**Steps to Run:-**

1) In cmd 1 execute following command to point db path

mongod.exe --dbpath "C:\data\db"

MLAB Creds :

User Name : Aditya Vibhute

Pswd : @Pvibhute21

Repair:-

mongod.exe --dbpath "C:\data\db" --repair

2) In cmd 2 execute following command to add db

https://www.youtube.com/watch?v=wtIvu085uU0

3) DB connect :-

<https://stackoverflow.com/questions/26626186/object-mongoclient-has-no-method-open?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa>

**Create a database in MongoDB.**

Basic syntax of **use DATABASE** statement is as follows −

To check your currently selected database, use the command **db**

>db

mydb

use DATABASE\_NAME

If you want to create a database with name **<mydb>**, then **use DATABASE**statement would be as follows −

>use mydb

switched to db mydb

If you want to check your databases list, use the command **show dbs**.

>show dbs

local 0.78125GB

test 0.23012GB

The dropDatabase() Method

MongoDB **db.dropDatabase()** command is used to drop a existing database.

Syntax

Basic syntax of **dropDatabase()** command is as follows −

db.dropDatabase()

This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

The createCollection() Method

MongoDB **db.createCollection(name, options)** is used to create collection.

Syntax

Basic syntax of **createCollection()** command is as follows −

db.createCollection(name, options)

go through :-

<https://www.tutorialspoint.com/mongodb/mongodb_create_collection.htm>

A **journaling file system** is a [file system](https://en.wikipedia.org/wiki/File_system) that keeps track of changes not yet committed to the file system's main part by recording the intentions of such changes in a data structure known as a "[journal](https://en.wikipedia.org/wiki/Journal_(computing))", which is usually a [circular log](https://en.wikipedia.org/wiki/Circular_log). In the event of a system crash or power failure, such file systems can be brought back online more quickly with a lower likelihood of becoming corrupted.[[1]](https://en.wikipedia.org/wiki/Journaling_file_system#cite_note-developerworks-1-1)[[2]](https://en.wikipedia.org/wiki/Journaling_file_system#cite_note-ostep-1-2)

The drop() Method

MongoDB's **db.collection.drop()** is used to drop a collection from the database.

Syntax

Basic syntax of **drop()** command is as follows −

db.COLLECTION\_NAME.drop()

drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

The insert() Method

To insert data into MongoDB collection, you need to use MongoDB's **insert()**or **save()** method.

Syntax

The basic syntax of **insert()** command is as follows −

>db.COLLECTION\_NAME.insert(document)

\_id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id,

3 bytes incrementer)

The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()**method.

Syntax

The basic syntax of **find()** method is as follows −

>db.COLLECTION\_NAME.find()

The pretty() Method

To display the results in a formatted way, you can use **pretty()** method.

Syntax

>db.mycol.find().pretty()

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Syntax** | **Example** | **RDBMS Equivalent** |
| Equality | {<key>:<value>} | db.mycol.find({"by":"tutorials point"}).pretty() | where by = 'tutorials point' |
| Less Than | {<key>:{$lt:<value>}} | db.mycol.find({"likes":{$lt:50}}).pretty() | where likes < 50 |
| Less Than Equals | {<key>:{$lte:<value>}} | db.mycol.find({"likes":{$lte:50}}).pretty() | where likes <= 50 |
| Greater Than | {<key>:{$gt:<value>}} | db.mycol.find({"likes":{$gt:50}}).pretty() | where likes > 50 |
| Greater Than Equals | {<key>:{$gte:<value>}} | db.mycol.find({"likes":{$gte:50}}).pretty() | where likes >= 50 |
| Not Equals | {<key>:{$ne:<value>}} | db.mycol.find({"likes":{$ne:50}}).pretty() | where likes != 50 |

**MongoDB Update() Method:**

The basic syntax of **update()** method is as follows −

>db.COLLECTION\_NAME.update(SELECTION\_CRITERIA, UPDATED\_DATA)

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'.

>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})

>db.mycol.find()

{ "\_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}

{ "\_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "\_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

>

By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

>db.mycol.update({'title':'MongoDB Overview'},

{$set:{'title':'New MongoDB Tutorial'}},{multi:true})

In **MongoDB**, the 'update ()' and '**save** ()' **methods**are used to update the document into a collection. The 'update ()' **method** is used to update the values present in the existing document whereas '**save** ()'**method** replaces the entire existing document with the new document which is passed in the '**save** ()'**method**.

**MongoDB Save() Method**

The **save()** method replaces the existing document with the new document passed in the save() method.

>db.COLLECTION\_NAME.save({\_id:ObjectId(),NEW\_DATA})

Eg. db.marks.save({"\_id": ObjectId("5ad80250f0917d8bb6da12c8"),"name":"Gamora","marks":70})

**The remove() Method**

MongoDB's **remove()** method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

* **deletion criteria** − (Optional) deletion criteria according to documents will be removed.
* **justOne** − (Optional) if set to true or 1, then remove only one document.

**Syntax**:

>db.COLLECTION\_NAME.remove(DELLETION\_CRITTERIA)

**Projection:**

In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

## The find() Method:

MongoDB's **find()** method, explained in [MongoDB Query Document](https://www.tutorialspoint.com/mongodb/mongodb_query_document.htm" \t "_blank) accepts second optional parameter that is list of fields that you want to retrieve. In MongoDB, when you execute **find()** method, then it displays all fields of a document. To limit this, you need to set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.

### Syntax

The basic syntax of **find()** method with projection is as follows −

>db.COLLECTION\_NAME.find({},{KEY:1})

Eg. db.marks.find({}, {"marks":98, \_id:0});

db.marks.find({}, {"name":0}) //Result :- name field will be hidden

db.marks.find({}, {"marks":0}) //Result :- marks field will be hidden

**Please note** **\_id** field is always displayed while executing **find()** method, if you don't want this field, then you need to set it as 0.

**The Limit() Method**

To limit the records in MongoDB, you need to use **limit()** method. The method accepts one number type argument, which is the number of documents that you want to be displayed.

**Syntax**

The basic syntax of **limit()** method is as follows −

>db.COLLECTION\_NAME.find().limit(NUMBER)

Eg. db.marks.find({}).limit(2) // Res : only first two documents will be shown

**MongoDB Skip() Method**

Apart from limit() method, there is one more method **skip()** which also accepts number type argument and is used to skip the number of documents.

Syntax

The basic syntax of **skip()** method is as follows −

>db.COLLECTION\_NAME.find().limit(NUMBER).skip(NUMBER)

Eg. db.marks.find({},{"name":"Gamora"}).limit(2).skip(1)

// skip(1) :- results to 1 field to skip

**The sort() Method**

To sort documents in MongoDB, you need to use **sort()** method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. **1 is used for ascending order while -1 is used for descending order**.

Eg. db.marks.find({},{\_id:0}).sort({"marks": 1}).pretty() //Ascending Order

db.marks.find({},{\_id:0}).sort({"marks": -1}).pretty() //Descending Order

**Indexes:** Indexes support the efficient resolution of queries.

**The ensureIndex() Method**

To create an index you need to use ensureIndex() method of MongoDB.

Syntax

The basic syntax of **ensureIndex()** method is as follows().

>db.COLLECTION\_NAME.ensureIndex({KEY:1})

Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use

-1.

**The aggregate() Method**

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL count(\*) and with group by is an equivalent of mongodb aggregation.

Eg. db.employee.aggregate([{$group : { \_id:"$gender", avg\_group:{$avg : "$age"} }}]) // res : object with avg number of male/female

Eg. db.employee.aggregate({$group : { \_id:"$gender", gender\_Group:{$sum : 1} }}) // res : object with avg no. of male female

**Mongoose Tutorial:**

[tutorial](http://tutorialtous.com/mongoose/mongooseterminology.php)

**Schema :**

"Mongoose Schema will create a mongodb collection and defines the shape of the documents within that collection".

var userSchema = new Schema({

userid : String,

chips : {type:double, default:0.0},

regdate : {type:Date, default: Date.now,required:true}

})

Ref :- [reference](http://tutorialtous.com/mongoose/mongooseschema.php)

var Mice = mongoose.model('Mice', MouseSchema, 'Mice');

// to prevent pluralize collection name

var dataSchema = new Schema({..}, { collection: 'data' })

**Using primary (pk) and foreign key (fk):**

While creating a database schema, make sure pk and fk mapped properly.If you have one document containing userId as pk and need to use same value in another document, then it should be used as fk.

**Note: Mongo db has no concept of fk,** we can retrieve documents from different collections with a single query using Mongoose.

var TeamSchema = new Schema({

name: {

type: String,

required: true

}

});

var EmployeeSchema = new Schema({

name: {

first: {

type: String,

required: true

},

last: {

type: String,

required: true

}

},

team: {

type: Schema.Types.ObjectId,

ref: 'Team'

}

}); // here team value is used from earlier document and ref Team is model name

var Team = mongoose.model('Team', TeamSchema);

[Ref](https://www.sitepoint.com/premium/books/full-stack-javascript-development-with-mean/preview/interacting-with-mongodb-using-mongoose-efe3bd3)

**Model:**

Schemas define a structure we will apply that Schema part to a model, means ****"Models are the constructors compiled from our schema definitions".****

We can create the Models using mongoose.model() constructor as

**var modelinstance= mongoose.model('collection-name',schemaname);**

for the above 'userSchema' example check the model creation

****var userModel = mongoose.model('users',userSchema);****

**Connect():**

Opens the default mongoose connection.  
**Syntax:**  
**connect(uri(s), [options], [callback])**  
Example**:**

var mongoose=require('mongoose');

mongoose.connect("mongodb://mongodb.tutorialtous.com/mongoose",function(error){

if(error){

console.log("error"+error);

}else{

console.log("open done")

}

});

**open()**

Opens the connection to MongoDB.   
  
Syntax:-  
  **open**(connection\_string, [database], [port], [options], [callback])

* connection\_string <String> mongodb://uri or the host to which we are connecting
* [database] <String> database name //optional
* [port] <Number> database port //optional
* [options] <Object> options //optional
  + db, server, replset, user, pass, auth are the options
* [callback] <Function> //optional

var mongoose=require('mongoose');

var con = mongoose.createConnection();

var cb = function(err){

if(!err)

console.log(err);

else

console.log("connection opened");

};

con.open("mongodb://localhost/tutorialtous",cb);

**OpenSet():**

Opens the connection to a replica set.  
  
Syntax:-  
   **openSet**(uris, [database], [options], [callback])

* uris <String> comma-separated mongodb:// URIs
* [database] <String> database name if not included in uris
* [options] <Object> passed to the internal driver
* [callback] <Function>

**Note:-**  
Functionality of openSet() will be similar to open() except openSet() works with replicaSets

**Difference in connect() and createConnection():**

So far we've seen how to connect to MongoDB using Mongoose's default connection. At times we may need multiple connections open to Mongo, each with different read/write settings, or maybe just to different databases for example. In these cases we can utilize mongoose.createConnection() which accepts all the arguments already discussed and returns a fresh connection for you.

**ReadyState:**

It is property of connection which holds state of the connection

| **State** | **Description** |
| --- | --- |
| 0 | disconnected |
| 1 | connected |
| 2 | connecting |
| 3 | disconnecting |

**Note: Mongoose get connected to db even before connection establishment.**

Mongoose lets you start using your models immediately, without waiting for mongoose to establish a connection to MongoDB.

That's because mongoose buffers model function calls internally. This buffering is convenient, but also a common source of confusion. Mongoose will not throw any errors by default if you use a model without connecting.

**Find():**

We need to call find() methods with Model Instances unlike save(), means like userModel.find() is valid. We can also use this method to search items in document as per input.

emo.find({age : {$gt : 27}},function(err, log){

if(err) return err;

else

console.log("Logg data obj", log);

}) // this will return Obj containing users age more than 27

**Syntax:-** find(conditions, [projection], [options], [callback])

[Ref](http://tutorialtous.com/mongoose/mongoosemodel.php)

**Update():**

Updates documents in the database without returning them.  
Syntax:-  
   **update(conditions, doc, [options], [callback])**  
Example:-

emo.update({age : "Aditya"},{$set: {age : 22}},function(err, log){

if(err) return err;

else

console.log("Logg data obj", log);

}) // this method can update only one document.

To update multiple documents add :- {multi : true}

**Remove():**

Removes document/s from the collection.  
  
Syntax:-  
   **remove(conditions, [callback])**

eg. emo.remove({age : 39} ,function(err, log){

if(err) return err;

else

console.log("Logg data obj", log);

})

**findOne():**

Finds a single document by its \_id field.   
Syntax:-  
   **findOne([conditions], [projection], [options], [callback])**

emo.findOne({age : 26} ,function(err, log){

if(err) return err;

else

console.log("Logg data obj", log);

})

**findOneAndRemove():**

Issue a mongodb findAndModify remove command.   
  
Syntax:-  
    **findOneAndRemove(conditions, [options], [callback])**

emo.findOneAndRemove({age : 26} ,function(err, log){

if(err) return err;

else

console.log("Logg data obj", log);

})

The difference between remove() and findOneAndRemove() is that it will remove only one document event multiple is set true.

**Note :** Foroperations below, we had used mlab database to update and find.

**findByIdAndUpdate():**

Finds a matching document, updates it according to the update arg, passing any options, and returns the found document (if any) to the callback. The query executes immediately if callback is passed else a Query object is returned.   
  
Syntax:-  
   **findByIdAndUpdate(id, [update], [options], [callback])**

emo.findByIdAndUpdate('5b827cfc0ff3981c281a51ff',{$set: {age : 76}}, function(err, logs){

if(err) return err

else

console.log("I am here ", logs);

})

**findById():**

Finds a single document by its \_id field. findById(id).   
  
Syntax:-  
    **findById(id, [projection], [options], [callback])**

Eg.

emo.findById('5b82c31373e94223d4a21168', function(err, logs){

if(err) return err

else

console.log("I am here ", logs);

})

**findByIdAndRemove():**

Issue a mongodb findAndModify remove command by a document's \_id field. findByIdAndRemove(id, ...) is equivalent to findOneAndRemove({ \_id: id }, ...).

**eg.**

emo.findByIdAndRemove('5b835ffe454a1501dc2bcdd9', function(err, logs){

if(err) return err

else

console.log("I am here ", logs);

})

**findByIdAndUpdate():**

Finds a matching document, updates it according to the update arg, passing any options, and returns the found document (if any) to the callback. The query executes immediately if callback is passed else a Query object is returned.

Syntax:-  
   **findByIdAndUpdate(id, [update], [options], [callback])**

Q. Update multiple value in document.

Q. Local db not saving old collection using mongoose

**findByIdAndUpdate()**

Finds a matching document, updates it according to the update arg, passing any options, and returns the found document (if any) to the callback. The query executes immediately if callback is passed else a Query object is returned.  
  
Syntax:-  
   **findByIdAndUpdate(id, [update], [options], [callback])**

eg.

emo.findByIdAndUpdate('5b82c5b5d0ee6703e84ac771', {$set : {name : "Hello"}}, function(err, logs){

if(err) return err

else

console.log("I am here ", logs);

})

**count()**

Counts number of matching documents in a database collection.

Syntax:-  
   **count(conditions, [callback])**

emo.count({}, function(err, logs){

if(err) return err

else

console.log("I am here ", logs);

}) // return number of enteries

**where():**

Creates a Query, applies the passed conditions, and returns the Query.   
  
Syntax:-  
   **where(path, [val])**

userModel.where('chips').lt(10000);

userModel.where('chips').lt(10000).where('country').eq('USA');

**Methods and Statics:**

Database logic should be encapsulated within the data model. Mongoose provides 2 ways of doing this, methods and statics. Methods adds an instance method to documents whereas Statics adds static “class” methods to the Models itself. like example below:

[Ref](https://osmangoni.info/posts/separating-methods-schema-statics-mongoose/)

**Version key (\_\_v)**

The versionKey is a property set on each document when first created by Mongoose. This keys value contains the internal revision of the document. The name of this document property is configurable. The default is \_\_v

[ref](https://mongoosejs.com/docs/guide.html#versionKey)

**Adding Date Field to DB**

Date as string :-

var myDateString = Date();

Date as object :-

var myDateObject = new Date();

db.empDatas.insert({ "name": "Harish", "lname": "Salvi", "dob": new Date()})

**$currentDate:**

Set current date to the document where lname="Salvi"

eg. db.empDatas.update({"lname":"Salvi"}, {$currentDate: {age: true}})

[Ref](https://docs.mongodb.com/manual/reference/operator/update/currentDate/#up._S_currentDate)

* a boolean true to set the field value to the current date as a Date, or
* a document { $type: "timestamp" } or { $type: "date" } which explicitly specifies the type. The operator is *case-sensitive* and accepts only the lowercase "timestamp" or the lowercase "date".

**Create time stamp:**

db.empDatas.update({"lname":"Salvi"}, {$currentDate: {age: {$type: "timestamp"},}})

### Timestamps:

BSON has a special timestamp type for *internal* MongoDB use and is **not** associated with the regular [Date](https://docs.mongodb.com/manual/reference/bson-types/#document-bson-type-date) type. Timestamp values are a 64 bit value where:

* the first 32 bits are a time\_t value (seconds since the Unix epoch)
* the second 32 bits are an incrementing ordinal for operations within a given second.

Within a single [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "bin.mongod" \o "bin.mongod) instance, timestamp values are always unique.

In replication, the [oplog](https://docs.mongodb.com/manual/reference/glossary/" \l "term-oplog) has a ts field. The values in this field reflect the operation time, which uses a BSON timestamp value