MONGODB HANDS-ON

After downloading MongoDB community server setup once done, head over to the C drive in which you have installed MongoDB. Go to program files and select the MongoDB directory.

```
C: -> Program Files -> MongoDB -> Server -> 4.0(version) -> bin
```

In the bin directory, you will find an interesting couple of executable files.

- mongod
- mongo

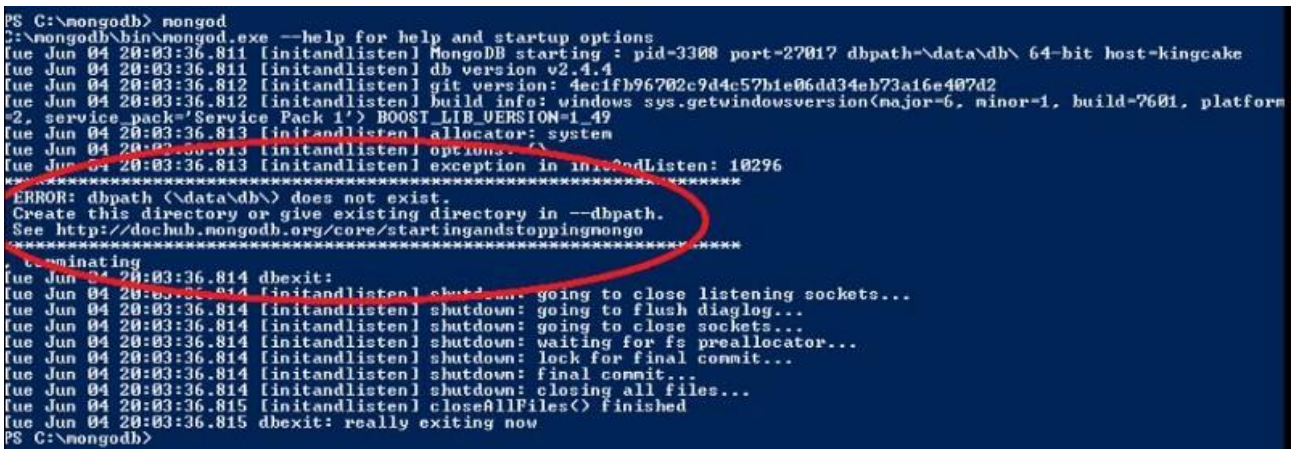
mongod stands for “Mongo Daemon”. mongod is a background process used by MongoDB. The main purpose of mongod is to manage all the MongoDB server tasks. For instance, accepting requests, responding to client, and memory management.

mongo is a command line shell that can interact with the client (for example, system administrators and developers).

Open up your command prompt inside your C drive and do the following:

```
C:\> mkdir data/dbC:\> cd dataC:\> mkdir db
```

The purpose of these directories is MongoDB requires a folder to store all data. MongoDB’s default data directory path is `/data/db` on the drive. Therefore, it is necessary that we provide those directories like so.
If you start the MongoDB server without those directories, you’ll probably see this following error:



Trying to start mongodb server without \data\db directories

After creating those two files, head over again to the bin folder you have in your mongodb directory and open up your shell inside it. Run the following command:

```
mongod
```

Now our MongoDB server is up and running! ?

In order to work with this server, we need a mediator. So open another command window inside the bind folder and run the following command:

```
mongo
```

After running this command, navigate to the shell which we ran mongod command (which is our server). You'll see a 'connection accepted' message at the end. That means our installation and configuration is successful!

Just simply run in the mongo shell:

```
db
```

```
Navindu@Navindu MINGW64 ~/Desktop
$ mongo
MongoDB shell version v4.0.5
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("ea1b013a-7d7c-43a5-8e66-2e884a9a3105")
}
MongoDB server version: 4.0.5
db
test
```

Setting up Environment Variables

To save time, you can set up your environment variables. In Windows, this is done by following the menus below:

`Advanced System Settings -> Environment Variables -> Path(Under System Variables) -> Edit`

Simply copy the path of our bin folder and hit OK! In my case it's `C:\Program Files\MongoDB\Server\4.0\bin`
Now you're all set!

Working with MongoDB

There's a bunch of GUIs (Graphical User Interface) to work with MongoDB server such as MongoDB Compass, Studio 3T and so on.

They provide a graphical interface so you can easily work with your database and perform queries instead of using a shell and typing queries manually.

1. Open up your command prompt and type `mongod` to start the MongoDB server.
2.

2. Open up another shell and type `mongo` to connect to MongoDB database server.

1. Finding the current database you're in

`db`

```
Navindu@Navindu MINGW64 ~/Desktop
$ mongo
MongoDB shell version v4.0.5
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("c2619dde-17ac-4a11-89c6-d36105e942a5")
}
MongoDB server version: 4.0.5
db
test
```

This command will show the current database you are in. `test` is the initial database that comes by default.

2. Listing databases

```
show databases
```

```
Navindu@Navindu MINGW64 ~/Desktop
$ mongo
MongoDB shell version v4.0.5
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("c2619dde-17ac-4a11-89c6-d36105e942a5")
}
MongoDB server version: 4.0.5
db
test
show databases
CrudDB  0.000GB
admin   0.000GB
config  0.000GB
local   0.000GB
```

3. Go to a particular database

```
use <your_db_name>
```

```
use local
switched to db local
db
local
```

You can check this if you try the command `db` to print out the current database name

4. Creating a Database

With RDBMS (Relational Database Management Systems) we have Databases, Tables, Rows and Columns.

But in NoSQL databases, such as MongoDB, data is stored in BSON format (a binary version of JSON). They are stored in structures called “collections”.

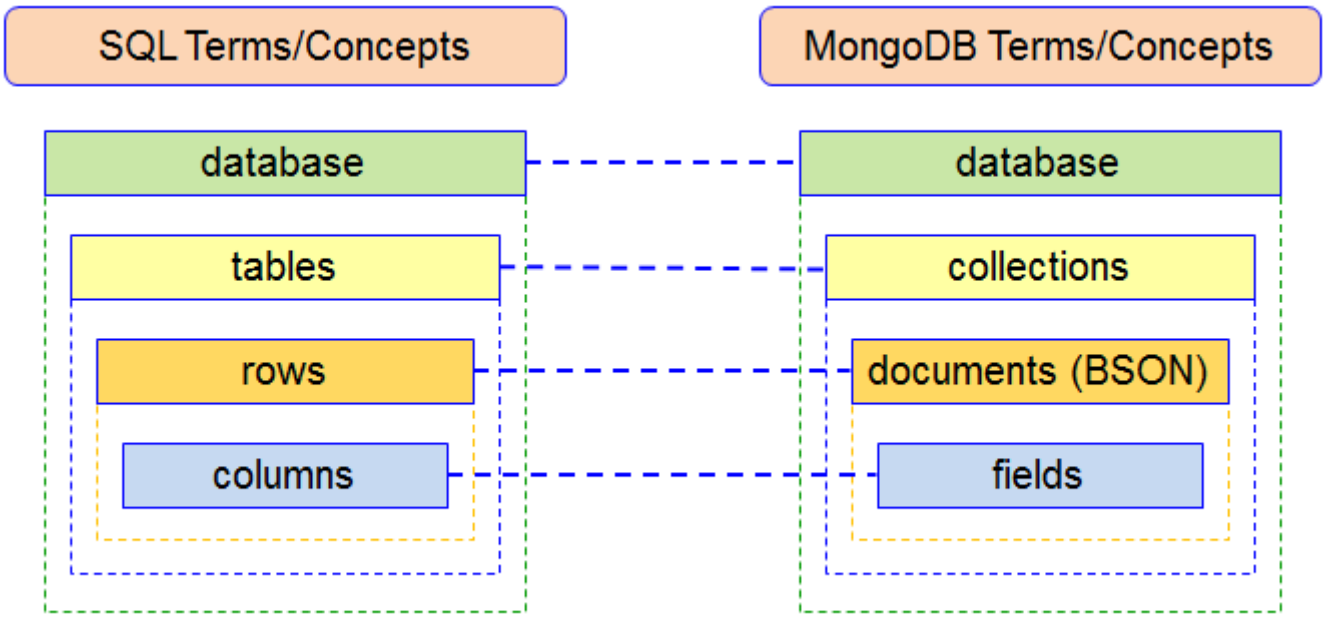
In SQL databases, these are similar to Tables.

Relational Database

Student_Id	Student_Name	Age	College
1001	Chaitanya	30	Beginnersbook
1002	Steve	29	Beginnersbook
1003	Negan	28	Beginnersbook

MongoDB

```
{
 "_id": ObjectId("....."),
 "Student_Id": 1001,
 "Student_Name": "Chaitanya",
 "Age": 30,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("....."),
 "Student_Id": 1002,
 "Student_Name": "Steve",
 "Age": 29,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("....."),
 "Student_Id": 1003,
 "Student_Name": "Negan",
 "Age": 28,
 "College": "Beginnersbook"
}
```



```
use <your_db_name>
```

In MongoDB server, if your database is present already, using that command will navigate into your database.

But if the database is not present already, then MongoDB server is going to create the database for you. Then, it will navigate into it.

After creating a new database, running the `show database` command will not show your newly created database. This is because, until it has any data (documents) in it, it is not going to show in your db list.

How Are Their Queries Different?

MySQL:

```
SELECT *

FROM customer
```

MongoDB:

```
db.customer.find()
```

Inserting records into the customer table:

MySQL:

```
INSERT INTO customer (cust_id, branch, status)

VALUES ('appl01', 'main', 'A')
```

MongoDB:

```
db.customer.insert({
```

```
        cust_id: 'appl01',

        branch: 'main',

        status: 'A'

})
```

Updating records in the customer table:

MySQL:

```
UPDATE customer

SET branch = 'main'

WHERE custage > 2
```

MongoDB:

```
db.customer.update({
        custage: { $gt: 2 }
   },
   {
        $set: { branch:'main' }
   },
   {
        multi: true
})
```

MySQL can be subject to SQL injection attacks, making it vulnerable. Since MongoDB uses object querying, where documents are passed to explain what is being queried, it reduces the risk of attack as MongoDB doesn't have a language to parse.

5. Creating a Collection

Navigate into your newly created database with the `use` command.

Actually, there are two ways to create a collection. One way is to insert data into the collection:

```
db.myCollection.insert({"name": "john", "age" : 22, "location": "colombo"})
```

This is going to create your collection `myCollection` even if the collection does not exist. Then it will insert a document with `name` and `age`. These are non-capped collections.
The second way is shown below:

2.1 Creating a Non-Capped Collection

```
db.createCollection("myCollection")
```

2.2 Creating a Capped Collection

```
db.createCollection("mySecondCollection", {capped : true, size : 2, max : 2})
```

In this way, you're going to create a collection without inserting data.

The `size : 2` means a limit of two megabytes, and `max: 2` sets the maximum number of documents to two.

6. Inserting Data

We can insert data to a new collection, or to a collection that has been created before.

There are three methods of inserting data.

1. `insertOne()` is used to insert a single document only.
2. `insertMany()` is used to insert more than one document.
3. `insert()` is used to insert documents as many as you want.

Below are some examples:

- **insertOne()**

```
db.myCollection.insertOne(
  {
    "name": "navindu",
    "age": 22
  }
```

```
)
```

- **insertMany()**

```
db.myCollection.insertMany([
  {
    "name": "navindu",
    "age": 22
  },
  {
    "name": "kavindu",
    "age": 20
  },

  {
    "name": "john doe",
    "age": 25,
    "location": "colombo"
  }
])
```

The `insert()` method is similar to the `insertMany()` method.

```
db.myCollection.insert({"name": "navindu", "age" : 22})
WriteResult({ "nInserted" : 1 })
```

7. Querying Data

```
db.myCollection.find()
```

```
db.myCollection.find()
{ "_id" : ObjectId("5c4af63bdfdc58d5ec8332ad"), "name" : "john", "age" : 22, "lo
cation" : "colombo" }
{ "_id" : ObjectId("5c4afe825e6ad6b667bd972d"), "name" : "navindu", "age" : 22 }
```

If you want to see this data in a cleaner, way just add `.pretty()` to the end of it. This will display document in pretty-printed JSON format.

```
db.myCollection.find().pretty()
```

```
db.myCollection.find().pretty()
{
        "_id" : ObjectId("5c4af63bdfdc58d5ec8332ad"),
        "name" : "john",
        "age" : 22,
        "location" : "colombo"
}
{
        "_id" : ObjectId("5c4afe825e6ad6b667bd972d"),
        "name" : "navindu",
        "age" : 22
}
```

`_id`?

How did that get there?

Well, whenever you insert a document, MongoDB automatically adds an `_id` field which uniquely identifies each document. If you do not want it to display, just simply run the following command

```
db.myCollection.find({}, _id: 0).pretty()
```

If you want to display some specific document, you could specify a single detail of the document which you want to be displayed.

```
db.myCollection.find(
  {
    name: "john"
  }
)
```

```
db.myCollection.find({ name: "john"})
{ "_id" : ObjectId("5c4af63bdfdc58d5ec8332ad"), "name" : "john", "age" : 22, "location" : "colombo"
}
```

Display people whose age is less than 25. You can use `$lt` to filter for this.

```
db.myCollection.find(
  {
    age : {$lt : 25}
  }
)
```

Similarly, `$gt` stands for greater than, `$lte` is “less than or equal to”, `$gte` is “greater than or equal to” and `$ne` is “not equal”.

8. Updating documents

```
db.myCollection.update({age: 20}, {$set: {age: 23}})
```

```
db.myCollection.update({age : 22}, {$set : {age : 23}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
db.myCollection.find();
{ "_id" : ObjectId("5c4af63bdfdc58d5ec8332ad"), "age" : 20 }
{ "_id" : ObjectId("5c4afe825e6ad6b667bd972d"), "name" : "navindu", "age" : 23 }
```

```
db.myCollection.update({name:"navindu"}, {location:"makola"});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
db.myCollection.find().pretty()
{ "_id" : ObjectId("5c4af63bdfdc58d5ec8332ad"), "age" : 20 }
{ "_id" : ObjectId("5c4afe825e6ad6b667bd972d"), "location" : "makola" }
```

```
// db.myCollection.update({name: "navindu"}, {$unset: age});
```

Removing a document

```
db.myCollection.remove({name: "navindu"});
```

10. Removing a collection

```
// db.myCollection.remove({});
```

Note, this is not equal to the `drop()` method. The difference is `drop()` is used to remove all the documents inside a collection, but the `remove()` method is used to delete all the documents along with the collection itself.

Logical Operators

MongoDB provides logical operators. The picture below summarizes the different types of logical operators.

Operand	Example	Meaning
&&	$variable1 && $variable2	Are both values true?
||	$variable1 || $variable2	Is at least one value true?
AND	$variable1 AND $variable2	Are both values true?
XOR	$variable1 XOR $variable2	Is at least one value true, but NOT both?
OR	$variable1 OR $variable2	Is at least one value true?
!	!$variable1	Is NOT something

Name	Description
`$and`	Joins query clauses with a logical `AND` returns all documents that match the conditions of both clauses.
`$not`	Inverts the effect of a query expression and returns documents that do *not* match the query expression.
`$nor`	Joins query clauses with a logical `NOR` returns all documents that fail to match both clauses.
`$or`	Joins query clauses with a logical `OR` returns all documents that match the conditions of either clause.

Display people whose age is less than 25, and also whose location is Colombo. What we could do?

We can use the `$and` operator!

```
db.myCollection.find({$and:[{age : {$lt : 25}}, {location: "colombo"}]});
```

Aggregation

Imagine if we had male and female students in a `recordBook` collection and we want a total count on each of them. In order to get the sum of males and females, we could use the `$group` aggregate function.

```
db.recordBook.aggregate([
  {
    $group : {_id : "$gender", result: {$sum: 1}}
  }
]);
```

```
db.recordBook.aggregate([{$group: {_id: "$gender", result: {$sum:1}}}]);
{ "_id" : "female", "result" : 1 }
{ "_id" : "male", "result" : 2 }
```

```
use LVCDB
db
db.LVCCollection.insert({"RegNo":"R003", "Name":"Anuj Singh",
"Location":"Bangalore", "Course":"Full Stack", "Mobile":443333333})

db.LVCCollection.insertMany( [
   {"RegNo":"R004", "Name":"Komal Singh", "Location":"Madurai", "Course":"Java",
"Mobile":12345432},
   {"RegNo":"R005", "Name":"Amit Singh", "Location":"Shimla", "Course":"Full Stack",
"Mobile":34567898},
   {"RegNo":"R006", "Name":"Reshma Singh", "Location":"UP", "Course":"Full Stack",
"Mobile":8766544332}
```

```
]);

show collections
show databases
db.LVCCollection.find( {} )
```

SHOW ALL RECORDS FROM COLLECTION

```
db.LVCCollection.find( {} )
```

```
db.LVCCollection.update({"RegNo": "R001"}, {$set: {"RegNo": "R009"}})
db.LVCCollection.find()
```

```
db.LVCCollection.aggregate([
  {
    $group : {_id : "RegNo", result: {$sum: 1}}
  }
]);
```

```
use LVCDB
db

db.LVCCollection.insert({"RegNo":"R009", "Name":"Kamal Singh",
"Location":"Bangalore", "Course":"Full Stack", "Mobile":54333333})

db.LVCCollection.insertMany( [
   {"RegNo":"R004", "Name":"Komal Singh", "Location":"Madurai", "Course":"Java",
"Mobile":12345432},
   {"RegNo":"R005", "Name":"Amit Singh", "Location":"Shimla", "Course":"Full Stack",
"Mobile":34567898},
   {"RegNo":"R006", "Name":"Reshma Singh", "Location":"UP", "Course":"Full Stack",
"Mobile":8766544332}
]);

db.createCollection("myCollection")
db.myCollection.insert({"RegNo":"R003", "Name":"Anuj Singh", "Location":"Bangalore",
"Course":"Full Stack", "Mobile":443333333})
db.myCollection.find( {} )
```

```
db.createCollection("mySecondCollection", {capped : true, size : 2, max : 3})
db.mySecondCollection.insert({"RegNo":"R006", "Name":"Ritu Saxena",
"Location":"Pune", "Course":"Angular", "Mobile":4435443333})
db.mySecondCollection.find( {} )
```

```
db.createCollection("myThirdCollection", {capped : true, size : 2, max : 4})
db.myThirdCollection.insert({"RegNo":"R006", "Name":"Ritu Saxena",
"Location":"Pune", "Course":"Angular", "Mobile":4435443333})
db.myThirdCollection.insertone({"RegNo":"R006", "Name":"Ritu Saxena",
"Location":"Pune", "Course":"Angular", "Mobile":4435443333})

db.myThirdCollection.find( {} )

show collections
show databases
db.LVCCollection.find( {} )

db.myCollection5.insertOne(
  {
    "name": "navindu",
    "age": 22
  }
)

db.myCollection5.find({})
db.myCollection.find({})
db.myCollection.find()

db

db.LVCCollection.find()
db.LVCCollection.find().pretty()
db.LVCCollection.find({}, _id: 0).pretty()
db.LVCCollection.find() {"_id":ObjectId("5f5baad8572a1ad446f3060c")}

db.LVCCollection.find(
  {
    age : {$lt:12345678}
  }
)


db.LVCCollection.update({"RegNo": "R001"}, {$set: {"RegNo": "R009"}})
db.LVCCollection.find()

db.LVCCollection.remove({Name:"Raj Kumar"});

db.mySecondCollection.remove({});

db.mySecondCollection.find({$and:[{Mobile : {$lt : 25}}, {location: "Mumbai"}]});

db.LVCCollection.aggregate([
  {
    $group : {_id : "RegNo", result: {$sum: 1}}
  }
]);
```

```
db.inventory.updateOne(
  { item: "paper" },
  {
    $set: { "size.uom": "cm", status: "P" },
    $currentDate: { lastModified: true }
  }
)
```

`db.collection.update()`	Either updates or replaces a single document that match a specified filter or updates all documents that match a specified filter. By default, the `db.collection.update()` method updates a **single** document. To update multiple documents, use the multi option.

Update a Single Document

The following example uses the `db.collection.updateOne()` method on the `inventory` collection to update the *first* document where `item` equals `"paper"`:

```
db.inventory.updateOne(
   { item: "paper" },
   {
     $set: { "size.uom": "cm", status: "P" }
   }
)
```

The following example uses the `db.collection.updateMany()` method on the `inventory` collection to update all documents where `qty` is less than `50`:

```
db.inventory.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" }


   }
)
```

- `db.collection.deleteMany()`
- `db.collection.deleteOne()`

Delete All Documents

To delete all documents from a collection, pass an empty filter document `{}` to the `db.collection.deleteMany()` method.

The following example deletes *all* documents from the `inventory` collection:

```
db.inventory.deleteMany({})
```

Delete Only One Document that Matches a Condition

To delete at most a single document that matches a specified filter (even though multiple documents may match the specified filter) use the `db.collection.deleteOne()` method.

The following example deletes the *first* document where `status` is `"D"`:

```
db.inventory.deleteOne( { status: "D" } )
```