MongoDB with Mongo Shell

jim stafford

Fall 2022 v2022-07-24: Built: 2022-12-07 06:14 EST

Table of Contents

Chapter 1. Introduction

This lecture will introduce working with MongoDB database using the Mongo shell.

1.1. Goals

The student will learn:

- basic concepts behind the Mongo NoSQL database
- to create a database and collection
- to perform basic CRUD operations with database collection and documents using Mongo shell

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. identify the purpose of a MongoDB collection, structure of a MongoDB document, and types of example document fields
2. access a MongoDB database using the Mongo shell
3. perform basic CRUD actions on documents
4. perform paging commands
5. leverage the aggregation pipeline for more complex commands

Chapter 2. Mongo Concepts

Mongo is a document-oriented database. This type of database enforces very few rules when it comes to schema. About the only rules that exist are:

- a primary key field, called `_id` must exist
- no document can be larger than 16MB

GridFS API Supports Unlimited Size Documents.

MongoDB supports unlimited size documents using the GridFS API. GridFS is basically a logical document abstraction over a collection of related individual physical documents — called "chunks" — abiding by the standard document-size limits

2.1. Mongo Terms

The table below lists a few keys terms associated with MongoDB.

Table 1. Mongo Terms

Mongo Term	Peer RDBMS Term	Description
Database	Database	a group of document collections that fall under the same file and administrative management
Collection	Table	a set of documents with indexes and rules about how the documents are managed
Document	Row	a collection of fields stored in binary JSON (BSON) format. RDBMS tables must have a defined schema and all rows must match that schema.
Field	Column	a JSON property that can be a single value or nested document. An RDBMS column will have a single type based on the schema and cannot be nested.
Server	Server	a running instance that can perform actions on the database. A server contains more than one database.
Mongos	(varies)	an intermediate process used when data is spread over multiple servers. Since we will be using only a single server, we will not need a `mongos`

2.2. Mongo Documents

Mongo Documents are stored in a binary JSON format called "BSON". There are many native types that can be represented in BSON. Among them include "string", "boolean", "date", "ObjectId", "array", etc.

Documents/fields can be flat or nested.

Example Document

```
{
  "field1": value1, ①
  "field2": value2,
  "field3": { ②
    "field31": value31,
    "field32": value32
  },
  "field4": [ value41, value42, value43 ], ③
  "field5": [ ④
    { "field511": value511, "field512": value512 },
    { "field521": value521}
    { "field513": value513, "field513": value513, "field514": value514 }
  ]
}
```

① example field with value of BSON type

② example nested document within a field

③ example field with type array — with values of BSON type

④ example field with type array — with values of nested documents

The follow-on interaction examples will use a flat document structure to keep things simple to start with.

Chapter 3. MongoDB Server

To start our look into using Mongo commands, lets instantiate a MongoDB, connect with the Mongo Shell, and execute a few commands.

3.1. Starting Docker-Compose MongoDB

One simple option we have to instantiate a MongoDB is to use Docker Compose.

The following snippet shows an example of launching MongoDB from the docker-compose.yml script in the example directory.

Starting Docker-Compose MongoDB

```
$ docker-compose up -d mongodb
Creating ejava_mongodb_1 ... done

$ docker ps --format "{{.Image}}\t{{.Ports}}\t{{.Names}}"
mongo:4.4.0-bionic  0.0.0.0:27017->27017/tcp mongo-book-example_mongodb_1 ①
```

① image is running with name ejava_mongodb_1 and server 27017 port is mapped also to host

This specific MongoDB server is configured to use authentication and has an admin account pre-configured to use credentials `admin/secret`.

3.2. Connecting using Host's Mongo Shell

If we have Mongo shell installed locally, we can connect to MongoDB using the default mapping to localhost.

Connect using Host's Mongo shell

```
$ which mongo
/usr/local/bin/mongo ①
$ mongo -u admin -p secret ② ③
MongoDB shell version v4.4.0
connecting to:
mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
```

① mongo shell happens to be installed locally

② password can be securely prompted by leaving off command line

③ URL defaults to mongodb://127.0.0.1:27017

3.3. Connecting using Guest's Mongo Shell

If we do not have Mongo shell installed locally, we can connect to MongoDB by executing the command in the MongoDB image.

Connecting using Guest's Mongo Shell

```
$ docker-compose exec mongodb mongo -u admin -p secret ① ②
MongoDB shell version v4.4.0
connecting to:
mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
```

① runs the `mongo` shell command within the `mongodb` Docker image

② URL defaults to mongodb://127.0.0.1:27017

3.4. Switch to test Database

We start off with three default databases meant primarily for server use.

Show Databases

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
```

We can switch to a database to make it the default database for follow-on commands even before it exists.

Switch Database

```
> use test ①
switched to db test
> show collections
>
```

① makes the `test` database the default database for follow-on commands

Mongo will create a new/missing database on-demand when the first document is inserted.

3.5. Database Command Help

We can get a list of all commands available to us for a collection using the `db.<collection>.help()` command. The collection does not have to exist yet.

Get Collection Command Help

```
> db.books.help() ①
DBCollection help
...
    db.books.insertOne( obj, <optional params> ) - insert a document, optional
parameters are: w, wtimeout, j
```

```
    db.books.insert(obj)
```

① command to list all possible commands for a collection

Chapter 4. Basic CRUD Commands

4.1. Insert Document

We can create a new document in the database, stored in a named collection.

The following snippet shows the syntax for inserting a single, new book in the `books` collection. All fields are optional at this point and the `_id` field will be automatically generated by the server when we do not provide one.

Insert One Document

```
> db.books.insertOne({title:"GWW", author:"MM", published:ISODate("1936-06-30")})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("606c82da9ef76345a2bf0b7f") ①
}
```

① `insertOne` command returns the `_id` assigned

MongoDB creates the collection, if it does not exist.

Created Collection

```
> show collections
books
```

4.2. Primary Keys

MongoDB requires that all documents contain a primary key with the name `_id` and will generate one of type `ObjectID` if not provided. You have the option of using a business value from the document or a self-generated uniqueID, but it has to be stored in the `_id` field.

The following snippet shows an example of an `insert` using a supplied, numeric primary key.

Example Insert with Provided Primary Key

```
> db.books.insert({_id:17, title:"GWW", author:"MM", published:ISODate("1936-06-30")})
WriteResult({ "nInserted" : 1 })

> db.books.find({_id:17})
{ "_id" : 17, "title" : "GWW", "author" : "MM", "published" : ISODate("1936-06-
30T00:00:00Z") }
```

4.3. Document Index

All collections are required to have an index on the `_id` field. This index is generated automatically.

Default _id Index

```
> db.books.getIndexes()
[
  { "v" : 2, "key" : { "_id" : 1 }, "name" : "_id_" } ①
]
```

① index on `_id` field in `books` collection

4.4. Create Index

We can create an index on one or more other fields using the createIndex() command.

The following example creates a non-unique, ascending index on the `title` field. By making it sparse — only documents with a `title` field are included in the index.

Create Example Index

```
> db.books.createIndex({title:1}, {unique:false, sparse:true})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}
```

4.5. Find All Documents

We can find all documents by passing in a JSON document that matches the fields we are looking for. We can find all documents in the collection by passing in an empty query (`{}`). Output can be made more readable by adding `.pretty()`.

Final All Documents

```
> db.books.find({}) ①
{ "_id" : ObjectId("606c82da9ef76345a2bf0b7f"), "title" : "GWW", "author" : "MM",
"published" : ISODate("1936-06-30T00:00:00Z") }

> db.books.find({}).pretty() ②
{
    "_id" : ObjectId("606c82da9ef76345a2bf0b7f"),
    "title" : "GWW",
    "author" : "MM",
    "published" : ISODate("1936-06-30T00:00:00Z")
}
```

① empty query criteria matches all documents in the collection

② adding `.pretty()` expands the output

4.6. Return Only Specific Fields

We can limit the fields returned by using a "projection" expression. `1` means to include. `0` means to exclude. `_id` is automatically included and must be explicitly excluded. All other fields are automatically excluded and must be explicitly included.

Return Only Specific Fields

```
> db.books.find({}, {title:1, published:1, _id:0}) ①
{ "title" : "GWW", "published" : ISODate("1936-06-30T00:00:00Z") }
```

① find all documents and only include the `title` and `published` date

4.7. Get Document by Id

We can obtain a document by searching on any number of its fields. The following snippet locates a document by the primary key `_id` field.

Get Document By Id

```
> db.books.find({_id:ObjectId("606c82da9ef76345a2bf0b7f")})
{ "_id" : ObjectId("606c82da9ef76345a2bf0b7f"), "title" : "GWW", "author" : "MM",
"published" : ISODate("1936-06-30T00:00:00Z") }
```

4.8. Replace Document

We can replace the entire document by providing a filter and replacement document.

The snippet below filters on the `_id` field and replaces the document with a version that modifies the `title` field.

Replace Document (Found) Example

```
> db.books.replaceOne(
{ "_id" : ObjectId("606c82da9ef76345a2bf0b7f")},
{"title" : "Gone WW", "author" : "MM", "published" : ISODate("1936-06-30T00:00:00Z")
})

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 } ①
```

① result document indicates a single match was found and modified

The following snippet shows a difference in the results when a match is not found for the filter.

Replacement Document (Not Found) Example

```
> db.books.replaceOne({ "_id" : "badId"}, {"title" : "Gone WW"})
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 } ①
```

① `matchCount` and `modifiedCount` result in 0 when filter does not match anything

The following snippet shows the result of replacing the document.

Replace Document Result

```
> db.books.findOne({_id:ObjectId("606c82da9ef76345a2bf0b7f")})
{
    "_id" : ObjectId("606c82da9ef76345a2bf0b7f"),
    "title" : "Gone WW",
    "author" : "MM",
    "published" : ISODate("1936-06-30T00:00:00Z")
}
```

4.9. Save/Upsert a Document

We will receive an error if we issue an `insert` a second time using an `_id` that already exists.

Example Duplicate Insert Error

```
> db.books.insert({_id:ObjectId("606c82da9ef76345a2bf0b7f"), title:"Gone WW", author:
"MMitchell", published:ISODate("1936-06-30")})
WriteResult({
"nInserted" : 0,
"writeError" : {
    "code" : 11000,
    "errmsg" : "E11000 duplicate key error collection: test.books index: _id_ dup key:
{ _id: ObjectId('606c82da9ef76345a2bf0b7f') }",
    }
})
```

We will be able to insert a new document or update an existing one using the `save` command. This very useful command performs an "upsert".

Example Save/Upsert Command

```
> db.books.save({_id:ObjectId("606c82da9ef76345a2bf0b7f"), title:"Gone WW", author:
"MMitchell", published:ISODate("1936-06-30")}) ①
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

① `save` command performs an upsert

4.10. Update Field

We can update specific fields in a document using one of the update commands. This is very useful when modifying large documents or when two concurrent threads are looking to increment a value in the document.

Example Update Field

```
> filter={ "_id" : ObjectId("606c82da9ef76345a2bf0b7f")} ①
> command={$set:{"title" : "Gone WW"} }
> db.books.updateOne( filter, command )

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
```

① using shell to store value in variable used in command

Update Field Result

```
> db.books.findOne({_id:ObjectId("606c82da9ef76345a2bf0b7f")})
{
    "_id" : ObjectId("606c82da9ef76345a2bf0b7f"),
    "title" : "Gone WW",
    "author" : "MM",
    "published" : ISODate("1936-06-30T00:00:00Z")
}
```

4.11. Delete a Document

We can delete a document using the `delete` command and a filter.

Delete Document by Primary Key

```
> db.books.deleteOne({_id:ObjectId("606c82da9ef76345a2bf0b7f")})
{ "acknowledged" : true, "deletedCount" : 1 }
```

Chapter 5. Paging Commands

As with most find() implementations, we need to take care to provide a limit to the number of documents returned. The Mongo shell has a built-in default limit. We can control what the database is asked to do using a few paging commands.

5.1. Sample Documents

This example has a small collection of 10 documents.

Count Documents

```
> db.books.count({})
10
```

The following lists the primary key, title, and author. There is no sorting or limits placed on this output

Document Titles and Authors

```
> db.books.find({}, {title:1, author:1})
{ "_id" : ObjectId("607c77169fca586207a97242"), "title" : "123Pale Kings and Princes",
"author" : "Lanny Miller" }
{ "_id" : ObjectId("607c77169fca586207a97243"), "title" : "123Bury My Heart at Wounded
Knee", "author" : "Ilona Leffler" }
{ "_id" : ObjectId("607c77169fca586207a97244"), "title" : "123Carrion Comfort",
"author" : "Darci Jacobs" }
{ "_id" : ObjectId("607c77169fca586207a97245"), "title" : "123Antic Hay", "author" :
"Dorcas Harris Jr." }
{ "_id" : ObjectId("607c77169fca586207a97246"), "title" : "123Where Angels Fear to
Tread", "author" : "Latashia Gerhold" }
{ "_id" : ObjectId("607c77169fca586207a97247"), "title" : "123Tiger! Tiger!", "author"
: "Miguel Gulgowski DVM" }
{ "_id" : ObjectId("607c77169fca586207a97248"), "title" : "123Waiting for the
Barbarians", "author" : "Curtis Willms II" }
{ "_id" : ObjectId("607c77169fca586207a97249"), "title" : "123A Time of Gifts",
"author" : "Babette Grimes" }
{ "_id" : ObjectId("607c77169fca586207a9724a"), "title" : "123Blood's a Rover",
"author" : "Daryl O'Kon" }
{ "_id" : ObjectId("607c77169fca586207a9724b"), "title" : "123Precious Bane", "author"
: "Jarred Jast" }
```

5.2. limit()

We can limit the output provided by the database by adding the `limit()` command and supplying the maximum number of documents to return.

Example limit() Command

```
> db.books.find({}, {title:1, author:1}).limit(3) ① ② ③
{ "_id" : ObjectId("607c77169fca586207a97242"), "title" : "123Pale Kings and Princes",
"author" : "Lanny Miller" }
{ "_id" : ObjectId("607c77169fca586207a97243"), "title" : "123Bury My Heart at Wounded
Knee", "author" : "Ilona Leffler" }
{ "_id" : ObjectId("607c77169fca586207a97244"), "title" : "123Carrion Comfort",
"author" : "Darci Jacobs" }
```

① find all documents matching `{}` filter

② return projection of `_id` (default), title `, and `author`

③ limit results to first 3 documents

5.3. sort()/skip()/limit()

We can page through the data by adding the `skip()` command. It is common that `skip()` is accompanied by `sort()` so that the follow on commands are using the same criteria.

The following snippet shows the first few documents after sorting by author.

Paging Example, First Page

```
> db.books.find({}, {author:1}).sort({author:1}).skip(0).limit(3) ①
{ "_id" : ObjectId("607c77169fca586207a97249"), "author" : "Babette Grimes" }
{ "_id" : ObjectId("607c77169fca586207a97248"), "author" : "Curtis Willms II" }
{ "_id" : ObjectId("607c77169fca586207a97244"), "author" : "Darci Jacobs" }
```

① return first page of `limit()` size, after sorting by `author`

The following snippet shows the second page of documents sorted by author.

Paging Example, First Page

```
> db.books.find({}, {author:1}).sort({author:1}).skip(3).limit(3) ①
{ "_id" : ObjectId("607c77169fca586207a9724a"), "author" : "Daryl O'Kon" }
{ "_id" : ObjectId("607c77169fca586207a97245"), "author" : "Dorcas Harris Jr." }
{ "_id" : ObjectId("607c77169fca586207a97243"), "author" : "Ilona Leffler" }
```

① return second page of `limit()` size, sorted by `author`

The following snippet shows the last page of documents sorted by author. In this case, we have less than the limit available.

Paging Example, First Page

```
> db.books.find({}, {author:1}).sort({author:1}).skip(9).limit(3) ①
{ "_id" : ObjectId("607c77169fca586207a97247"), "author" : "Miguel Gulgowski DVM" }
```

① return last page sorted by author

Chapter 6. Aggregation Pipelines

There are times when we need to perform multiple commands and reshape documents. It may be more efficient and better encapsulated to do within the database versus issuing multiple commands to the database. MongoDB provides a feature called the Aggregation Pipeline that performs a sequence of commands called stages.

The intent of introducing the Aggregation topic is for those cases where one needs extra functionality without making multiple trips to the database and back to the client. The examples here will be very basic.

6.1. Common Commands

Some of these commands are common to `db.<collection>.find()`:

- criteria
- project
- sort
- offset
- limit

The primary difference between aggregate's use of these common commands and `find()` is that `find()` can only operate against the documents in the collection. `aggregate()` can work against the documents in the collection and any intermediate reshaping of the results along the pipeline.

Downstream Pipeline Stages do not use Collection Indexes

Only initial aggregation pipeline stage commands — operating against the database collection — can take advantage of indexes.

6.2. Unique Commands

Some commands unique to aggregation include:

- group - similar to SQL's "group by" for a JOIN, allowing us to locate distinct, common values across multiple documents and perform a group operation (like `sum`) on their remaining fields
- lookup - similar functionality to SQL's JOIN, where values in the results are used to locate additional information from other collections for the result document before returning to the client
- ...(see Aggregate Pipeline Stages documentation)

6.3. Simple Match Example

The following example implements functionality we could have implemented with `db.books.find()`. It uses 5 stages:

- `$match` - to select documents with title field containing the letter `T`
- `$sort` - to order documents by `author` field in descending order
- `$project` - return only the `_id` (default) and `author` fields

- `$skip` - to skip over 0 documents
- `$limit` - to limit output to 2 documents

Aggregate Simple Match Example

```
> db.books.aggregate([
    {$match: {title:/T/}},
    {$sort: {author:-1}},
    {$project:{author:1}},
    {$skip:0},
    {$limit:2} ])
{ "_id" : ObjectId("607c77169fca586207a97247"), "author" : "Miguel Gulgowski DVM" }
{ "_id" : ObjectId("607c77169fca586207a97246"), "author" : "Latashia Gerhold" }
```

6.4. Count Matches

This example implements a count of matching fields on the database. The functionality could have been achieved with `db.books.count()`, but is gives us a chance to show a few things that can be leveraged in more complex scenarios.

- `$match` - to select documents with title field containing the letter `T`
- `$group` - to re-organize/re-structure the documents in the pipeline to gather them under a new, primary key and to perform an aggregate function on their remaining fields. In this case we are assigning all documents the `null` primary key and incrementing a new field called `count` in the result document.

Aggregate Count Example

```
> db.books.aggregate([
    {$match:{ title:/T/}},
    {$group: {_id:null, count:{ $sum:1}}} ]) ①
{ "_id" : null, "count" : 3 } /②
```

① create a new document with field `count` and increment value by 1 for each occurrence

② the resulting document is re-shaped by pipeline

The following example assigns the primary key (`_id`) field to the `author` field instead, causing each document to a distinct `author` that just happens to have only 1 instance each.

Aggregate Count Example with Unique Primary Key

```
> db.books.aggregate([
    {$match:{ title:/T/}},
    {$group: {_id:"$author", count:{ $sum:1}}} ]) ①
{ "_id" : "Miguel Gulgowski DVM", "count" : 1 }
{ "_id" : "Latashia Gerhold", "count" : 1 }
{ "_id" : "Babette Grimes", "count" : 1 }
```

① assign primary key to `author` field

Chapter 7. Helpful Commands

This section contains a set if helpful Mongo shell commands. It will be populated over time.

7.1. Default Database

We can invoke the Mongo shell with credentials and be immediately assigned a named, default database.

- authenticating as usual
- supplying the database to execute against
- supplying the database to authenticate against (commonly `admin`)

The following snippet shows an example of authenticating as `admin` and starting with `test` as the default database for follow-on commands.

Example Set Default Database Command

```
$ docker-compose exec mongodb mongo test -u admin -p secret --authenticationDatabase
admin
...
> db.getName()
test
> show collections
books
```

7.2. Command-Line Script

We can invoke the Mongo shell with a specific command to execute by using the `--eval` command line parameter.

The following snippet shows an example of listing the contents of the `books` collection in the `test` database.

Example Script Command

```
$ docker-compose exec mongodb mongo test -u admin -p secret --authenticationDatabase
admin --eval 'db.books.find({},{author:1})'

MongoDB shell version v4.4.0
connecting to: mongodb://127.0.0.1:27017/test?authSource=admin&compressors=disabled&g
ssapiServiceName=mongodb
Implicit session: session { "id" : UUID("47e146a5-49c0-4fe4-be67-cc8e72ea0ed9") }
MongoDB server version: 4.4.0
{ "_id" : ObjectId("607c77169fca586207a97242"), "author" : "Lanny Miller" }
{ "_id" : ObjectId("607c77169fca586207a97243"), "author" : "Ilona Leffler" }
{ "_id" : ObjectId("607c77169fca586207a97244"), "author" : "Darci Jacobs" }
{ "_id" : ObjectId("607c77169fca586207a97245"), "author" : "Dorcas Harris Jr." }
```

```
{ "_id" : ObjectId("607c77169fca586207a97246"), "author" : "Latashia Gerhold" }
{ "_id" : ObjectId("607c77169fca586207a97247"), "author" : "Miguel Gulgowski DVM" }
{ "_id" : ObjectId("607c77169fca586207a97248"), "author" : "Curtis Willms II" }
{ "_id" : ObjectId("607c77169fca586207a97249"), "author" : "Babette Grimes" }
{ "_id" : ObjectId("607c77169fca586207a9724a"), "author" : "Daryl O'Kon" }
{ "_id" : ObjectId("607c77169fca586207a9724b"), "author" : "Jarred Jast" }
```

Chapter 8. Summary

In this module we learned:

- to identify a MongoDB collection, document, and fields
- to create a database and collection
- access a MongoDB database using the Mongo shell
- to perform basic CRUD actions on documents to manipulate a MongoDB collection
- to perform paging commands to control returned results
- to leverage the aggregation pipeline for more complex commands