

## Select a Database

Before starting this lab make sure you have a running MongoDB DBMS (see previous instructions). After starting the `mongosh` client your session will use the `test` database by default. A database is equivalent to a **user schema** in the relational world. Note to start the mongo client run `mongosh.exe` . (found in the `mongosh` bin folder)

```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
C:\>mongosh
Current Mongosh Log ID: 617bc06e0004872e7b5ca02f
Connecting to:  mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
Using MongoDB:  5.0.3
Using Mongosh:  1.0.7
```

You should have a command prompt as follows:

```
test>
```

1. Issue the following operation at the command libeto report the name of the current database:

```
db
```

2. From the `mongo` shell, display the list of databases, with the following operation:

```
show dbs
[show databases - is another way]
```

3. Switch to a new database named `mydb`, with the following operation:

```
use mydb
```

Confirm that your session has the `mydb` database as context, by checking the value of the `db` object, which returns the name of the current database, as follows:

```
db
```

Note: At this point, if you issue the `show dbs` operation again, it will not include the `mydb` database. MongoDB will not permanently create a database until you insert data into that database. `show databases` also returns a list of databases.

## Display mongo Help

At any point, you can access help for the `mongo` shell using the following operation:

```
help
```

Furthermore, you can append the `.help()` method to some JavaScript methods, any cursor object, as well as the `db` and `db.collection` objects to return additional help information.

## Create a Collection and Insert Documents

Insert documents into a new `collection` named `testData` within the new `database` named `mydb`. A collection is akin to a table in the relational world. However, a collection has no explicit definition in the database.

MongoDB will create a collection implicitly upon its first use. You do not need to create a collection before inserting data. Furthermore, because MongoDB uses `dynamic schemas`, you also need not specify the structure of your documents before inserting them into the collection.

1. Create two simple documents named `j` and `k` by using the following sequence of JavaScript operations:

```
j = { "name" : "mongo" }  
k = { "x" : 3 }
```

2. Insert the `j` and `k` documents as well another JSON document into the `testData` collection with the following sequence of operations ( note the dbms will save these very different documents in the in the collection. The DBMs does not complain!:

```
db.testData.insertOne( j )  
db.testData.insertOne( k )  
db.testData.insertOne( {  
    "_id" : 1,  
    "username" : "johnclayton",  
    "password" : "bf38395d61b821748ae1"  
})
```

When you insert the first document, the `mongod` daemon will create both the `mydb` database and the `testData` collection.

3. Confirm that the `testData` collection exists in the `mydb` database. Issue the following operations one at a time:

```
show collections
db
show dbs
```

At this point, the only collection is `testData`. All `mongod` databases also have a `system.indexes` collection.

4. Confirm that the documents exist in the `testData` collection by issuing a query on the collection using the `find()` method:

```
db.testData.find().pretty()
```

This operation returns the following results. The `ObjectId` values will be unique:

```
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name":"mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{
  "_id" : 1,
  "username" : "johnclayton",
  "password" : "bf38395d61b821748ae1"
}
```

All MongoDB documents must have an `_id` field with a unique value. These operations do not explicitly specify a value for the `_id` field, so `mongo` creates a unique `ObjectId` value for the field before inserting it into the collection. `_id` acts the primary key. Create documents as follows

```
db.testData.insertOne({_id:123456, "name":"john"})
db.testData.insertOne({_id:123456, "name":"mary"})
```

What do you observe?

Inserting a duplicate value for any key that is part of a *unique index*, such as `_id`, throws an exception.

Let us search for John. Try each statement one at a time. What do you observe?

```
db.testData.find({"name":"john"})
db.testData.find({"name":"john"}, {"name":0})
db.testData.find({"name":"john"}, {"name":1, "_id":0})
```

You should notice in the 2<sup>nd</sup> {} curly braces we can turn on or off what name value pairs are displayed.

### Deleting a Database

After a database has been created, it exists in MongoDB until the administrator deleted it. To delete a database from the MongoDB shell, use the `dropDatabase()` method for example, to delete the `mydn` database, you use the following commands to change to the `mydb` database and delete it.

```
use mydb
db.dropDatabase()
```

Be aware that `dropDatabase()` removes the current database, but it does not change the current database handle, If you drop a database and then create a collection using the handle without changing the current database first, you will recreate the dropped database.

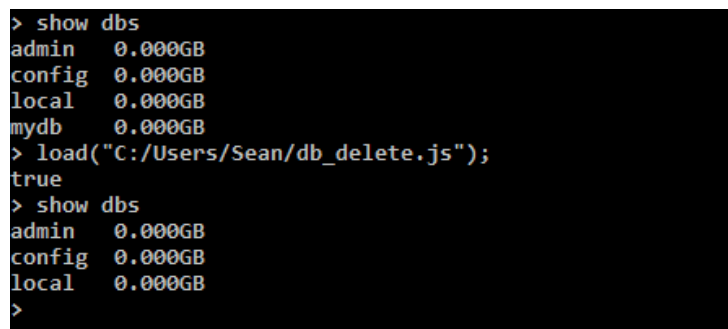
To delete a database using JavaScript, You create an instance of the database from a Connection object and then call `dropDatabase()` using that object

```
mongo = new Mongo("localhost");
myDB = mongo.getDB("mydb");
myDB.dropDatabase();
```

Create a javascript file with the above commands.

In your Mongo Shell Run the script as follows

```
load("C:/Users/Sean/db_delete.js");
-- change to suit your environment
```

A screenshot of a MongoDB shell session. It shows the command `load("C:/Users/Sean/db_delete.js");` being executed, which returns `true`. Then, the command `show dbs` is run twice. The first time, it shows four databases: `admin` (0.000GB), `config` (0.000GB), `local` (0.000GB), and `mydb` (0.000GB). The second time, after the script execution, `mydb` is no longer listed, indicating it has been successfully deleted.

```
> show dbs
admin    0.000GB
config  0.000GB
local    0.000GB
mydb     0.000GB
> load("C:/Users/Sean/db_delete.js");
true
> show dbs
admin    0.000GB
config  0.000GB
local    0.000GB
>
```

5. You can use a `db.collection.insertMany()` inserts multiple documents into a collection. It takes the form of an array e.g. Here we create 4 documents in the `users` collection. Check the collection for the documents. How many collections do we now have in the `mydb` database?

```
db.users.insertMany([ {"username": "patdonohoe","dba_rights": true,
  "password": "bf38395d61b821748ae1"},

  {"username": "johnclayton","dba_rights": true,
  "password": "bf38395d61b821748ae1"},

  {"username": "mattmurphy","dba_rights": false,
  "password": "bf38395d61b821748ae1"},

  {"username": "mattmurphy","dba_rights": false,
  "password": "bf3787871b821748ae1"}  ])
```

### For you to do!

- Call `db.users.find().pretty()` What do you notice? We can see that an `"_id"` key was added and that the other key/value pairs were saved as we entered them. The reason for the sudden appearance of the `"_id"` If there is no `"_id"` key present when a document is inserted, one will be automatically added to the inserted document. This can be handled by the MongoDB server but will generally be done by the driver on the client side.
- Try out `db.users.findOne()` . Good if you want to get a quick look a sample document in the collection. With the `findOne()` method you can return a single *document* from a MongoDB collection. The `findOne()` . method takes the same parameters as `find()` , but returns a document rather than a cursor
- To delete a document(s) there are two methods `deleteOne` and `deleteMany` for this purpose. Both of these methods take a filter document as their first parameter. The filter specifies a set of criteria to match against in removing documents. To delete the document with the `"_id"` value of 4 , we use `deleteOne` in the mongo shell as illustrated here: `db.users.deleteOne({"dba_rights": false});`

**Note:** if there is more than one that meets the criteria, it deletes the first one it finds

- Try out `db.users.deleteMany({"dba_rights": true});` and see what it does!
- To delete the collection use the `drop()` method. e.g. `db.users.drop()`
- i) Insert 3 documents into a student collection that has fields `_id` ( to be student id), `studentname`, `address` ( containing fields `street no`, `street name` and `city`) and `area of study`. ii) Delete one of the documents meeting a criteria. iii) Remove the collection.

**Note:**

```
db.collection.find(
{criteria},
{projection}
)
```

In this example:

**criteria** is a JSON document that will specify the criteria for the selection of documents inside a collection by using some operators

**projection** is a JSON document that will specify which document's fields in a collection will be returned as the query result

e.g.

```
db.testData.find({"name":"john"},
{"name":1, "_id":0})
```

In this example:

`{"name":"john"}` is the criteria

`{"name":1, "_id":0}` is the projection (1 & 0 are boolean values representing true & false)

**For you to do!** Insert your users documents again and write some queries. Write some queries to search for certain values in your users collection. Display certain name:value pairs.

### Insert Documents using a For Loop or a JavaScript Function

You can add documents to a new or existing collection by using a JavaScript `for` loop run from the [mongo](#) shell.

1. From the [mongo](#) shell, insert new documents into the `mynewcollection` collection using the following `for` loop. If the `mynewcollection` collection does not exist, MongoDB creates the collection implicitly.

```
for (var i = 1; i <= 25; i++) db.mynewcollection.insertOne( { x : i
} )
```

2. You can create a JavaScript function in your mongo shell session to generate the above data. The `insertData()` JavaScript function, shown here, creates new data for use in testing or training by either creating a new collection or appending data to an existing collection:

```
function insertData(dbName, colName, num) {

    var col = db.getSiblingDB(dbName).getCollection(colName);
    for (i = 0; i < num; i++) {
        col.insertOne({x:i});
    }
    print(col.countDocuments());
}
```

The `insertData()` function takes three parameters: a database, a new or existing collection, and the number of documents to create. The function creates documents with an `x` field that is set to an incremented integer

**Exercise:** Copy the above function into the Mongo shell and run and create it. Call it as follows:

```
insertData("mynewdb", "mynewcollection", 100)
```

This operation inserts 100 documents into the `mynewcollection` collection in the `mynewdb` database. If the collection and database do not exist, MongoDB creates them implicitly before inserting documents.

Note: functions only exist for the current sessions. For functions you use regularly you can store them in your `.mongorc.js` file. The `mongo` shell reads this file and loads the function for you every time you start a session.

## Working with the Cursor

We can use the **find** interface to execute a query in MongoDB. The find interface will select the documents in a collection and return a cursor for the selected documents. The `mongo` shell then iterates over the cursor to display the results. Rather than returning all results at once, the shell iterates over the cursor 20 times to display the first 20 results and then waits for a request to iterate over the remaining results. In the shell, use `it` to iterate over the next set of results.

The procedures in this section show other ways to work with a cursor. For comprehensive documentation on cursors, see [Iterate the Returned Cursor](#).

## Iterate over the Cursor with a Loop

We will assume the database named `mynewdb` and a collection named `mynewcollection`.

1. In the MongoDB JavaScript shell, query the `mynewcollection` collection you created in the `mynewdb` database and assign the resulting cursor object to the `c` variable:

```
use mynewdb
var c = db.mynewcollection.find()
```

2. Print the full result set by using a `while` loop to iterate over the `c` variable:

```
while (c.hasNext() ) printjson( c.next() )
```

The `hasNext()` function returns true if the cursor has documents.

The `next()` method returns the next document. The `printjson()` method renders the document in a JSON-like format.

## Use Array Operations with the Cursor

The following procedure lets you manipulate a cursor object as if it were an array:

1. In the [mongo](#) shell, query the `mynewcollection` collection and assign the resulting cursor object to the `c` variable:

```
var c = db. mynewcollection.find()
```

2. To find the document at the array index 4, use the following operation:

```
printjson( c [ 4 ] )
```

When you access documents in a cursor using the array index notation, [mongo](#) first calls the `cursor.toArray()` method and loads into RAM all documents returned by the cursor. The index is then applied to the resulting array. This operation iterates the cursor completely and exhausts the cursor.

Note for very large result sets, [mongo](#) may run out of available memory.

## Return a Single Document from a Collection

As you know, with the [findOne\(\)](#) method you can return a single *document* from a MongoDB collection. The [findOne\(\)](#) method takes the same parameters as [find\(\)](#), but returns a document rather than a cursor. To retrieve one document from the `mynewcollection` collection, issue the following command:

```
db. mynewcollection.findOne()
```



## Limit the Number of Documents in the Result Set

To increase performance, you can constrain the size of the result by limiting the amount of data your application must receive over the network.

To specify the maximum number of documents in the result set, call the [limit\(\)](#) method on a cursor, as in the following command:

```
db. mynewcollection.find().limit(3)
```

MongoDB will return the following result, with different [ObjectId](#) values:

```
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be6"), "x" : 0 }  
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be7"), "x" : 1 }  
{ "_id" : ObjectId("51a7dc7b2cacf40b79990be8"), "x" : 2 }
```

### A few exercises!

- a) Create an appropriate collection to hold movies documents. Using insertOne() method insert two movies with appropriate values for the fields. Fields to include are **imdb** (will be the primary key), **title**, **year** and **type**. For one of the documents you are inserting set **\_id** to be tt0075148.
- b) Now insert 4 movies using insertMany method (bulk insert) with the second document having **\_id** of tt0075148. What do you notice?

By default, if there is an error in a bulk insert the insert will stop at the error i.e. the following documents are not loaded. This is deemed an “ordered” insert where {ordered:true} is set for the insertMany method. Add {ordered:false} to insertMany method following the array of movie documents to allow the insert to carry on. Check the collection to see the movie documents in the collection.

```
db.movies.insertMany(  
  [  
    {  
      DOC1  
    },  
    {  
      DOC2  
    },  
    {  
      DOC3  
    },  
    {  
      DOC4  
    },  
  ],  
  {ordered: true}  
);
```

**Note** DOC1, DOC2, DOC3 and DOC4 are your 4 JSON movie documents

- c) Create a books collection and insert book\_id of 123345, title of Mother Night and author of Kurt Vonnegut.
- d) Insert 4 more books using a bulk insert
- e) Query the books collection showing only the title and author of the books.
- f) Execute the following:  
**db.books.update({"book\_id":12345}, {\$set: {"quantity": 10 }})**  
**db.books.update({"book\_id":12345}, {\$set: {"quantity": 10 }}, {"upsert": true})**  
Examine the document what has happened?

## A note on Datatypes and Objectid

### Common Datatypes

MongoDB adds support for a number of common data types while keeping JSON's essential key/value-pair nature. Exactly how values of each type are represented varies by language, but this is a list of the commonly supported types and how they are represented as part of a document in the shell. The most common types are **(try out the inserts and find them in the collection)**:

**Null** - The null type can be used to represent both a null value and a nonexistent field:

```
db.datatypes.insert({"x" : null})
```

**Boolean** - There is a boolean type, which can be used for the values true and false :

```
db.datatypes.insert(x" : true} )
```

**Number** - The shell defaults to using 64-bit floating-point numbers. Thus, these numbers both look "normal" in the shell:

```
db.datatypes.insert({"x" : 3.14})
db.datatypes.insert({"x" : 3} )
```

For integers, use the NumberInt or NumberLong classes, which represent 4-byte or 8-byte signed integers, respectively.

```
db.datatypes.insert({"x" : NumberInt("3")})
db.datatypes.insert({"x" : NumberLong("3")} )
```

**String** - Any string of UTF-8 characters can be represented using the string type:

```
db.datatypes.insert({"x" : "foobar"} )
```

**Date** - MongoDB stores dates as 64-bit integers representing milliseconds since the Unix epoch (January 1, 1970). The time zone is not stored:

```
db.datatypes.insert({"x" : new Date() } )
```

### Regular expression

Queries can use regular expressions using JavaScript's regular expression syntax:

```
db.datatypes.insert({"x" : /foobar/i} )
```

### Array

**Sets** or lists of values can be represented as arrays:

```
db.datatypes.insert({"x" : ["a", "b", "c"]} )
```

### Embedded document

Documents can contain entire documents embedded as values in a parent document

```
db.datatypes.insert({"x" : {"foo" : "bar"}})
```

## Object ID

An object ID is a 12-byte ID for documents: The "\_id" key's value can be any type, but it defaults to an ObjectId . In a single collection, every document must have a unique value for "\_id" , which ensures that every document in a collection can be uniquely identified. The ObjectId class is designed to be lightweight, while still being easy to generate in a globally unique way across different machines. MongoDB was designed to be a distributed database, it was important to be able to generate unique identifiers in a sharded environment.

ObjectIds use 12 bytes of storage, which gives them a string representation that is 24 hexadecimal digits: 2 digits for each byte.

The 12 bytes of an ObjectId are generated as follows:

- The first four bytes of an ObjectId are a timestamp in seconds since the epoch. This provides a couple of useful properties:
- The next five bytes of an ObjectId are a random value.
- The final three bytes are a counter that starts with a random value to avoid generating colliding ObjectIds on different machines.

These first nine bytes of an ObjectId therefore guarantee its uniqueness across machines and processes for a single second. The last three bytes are simply an incrementing counter that is responsible for uniqueness within a second in a single process. This allows for up to  $2^{24}$  (16,777,216) unique ObjectIds to be generated per process in a single second.

```
db.datatypes.insert({"x" : ObjectId()})
```