Monger, a Clojure client for MongoDB

- Home
- All guides
- API reference
- Community
- Code
- Change log
- More Clojure libraries
- Donate
- Clojure Docs

About this guide

This guide combines an overview of Monger with a quick tutorial that helps you to get started with it. It should take about 10 minutes to read and study the provided code examples. This guide covers:

- Feature of Monger, why Monger was created
- Clojure and MongoDB version requirements
- How to add Monger dependency to your project
- Basic operations (created, read, update, delete)
- Overview of Monger Query DSL
- Overview of how Monger integrates with libraries like clojure.data.json and JodaTime.

What version of Monger does this guide cover?

This guide covers Monger 3.1 (including preview releases).

Monger Overview

Monger is an idiomatic Clojure wrapper around MongoDB Java driver. It offers powerful expressive query DSL, strives to support every MongoDB 2.0+ feature, has minimal performance overhead and is well maintained.

What Monger is not

Monger is not a replacement for the MongoDB Java driver, instead, Monger is symbiotic with it. Monger does not try to offer object/document mapping functionality. With Monger, you work with native Clojure and Java data structures like maps, vectors, strings, dates and so on. This approach has pros and cons but (we believe) closely follows Clojure's philosophy of reducing incidental complexity. It also fits MongoDB data model very well.

Supported Clojure Versions

Monger requires Clojure 1.6+. The most recent stable release is highly recommended.

Supported MongoDB Versions

Monger uses MongoDB Java driver 3.x under the hood and thus supports MongoDB 2.2 and later versions. Please note that some features may be specific to recent MongoDB releases.

Adding Monger Dependency To Your Project

Monger artifacts are released to Clojars.

With Leiningen

```
[com.novemberain/monger "3.1.0"]
```

With Maven

Add Clojars repository definition to your `pom.xml`:

```
<repository>
  <id>clojars.org</id>
  <url>http://clojars.org/repo</url>
</repository>
```

And then the dependency:

```
<dependency>
  <groupId>com.novemberain</groupId>
  <artifactId>monger</artifactId>
  <version>3.1.0</version>
</dependency>
```

Connecting to MongoDB

Before using Monger, you need to connect to MongoDB and choose a database to work with. Monger supports working with multiple connections and databases.

To connect, use `monger.core/connect` function which returns a connection:

```
(ns my.service.server
  (:require [monger.core :as mg])
  (:import [com.mongodb MongoOptions ServerAddress]))

;; localhost, default port
(let [conn (mg/connect)])

;; given host, default port
(let [conn (mg/connect {:host "db.megacorp.internal"})])


;; given host, given port
(let [conn (mg/connect {:host "db.megacorp.internal" :port 7878})])

;; using MongoOptions allows fine-tuning connection parameters,
;; like automatic reconnection (highly recommended for production environment)
(let [^MongoOptions opts (mg/mongo-options {:threads-allowed-to-block-for-connection-multiplier 300})
      ^ServerAddress sa  (mg/server-address "127.0.0.1" 27017)
      conn               (mg/connect sa opts)]
  )
```

To choose a database, use `monger.core/get-db`:

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

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")])
```

Connecting Using URI (Heroku, CloudFoundry, etc)

In certain environments, for example, Heroku or other PaaS providers, the only way to connect to MongoDB is via connection URI.

Monger provides `monger.core/connect-via-uri` function that combines `monger.core/connect`, `monger.core/get-db`, and authentication and works with string URIs like `mongodb://userb71148a:0da0a696f23a4ce1ecf6d11382633eb2049d728e@cluster1.mongohost.com:27034/app81766662`.

`monger.core/connect-via-uri` returns a map with two keys:

- `:conn`
- `:db`

It can be used to connect with or without authentication, for example:

```
;; connect without authentication
```

```
(let [uri               "mongodb://127.0.0.1/monger-test4"
      {:keys [conn db]} (mg/connect-via-uri uri)])

;; connect with authentication
(let [uri               "mongodb://clojurewerkz/monger!:monger!@127.0.0.1/monger-test4"
      {:keys [conn db]} (mg/connect-via-uri uri)])

;; connect using connection URI stored in an env variable, in this case, MONGOHQ_URL
(let [uri               (System/genenv "MONGOHQ_URL")
      {:keys [conn db]} (mg/connect-via-uri uri)])
```

It is also possible to pass connection options as query parameters:

```
(let [uri               "mongodb://localhost/test?maxPoolSize=128&waitQueueMultiple=5;waitQueueTimeoutMS=150;socketTimeoutMS=5500&autoConnectRetry=true;safe=false&w=1;wtimeout=2500;fsync=true"
      {:keys [conn db]} (mg/connect-via-uri uri)])
```

Authentication

To authenticate, use `monger.credentials/create` which takes the admin database, username, and password as char array:

```
(ns monger.docs.examples
  (:require [monger.core :as mg]
            [monger.credentials :as mcr]))

(let [admin-db   "admin"
      u    "username"
      p    (.toCharArray "password")
      cred (mcr/create u admin-db p)
      host "127.0.0.1"]
  (mg/connect-with-credentials host cred))
```

The function will return a `MongoClient` object if authentication succeeds and raises an exception if authentication fails.

Disconnecting

To disconnect, use `monger.core/disconnect`:

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

(let [conn (mg/connect)]
  (mg/disconnect conn))
```

Monger 2.0+ Public API Convention

Monger versions prior to 2.0 used dynamic vars (shared state) to store connection, database, and GridFS references. Monger 2.0 is different: it accepts them as an explicit argument. For example, functions that operate on documents require a database reference as their 1st argument:

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

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")]
  (mc/insert db "documents" { :_id (ObjectId.) :first_name "John" :last_name "Lennon" })
  (mc/insert-and-return db "documents" {:name "John" :age 30}))
```

The same is true for functions operate on databases (they require an explicit connection argument), GridFS operations (a GridFS instance needs to be passed in), and so on.

How to Insert Documents with Monger

To insert documents, `insert`, `insert-and-return` and `insert-batch` functions in the `monger.collection` namespace are used.

```
(ns my.service.server
  (:require [monger.core :as mg]
            [monger.collection :as mc])
  (:import [org.bson.types ObjectId]
           [com.mongodb DB WriteConcern]))

;; localhost, default port
(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")]
  ;; with a generated document id, returns the complete
  ;; inserted document
  (mc/insert-and-return db "documents" {:name "John" :age 30})

  ;; with explicit document id (recommended)
  (mc/insert db "documents" { :_id (ObjectId.) :first_name "John" :last_name "Lennon" })

  ;; multiple documents at once
  (mc/insert-batch db "documents" [{ :first_name "John" :last_name "Lennon" }
                                   { :first_name "Paul" :last_name "McCartney" }])

  ;; without document id (when you don't need to use it after storing the document)
  (mc/insert db "documents" { :first_name "John" :last_name "Lennon" })

  ;; with a different write concern
  (mc/insert db "documents" { :_id (ObjectId.) :first_name "John" :last_name "Lennon" } WriteConcern/JOURNAL_SAFE))
```

`monger.collection/insert` returns write result that `monger.result/acknowledged?` and other `monger.result` functions can operate on.

`monger.collection/insert-and-return` returns the exact documented inserted, including the generated document id.

`monger.collection/insert-batch` is a recommended way of inserting batches of documents (from tens to hundreds of thousands) because it is very efficient compared to sequentially or even concurrently inserting documents one by one.

Write Failures

When a write fails, with a write concern that doesn't ignore errors, an exception will be thrown.

For the list of available options, see MongoDB Java driver API reference on WriteConcern.

Document ids (ObjectId)

If you insert a document without the `:_id` key, MongoDB Java driver that Monger uses under the hood will generate one for you. Unfortunately, it does so by mutating the document you pass it. With Clojure's immutable data structures, that won't work the way MongoDB Java driver authors expected.

So it is highly recommended to always store documents with the `:_id` key set. If you need a generated object id. You do so by instantiating `org.bson.types.ObjectId` without arguments:

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

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

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.

Default WriteConcern

To set default write concern Monger use, use `monger.core/set-default-write-concern!`:

```
(ns my.service.server
  (:require [monger.core :as mg])
  (:import com.mongodb.WriteConcern))

(mg/set-default-write-concern! WriteConcern/FSYNC_SAFE)
```

Most functions in the Monger API that can work with different write concerns accept it as a positional argument or an option.

Safe By Default

By default Monger will use `WriteConcern/ACKNOWLEDGED` as write concern. Historically, MongoDB Java driver (as well as other official drivers) have **very unsafe defaults** when no exceptions are raised, even for network issues. This does not sound like a good default for most applications: many applications use MongoDB because of the flexibility, not extreme write throughput requirements.

Monger's default is and always will be on the safe side, regardless of what the Java driver is.

How to Find Documents with Monger

Monger provides two ways of finding documents:

- Using finder functions in the `monger.collection` namespace
- Using query DSL in the `monger.query` namespace

The former is designed to cover simple cases better while the latter gives you access to full power of MongoDB querying capabilities and extra features like pagination.

Using Finder Functions

Finder functions in Monger return either Clojure maps (commonly used) or Java driver's objects like `DBObject` and `DBCursor`.

For example, `monger.collection/find` returns a `DBCursor`:

```
(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-maps` is similar to `monger.collection/find` but converts `DBObject` instances to Clojure 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 })
```

`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))
```

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 } })
```

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 } }))
```

Counting Documents

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

```
(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"]
  ;; removes all documents
  (mc/remove db coll)
  (mc/empty? db coll)
  ;= true
  (mc/any? db coll)
  ;= false
  (mc/insert-and-return db coll {:name "Monger"})
  (mc/count db coll)
  ;= 1
```

Query DSL Overview

For cases when it is necessary to combine sorting, limiting or offseting results, pagination and even more advanced features like cursor snapshotting or manual index hinting, Monger provides a very powerful query DSL. Here is what it looks like:

```
(with-collection db "movies"
  (find { :year { $lt 2010 $gte 2000 } :revenue { $gt 20000000 } })
  (fields [ :year :title :producer :cast :budget :revenue ])
  ;; note the use of sorted maps with sort
  (sort (sorted-map :revenue -1))
  (skip 10)
  (limit 20)
  (hint "year-by-year-revenue-idx")
  (snapshot))
```

It is easy to add new DSL elements, for example, adding pagination took literally less than 10 lines of Clojure code. Here is what it looks like:

```
(with-collection db coll
                  (find {})
                  (paginate :page 1 :per-page 3)
                  (sort (sorted-map :title 1))
                  (read-preference ReadPreference/PRIMARY))
```

Query DSL supports composition, too:

```
(let
    [top3               (partial-query (limit 3))
     by-population-desc (partial-query (sort (sorted-map :population -1)))
     result             (with-collection db coll
                          (find {})
                          (merge top3)
                          (merge by-population-desc))]
  ;; ...
  )
```

Learn more in our Querying guide.

How to Update Documents with Monger

Monger's update API follows the following simple rule: the "syntax" for condition and update document structure is the same or as close as possible to MongoDB shell and the official drivers. In addition, Monger provides several convenience functions for common cases, for example, finding documents by id.

Regular Updates

`monger.collection/update` is the most commonly used way of updating documents. `monger.collection/update-by-id` is useful when document id is known:

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

;; updates a document by id
(mc/update-by-id db "scores" oid {:score 1088})
```

Upserts

MongoDB supports upserts, "update or insert" operations. To do an upsert with Monger, use `monger.collection/update` function with `:upsert` option set to true:

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

;; updates score for player "sam" if it exists; creates a new document otherwise
(mc/update db "scores" {:player "sam"} {:score 1088} {:upsert true})
```

Note that upsert only inserts one document. Learn more about upserts in this MongoDB documentation section.

Atomic Modifiers

Modifier operations are highly-efficient and useful when updating existing values; for instance, they're great for incrementing counters, setting individual fields, updating fields that are arrays and so on.

MongoDB supports modifiers via update operation and Monger API works the same way: you pass a document with modifiers to `monger.collection/update`. For example, to increment number of views for a particular page:

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

(mc/update db "visits" {:url "http://megacorp.com"} {$inc {:visits 1}})
```

How to Remove Documents with Monger

Documents are removed using `monger.collection/remove` function. `monger.collection/remove-by-id` is useful when document id is known.

```
(ns my.service.server
  (:require [monger.core :as mg]
            [monger.collection :refer [insert update update-by-id remove-by-id] :as mc])
  (:import org.bson.types.ObjectId
           [com.mongodb DB WriteConcern]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      coll "documents"]
  ;; insert a few documents
  (mc/insert coll { :language "English" :pages 38 })
  (mc/insert db coll { :language "Spanish" :pages 78 })
  (mc/insert db coll { :language "Unknown" :pages 87 })

  ;; remove multiple documents
  (mc/remove db coll { :language "English" })

  ;; remove ALL documents in the collection
  (mc/remove db coll)
  ;; remove document by id
  (let [oid (ObjectId.)]
    (mc/insert db coll { :language "English" :pages 38 :_id oid })
    (remove-by-id db coll oid)))
```

Integration With Other Libraries

Monger heavily relies on relatively recent Clojure features like protocols to integrate with libraries like Cheshire or clj-time (Joda Time). As the result you can focus on your application instead of figuring out how to glue two libraries together.

Cheshire (or clojure.data.json)

Many applications that use MongoDB and Monger have to serialize documents stored in the database to JSON and pass them to other applications using HTTP or messaging protocols such as AMQP 0.9.1 or ZeroMQ.

This means that MongoDB data types (object ids, documents) need to be serialized. While BSON, data format used by MongoDB, is semantically very similar to JSON, MongoDB drivers do not typically provide serialization to JSON and JSON serialization libraries typically do not support MongoDB data types.

Monger provides a convenient feature for Cheshire, a pretty popular modern JSON serialization library for Clojure. The way it works is Monger will add custom serializes for MongoDB Java driver data types: `org.bson.types.ObjectId` and `com.mongodb.DBObject` if you opt-in for it. To use it, you need to add Cheshire dependency to your project, for example (with Leiningen)

```
[cheshire "5.1.1"]
```

and then require `monger.json` namespace like so:

```
(ns mycompany.myservice
  (:require monger.json))
```

when loaded, code in that namespace will extend necessary protocols and that's it. Then you can pass documents that contain object ids in them to JSON serialization functions from `cheshire.custom` and everything will just work.

This feature is optional: Monger does not depend on `Cheshire` or `clojure.data.json` and won't add unused dependencies to your project.

clojure.data.json Version Compatibility

Monger only works `clojure.data.json` `0.2.x` and `0.1.x`. Support for versions earlier than `0.2.x` will be dropped in one of the future releases.

clj-time, Joda Time

Because of various shortcomings of Java date/time classes provided by the JDK, many projects choose to use Joda Time to work with dates.

To be able to insert documents with Joda Time date values in them, you need to require `monger.joda-time` namespace:

```
(ns mycompany.myservice
  (:require monger.joda-time))
```

Just like with `clojure.data.json` integration, there is nothing else you have to do. This feature is optional: Monger does not depend on `clj-time` or `Joda Time` and won't add unused dependencies to your project.

clojure.core.cache

Monger provides a MongoDB-backed cache implementation that conforms to the `clojure.core.cache` protocol. It uses capped collections for caches. You can use any many cache data structure instances as your application may need.

This topic is covered in the Integration with 3rd party libraries guide.

Wrapping Up

Congratulations, you now know how to do most common operations with Monger. Monger and MongoDB both have much more to them to explore. Other guides explain these and other features in depth, as well as rationale and use cases for them.

To stay up to date with Monger development, follow @ClojureWerkz on Twitter and join our mailing list about Monger, Clojure and MongoDB.

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:

- Connecting to MongoDB
- Inserting documents
- Querying & finders
- 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
- Miscellaneous topics

Tell Us What You Think!

Please take a moment to tell us what you think about this guide on Twitter or the Clojure MongoDB 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