seq`, but you can simulate their functionality by using helpers from the cursor namespace. Here's a little usage example, that simulates the `find-map` functionality:

```
(require '[monger.cursor :refer [make-db-cursor add-options format-as]])

(let [db-cur (make-db-cursor db :languages {:language "Clojure"})]
 (-> db-cur
     (add-options :notimeout)
     (format-as :map)))
```

Table of Options

keyword	Bytes class value	description
:awaitdata	QUERYOPTION_AWAITDATA	Use with TailableCursor.
:notimeout	QUERYOPTION_NOTIMEOUT	The server normally times out idle cursors after an inactivity period (10 minutes) to prevent excess memory use.
:oplogreplay	QUERYOPTION_OPLOGREPLAY	Internal replication use only - driver should not set
:partial	QUERYOPTION_PARTIAL	Use with sharding (mongos).
:slaveok	QUERYOPTION_SLAVEOK	When turned on, read queries will be directed to slave servers instead of the primary server.
:tailable	QUERYOPTION_TAILABLE	Tailable means cursor is not closed when the last data is retrieved.

What To Read Next

The documentation is organized as a number of guides, covering all kinds of topics.

We recommend that you read the following guides first, if possible, in this order:

- Updating documents
- Deleting documents
- Indexing and other collection operations
- Integration with 3rd party libraries
- Map/Reduce
- GridFS support
- Using MongoDB Aggregation Framework
- Using MongoDB commands

Tell Us What You Think!

Please take a moment to tell us what you think about this guide on Twitter or the Monger mailing list

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

comments powered by Disqus

This website was developed by ClojureWerkz team.

Follow us on Twitter: ClojureWerkz, Michael Klishin, Alex P

Artwork by zuk13