About this guide

This guide covers:

- Inserting documents
- Inserting batches of documents
- Checking database responses

- Validating data with Validateur, a Clojure validation library
- Setting default write concern
- Changing write concern for individual operations

What version of Monger does this guide cover?

This guide covers Monger 3.1 (including preview releases).

Inserting documents

If we don't count upserts, there are three insert functions in Monger: `monger.collection/insert`, `monger.collection/insert-and-return`, and `monger.collection/insert-batch`.

Let's first take a look at the former. It takes three arguments: a database, collection name and document to insert:

```
(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" }))
```

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

`monger.collection/insert-and-return` is an alternative insertion function that returns the exact documented inserted, including the generated document id:

```
(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-and-return db "documents" {:name "John" :age 30}))
```

Because `monger.collection/insert-and-return` returns the document inserted and not a write result.

Document can be either a Clojure map (in the majority of cases, it is) or an instance of `com.mongodb.DBObject` (referred to later as `DBObject`). In case your application obtains DBObjects from other libraries (for example), you can insert those:

```
(ns doc.examples
  (:require [monger.core :as mg]
            [monger.collection :as mc])
  (:import [com.mongodb BasicDBObject BasicDBList]
           java.util.ArrayList))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")
      db-obj (doto (BasicDBObject.)
               (.put "int" 101)
               (.put "dblist" (doto (BasicDBList.)
                                (.put "0" 0)
                                (.put "1" 1)))
               (.put "list" (ArrayList. ["red" "green" "blue"])))]
  (mc/insert db "documents" db-obj))
```

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.

Generating random UUIDs

To generate a new psuedo random UUID (type 4), combine `java.util.UUID/randomUUID` and `clojure.core/str`:

```
(str (java.util.UUID/randomUUID))
```

or just use `monger.util/random-uuid`.

Collection (Array) Fields

Document fields that you need to be stored as arrays are typically passed as vectors, although they can also be lists, lazy sequences or your own data types that implement `java.util.List`:

```
(ns doc.examples
  (:require [monger.collection :as mc]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")]
  (mc/insert db "owners" {:name "John" :age 30 :pets ["Sam" "Chelsie"]}))
```

Nested Documents

Nested documents are inserted just the way one would expect: as nested Clojure maps (or DBObjects):

```
(ns doc.examples
  (:require [monger.core :as mg]
            [monger.collection :as mc]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")]
  (mc/insert db "owners" {:name "John" :age 30 :address {:country "Italy" :state "Lazio" :city "Rome" :zip 00199}}))
```

Serialization of Clojure Data Types to DBObject and DBList

Monger will construct `DBObject` instances from Clojure data types for you. The serialization process has very low overhead and supports almost all core Clojure data types:

- Maps
- Lists, vectors, sets
- Keywords (serialized as strings)
- Symbols (serialized as strings)
- Ratios (serialized as doubles)

Several data types are serialized transparently because they are just Java data types:

- Strings
- Integers, longs, doubles
- Dates (`java.util.Date`)

In case you need to manually convert a DBObject to Clojure map or vice versa, Monger has two functions that do that, `monger.conversion/to-db-object` and `monger.conversion/from-db-object`:

```
(ns doc.examples
  (:require [monger.conversion :refer :all]))

(to-db-object {:int 1 :string "Mongo" :float 22.23 :map {:int 10 :string "Clojure" :float 11.9 :list '(1 "a" :b) :map {:key "value"}}})

(let [db-obj (doto (BasicDBObject.)
               (.put "int" 101)
               (.put "dblist" (doto (BasicDBList.)
```

```
                                (.put "0" 0)
                                (.put "1" 1)))
                (.put "list" (ArrayList. ["red" "green" "blue"])))]
  (from-db-object db-obj))
```

the latter takes an optional argument that controls whether document fields are converted to Clojure keywords. When false, field names become string keys in the result.

Inserting Batches Of Documents

Sometimes you need to insert a batch of documents all at once and you need it to be done efficiently. MongoDB supports batch inserts feature. To do it with Monger, use `monger.collection/insert-batch` function:

```
(ns doc.examples
  (:require [monger.core :as mg]
            [monger.collection :as mc]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")]
  (mc/insert-batch db "documents" [{:name "Alan" :age 27 :score 17772}
                                   {:name "Joe" :age 32 :score 8277}
                                   {:name "Macy" :age 29 :score 8837777}]))
```

Please make sure to read MongoDB documentation on error handling of batch inserts

Checking Insertion Results

In real world applications, things often go wrong. Insert operations may fail for one reason or another (from duplicate `_id` key to network outages to hardware failures to everything in between). Monger provides `monger.result` namespace with several functions that check MongoDB responses for success:

```
(ns doc.examples
  (:require [monger.core :as mg]
            [monger.collection :as mc]
            [monger.result :refer [acknowledged?]]))

(let [conn (mg/connect)
      db   (mg/get-db conn "monger-test")]
  (mc/insert-batch db "documents" [{:name "Alan" :age 27 :score 17772}
                                   {:name "Joe" :age 32 :score 8277}
```

```
                                    {:name "Macy" :age 29 :score 8837777}])
  (println (acknowledged? response)))
```

Please note that responses will carry error/success information only with safe write concern ("safe mode") which is Monger's default. To learn more, see MongoDB documentation on error handling.

Validating Data With Validateur, a Clojure data validation library

Monger does not handle data validation but it comes with a small standalone Clojure validation library called Validateur. Validateur is inspired by Ruby's ActiveModel. It has two central concepts: validators and validation sets.

Validateur is functional: validators are functions, validation sets are higher-order functions, validation results are returned as values.

With Validateur you define validation sets that compose one or more validators:

```
(ns my.app
  (:require [validateur.validation :refer :all]))

(validation-set
  (presence-of :email)
  (presence-of [:address :street])
  ;; reaches into associative data structures, like clojure.core/get-in and /update-in.
  (presence-of [:card :cvc]))
```

Any function that returns either a pair of

```
[true #{}]
```

to indicate successful validation or

```
[false set-of-validation-error-messages]
```

to indicate validation failure and return error messages can be used as a validator. Validation sets are then passed to `validateur.core/valid?` together with a map to validate:

```
(let [v (validation-set
          (presence-of :name)
          (presence-of :age))]
    (is (valid? v {:name "Joe" :age 28}))
    (is (not (invalid? v {:name "Joe" :age 28}))))
```

```
    (is (not (valid? v {:name "Joe"}))))
```

`validateur.core/invalid?` is a complement to `validateur.core/valid?`.

TBD: examples with custom validators

Setting Default Write Concern

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

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

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

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

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.

Changing Write Concern For Individual Operations

In many applications, most operations are not performance-sensitive but a few are. Some kinds of data can be lost but some is absolutely cruicial to system/company operation. For those cases, MongoDB and Monger allow developers to trade some write throughput for safety (or vice versa) by specifying a different write concern value for individual operations:

```
(ns doc.examples
  (:require [monger.core :as mg]
            [monger.collection :as mc])
  (:import com.mongodb.WriteConcern))

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

```
    db   (mg/get-db conn "monger-test")]
;; for data that you can afford to lose
(mc/insert "events" {:type "pages.view" :url "http://megastartup.com/photos/9b50311c"} WriteConcern/NORMAL)
;; for data that you absolutely cannot afford to lose and want to be replicated
;; before the function call returns
(mc/insert "accounts" {:email "joe@example.com" :password_hash "..." :password_salt "..."} WriteConcern/REPLICAS_SAFE))
```

When doing so, please keep MongoDB's differences in error handling in mind.

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:

- 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

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