Command Line Help

To see the list of options and help for starting the mongo shell, use the --help option from the command line:

mongo --help

Shell Help

To see the list of help, in the mongo shell, type help:

help

Database Help

In the mongo shell:

To see the list of databases on the server, use the show dbs command:

show dbs

New in version 2.4: show databases is now an alias for show dbs

To see the list of help for methods you can use on the db object, call the db.help() method:

db.help()

To see the implementation of a method in the shell, type the db.<method name> without the parenthesis (()), as in the following example which will return the implementation of the method db.updateUser():

db.updateUser

Collection Help

In the mongo shell:

To see all the database

show dbs

To see the list of collections in the current database, use the show collections command:

show collections

To see the help for methods available on the collection objects (e.g. db.<collection>), use the db.<collection>.help() method:

db.collection.help()

<collection> can be the name of a collection that exists, although you may specify a collection that doesn’t exist.

To see the collection method implementation, type the db.<collection>.<method> name without the parenthesis (()), as in the following example which will return the implementation of the save() method:

db.listcollections()

db.collection.save

Cursor Help

When you perform read operations with the find() method in the mongo shell, you can use various cursor methods to modify the find() behavior and various JavaScript methods to handle the cursor returned from the find() method.

To list the available modifier and cursor handling methods, use the db.collection.find().help() command:

db.collection.find().help()

<collection> can be the name of a collection that exists, although you may specify a collection that doesn’t exist.

To see the implementation of the cursor method, type the db.<collection>.find().<method> name without the parenthesis (()), as in the following example which will return the implementation of the toArray() method:

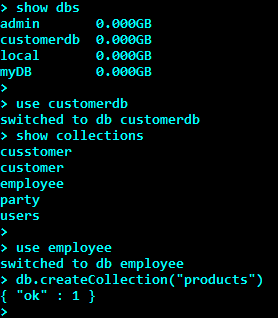
db.collection.find().toArray

The [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell is an interactive JavaScript interface to MongoDB. You can use the [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell to query and update data as well as perform administrative operations.

mongo also provides a fully functional JavaScript environment for use with a MongoDB.

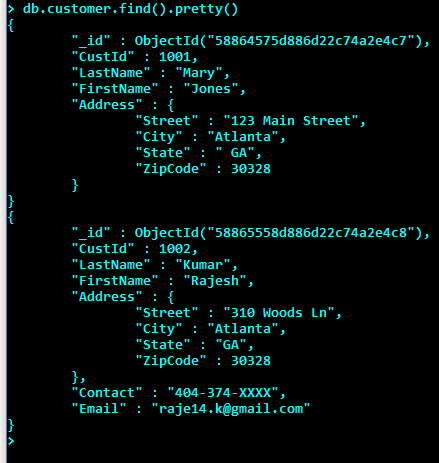
Working with the mongo Shell ; The operation should return test, which is the default database

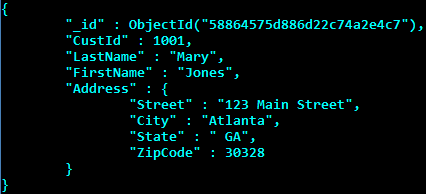
To switch databases, issue the use <db>

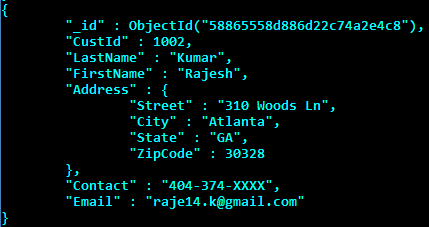


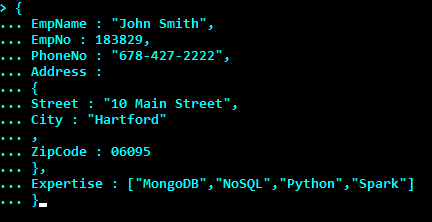
> db.customer.insert( { CustId : 1002, LastName : "Kumar", FirstName : "Rajesh", Address : { Street : "310 Woods Ln", City : "Atlanta", State : "GA", ZipCode : 30328, }, Contact: "404-374-XXXX", Email:"raje14.k@gmail.com"})

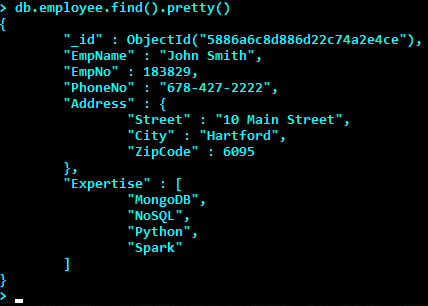












> db.employee.find().pretty()

{

"\_id" : ObjectId("5886a6c8d886d22c74a2e4ce"),

"EmpName" : "John Smith",

"EmpNo" : 183829,

"PhoneNo" : "678-427-2222",

"Address" : {

"Street" : "10 Main Street",

"City" : "Hartford",

"ZipCode" : 6095

},

"Expertise" : [

"MongoDB",

"NoSQL",

"Python",

"Spark"

]

}

{

"\_id" : ObjectId("5886a8d6d886d22c74a2e4cf"),

"EmpName" : "Rajesh Kumar",

"EmpNo" : 374528,

"PhoneNo" : "678-427-2222",

"Address" : {

"Street" : "123 Marsh Trail",

"City" : "Atlanta",

"ZipCode" : "30328"

},

"Expertise" : [

"MongoDB",

"NoSQL",

"Python",

"Spark"

]

}

{

"\_id" : ObjectId("5886aa27d886d22c74a2e4d0"),

"EmpName" : "Rajesh Kumar",

"EmpNo" : 374528,

"PhoneNo" : "678-427-2222",

"Address" : {

"Street" : "123 Marsh Trail",

"City" : "Atlanta",

"ZipCode" : "30328"

},

"Expertise" : [

"MongoDB",

"NoSQL",

"Python",

"Spark"

],

"CurrentRole" : "Data Modeler"

}

{

"\_id" : ObjectId("5886aaa8d886d22c74a2e4d1"),

"EmpName" : "Rajesh Kumar",

"EmpNo" : 374528,

"PhoneNo" : "678-427-2222",

"Address" : {

"Street" : "123 Marsh Trail",

"City" : "Atlanta",

"ZipCode" : "30328"

},

"Expertise" : [

"MongoDB",

"NoSQL",

"Python",

"Spark"

],

"CurrentRole" : "Data Modeler",

"ClientLocation" : "Hartford"

}

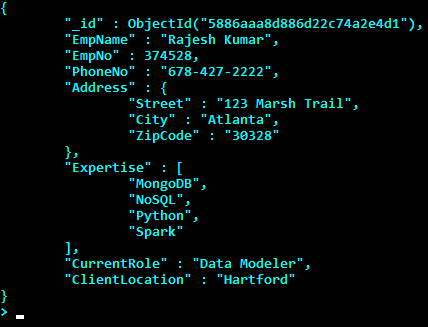
>

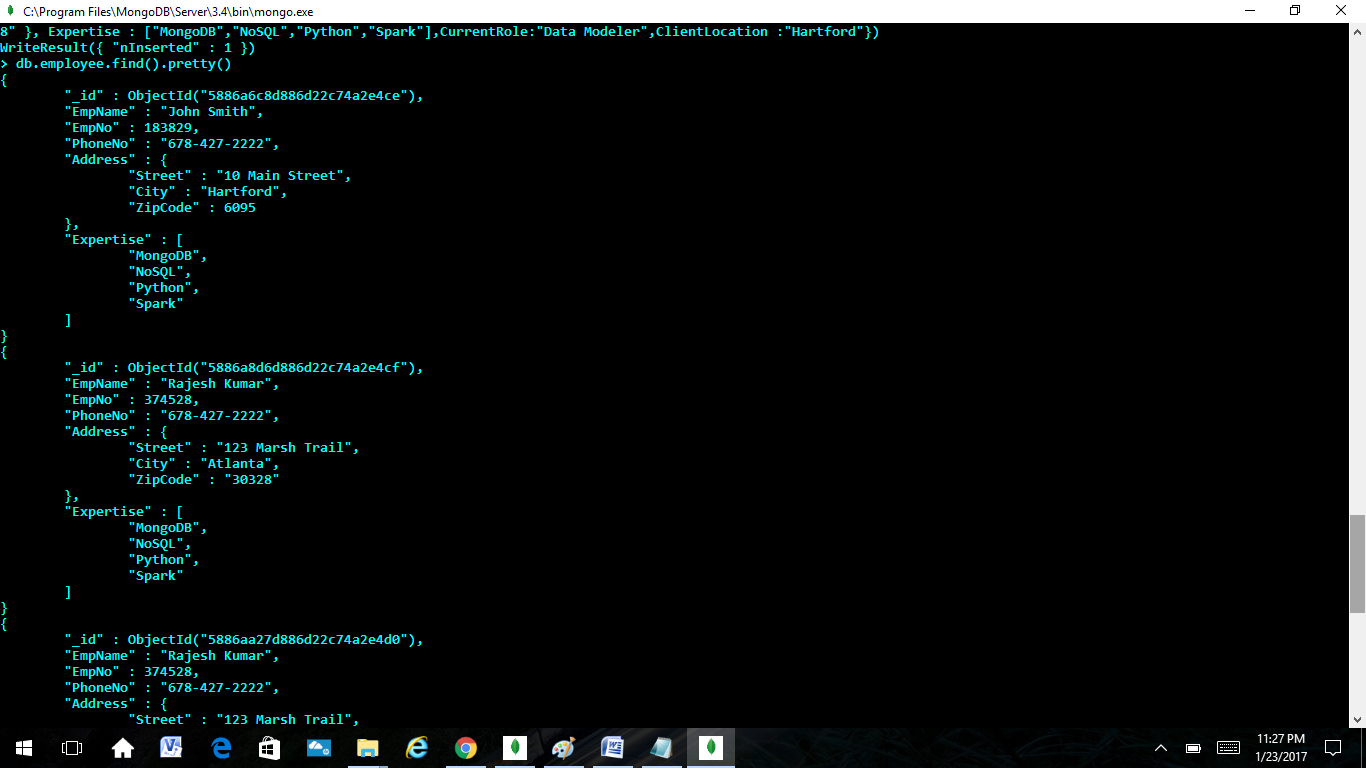
> db.employee.insert( { EmpName : "Rajesh Kumar", EmpNo : 374528, PhoneNo : "678-427-2222", Address : { Street : "123 Marsh Trail", City : "Atlanta" , ZipCode : "30328" }, Expertise : ["MongoDB","NoSQL","Python","Spark"]})

WriteResult({ "nInserted" : 1 })

> db.employee.insert( { EmpName : "Rajesh Kumar", EmpNo : 374528, PhoneNo : "678-427-2222", Address : { Street : "123 Marsh Trail", City : "Atlanta" , ZipCode : "30328" }, Expertise : ["MongoDB","NoSQL","Python","Spark"],CurrentRole:"Data Modeler",ClientLocation :"Hartford"})

WriteResult({ "nInserted" : 1 })





Aggregation

The Aggregation Framework The aggregation framework lets you transform and combine documents in a collection. Basically, you build a pipeline that processes a stream of documents through several building blocks: filtering, projecting, grouping, sorting, limiting, and skipping. For example, if you had a collection of magazine articles, you might want find out who your most prolific authors were. Assuming that each article is stored as a document in MongoDB, you could create a pipeline with several steps:

1. Project the authors out of each article document.

2. Group the authors by name, counting the number of occurrences.

3. Sort the authors by the occurrence count, descending.

4. Limit results to the first five. Each of these steps maps to a aggregation framework operator: 1. {"$project" : {"author" : 1}} This projects the author field in each document.

The syntax is similar to the field selector used in querying: you can select fields to project by specifying "fieldname" : 1 or exclude fields with "fieldname" : 0. After this operation, each document in the results looks like: {"\_id" : id, "au thor" : "authorName"}. These resulting documents only exists in memory and are not written to disk anywhere. 2. {"$group" : {"\_id" : "$author", "count" : {"$sum" : 1}}} This groups the authors by name and increments "count" for each document an author appears in. First, we specify the field we want to group by, which is "author". This is indicated by the "\_id" : "$author" field. You can picture this as: after the group there will be one result document per author, so "author" becomes the unique identifier ("\_id"). The second field means to add 1 to a "count" field for each document in the group. Note that the incoming documents do not have a "count" field; this is a new field created by the "$group". At the end of this step, each document in the results looks like: {"\_id" : "author Name", "count" : articleCount}. 3. {"$sort" : {"count" : -1}} This reorders the result documents by the "count" field from greatest to least. 4. {"$limit" : 5} This limits the result set to the first five result documents. To actually run this in MongoDB, pass each operation to the aggregate() function: >

db.articles.aggregate({"$project" : {"author" : 1}}, ... {"$group" : {"\_id" : "$author", "count" : {"$sum" : 1}}}, ... {"$sort" : {"count" : -1}}, ... {"$limit" : 5}) { "result" : [ { "\_id" : "R. L. Stine", "count" : 430 }, { "\_id" : "Edgar Wallace", "count" : 175 }, { "\_id" : "Nora Roberts", "count" : 145 }, { "\_id" : "Erle Stanley Gardner",

"count" : 140 }, { "\_id" : "Agatha Christie", "count" : 85 } ], "ok" : 1 } aggregate() returns an array of result documents, showing the five most prolific authors