MongoDB

# Introduction to MongoDB

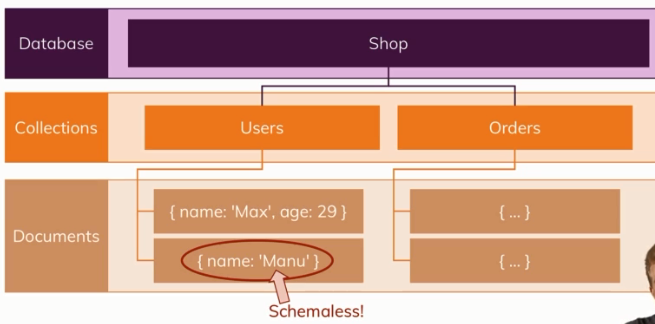
## Introduction

* MongoDB is an exciting technology, an exciting database solution and even more than just a database solution.
* You can use it in basically any project you're building. If that's a website, a mobile app, a desktop app or if you're working as a data scientist, you're probably working with some kind of data, you need to store data, you need to fetch data and you need to do all of that **fast and efficiently** without worrying too much about database configuration, setup and data structures and mongodb is an awesome solution for that.

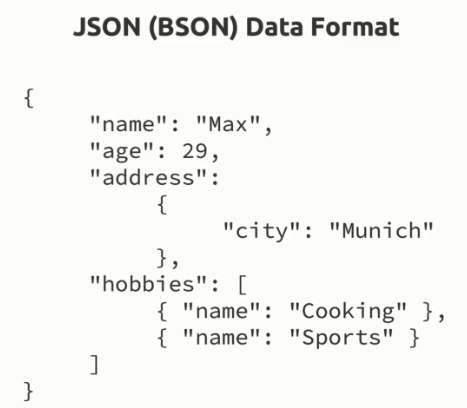
## What is MongoDB

* MongoDB is a database solution.
* The name mongodb is stemming from the word **humongous** because this database is built to store lots and lots of data and not just from a data size perspective but also in a sense that you can store lots of data and you can then work with it efficiently which of course also is super important.

## How MongoDB is different than other Databases / How it works



* mongodb is most importantly a database server that allows you to then run different databases on it.
* Relational DBs (MySQL, Oracle) have tables, in mongodb we have so-called collections.
* We can have multiple databases and multiple collections per database.
* Now inside of a collection, you have so-called documents which look like JSON objects and indeed this is basically how you store data in mongodb.
* The documents inside a collection are **schemaless**, in the sense that the second document may not look like the first one and this is a flexibility mongodb gives you.
* It is really all about flexibility, where SQL based databases are very strict about the data you have to store in there, mongodb is more flexible, you can store totally different data in one and the same collection and therefore, your database can grow with your application and your application needs.
* We can store nested data within a document (embedded document). This allows you to create complex relations between data and store them in one and the same document which makes working with it and fetching it super efficient.
* The data format used to store documents in mongodb is called **BSON (Binary JSON)**. Behind the scenes on the server, mongodb converts your JSON data to a **binary** version of it, which can basically be stored and can be queried more efficiently. But we as developers use the JSON data to interact with MongoDB.



* The whole theme of mongodb really is the **flexibility and the optimization for usability** and that is really what sets mongodb apart from other database solutions and which makes it so awesome and so efficient from a performance perspective too because you can query data in the format you need it instead of running complex restructurings on the server.

## The Key MongoDB Characteristics

* Mongodb is a so-called **NoSQL** solution because it's basically following an opposite concept or philosophy than all the SQL based databases do.

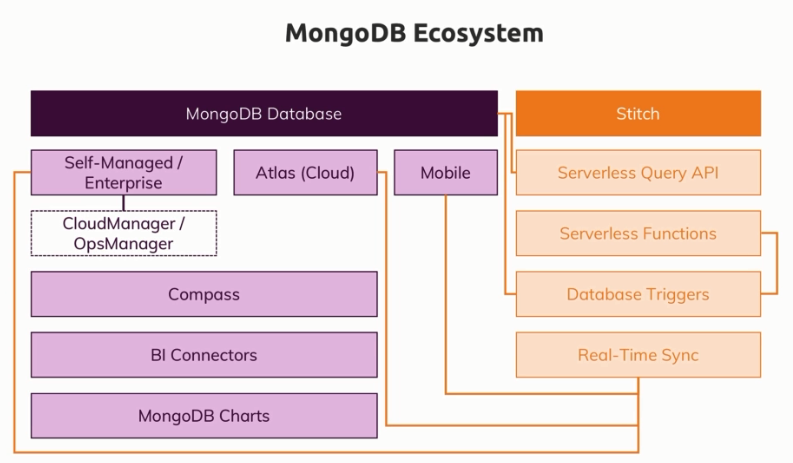
### No Schema

* Instead of normalizing data which means storing it, distribute it across multiple tables where every table has a clear schema and then using a lot of relations, mongodb goes for storing data together in a document and it also doesn't force a schema on you.
* So we got no schemas in mongodb. If we have multiple documents in one and the same collection, they can have different structures.
* This can of course lead to messy data but it's still your responsibility as a developer to work with clean data and to implement a solution that works but on the other hand, it gives you a **lot of flexibility** and flexibility is always good.
* You can use mongodb for apps that might still evolve, where the exact data requirements are just not set yet, you can get started and you can always add data with more information in a collection in the same collection at a later point of time.

### No / Less Relations

* With mongodb, you also work with less relations.
* With embedded documents, one core thing of mongodb indeed is that you have less less collections which you connect but instead that you store data together and this is where the efficiency is derived from.
* Since data is stored together, when your application is fetching data, it doesn't need to reach out to collection A, merge it with collection B, merge it with collection C, instead it goes to collection A then mongodb has a very efficient querying mechanism behind the scenes so that it can go through all the data very fast when looking for a specific document, so this will be super fast and then it finds that document and it's done, it doesn't need to do any merging most of the time.
* So this is really where the **speed, the performance and flexibility** comes from.
* And this hopefully speaks for itself that this can be useful when building applications and this is also why NoSQL solutions and amongst them most of all mongodb is super popular **for read write heavy applications**, applications with a lot of workload, applications that store a lot of data, let's say smart devices which send some sensor data every second, for such applications but also for building an online shop or a blog, mongodb is an amazing solution due to the performance and the flexibility it gives you.
* If there is any relational data, that has to be merged manually (kind of).

## Understanding MongoDB Ecosystem

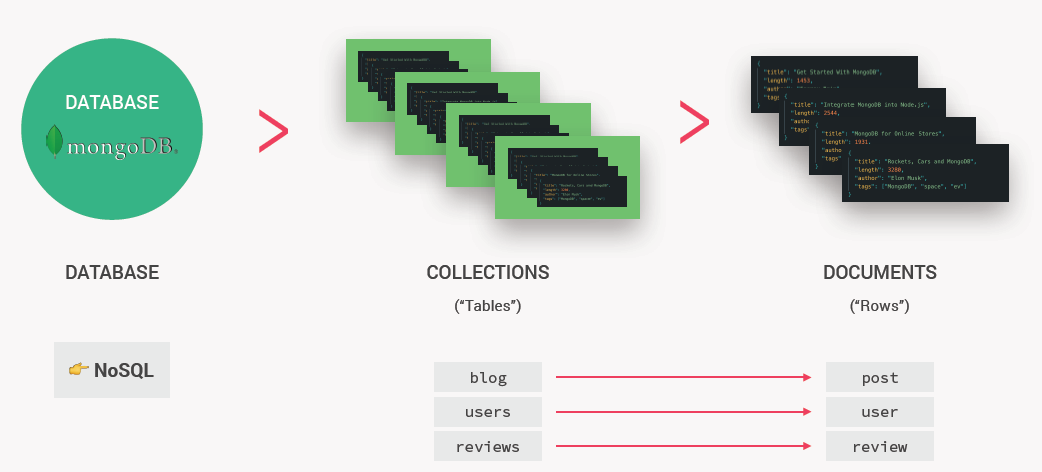


* The company behind the mongodb database solution is also called mongodb and actually, they have a couple of other products too.
* MongoDB Ecosystem has below systems/features.
* Obviously we got the **mongodb database** which is the core asset, the core feature.

There you have a **self-managed and an enterprise solution**.

* Then they have some tools (**CloudManager/OpsManager**) that are related to managing that database that is more of a system admin, database admin task.
* And they have a **cloud solution (Atlas),** which is an amazing solution for you as a developer, because all the managing a system administrator would have to do is done for you there and you can focus on the data and your logic there.
* They also have a **mobile solution** which basically means you can install mongodb on a mobile device to store data there directly and even work **without an internet connection**.
* They got a graphical user interface **(compass)** which allows you to connect to your database and look into it with a nice user interface.
* And we also get some other tools like BI connectors or Mongodb charts if you're more on the data science area. This is not really something where you will write a lot of code. These are more tools that allow you to connect different analytics tools if you need that.
* Now this is the entire database world and this is of course their core feature hence they have a lot of products there as you can see.
* Now they also got a relatively new offering which is called **stitch** and stitch is basically their **serverless backend solution**. So this is a bit decoupled from the database, still strongly connected but also it offers you more than just data storage. Stitch for example gives you a serverless query API. This is essentially a toolset, a tool which you can use to efficiently query your database directly from inside your client side apps, so from inside your react app or anything like that.
* You also have **serverless functions** which allows you to execute code in the cloud on demand. So now this is totally unrelated to the database, you can execute any javascript code in the cloud there and if you know something like AWS Lambda or Google cloud functions, this is your equivalent.
* We also have **database triggers** which is basically a service that allows you to listen to events in a database, like document was inserted and then execute a function in response to that.
* There is a feature called **real time sync** which basically is built to synchronize a database in a cloud with that mobile offline supporting database you might have on mobile devices.

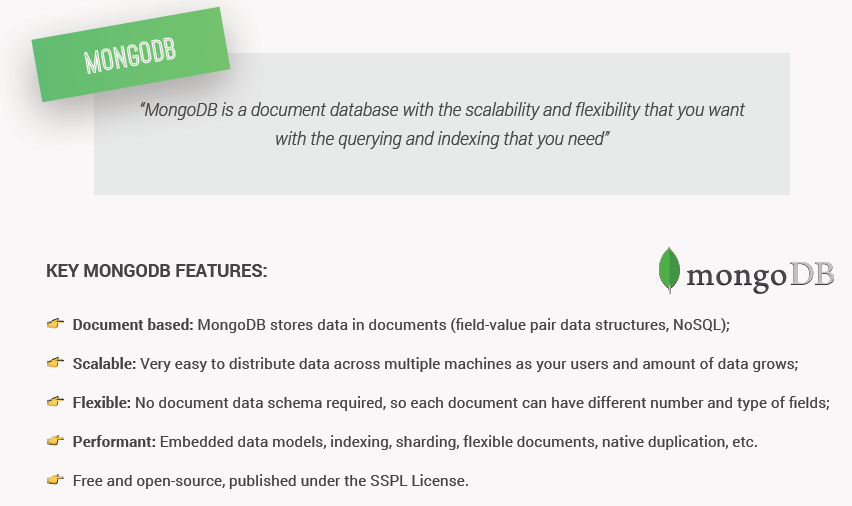
## Overview of MongoDB



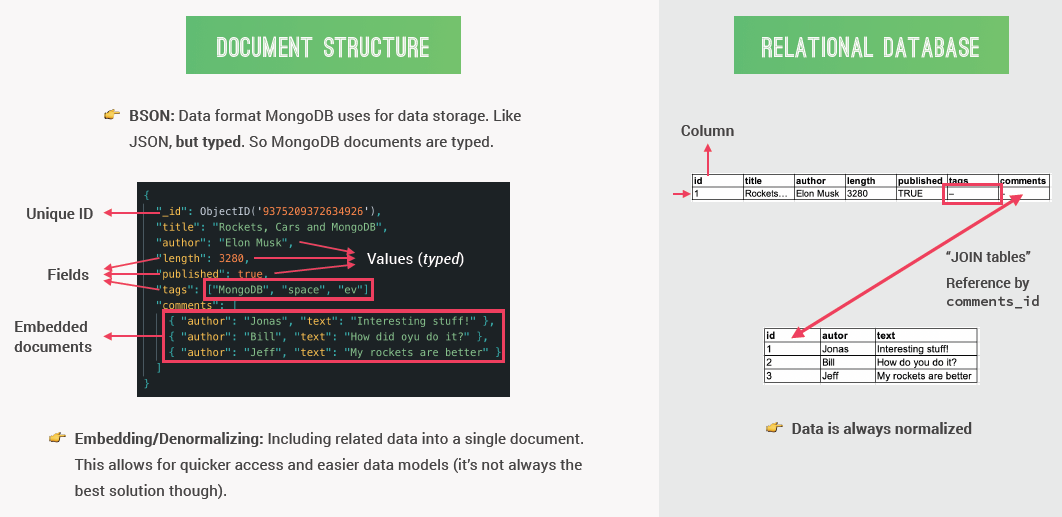
* In MongoDB (or just “mongo”), each database can contain one or more collections. A collection is like a table of data.
* Each collection can contain one or more data structures called documents. A document is like a row in a table. So each document contains the data about one single entity, for example, one blog post or one user or one review, etc.
* The collection is like the parent structure that contains all these entities. For example, a blog collection for all posts, users collection or a reviews collection.
* The document has a data format like JSON.

## Features of mongoDB

* MongoDB is a document database with the **scalability** and **flexibility** that you want, and with the **querying** and **indexing** that you need.
* MongoDB is a free and open-source database.
* MongoDB is a great database system to build many types of modern, scalable and flexible web applications.
* Mongo is probably the most used database with Node.js.



## Documents, BSON and Embedding



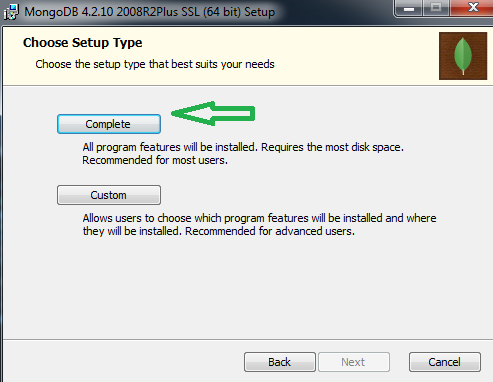
* MongoDB uses a data format similar to JSON for data storage called BSON. It looks basically the same as JSON, but it's **typed**, meaning that all values will have a data type such as string, Boolean, date, integer, double, object or more.
* These BSON documents will also have fields and data is stored in key value pairs. Field is like column in relational DB.
* We can have **multiple values to same field** in the form of arrays (see ‘tags’ field above), which is something complicated to do with relational DB.
* Another extremely important feature in MongoDB is the concept of **embedded documents**. The process of embedding, or de-normalizing as we can also call it, is basically to include so to embed some related data all into one single document. E.g. all the comments above.
* Embedded document makes a database more performant in some situations because this way, it can be easier to read all the data that we need all at once.
* The opposite of embedding or de-normalizing is normalizing, and that's how the data is always modeled in relational databases.
* The maximum size for each document is currently **16 MB**, but this might increase in the future.
* Each document contains a **unique ID**, which acts as a primary key of that document. It's automatically generated with the ObjectID data type each time there is a new document. So we don't have to worry about it.

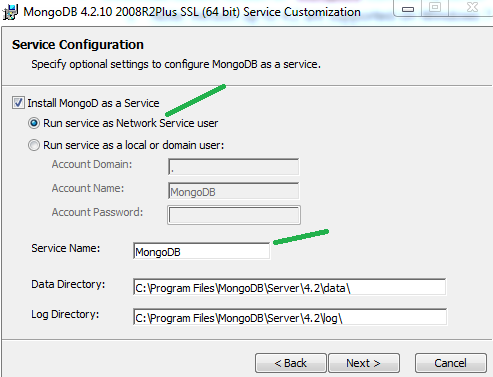
## Setting up MongoDB on Windows

Here we will install the MongoDB server and optionally MongoDB Compass (UI tool), then will create a directory to store the mongodb databases, then will see how to start and connect to the mondogb server and finally adding the mongodb in System Path Variable so that it can be run from anywhere.

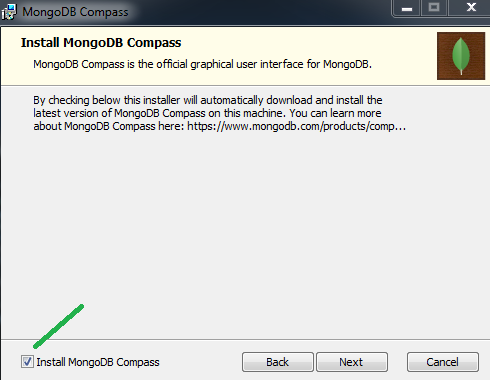
### Installing MongoDB

* Go to <https://www.mongodb.com/>
* Download current version of “MongoDB Community Server” installable (.msi) for Windows.
  + Note: Version up to 4.2 are supported on Windows 7





* + You can either select ‘Install MongoDB Compass’ so that the Compass tool will also be installed. Or you can deselect this option and separately install MongoDB Compass from the mongodb website.



### Create folder to store the MongoDB database

* Now once the MongoDB is installed, we need to create a directory in which MongoDB will store our data.
  + Go to the disk where MongoDB is installed.
  + If let’s say it is C: drive, then create folder ‘data’ under C drive and another folder ‘db’ under the ‘data’ folder like this –   
    C:\data  
    C:\data\db – This is the place where MongoDB will store our databases.
  + By **default**, MongoDB expects the data and data\db folders directly under root (For windows, it should be directly under C: drive where MongoDB is installed)
  + If you don’t have these default folders at this exact same path(C:\data\db), the mongodb server will fail to start.
  + If you want to provide a different path, e.g. say C:\Users\Sameer\data\db, then while starting the MongoDB Server, we need to specify this path as –

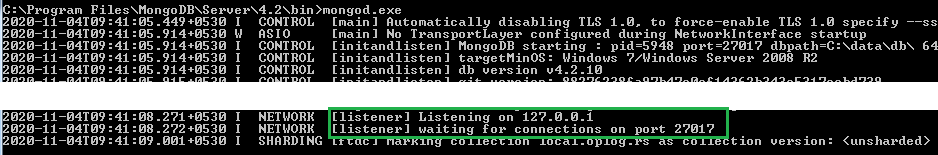
mongod --dbpath “C:\Users\Sameer\data\db”

* Few files at MongoDB installed location - C:\Program Files\MongoDB\Server\4.2\bin
  + mongod.exe – To start the MongoDB server.
  + mongo.exe – This is MongoDB Shell through which we can interact with our MongoDB databases and perform DB operations.
  + mongoimport.exe – used for importing data.

### Starting the MongoDB server and Connecting using MondoDB Shell

* Go to C:\Program Files\MongoDB\Server\4.2\bin and run mongod.exe to start the **server**.

C:\Program Files\MongoDB\Server\4.2\bin>**mongod**

* After you run this file, you will see an message in the end – “[listener] waiting for connections on port 27017”  
  
* MongoDB dy default runs on port 27017. If you want to change this, you need to start the server with ‘port’ flag like this –

>**mongod** --port 27018

* So we've basically started a server and now we need a shell to connect to the server to be able to manipulate our databases (Create, Delete, Query, etc.).
* Note:
  + On windows, you may not have to start the mongodb server (mongod) manually because on windows it is running as a “Service” so it is anyways running in the background. However if you get error while connecting to Shell (below), then you can manually start the server.
  + To stop the background MongoDB server which is running as a service, execute –

net stop MongoDB

To start the MongoDB server as a service in background, execute this –

net start MongoDB



* To run the **Shell**, we need to run the mongo.exe file

C:\Program Files\MongoDB\Server\4.2\bin>**mongo**

* After running this command, you can see that we've automatically connected to the default port (27017). And you will see the shell to interact with the MongoDB database.



* If you have used a different port to run MongoDB server, then you have to specify same port explicitly while running the Shell. E.g. >**mongo –-port 27018**
* To ‘double’ test the connection, we can run the db command and you should see output as ‘test’ which is the default database we get.

> db

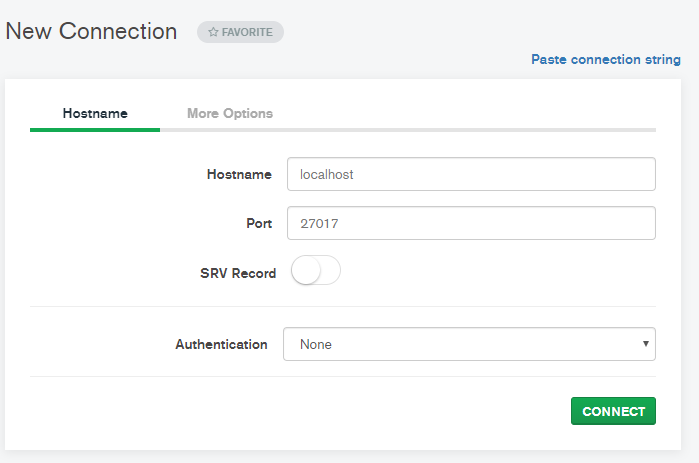
Test

### Set System Environment Variables

* In order to run the Mongo server from any directory, we need to set path for the mongodb in System Environment variables.
* Add path ‘C:\Program Files\MongoDB\Server\4.2\bin’ to the ‘Path’ System Environment Variables on your machine.

## MongoDB Compass

* Instead of using the terminal to work with MongoDB, we can also use a graphical user interface app called MongoDB Compass.
* For MongoDB Compass installation, refer – [Installing MongoDB](#_Installing_MongoDB)
* In order to create a new connection to our local database in Compass, make sure you have the Mongo server running in the background.
* Open MongoDB Compass -> New Connection -> click Connect. Because by default it has the local connection settings (Host: localhost and Port: 27017)



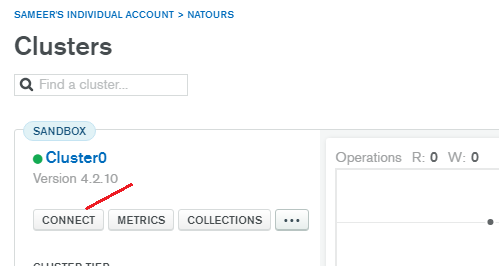
* Compass is just a graphical user interface for doing the exact same stuff that we can do with terminal/Shell.
* In Compass, we can really do all kinds of stuff like aggregations, define some schemas or analyze schemas, take look at indexes, performances and all that good stuff.

## Creating Remote Database Hosted on MongoDB Atlas

* We can either use local database (hosted on our machine) or use a remote database hosted on a service called Atlas, which is actually owned by the same company that involves MongoDB.
* Go to MongoDB website ( <https://www.mongodb.com> ) and select product MongoDB Atlas or you can use this link directly <https://www.mongodb.com/cloud/atlas>
* Atlas is a so-called database as a service provider which takes all the pain of managing and scaling databases away from us.
  + So that already is a huge advantage for us but it's also **extremely useful to always have our data basically in the Cloud**, because this way we can develop our application from everywhere.
  + And, even more importantly, we don't have to export data from the local database and then upload it to a hosted database, once we are ready to deploy our application.
* Create a cluster in the link given above.
* An Atlas cluster is basically like an instance of our database.
* Once a cluster is created, we now have a blank empty database, ready to connect to our own development computer.

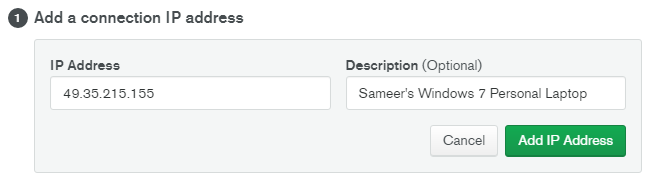
## Connecting to the Remote Hosted Database

* We can connect our remote hosted database with our Compass app and also with the Mongo shell.
* In the MongoDB Atlas app, click on ‘Connect’ under the cluster.

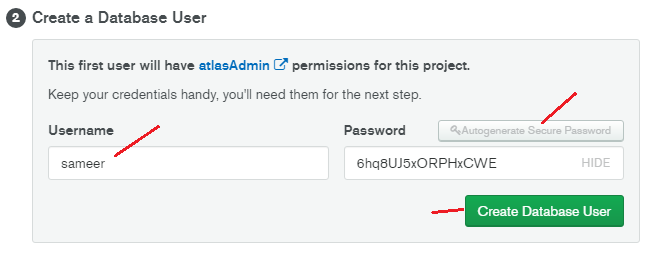


* First, we need to add our current IP address, so that our computer is actually able to connect to this cluster.





* Then, we need to create a database user.



Copy the password and add it into our Node application’s config.env file.

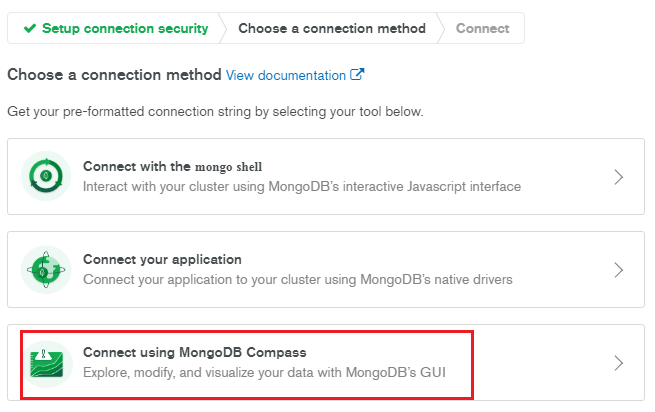
DATABASE\_PASSWORD=6hq8UJ5xORPHxCWE

Later, when we will then connect our application to the database, we will then use this environment variable to create that connection.

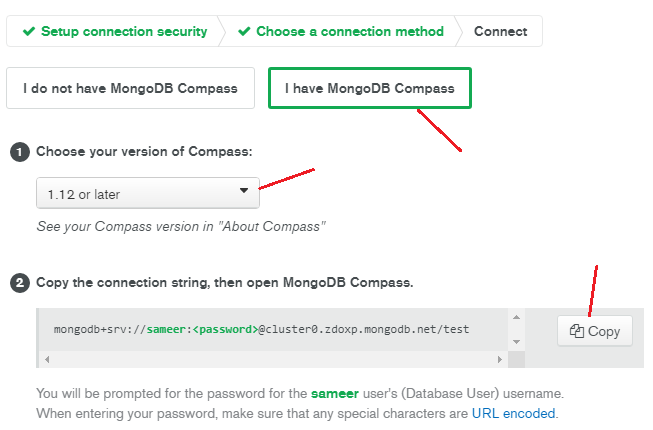
* Next, we need to choose a connection method.



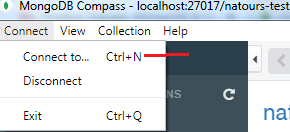
Select ‘Connect using MongoDB Compass’

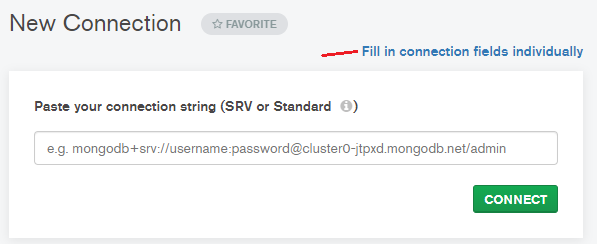


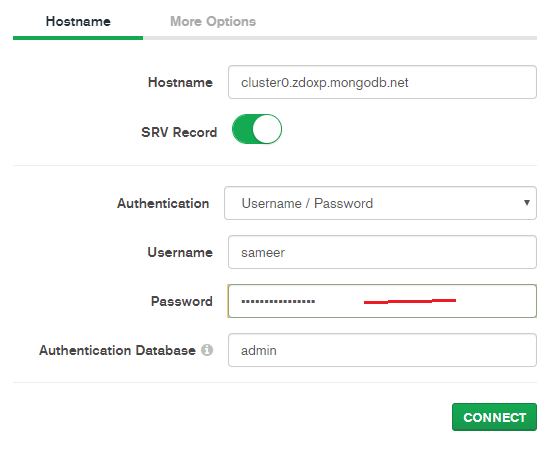
Select ‘I have MongoDB Compass’ (if not, first install it). Then select installed Compass version and then copy the connection string so that we can use our Remote database from our Compass app.



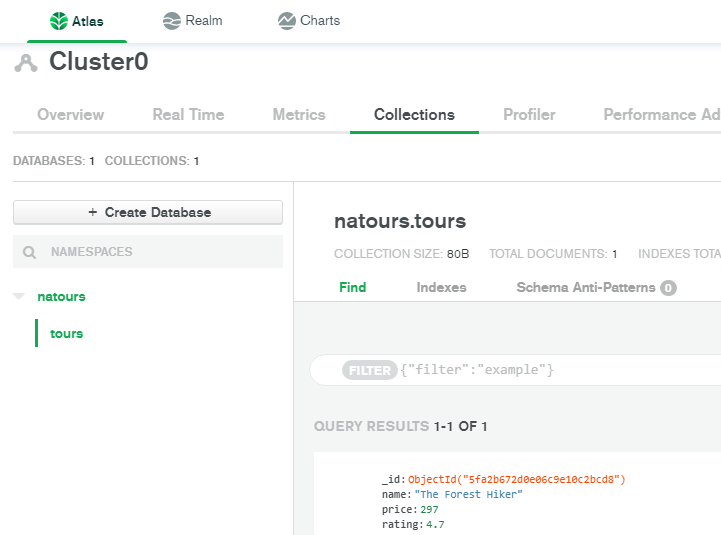
* Open MongoDB Compass and navigate to

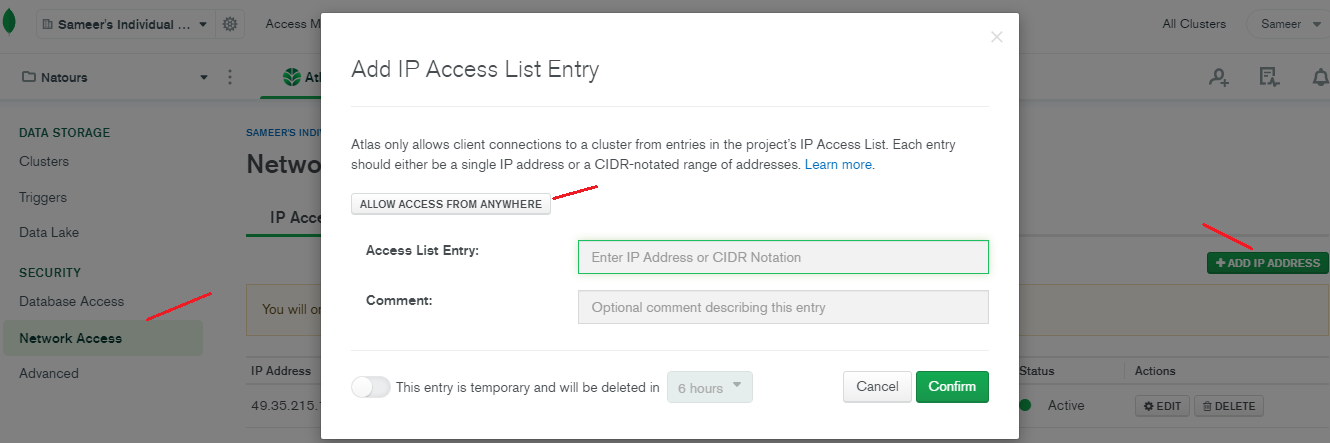


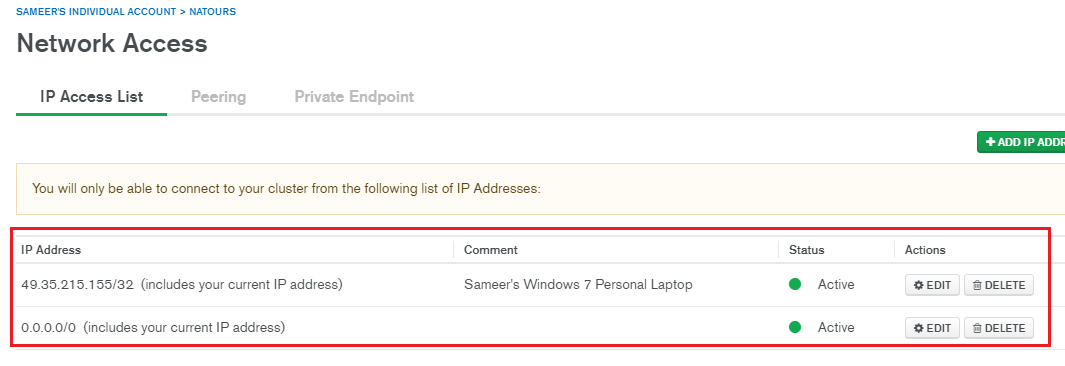
* Paste the copied connection string to below textbox and click on ‘Fill in connection fields individually’. There enter the database password which had created in previous step. And click on Connect.  
  



* Now we are connected to the Remote Database. From here on, we can the usual business of creating database, collections, documents, etc.
* Any DBs/collections/documents created here will be created on the Remote Host.
* To see these in Atlas, go to Atlas website and go under Clusters.

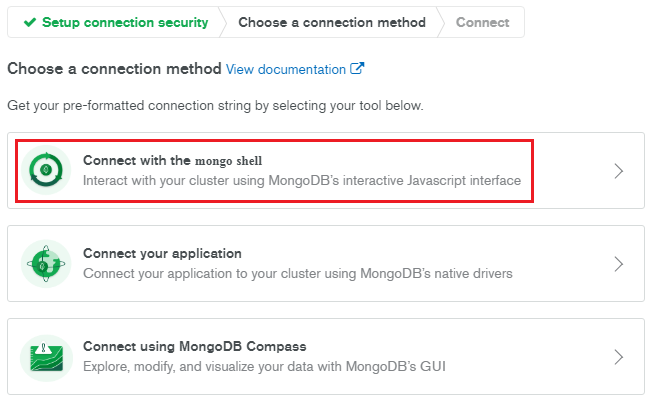


* Now another thing that we can and should do, is to **allow access** from everywhere to this cluster. In the beginning, we whitelisted our IP in order to grant access to our current computer to this cluster. But if you happen to switch computers during development, you might need to whitelist the IP of that computer as well, because otherwise you might not be able to connect. If we're not really dealing with sensitive data (test projects), we can simply whitelist every single IP in the world and allow access from everywhere. (of course, we will always still need our username and our password)
* For this, do this –   
  
* We can update this setting later on, if we want

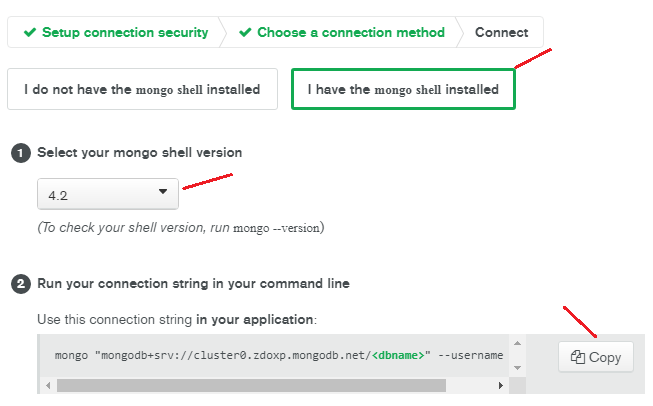


## Connecting Mongo Shell to the Remote Cluster

* Go to Cluster -> Connect -> Select ‘Connect using the mongo shell’



* Select appropriate option and copy the connection string



* Open New command prompt and paste the copied text. This text is the complete command to connect to our cluster.
* Now replace the text <dbname> with your actual database name .e.g natours. And hit enter.

E.g.

>mongo "mongodb+srv://cluster0.zdoxp.mongodb.net/**natours**" --username sameer

* Then it will ask for the password. Copy the database password which we created above and paste it and hit enter again.
* You should be connected now. To confirm it, enter command show dbs. You should see your DB.

MongoDB Enterprise atlas-kcxepu-shard-0:PRIMARY> show dbs

admin 0.000GB

local 1.252GB

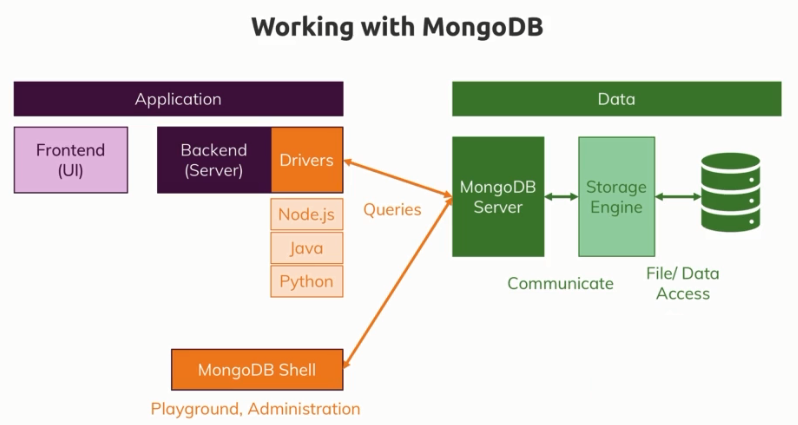
natours 0.000GB

* Now we are all set to go ahead.

## Shell vs Driver

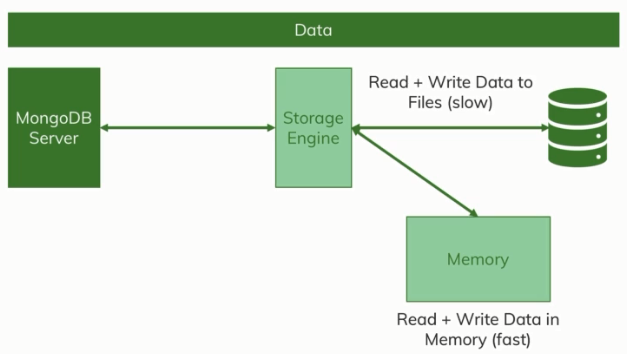
* Use Shell, to see all the commands and the different ways of using these commands because the shell is a great neutral ground for working with mongodb.
* Drivers are packages that you install/use for the different programming languages your app might be written in, and these drivers are then your bridges between your programming language and the mongodb server.
* And in these drivers, you use the same commands as you use in the shell, just adjusted to the syntax of the language you are working with of course.

## Working with MongoDB



* Now the mongodb server will actually not directly write the data into files but talk to a so-called **storage engine** which you could replace with your favorite storage engine but the default one called **Wired Tiger** is actually an awesome storage engine which allows you to efficiently work with your data, store it efficiently and so on.

### A closer look



* At that data layer – with the server, the storage engine and the file system, we actually have to differentiate between writing and reading from files which is a bit slower and writing and reading from memory which is faster.
* The storage engine actually does both, it loads a chunk of data into memory and manages that, such that the data you often use is in memory if possible.
* It also writes data in memory at first so that this is really fast but then of course it also goes ahead and stores data in the database files.
* In general, you need to be aware that you always talk to the mongodb server and behind the scenes, the server talks to the storage engine which manages your data and stores it in files in the end but also in memory in between so that you can work with the data in a very fast way. So this is how the mongodb server works behind the scenes.

# Understanding the Basics and CRUD Operations

## Understanding Databases, Collections and Documents

* In a MongoDB world, you have one or more **databases** on your database server and each database can hold one or more **collections**, a collection would be a table in a SQL database. Now in that collection, you have these **documents**, multiple documents per collection and the documents are really the data pieces you're storing in your database.
* Now important, when working with MongoDB, the databases, the collections and the documents are all automatically created for you, they are **created implicitly** when you start working with them, when you start storing data. This is a cool feature which makes getting started super simple.

## Default Databases

* When you run show dbs command, you see admin, config and local as databases.
* These default databases here simply exist to store configuration for this database server or for example the admin database will allow you to create users and roles and how people can use and interact with the database.

## Creating a Local Database

* We will create a local database using the **Mongo Shell** (mongo.exe) which is the terminal application.
* To clear the terminal – Ctrl + L
* To create a database, we use the "use" command inside the Mongo Shell, and then the name of the database that we want to create. This command here is also used to switch to an already existing database, but if we try to basically **switch to** a database that doesn't yet exist, it will then **create** a new one (actually it will create db if we add at least one document inside a collection).
* E.g. Here it created the database and switched to it.

> use natours-test

switched to db natours-test

* The data in mongodb is always a Document and the documents are part of a Collection. So we first need to create or use a collection in order to operate on documents.
* ‘db’ stands for current database.
* To create a document inside a collection, we use insertOne() or insertMany() operators like this

> db.tours.insertOne({ name: "The Forest Hiker", price: 297, rating: 4.7 })

{

"acknowledged" : true,

"insertedId" : ObjectId("5fa234442873acd05244dc3b")

}

* + In the insertOne() method we pass BSON string representing our document.
  + Use double quotes instead of single quotes.
  + Doubles quotes for key is optional.
  + Here ‘tours’ is a collection name. If it doesn’t exist, it will be created.
  + As you can see from the output of the command, mongodb automatically created unique ID property for our document.
* To query/see the documents inside a collection, use command db.<collection\_name>.find()

E.g.

> db.tours.find()

{ "\_id" : ObjectId("5fa234442873acd05244dc3b"), "name" : "The Forest Hiker", "price" : 297, "rating" : 4.7 }

* As you can see, mongodb works with the Javascript syntax and JSON, which is one of the main reasons why MongoDB is so popular for Node JS applications.
* To see all the databases in mongodb, use command show dbs.

> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

natours-test 0.000GB

* To see collections inside a database, use show collections.

> show collections

tours

* To exit/quit the mongo shell, use quit() operator.

## ObjectId Object

* ObjectId object is simply a special type of data provided by mongodb which is a unique ID that also allows you to sort your documents because it will also have some **timestamp** data in there. So this id also can be used for **sorting** because it's guaranteed that if you would add another document, that would be treated as a more recent document than the already inserted document.
* Each document needs a unique id in the field “\_id”. If not given explicitly, mongodb creates automatically in the form of ObjectId() and it also maintains **insertion order**.

## JSON vs BSON

* Mongodb does not use JSOB but BSON which stands for **Binary JSON** for storing data in your database.
* Now you don't really need to care too much about that because all you (developers) will write and see will be JSON code and this is also what we see when we retrieve data from mongodb. But behind the scenes, mongodb actually uses BSON data.
* This conversion (JSON to BSON) is done by the **mongodb drivers**. So this basically takes your JSON code and stores it in binary data and this is simply done because it is more efficient to store than JSON data, so it's faster and more efficient from a space or size perspective and additionally it supports additional types because for example that ObjectId is not valid normal JSON. And there are some other types too like for example there are different types of numbers, with decimals and very big numbers and these are stored in different ways behind the scenes.
* We as developers will always write data in JSON format.
* While creating documents, we can skip double quotation marks for keys provided there is no space in a key.

## \_id field in the document

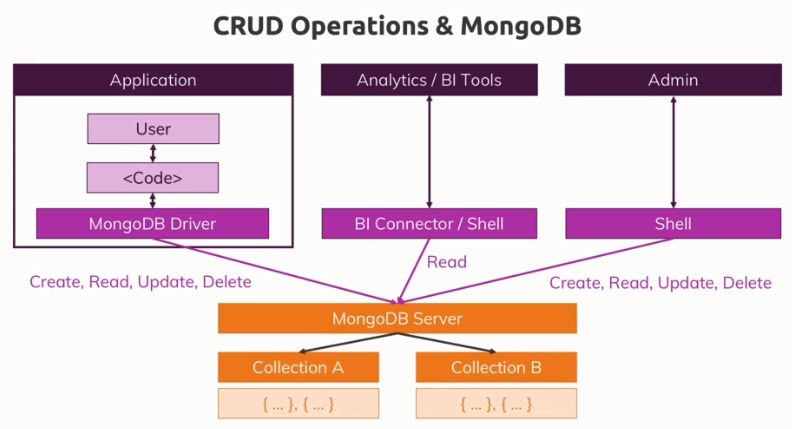
* When we insert documents, we get “\_id” as an auto-generated field.
* We don't have to use the auto-generated ID, we just have to ensure that we have a unique ID and if we can ensure this, we can also assign IDs on our own.
* Each document must have unique key.
* E.g.

> db.flightData.insertOne({ "departureAirport": "MUC", "arrivalAirport": "SFO", "aircraft": "Airbus A380", "distance": 12000, **"\_id": "MUC-SFO"** })

## Embedded Documents

* Nesting of documents inside one document.
* We can have up to 100 levels of nesting in mongodb (but we rarely need 3 or 4 levels of nesting).
* Another hard limit is the overall document’s size has to be below 60 MB.
* We can have arrays of embedded documents.
* Arrays can hold any kind of data.
* You can have documents in an array and you can also have arrays in documents.
* Embedded or nested documents and arrays gives you a lot of flexibility to structure your data.

## CRUD operations on Documents



### Creating Documents

* To create a document inside a collection, we use insertOne() or insertMany() operators.
* MongoDB documents are very flexible, i.e. they do not all have to have the same structure (within the same collection).
* We have seen insertOne() above, let’s use insertMany()
* To create multiple documents inside a collection, use insertMany()

E.g.

> db.tours.insertMany([{ name: "The Sea Explorer", price: 497, rating: 4.8}, { name: "The Snow Adventurer", price: 997, rating: 4.9, difficulty: "easy"}])

{

"acknowledged" : true,

"insertedIds" : [

ObjectId("5fa294d58251e593f8ea0992"),

ObjectId("5fa294d58251e593f8ea0993")

]

}

* + Here we have added 2 documents and the second document has an extra field called ‘difficulty’.
  + For the created documents, we got 2 auto generated IDs.

### Querying/Reading Documents

#### Theory

* find() method does not give us collection of data, but it gives us a so called **cursor** object.
* Find() does not give us an array of all the documents in a collection and that makes a lot of sense because that collection could be very big and if find() would immediately send us back all documents and you think about a collection with let's say 20 million documents, then this would take super long but most importantly, it would send a lot of data over the wire. So instead of that, it gives us back cursor object which is an object with a lot of metadata behind it that allows us to loop through the results and that is what that “it” command does in Mongo Shell, “it” basically use that cursor to fetch the next bunch of data.
* By default, using just find() in Mongo Shell returns us first 20 documents.
* Since find() returns a cursor, there are methods which can be used on it. E.g.
  + toArray() – to get all the data at once, we can use   
    > db.tours.find().toArray()
  + forEach() – to loop through all the documents. (Very efficient)   
    > db.tours.find().forEach((doc) => printjson(doc))
* find().pretty() works but findOne().pretty() fails because **pretty() is a method that simply exists on the cursor**. find() returns a cursor however findOne() does not give us a cursor because it only gives us one document anyways.
* For the other methods like insert, update and delete, cursors do not exist because these methods don't fetch data, they manipulate it.

#### Examples

* To query for all the documents in a certain collection, use find() operator without any arguments.

E.g.

> db.tours.find()

* To query for a certain document with some filtering criteria, just pass the filtering criteria as a BSON object (filter object) to the find() operator.

E.g.

> db.tours.find( { name: "The Forest Hiker" } )

> db.tours.find( { difficulty: "easy" } )

* We can also use some special query operators for more granular filtering. E.g. less than ($lt), greater than ($gt), less than or equal to ($lte), greater than or equal to ($gte).

E.g.

> db.tours.find( { price: {$lte: 500} } )

Here we want to get all documents whose price is less than or equal to 500.

* The special Mongo operators start with $ sign.
* We can use more than one search criteria (AND) to query the documents.

E.g.

> db.tours.find( { price: {$lt: 500}, rating: {$gte: 4.8} } )

Here we want to get all documents whose price is less than to 500 AND rating greater than or equal to 4.8.

* We can also use OR criteria to query the documents, i.e. to get documents satisfying either of the conditions. For this, we use $or operator, which takes an array of conditions.

E.g.

> db.tours.find({ $or: [{ price: {$lt: 500 }}, { rating: {$gte: 4.8 }}] })

Here we want to get documents having price less than 500 OR having rating greater than or equal to 4.8

* Besides our filter object, we can also pass in an object for **projection**. Projection means we simply want to select some of the fields in the output.
* For projections, we need to 0 or 1 for the document fields. **0 means exclusion, 1 means inclusion**. We cannot mix both 0 and 1 for different fields in the same query.
* E.g. If we want only name from the above query,

> db.tours.find({ $or: [ { price: { $lt: 500 } }, { rating: { $gte: 4.8 }} ] }, {**name: 1**})  
Here we want to get documents (with names only) having price less than 500 OR having rating greater than or equal to 4.8

* E.g. 2) If you want all fields except ‘name’

> db.tours.find({ $or: [ { price: { $lt: 500 } }, { rating: { $gte: 4.8 }} ] }, {**name: 0**})

* E.g. 2) If you want all fields except ‘name’ and ‘price’

> db.tours.find({ $or: [ { price: { $lt: 500 } }, { rating: { $gte: 4.8 }} ] }, {**name: 0, price: 0**})

* We cannot mix both 0 and 1 for different fields in the same query.

E.g.

> db.tours.find({ price: { $lt: 500 } }, {name: 0, price: 1})

Error: error: {

"ok" : 0,

"errmsg" : "Projection cannot have a mix of inclusion and exclusion.",

"code" : 2,

"codeName" : "BadValue"

}

* By default, “\_id” field is always returned even with projections. We have to explicitly mention in the projection if we don’t want it by setting “\_id”: 0

> db.tours.find({ price: { $lt: 500 } }, {name: 1, **\_id: 0**})

* **Querying embedded documents**. Note: We have to use double quotes.

> db.flightData.find({**"status.description"**: "on-time"}).pretty()

{

"\_id" : ObjectId("5ff856d1c1bd95b8458e8602"),

"departureAirport" : "MUC",

"arrivalAirport" : "SFO",

"aircraft" : "Airbus A380",

"status" : {

"description" : "on-time",

"lastUpdated" : "1 hr ago"

}

}

* **Querying** **arrays**.

> db.passengers.find({**hobbies: "sports"**}).pretty()

{

"\_id" : ObjectId("5ff9723ef82ed11a0231da5b"),

"name" : "Albert Twostone",

"age" : 68,

**"hobbies" : [**

**"sports",**

**"cooking"**

**]**

}

### Updating Documents

* To update one document, we use updateOne() operator. And for updating many, we have updateMany() operator.
* First we need to select the documents we want to update and second we need to pass in the data that should be updated.
* The first argument is basically a filter object, we basically need to query for the documents that we want to update. The filter object can be a complex object.
* E.g.

> db.tours.updateOne({ name: "The Snow Adventurer" }, { $set: { price: 597 } })

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

Here the first argument is filter object ({ name: "The Snow Adventurer" }, and the second argument is to set price of 597 using the $set operator.

* In case of updateOne(), if the filter query returns more than one documents, then only first document will be updated. So in order to update multiple documents, use updateMany()
* E.g.

> db.tours.updateMany({ price: {$gt: 500}, rating: {$gte: 4.7} }, { $set: {premium: true} })

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

Here we have set a new field ‘premium’ to true for the documents having price greater than 500 and rating greater than or equal to 4.7

* With updateMany() or updateOne(), we only update parts of the document. But we can also completely replace the content of the document using replaceOne() or replaceMany().
* replaceOne() and replaceMany() use the same arguments as updateOne() and updateMany().

### Deleting Documents

* deleteOne() will delete only the first document matching our query. And deleteMany() will delete all the documents matching our query.
* E.g.

> db.tours.deleteMany({ rating: {$lt: 4.8} })

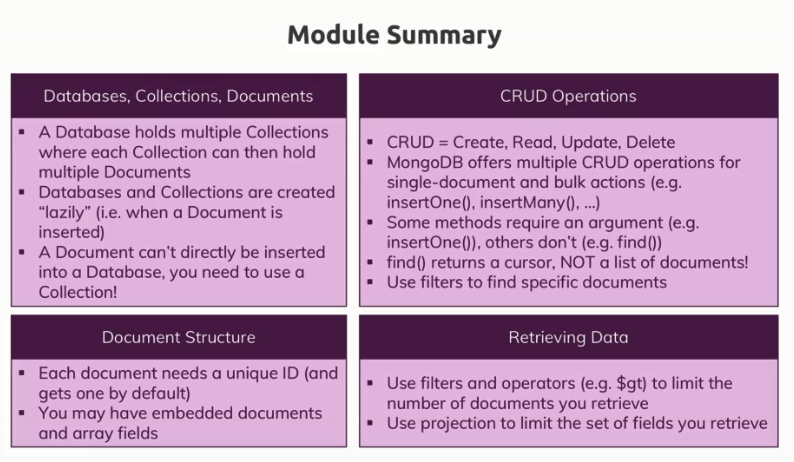
Here we delete all documents having rating less than 4.8

* To delete all the documents in a collection, pass empty filtering object like this

> db.tours.deleteMany({})

Because the empty object is basically a condition that all of the documents always match.

## Summary



# Tips and Tricks