1. **Introduction MongoDB**

**==========================================**

**First Download & Install MongoDB Community Servers**

<https://www.mongodb.com/home>

Go To Products -> Community Servers -> Download mongodb-windows-installer.( size : 482 MB)

<https://www.mongodb.com/try/download/community>

To Check-> windows start Menu -> Services -> Find MongoDB -> Right Click Start/Stop Server.

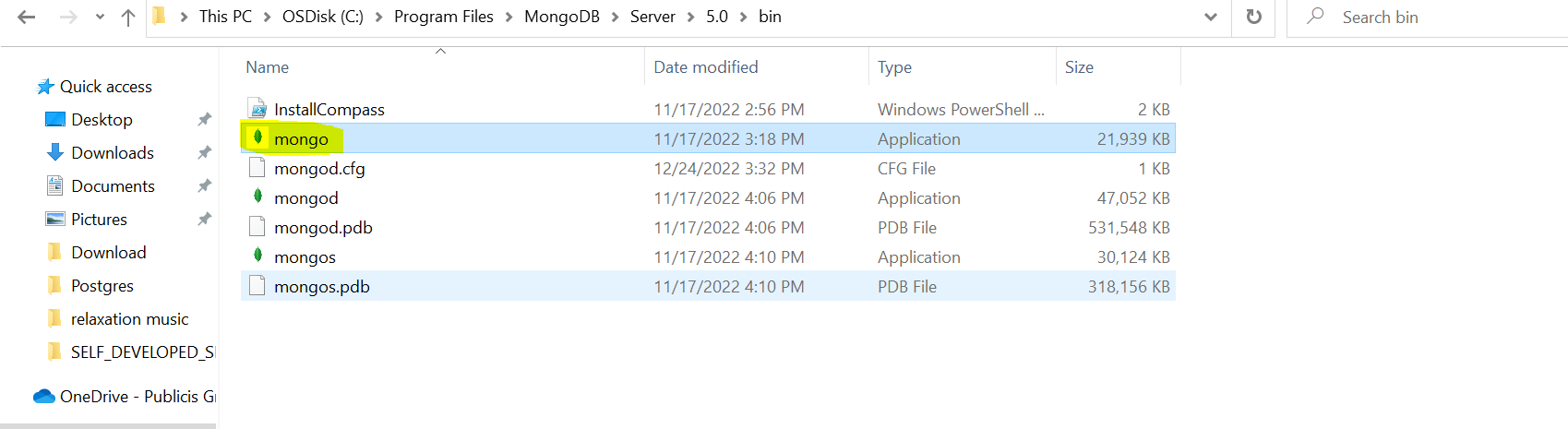
Alternatively open CMD -> Run AS Administrator -> net start/stop MongoDB

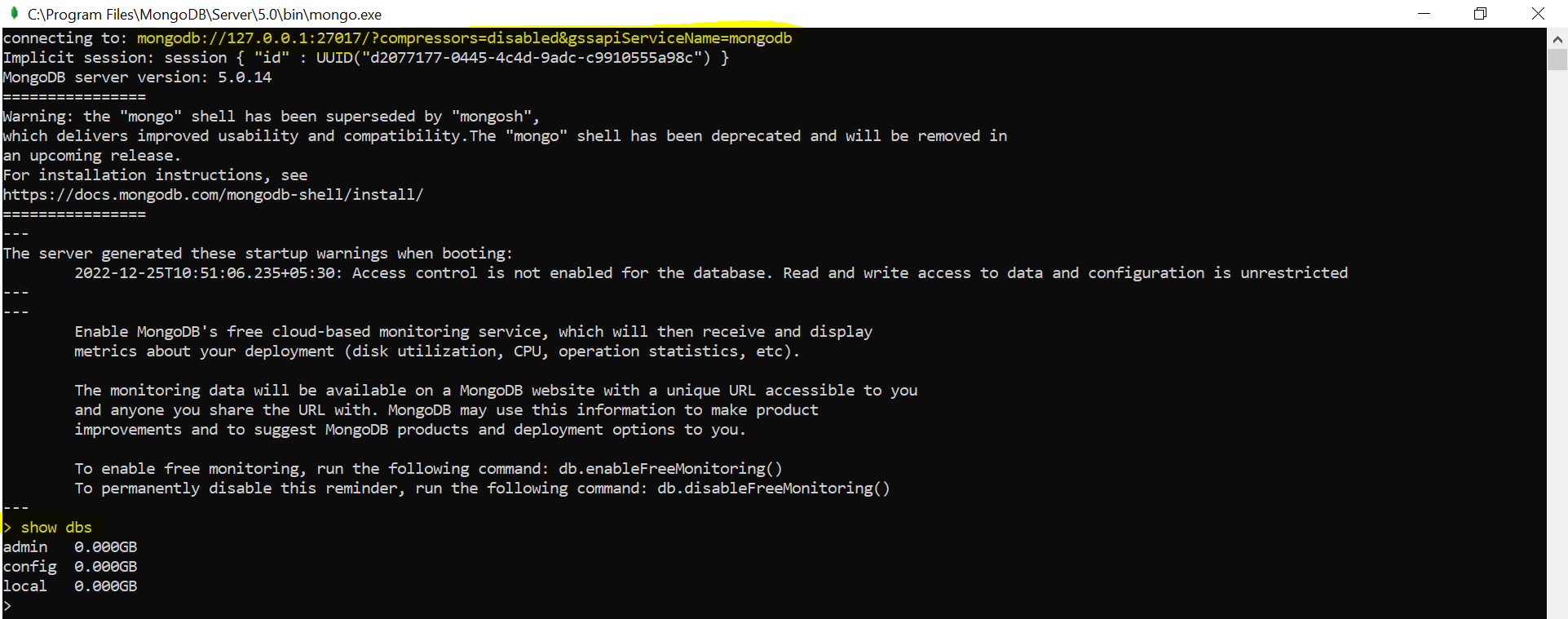
Now Our Server is Running…

In Order to use that Server, we need a **client,** orit’s called a **Shell** which allows us to send a **command** or **query** to the **server.**

For this you can open up the following path **C:\Program Files\MongoDB\Server\5.0\bin**

Now Run this **Mongo.exe** file.



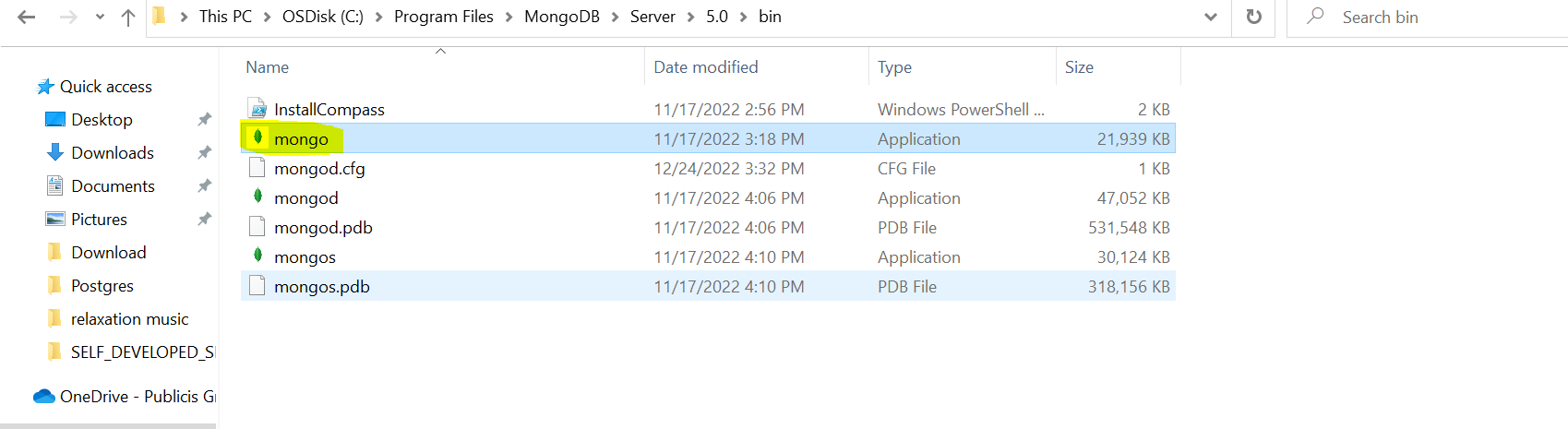


Now its connected to the MongoDB Server.

**Second Download & Install MongoDB Shell**

Now before going into deep dive or before writing queries to our mongodb database there is one additional tool we might need to install and that is **MongoDB Shell**.

Actually, we already did installed a **shell** together with the mongodb server in last discussion and we already did start the shell there. That was that thing that tool which we executed in the command line or by double clicking on it on windows to then connect to our running mongodb server. So, that tool we will use for inputting commands for sending commands to the server. but that shell might be the old shell but there is a newer shell available and you either already have that or you need to install it separately. If u already installed MongoDB shell named “**mongosh**” you already have the new shell but we don’t have in our bin folder.

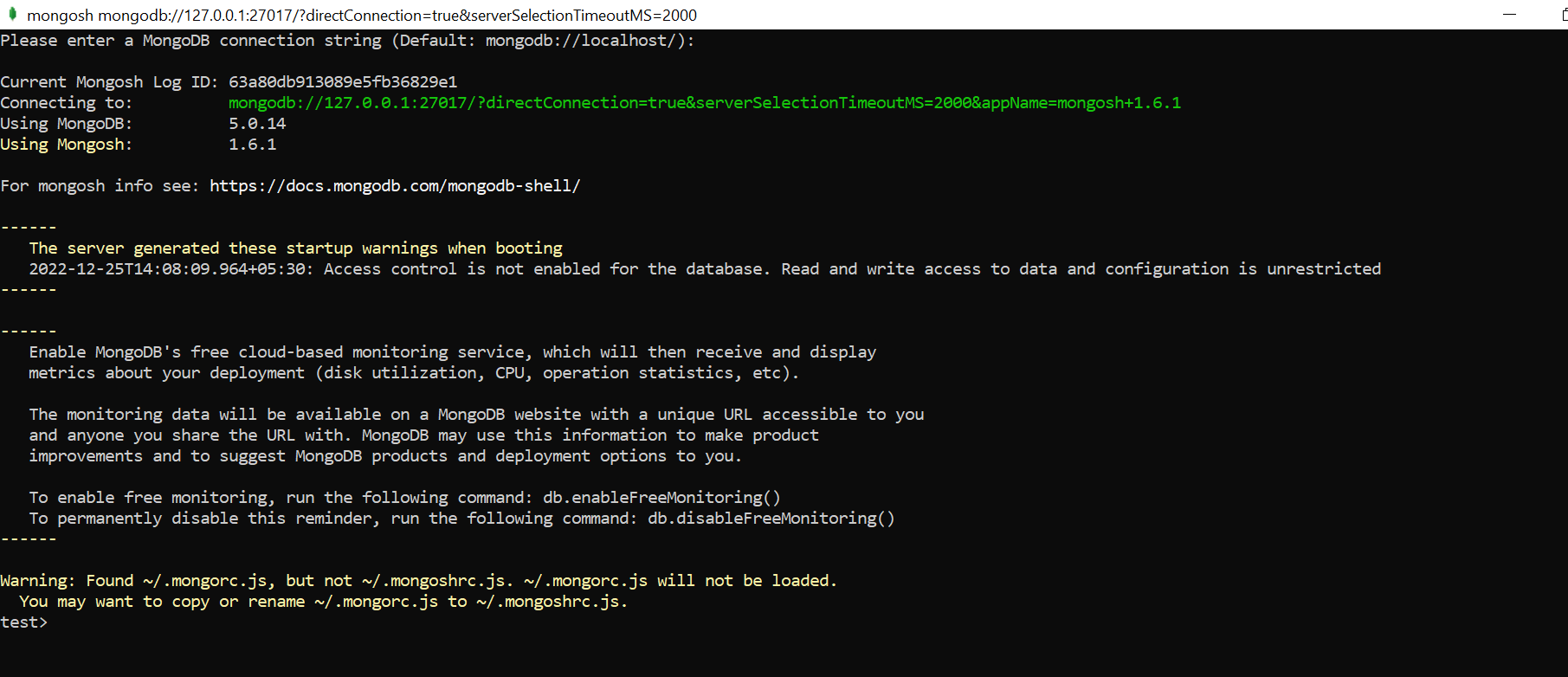


Now to get that newer shell you can search for **MongoDB Shell**.

Download link-> <https://www.mongodb.com/try/download/shell>

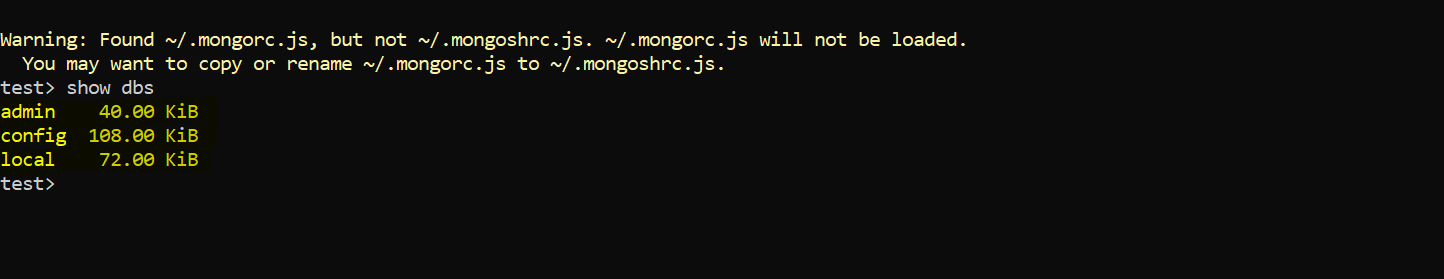
On Windows-> start menu -> mongosh -> execute this command tool

First need to start your mongodb server then this shell will connect to your running server.



It will open mongosh shell cmd tool we just need to do enter.

We can now use this shell to interact with the database.



**>> Show dbs**

**Third Download & Install mongoimport**

In this course, we'll also use another tool called **mongoimport**. It's a tool / local command which we will occasionally use in this course to import some prepared data into a MongoDB database.

Since it wasn't installed together with the MongoDB server + client, you need to install it as a separate tool, following the installation instructions you find here

**Installation**: [https://docs.mongodb.com/database-tools/installation/installation windows/#installation](https://docs.mongodb.com/database-tools/installation/installation%20windows/#installation)

**1 Download the Database Tools MSI installer.**

Open the [MongoDB Download Center](https://www.mongodb.com/try/download/database-tools?tck=docs_databasetools)(https://www.mongodb.com/try/download/database-tools). Using the drop-down menu on the right-hand side of the page:

1. Select the Windows x86\_64 Platform
2. Select the msi Package
3. Click the **Download** button

**2 Run the MSI installer.**

Double-click the downloaded MSI installer to install the Database Tools. During the install you may customize the installation directory if desired.

**3 Make the DB Tools available in your PATH.**

You may wish to make the Database Tools available in your system's PATH environment variable, which allows referencing each tool directly on the command prompt by name, without needing to specify its full path, or first navigating to its parent directory.

Once you've installed the Database Tools, follow the instructions below to add the install directory to your system's PATH environment variable.:

1. Open the **Control Panel**.
2. In the **System and Security** category, click **System**.
3. Click **Advanced system settings**. The **System Properties** modal displays.
4. Click **Environment Variables**.
5. In the *System variables* section, select Path and click **Edit**. The **Edit environment variable** modal displays.
6. Click **New** and add the filepath to the location where you installed the Database Tools.
7. Click **OK** to confirm your changes. On each other modal, click **OK** to confirm your changes.

Once set, you can run any of the Database Tools directly from your command prompt.

So now that everything is set up and you should have a working mongodb server and the shell connected to it.

**>>> cls** --- clear the screen

**>>> show dbs** --- to see all existing databases

Now first of all, let's connect to a database with the **use** command and you can connect to a brand new

database by simply typing its name

even if this doesn't exist yet, it will be created on the fly once you start inserting data. Now in there.

**>>> use shop ---** even shop db is not there but it will create and use directly.

you can create a new **collection** also on the fly because no collection exists right now and you do this

by typing db, this refers to the database you are currently connected to, so, shop in my case and then the name of the **collection** where you want to insert data, let's say products and this also doesn't need to exist yet. And then you can insert one new product with the insertOne command, and you pass this json object

to it

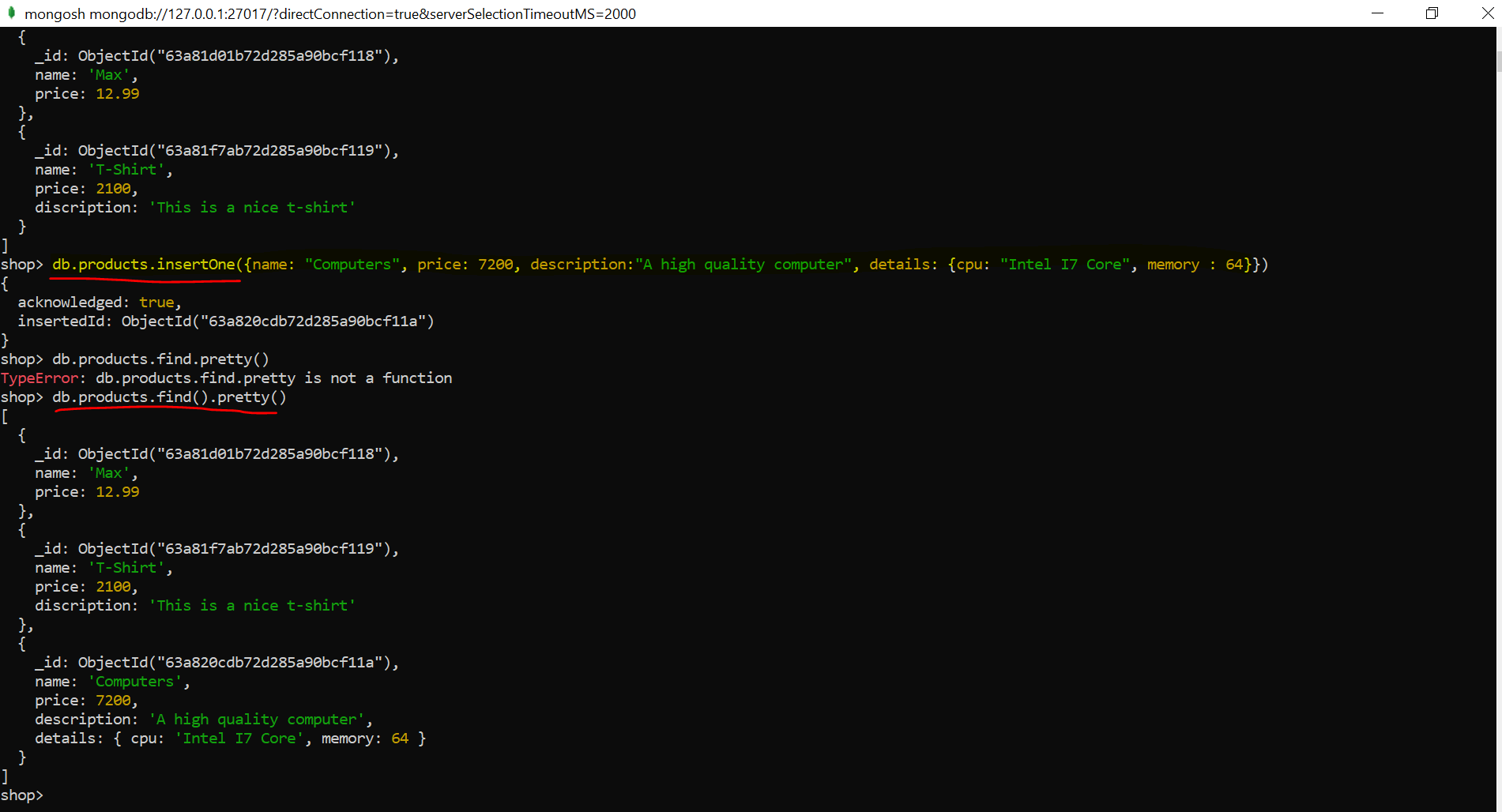
**>>> db.products.insertOne({name:"Max", price:12.99})**

if you add no other arguments this will give you all the data in that collection, find.

**>>>** **db.products.find()**

you can add .**pretty** at the end to simply output it in a prettier way

**>>> db.products.find().pretty()**



**If u want to use MongoDB with any application u need to use the Driver respectively.**

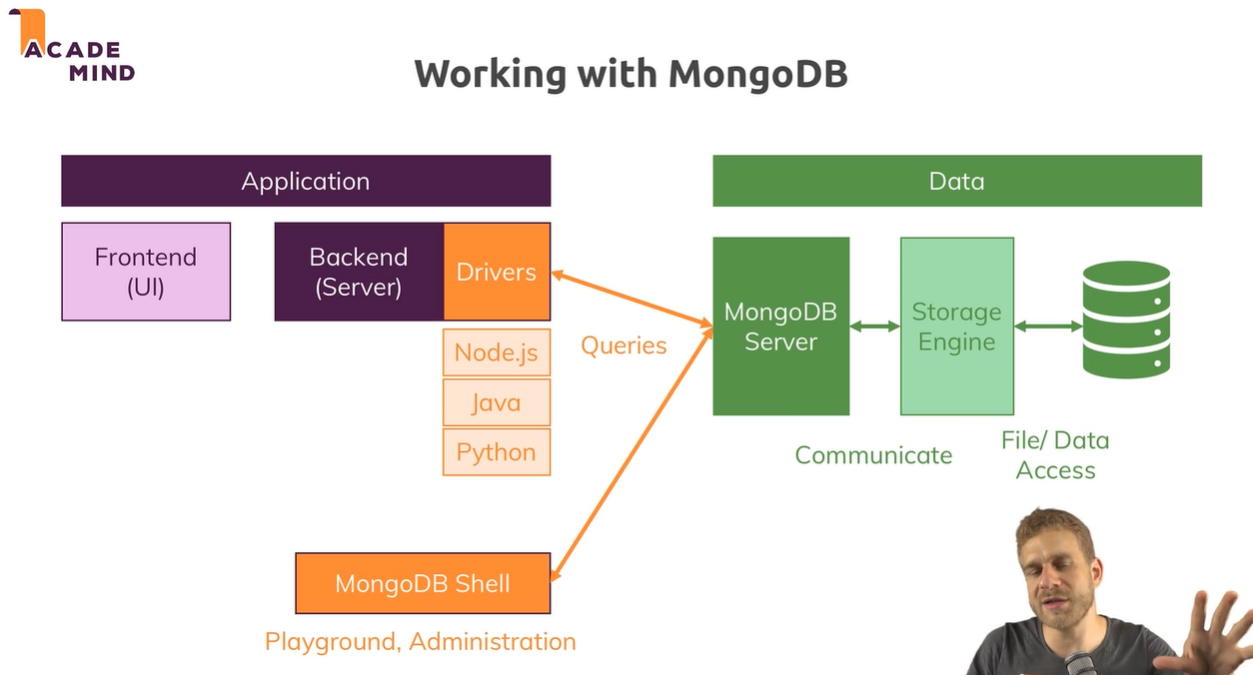
[**https://www.mongodb.com/docs/**](https://www.mongodb.com/docs/)

**Resources -> Drivers -> Java Drivers**

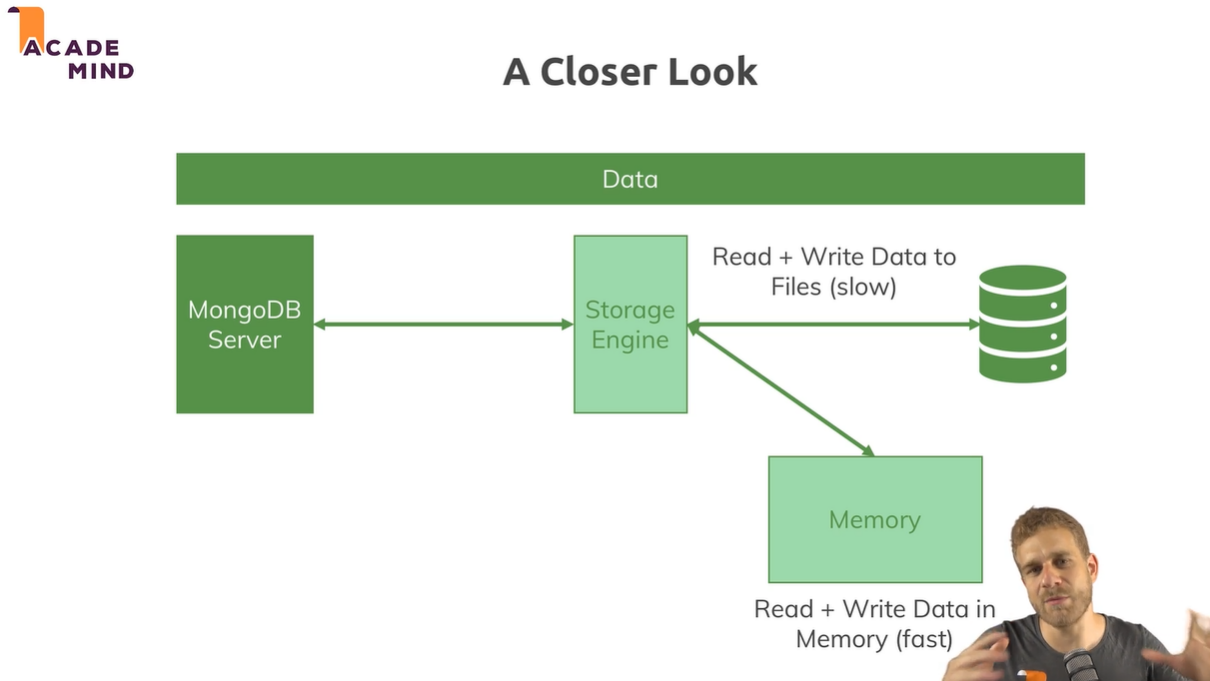
<https://www.mongodb.com/docs/drivers/java-drivers/>

<https://www.mongodb.com/docs/drivers/java/sync/current/>

**Overview: Working with MongoDB**

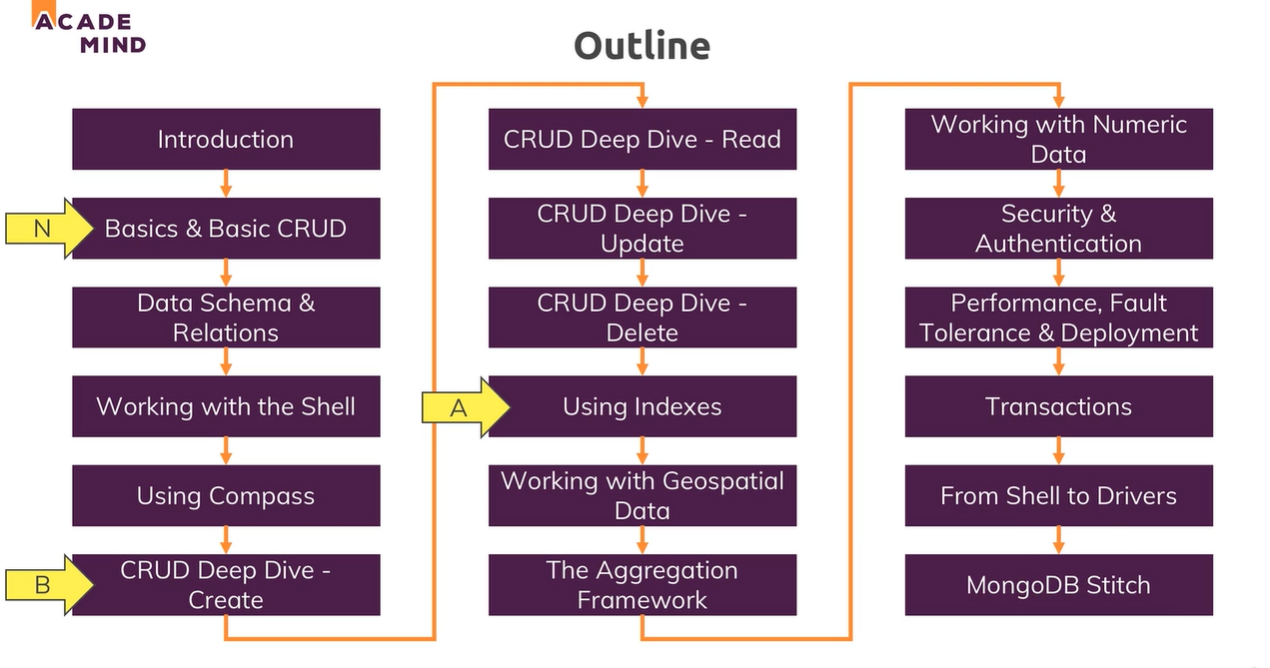


Now let me quickly give you an overview over how exactly you will work with mongodb and what Mongodb exactly does behind the scenes. So, we can generally differentiate between the application, so let's say your java application you are creating and the data layer. Now in your application, you will have some frontend that can be a single page application, that can be a mobile app, that can be the views your level or your java app renders. You'll then have a backend, your server where you have your server-side logic and then you have your data layer, so you have the database, the data storage, the files on a file system which hold your data in the end. You have the mongodb server and you have on the backend server where you will write your code, you have these drivers for the different languages. Now the drivers will interact with the mongodb server, the mongodb server is what we started with the mongod command by the way. 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. So, the mongodb basically gets the query from your driver or from your shell and then just knows I want to insert something and basically forwards that information after doing some other things to the storage engine and the storage engine then stores it in files in the end.



Now if we have a closer look at that data layer with the server, the storage engine, and the file system, then 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 of 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 and I will dive a little deeper into that later in the course but in general, you need to be aware that you always talk to the mongodb server and behind that server, 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 you could say.

**Overall Course Details:**



1. Understanding the Basics & CRUD Operations

========================================

**Points to Remember: -**

1. **Databases, Collections, Documents**

* A Database holds multiple Collections where each Collection can then hold multiple Documents.
* Databases and Collections are created “lazily” (i.e when a Document is inserted)
* A Document can’t directly be inserted into a Database, you need to use a Collection!

1. **Document Structure**

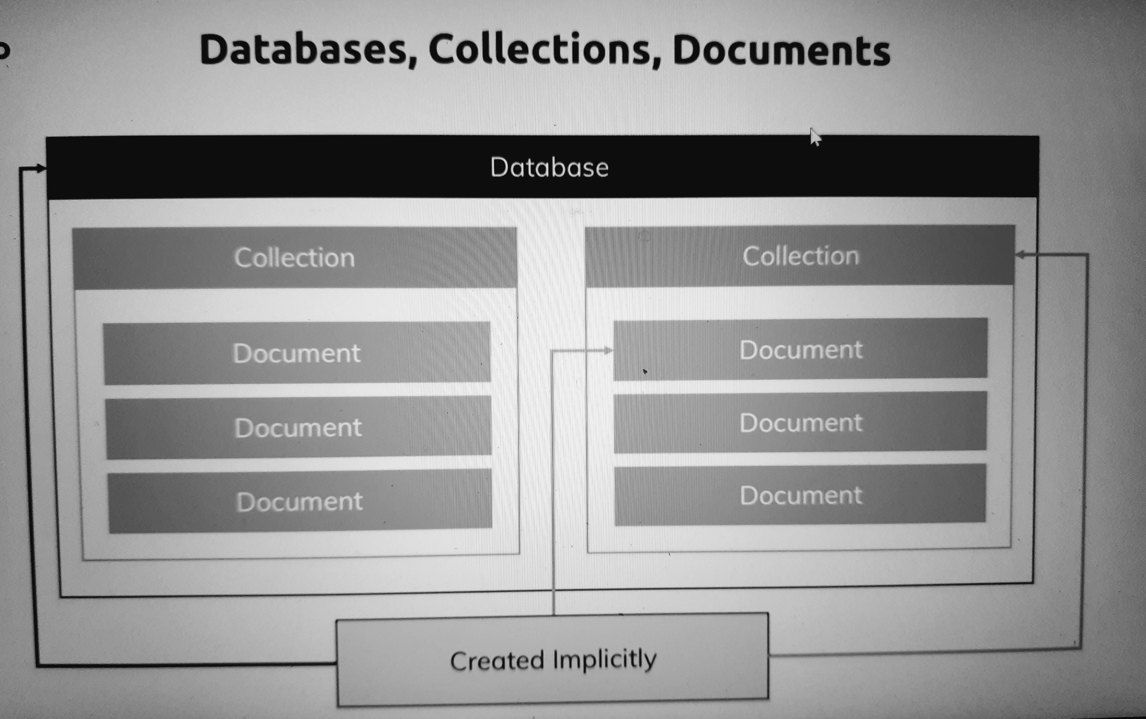
* Each document needs a unique ID (and gets one by default)
* You may have embedded documents and array fields

1. **CRUD Operations**

* CRUD = Create, Read, Update, Delete
* MongoDB offers multiple CRUD operations for single-document and bulk actions (e.g. insertOne(), insertMany(), …)
* Some methods require an argument (e.g. insertOne()), others don’t (e.g. find())
* find()returns a cursor, NOT a list of documents!
* Use filters to find specific documents

1. **Retrieving Data**

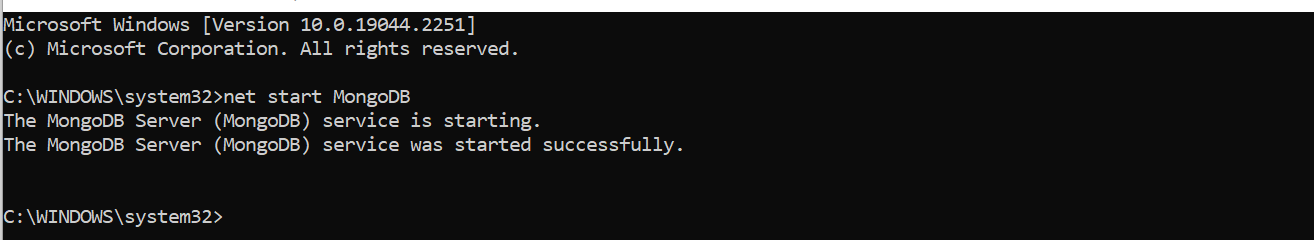
* Use filters and operators(e.g. $gt) to limit the number of documents you retrieve
* Use projection to limit the set of fields you retrieve.



it's really important that you understand how databases collections and documents are related. 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, you will see in this module already that 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. That is really convenient, later in the course we'll also learn a way of explicitly creating new collections which allows you to configure them a bit further but this is a cool feature which makes getting started super simple. Now this is the core concept you always should keep in mind and the relation between databases collections and documents you always should keep in mind.

Let’s start our mongodb server…

Run cmd as Admin > type **net start MongoDB**

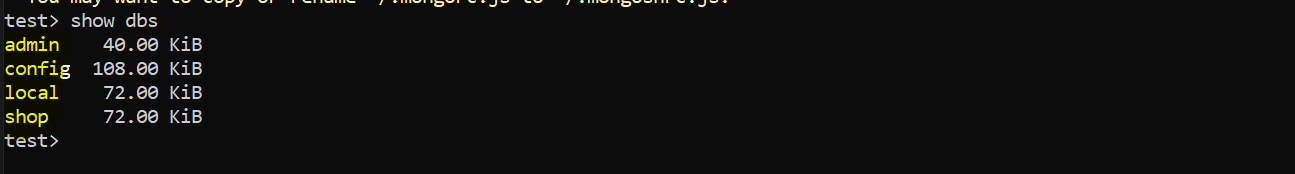


Start **Client** to send command to mongodb server

Start Menu > mongosh > enter > running on default port **27017**

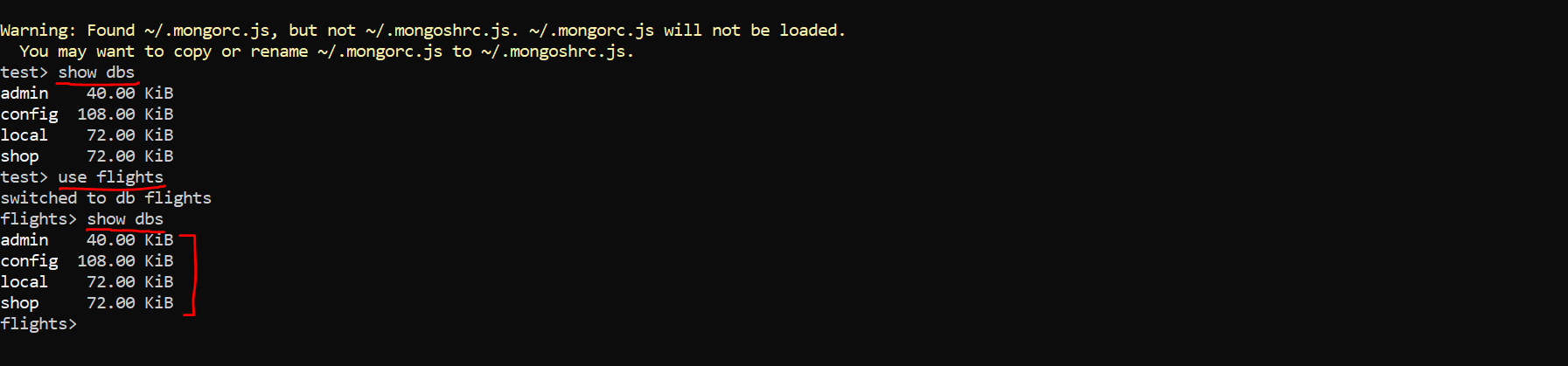
Our mongo shell will connect to our mongodb server.

>>> **show dbs** --- to see which all databases avails in our mongo server.

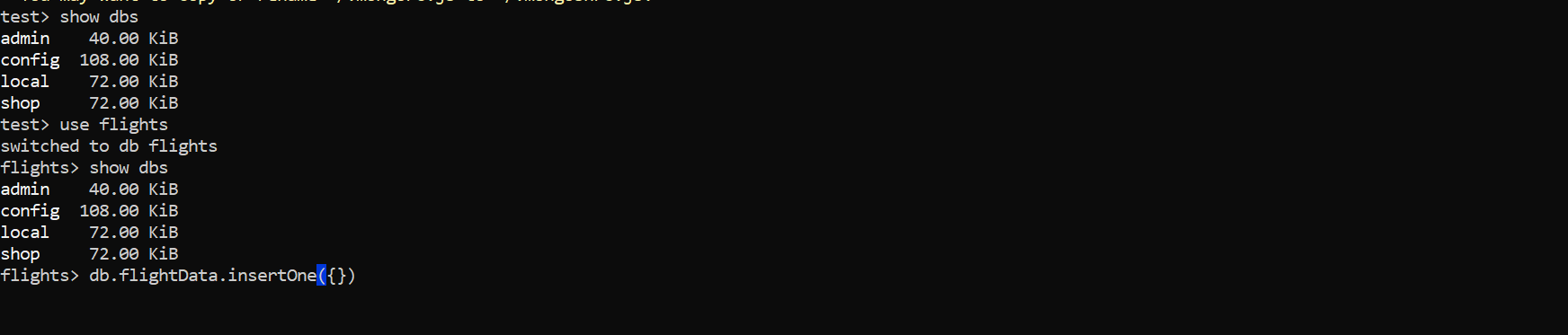


The default databases here simply exist to store configuration for this database server or for example the admin database will allow you to create users and roles and how people can use and interact with the database

we can switch to a database with the **use** command though and you can even switch to databases which don't exist yet.

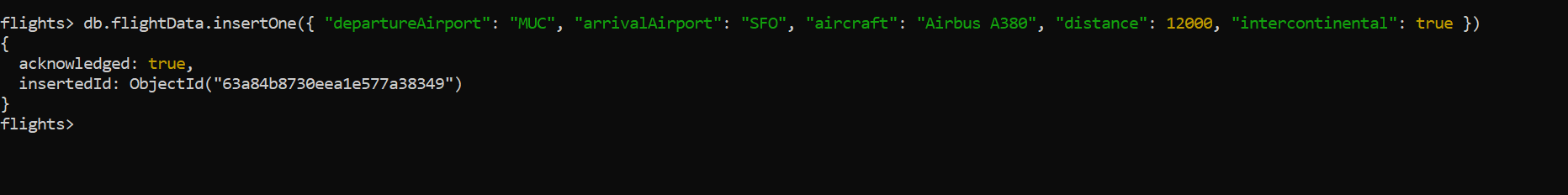


So, for example here I want to start with some flight data, so I'll name it flights and you can choose any name you want. If you hit enter, you see that we switched to the db flights but if I type show dbs, we still don't see that even though I did switch to it. The reason for this is that this database does not get created before we start entering data in there but if we do start entering data, it will be created automatically and implicitly, so we don't have to create the database in advance.



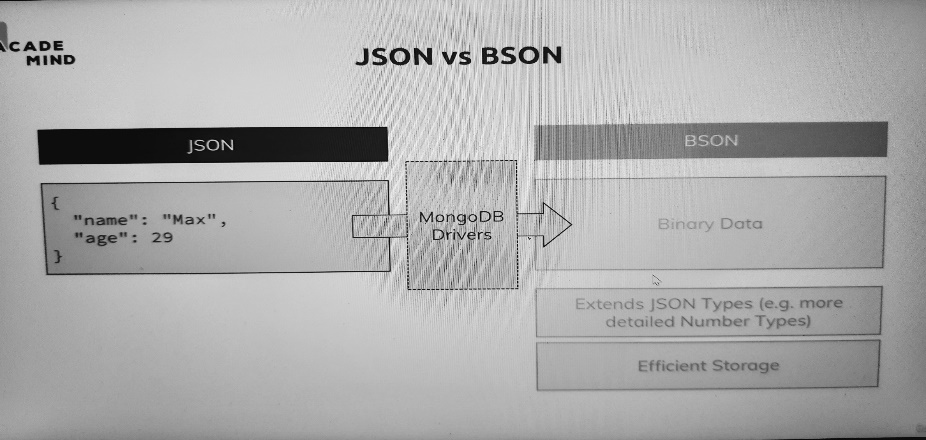
Now we get our flights database and in there, let's say we want to store some flight data. so, in our case the flights database here with **db** and then you chain commands by typing a dot and then the command and one command in quotation marks is that we type the name of a collection we want to interact with. Now a document is always defined with curly braces, now this is a collection which also doesn't exist yet but just like a database, it will be created on demand once we start inserting data. Now to insert data, we type another dot got the insertOne command for example. As the name suggests, this inserts one document into this collection.

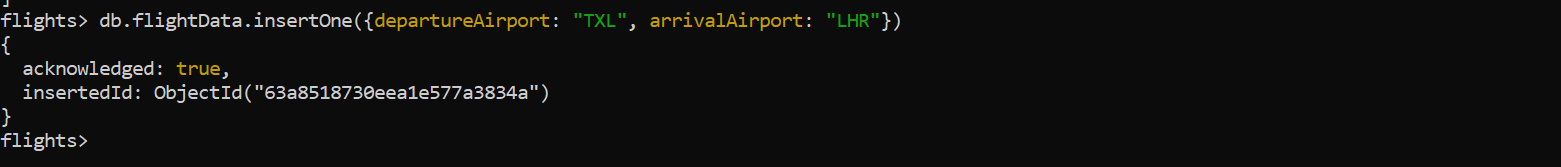
and now between these parentheses at the end here, we pass the data we want to store. you will know this syntax. there you also create objects with curly braces. To be very precise, we're not creating a javascript object here but a json document and json documents are also delimited or surrounded by curly braces. Now in that document, we now have key value pairs.





**JSON VS BSON**

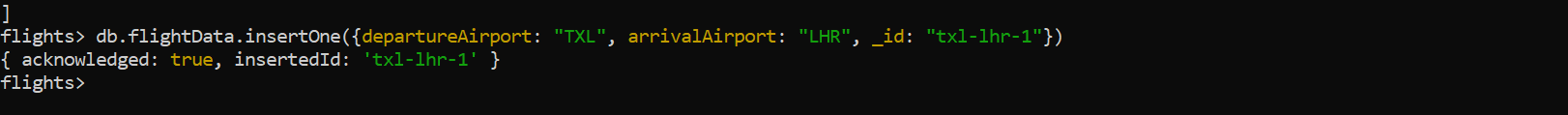




let's come back to that json thing because actually, mongodb does not use json but bson on which stands for binary json for storing data in your database. you can actually omit the quotation marks here as long as there is no whitespace in the name. If you add the data like this, you still see, this is accepted



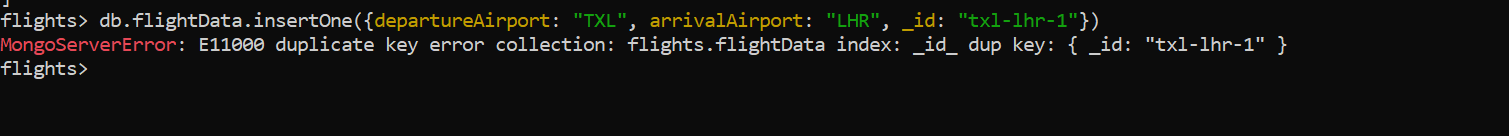
In there even though they have different schemas. but I also want you to take away and keep in mind that in mongodb, two documents in the same collection don't have to have the same schema. Often you will have at least some intersections or complete the equal schemas but it's not a must.



One other thing you have to know is that this ID which is auto-generated, you don't have to use the auto-generated ID, you just have to ensure that you have a unique ID but if you can ensure this, you can also assign IDs on your own. So, if I add this same entry again with departure airport and arrival airport, I can add \_ID in there as an extra field and it has to be \_ID, so not just ID, also not ID with a capital character, \_id like this and then you can use any ID you want, a number or another string like txl-lhr-1. This can be done and you see this was also accepted and inserted with this ID.



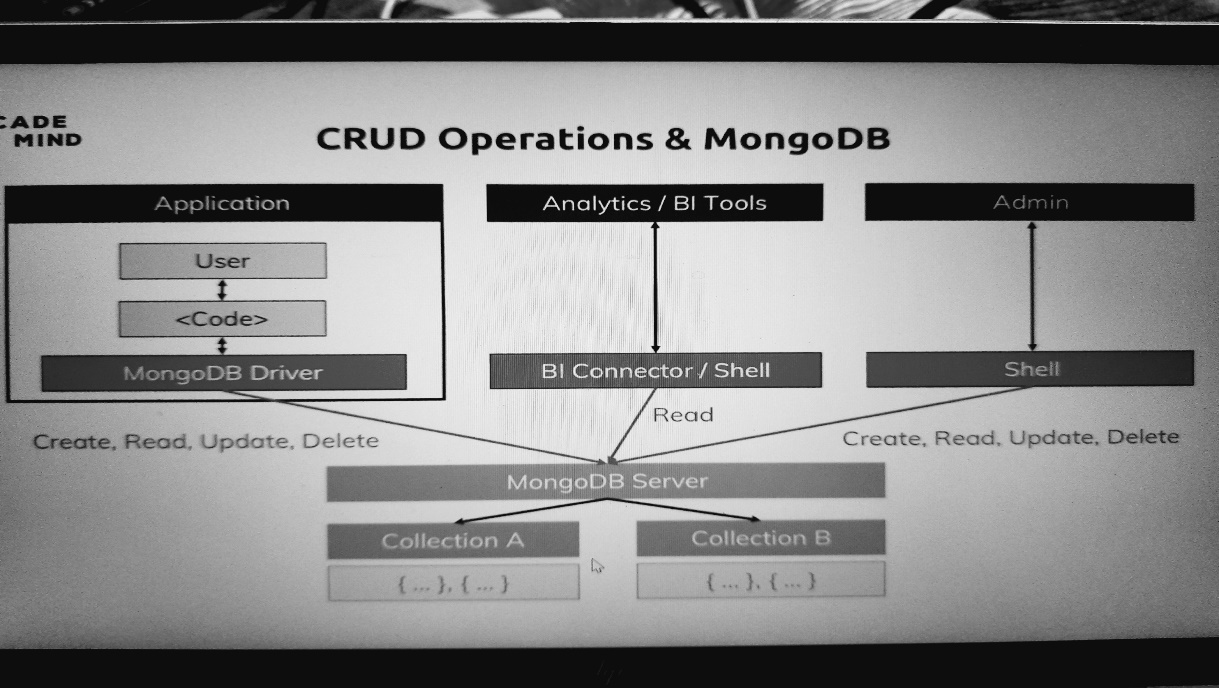
new one does now not have an objectId which was generated for us by mongodb but our own ID. If I try to enter the same entry again with the same ID though and that's the only thing which is the problem here, not the other data, the other data maybe equal as you can see here but if I use the same id again, we get an error and in the end, this error is simply telling us that we get a duplicate key error in this collection.



and that the duplicate key is the ID which is not allowed. So, this is the error we're seeing here and this is something you should also pay close attention to.

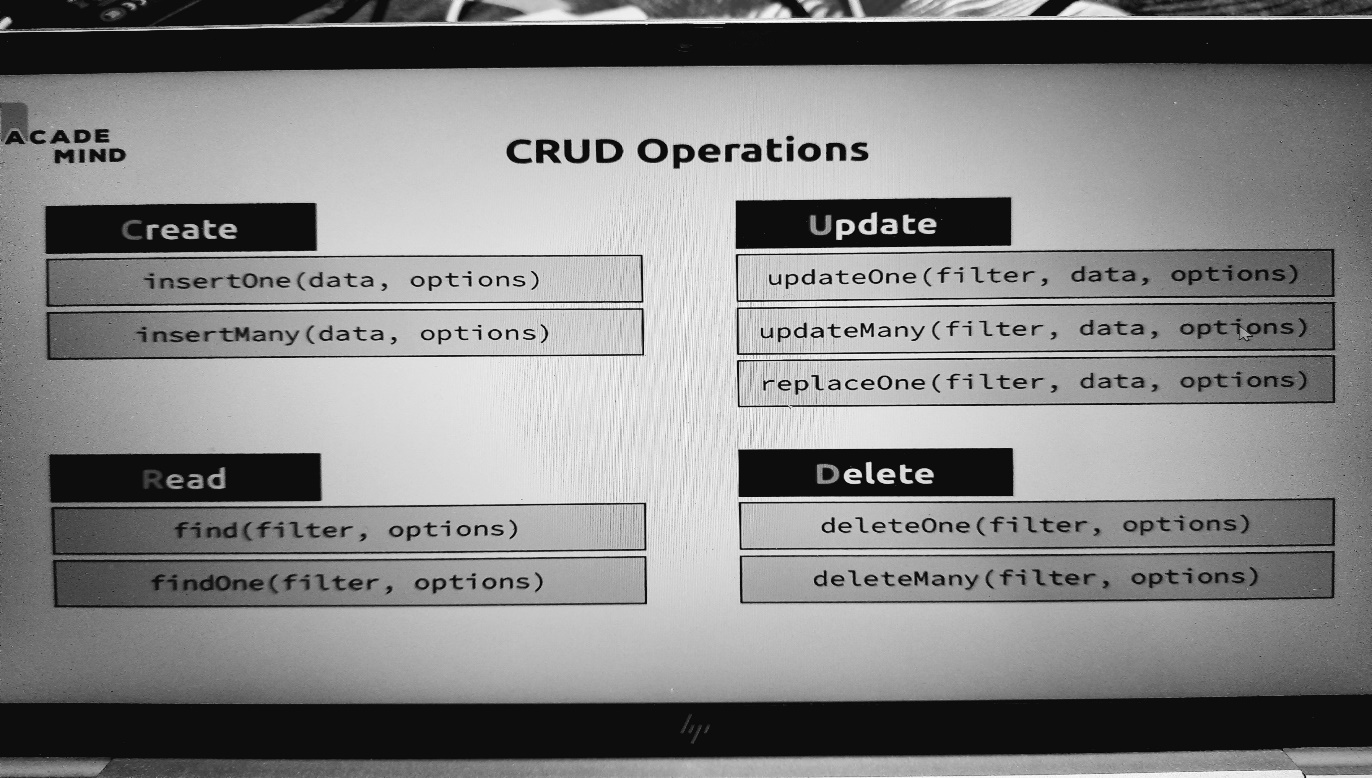
**CRUD Operations in MongoDB**

You can use mongodb for a broad variety of things, you might be building an application, or you have some data which is shipped to you as one file and you just want to run some analytics on it or you're an administrator of a database, of a mongo database, in all these cases, you will have to interact with the database. Now in the application case, you might have an app, a mobile app or a website where user interacts with your code in the end and that code can be written in Nodejs, php, can be a desktop app written in C++, really no limits there and you will have your mongodb driver included in your application then. In the other cases like for analytics, you might be using the business intelligence connector mongodb offers or you directly use the shell, or some other import mechanism provided by your favorite BI tool or as an administrator, you'll work with the shell like we're doing it right now. In all these cases, you want to interact with your mongodb server and for an application like a website, you typically want to be able to create elements, you want to be able to read, update and delete them.

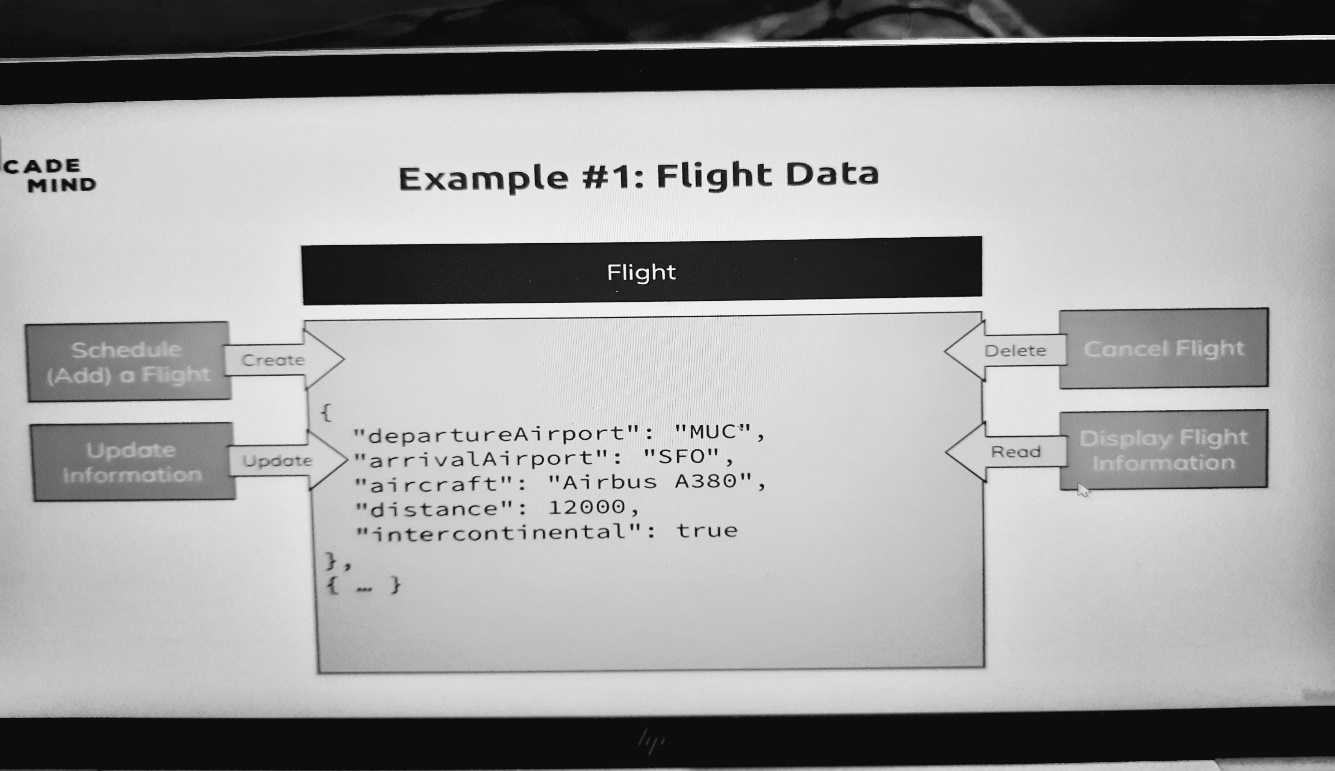


Let's say you were building a blog, you want to be able to create new posts, to deliver those posts to your users, you want to be able to update these posts, so, to change them or maybe even delete them. With analytics, you also at least want to be able to read the data, you might never change it, but you want to read it and as an admin, you might want to do all these things to fix some data or to play around with it as we are doing it. So, these are these core crud operations and, in the end, these operations are the only thing you do with your data, this is what you have your data for, to create it, manage it, read it, all these things and you do these things with the mongodb server and there of course, in the end on a collection and the documents in that collection.

**CRUD Operations**



insertOne allows us to insert some data into a collection and all these methods I'm showing you are directly executed on a collection as we did it in the last lectures and we can also pass some options along with insertOne Besides inserting one document, we can also insert many documents by the way, so if you have a use case where you need to insert more than one document at a time, you can do that too, this is important for creating data. Now for reading data, we got find and find also allows us to pass in some arguments. Right now we're calling it without anything, just find empty parentheses, now you can pass in a filter and filters are very powerful, they allow you to narrow down the data you're retrieving, something you of course often want to do in a database and you can also pass some options to find and again, we'll take a close look at all these options and filters throughout the course too. You also got find one, the difference here is that find finds all matching documents, find one simply gives you the first matching document it finds. Update one is then useful for changing one piece of data and just as we can insert many documents at once, we also have update many. Now the update functions take three arguments because we can pass a filter to narrow down which document or documents to change then the data describing the change, so what should be changed and again some options we can use to configure that process. We also got replace one in case we just want to replace a document entirely with a new one. There also is delete, we can delete one or delete many documents in a collection and there we also pass in a filter to find out which document or documents to delete.



Now let's see all these things at an example, again having a look at our flight data here which could look like this, basically the data we already worked with and there, let's say this is the data we're using on our application. Our application might be a website where people working at an airport or at an airline can schedule a new flight, therefore they would create an entry in our flight data collection, they can change information about this flight so they can update information, for example if the aircraft changed, they can cancel the flight let's say so they can delete it and they can also display flight information on their own website in the mobile apps for their passengers and therefore, they need to be able or you need to be able to read the data from this collection. This is our little example here using the data we already set up. So, let's get started and play around with this crud operations and our data.

**Finding, Inserting, Deleting & Updating Elements**

you see we got three documents in there we can simply use our flight data collection because all these commands are always executed on a collection where you want to add or delete documents and there, we got deleteOne or deleteMany. Delete one takes a filter to find out which document to delete and now we could filter for all kinds of things. A filter is defined as a document, so with curly braces and then in its simplest form, you now simply define which key and which value you want to delete or the document with that key and value you want to delete. So for example here, we could say departure airport, if that is equal to txl so to Berlin, then I want to delete that document and what this would do is it would go ahead, find the first document in our database where the departure airport is txl and delete it. if I just execute deleteMany like this, it fails though, so I need to pass in a criteria. so let's first of all update them real quick to add a new field to both documents so that they have something in common, so that we can delete them in bulk, so that I can also show you this. So for this, let's go back to the collection and now use update one, update one updates as the name suggests, one element and for update one, we first of all define a filter where we say which document should be updated The second argument then is how I want to change it and now the question is what do you pass there. Now let's say we want to add a new field and therefore we pass a document here, marker which hold some text, delete. but if I hit enter, I actually get an error and the error is that the update operation document must contain an atomic operator or atomic operators. Now this is a very valid error and there's not really much you can do with it if you're just starting with mongodb. instead, what you do pass here is a document with curly braces with a special keyword, $set and this is something you can remember, whenever you see something with dollar sign in mongodb, it's a reserved operator or word, $set is simply identified by mongodb when used in the update one operation to describe the changes you want to make. The value of $set then is a document with curly braces and we need to close another pair of curly braces for our surrounding curly braces now marked in purple and now what this does is it tells mongodb hey for this document you're finding, please set this value and if this value did exist, it would change it to delete and if it does not exist as in our case, it will add it.

**>>> db.flightData.find().pretty()**

**[**

**{**

**\_id: ObjectId("63a84b8730eea1e577a38349"),**

**departureAirport: 'MUC',**

**arrivalAirport: 'SFO',**

**aircraft: 'Airbus A380',**

**distance: 12000,**

**intercontinental: true**

**},**

**{**

**\_id: ObjectId("63a8518730eea1e577a3834a"),**

**departureAirport: 'TXL',**

**arrivalAirport: 'LHR'**

**},**

**{ \_id: 'txl-lhr-1', departureAirport: 'TXL', arrivalAirport: 'LHR' }**

**]**

**>>> db.flightData.deleteOne({departureAirport: "TXL"})**

**{ acknowledged: true, deletedCount: 1 }**

**>>> db.flightData.find().pretty()**

**[**

**{**

**\_id: ObjectId("63a84b8730eea1e577a38349"),**

**departureAirport: 'MUC',**

**arrivalAirport: 'SFO',**

**aircraft: 'Airbus A380',**

**distance: 12000,**

**intercontinental: true**

**},**

**{ \_id: 'txl-lhr-1', departureAirport: 'TXL', arrivalAirport: 'LHR' }**

**]**

**>>> db.flightData.deleteMany()**

**MongoshInvalidInputError: [COMMON-10001] Missing required argument at position 0 (Collection.deleteMany)**

**>>> db.flightData.updateOne({distance: 12000}, {marker: delete})**

**MongoInvalidArgumentError: Update document requires atomic operators**

**>>> db.flightData.updateOne({distance: 12000}, {$set: {marker: 'delete'}})**

**{**

**acknowledged: true,**

**insertedId: null,**

**matchedCount: 1,**

**modifiedCount: 1,**

**upsertedCount: 0**

**}**

**>>> db.flightData.find().pretty()**

**[**

**{**

**\_id: ObjectId("63a84b8730eea1e577a38349"),**

**departureAirport: 'MUC',**

**arrivalAirport: 'SFO',**

**aircraft: 'Airbus A380',**

**distance: 12000,**

**intercontinental: true,**

**marker: 'delete'**

**},**

**{ \_id: 'txl-lhr-1', departureAirport: 'TXL', arrivalAirport: 'LHR' }**

**]**

**>>> db.flightData.updateMany({}, {$set: {marker : 'toDelete'}})**

**{**

**acknowledged: true,**

**insertedId: null,**

**matchedCount: 2,**

**modifiedCount: 2,**

**upsertedCount: 0**

**}**

**>>> db.flightData.find().pretty()**

**[**

**{**

**\_id: ObjectId("63a84b8730eea1e577a38349"),**

**departureAirport: 'MUC',**

**arrivalAirport: 'SFO',**

**aircraft: 'Airbus A380',**

**distance: 12000,**

**intercontinental: true,**

**marker: 'toDelete'**

**},**

**{**

**\_id: 'txl-lhr-1',**

**departureAirport: 'TXL',**

**arrivalAirport: 'LHR',**

**marker: 'toDelete'**

**}**

**]**

**>>> db.flightData.deleteMany({marker: "toDelete"})**

**{ acknowledged: true, deletedCount: 2 }**

**>>> db.flightData.find().pretty()**

**B L A N K**

**>>> db.flightData.insertMany([{"departureAirport": "MUC", "arrivalAirport": "SFO", "aircraft": "Airbus A380", "distance": 12000, "intercontinental": true }, { "departureAirport": "LHR", "arrivalAirport": "TXL", "aircraft": "Airbus A320", "distance": 950, "intercontinental": false }])**

**{**

**acknowledged: true,**

**insertedIds: {**

**'0': ObjectId("63a9d09be5e6e4fb6c8bc1da"),**

**'1': ObjectId("63a9d09be5e6e4fb6c8bc1db")**

**}**

**}**

**>>> db.flightData.find().pretty()**

**[**

**{**

**\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),**

**departureAirport: 'MUC',**

**arrivalAirport: 'SFO',**

**aircraft: 'Airbus A380',**

**distance: 12000,**

**intercontinental: true**

**},**

**{**

**\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),**

**departureAirport: 'LHR',**

**arrivalAirport: 'TXL',**

**aircraft: 'Airbus A320',**

**distance: 950,**

**intercontinental: false**

**}**

**]**

insertMany now allows you well as the name suggests, to insert many elements and this is actually not done by adding many elements separated by commas but instead by passing an array and there you also see this order t automatically keeps which I mentioned. It's almost the same ID but the second ID has a b at the end so mongodb keeps the order here.

**Diving Deeper into Finding Data**

we always used this way of finding data, find with no arguments. Now just as you can filter for specific records or documents when deleting or updating, you can also do so when finding. instead of passing nothing, you can pass a document to it which will be interpreted as a filter for your collection and you can filter for anything that makes sense.

You could filter for a name equal to Max but of course this returns nothing because in our collection, we get these two flights and there is no name field in there but what we can do of course is we can use flight data and find all flights that are let's say intercontinental.

**>>> db.flightData.find({name : 'max'})**

B L A N K

so we use that key here. We do so by adding a document, then the name and this can be wrapped by quotation marks, but it doesn't have to and then let's say I want to find all flights where this is true, this would be a valid filter.

**>>> db.flightData.find({intercontinental: true}).pretty()**

**[**

**{**

**\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),**

**departureAirport: 'MUC',**

**arrivalAirport: 'SFO',**

**aircraft: 'Airbus A380',**

**distance: 12000,**

**intercontinental: true**

**}**

**]**

**flights> db.flightData.find({distance: 12000})**

**[**

**{**

**\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),**

**departureAirport: 'MUC',**

**arrivalAirport: 'SFO',**

**aircraft: 'Airbus A380',**

**distance: 12000,**

**intercontinental: true**

**}**

**]**

if we want to have a greater than check, we can do that too. And there we just need to pass another document or object, we need to pass another object as a value for distance and here we again use one of these special operators provided by mongodb, $**gt** which stands for greater than and no worries, you don't need to learn them by heart right now. Gt does exist, $**gt** and that means greater than and then you enter the value which you want to have as a greater than value.

**>>> db.flightData.find({distance:{$gt: 900}})**

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),

departureAirport: 'MUC',

arrivalAirport: 'SFO',

aircraft: 'Airbus A380',

distance: 12000,

intercontinental: true

},

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false

}

]

So if I want to find all, I should use find, if I am only interested in the first matching element, findOne is the solution. Important is it will still respect that filter but then not look for all matching elements but only the first matching element.

**>>> db.flightData.findOne({distance: {$gt: 900}})**

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),

departureAirport: 'MUC',

arrivalAirport: 'SFO',

aircraft: 'Airbus A380',

distance: 12000,

intercontinental: true

}

**>>> db.flightData.findOne({distance: {$lt: 1100}})**

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false

}

**>>> db.flightData.find({distance: {$lt: 1100}})**

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false

}

]

**>>> db.flightData.find({distance: {$eq: 950}})**

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false

}

]

**>>> db.flightData.find({distance: {$eq: 900}})**

**B L A N K**

**update() VS updateMany()**

a bit like updateMany, updateMany was used to update all matching elements and update would also update all matching elements. I recommend using updateOne and many to avoid this issue with update.

**>>> db.flightData.updateOne({\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da")}, {$set: {delayed: true}})**

{

acknowledged: true,

insertedId: null,

matchedCount: 1,

modifiedCount: 1,

upsertedCount: 0

}

**>>> db.flightData.find()**

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),

departureAirport: 'MUC',

arrivalAirport: 'SFO',

aircraft: 'Airbus A380',

distance: 12000,

intercontinental: true,

delayed: true

},

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false

}

]

**>>> db.flightData.updateMany({}, {$set: {flightName: "Indigo"}})**

{

acknowledged: true,

insertedId: null,

matchedCount: 2,

modifiedCount: 2,

upsertedCount: 0

}

**>>> db.flightData.find()**

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),

departureAirport: 'MUC',

arrivalAirport: 'SFO',

aircraft: 'Airbus A380',

distance: 12000,

intercontinental: true,

delayed: false,

flightName: 'Indigo'

},

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false,

flightName: 'Indigo'

}

]

**Understanding find() & Cursor Object**

**>>> cls** To Clear Screen

now let me add another kind of data. I'll clear the console with the cls command for that and now I'll add a passenger's collection to my flight's database, and I want to insertMany passengers in there. Now you find a file with name **passengers**.**json** file with an array of passenger data.

**>>> db.passengers.insertMany([ { "name": "Max Schwarzmueller", "age": 29 }, { "name": "Manu Lorenz", "age": 30 }, { "name": "Chris Hayton", "age": 35 }, { "name": "Sandeep Kumar", "age": 28 }, { "name": "Maria Jones", "age": 30 }, { "name": "Alexandra Maier", "age": 27 }, { "name": "Dr. Phil Evans", "age": 47 }, { "name": "Sandra Brugge", "age": 33 }, { "name": "Elisabeth Mayr", "age": 29 }, { "name": "Frank Cube", "age": 41 }, { "name": "Karandeep Alun", "age": 48 }, { "name": "Michaela Drayer", "age": 39 }, { "name": "Bernd Hoftstadt", "age": 22 }, { "name": "Scott Tolib", "age": 44 }, { "name": "Freddy Melver", "age": 41 }, { "name": "Alexis Bohed", "age": 35 }, { "name": "Melanie Palace", "age": 27 }, { "name": "Armin Glutch", "age": 35 }, { "name": "Klaus Arber", "age": 53 }, { "name": "Albert Twostone", "age": 68 }, { "name": "Gordon Black", "age": 38 }])**

{

acknowledged: true,

insertedIds: {

'0': ObjectId("63aabcc2be806e3b8eec584e"),

'1': ObjectId("63aabcc2be806e3b8eec584f"),

'2': ObjectId("63aabcc2be806e3b8eec5850"),

'3': ObjectId("63aabcc2be806e3b8eec5851"),

'4': ObjectId("63aabcc2be806e3b8eec5852"),

'5': ObjectId("63aabcc2be806e3b8eec5853"),

'6': ObjectId("63aabcc2be806e3b8eec5854"),

'7': ObjectId("63aabcc2be806e3b8eec5855"),

'8': ObjectId("63aabcc2be806e3b8eec5856"),

'9': ObjectId("63aabcc2be806e3b8eec5857"),

'10': ObjectId("63aabcc2be806e3b8eec5858"),

'11': ObjectId("63aabcc2be806e3b8eec5859"),

'12': ObjectId("63aabcc2be806e3b8eec585a"),

'13': ObjectId("63aabcc2be806e3b8eec585b"),

'14': ObjectId("63aabcc2be806e3b8eec585c"),

'15': ObjectId("63aabcc2be806e3b8eec585d"),

'16': ObjectId("63aabcc2be806e3b8eec585e"),

'17': ObjectId("63aabcc2be806e3b8eec585f"),

'18': ObjectId("63aabcc2be806e3b8eec5860"),

'19': ObjectId("63aabcc2be806e3b8eec5861"),

'20': ObjectId("63aabcc2be806e3b8eec5862")

}

}

and this should insert many elements with the insertMany command with all these automatically generated IDs. there are our passengers. Now on first sight, this might look good but actually if you have a look at the last entry, Albert Twostone and then you have a look at the data you found attached, you will see that Albert is not the last entry, Gordon Black actually is. Now the order here is the same order as it was added to the database, so where is Gordon? Well you also see the type it for more, if we type it, then we see Gordon Black. Now this it command is a feature shown in the shell, I will show you another way of getting the data if you were using a mongodb driver in an application in a second but the fact that we had used that is not shell specific, instead find, so passengers find or the find command in general no matter on which collection you use it gives you back a cursor and not all the data. Previously, it looked like it always gives us back all the data but actually it gives us a so-called cursor object

**>>> db.passengers.find().pretty()**

[

{

\_id: ObjectId("63aabcc2be806e3b8eec584e"),

name: 'Max Schwarzmueller',

age: 29

},

{

\_id: ObjectId("63aabcc2be806e3b8eec584f"),

name: 'Manu Lorenz',

age: 30

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5850"),

name: 'Chris Hayton',

age: 35

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5851"),

name: 'Sandeep Kumar',

age: 28

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5852"),

name: 'Maria Jones',

age: 30

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5853"),

name: 'Alexandra Maier',

age: 27

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5854"),

name: 'Dr. Phil Evans',

age: 47

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5855"),

name: 'Sandra Brugge',

age: 33

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5856"),

name: 'Elisabeth Mayr',

age: 29

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5857"),

name: 'Frank Cube',

age: 41

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5858"),

name: 'Karandeep Alun',

age: 48

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5859"),

name: 'Michaela Drayer',

age: 39

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585a"),

name: 'Bernd Hoftstadt',

age: 22

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585b"),

name: 'Scott Tolib',

age: 44

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585c"),

name: 'Freddy Melver',

age: 41

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585d"),

name: 'Alexis Bohed',

age: 35

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585e"),

name: 'Melanie Palace',

age: 27

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585f"),

name: 'Armin Glutch',

age: 35

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5860"),

name: 'Klaus Arber',

age: 53

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5861"),

name: 'Albert Twostone',

age: 68

}

]

Type "it" for more

flights> it

[

{

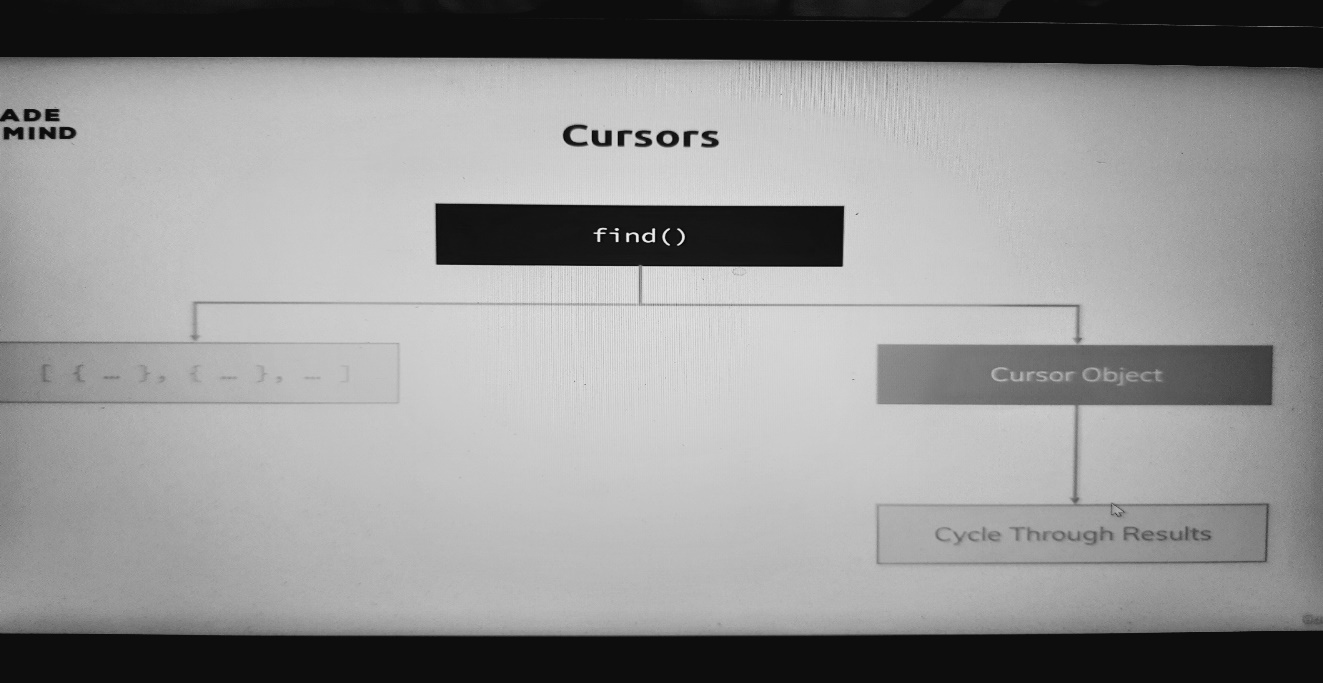
\_id: ObjectId("63aabcc2be806e3b8eec5862"),

name: 'Gordon Black',

age: 38

}

]



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 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 cycle through the results and that is what that it command did, it basically used that cursor to fetch the next bunch of data. So find here really just gives us back that cursor and we can actually see that by using some other methods here

for example there is **toArray**

**>>> db.passengers.find().toArray()**

**[**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec584e"),**

**name: 'Max Schwarzmueller',**

**age: 29**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec584f"),**

**name: 'Manu Lorenz',**

**age: 30**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5850"),**

**name: 'Chris Hayton',**

**age: 35**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5851"),**

**name: 'Sandeep Kumar',**

**age: 28**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5852"),**

**name: 'Maria Jones',**

**age: 30**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5853"),**

**name: 'Alexandra Maier',**

**age: 27**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5854"),**

**name: 'Dr. Phil Evans',**

**age: 47**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5855"),**

**name: 'Sandra Brugge',**

**age: 33**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5856"),**

**name: 'Elisabeth Mayr',**

**age: 29**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5857"),**

**name: 'Frank Cube',**

**age: 41**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5858"),**

**name: 'Karandeep Alun',**

**age: 48**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5859"),**

**name: 'Michaela Drayer',**

**age: 39**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585a"),**

**name: 'Bernd Hoftstadt',**

**age: 22**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585b"),**

**name: 'Scott Tolib',**

**age: 44**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585c"),**

**name: 'Freddy Melver',**

**age: 41**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585d"),**

**name: 'Alexis Bohed',**

**age: 35**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585e"),**

**name: 'Melanie Palace',**

**age: 27**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585f"),**

**name: 'Armin Glutch',**

**age: 35**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5860"),**

**name: 'Klaus Arber',**

**age: 53**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5861"),**

**name: 'Albert Twostone',**

**age: 68**

**},**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5862"),**

**name: 'Gordon Black',**

**age: 38**

**}**

**]**

if you execute this, now you see Gordon Black as the last element and now there is no type it to see more because toArray will actually go ahead and exhaust the cursor, so go through the entire list and fetch all the documents and not stop after the first 20, which by the way is simply a feature by the mongodb shell, it gives you the first 20 documents automatically but then stops, toArray simply gets them all and gives you an array.

Now that might be fine here but of course that is also not optimal if you've got a lot of elements. In your application code, you would most likely use something like passengers find and then there is a **forEach** method and **forEach** here simply allows you to write some code to do something on every element that is in your database, on every document you got in there. Now the exact syntax you can use is something you can find in the driver docs for your driver, so if you're using PHP, check out the PHP mongodb driver docs and you will see how to use that forEach function. In javascript and the shell is based on javascript, you could pass a javascript function to it using an arrow function here where you will get an argument passed into the function automatically and that will be your document and you can name this argument however you want, so in our case it will be our passenger data and then you can use it in here and here in the shell, there is a print json command which we can use to pass or to well yeah pass our json data which we want to output like passenger data. And now we also get all the documents because now what we're doing is on the cursor which find does return, we execute the built-in forEach method to execute this method here, this anonymous method on every element and the cursor will simply go through our entire collection, fetch all documents one at a time, execute that function on it, pass the document into the function and then here we just happened to print it out like this in your application with your favorite language.

**>>> db.passengers.find().forEach((passengersData) => {printjson(passengersData)})**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec584e"),**

**name: 'Max Schwarzmueller',**

**age: 29**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec584f"),**

**name: 'Manu Lorenz',**

**age: 30**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5850"),**

**name: 'Chris Hayton',**

**age: 35**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5851"),**

**name: 'Sandeep Kumar',**

**age: 28**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5852"),**

**name: 'Maria Jones',**

**age: 30**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5853"),**

**name: 'Alexandra Maier',**

**age: 27**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5854"),**

**name: 'Dr. Phil Evans',**

**age: 47**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5855"),**

**name: 'Sandra Brugge',**

**age: 33**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5856"),**

**name: 'Elisabeth Mayr',**

**age: 29**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5857"),**

**name: 'Frank Cube',**

**age: 41**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5858"),**

**name: 'Karandeep Alun',**

**age: 48**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5859"),**

**name: 'Michaela Drayer',**

**age: 39**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585a"),**

**name: 'Bernd Hoftstadt',**

**age: 22**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585b"),**

**name: 'Scott Tolib',**

**age: 44**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585c"),**

**name: 'Freddy Melver',**

**age: 41**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585d"),**

**name: 'Alexis Bohed',**

**age: 35**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585e"),**

**name: 'Melanie Palace',**

**age: 27**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec585f"),**

**name: 'Armin Glutch',**

**age: 35**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5860"),**

**name: 'Klaus Arber',**

**age: 53**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5861"),**

**name: 'Albert Twostone',**

**age: 68**

**}**

**{**

**\_id: ObjectId("63aabcc2be806e3b8eec5862"),**

**name: 'Gordon Black',**

**age: 38**

**}**

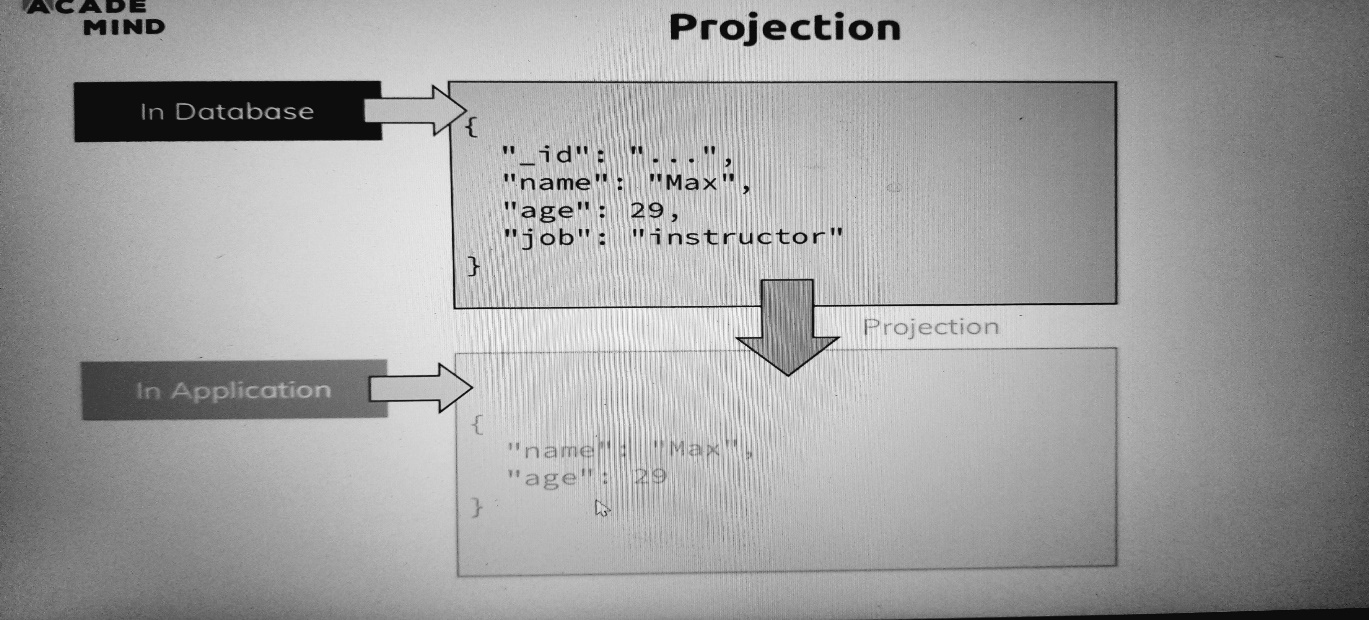
Find does not by default give you all the documents even though it looked like it did because the shell actually gives you the first 20 documents by default, instead find gives you a cursor and then it's up to you to use that cursor with toArray to force it to get all documents and put them into an array or better for large sets of documents, with forEach and there are more methods available which you find in your driver docs.

Technically you need to know that with forEach, it will really just fetch the next document for every loop cycle, so this is very efficient here because it does not fetch all the data in advance, load it into memory and then force you to go through it with forEach, instead it really just fetches the data on the demand therefore of course not overusing your bandwidth and not loading too much into memory. So, this is the important thing to take away, find gives you a cursor and not the element itself.

This also is the reason why on find one, we could not use pretty, you remember if I use passengers find one, this works, find one pretty fails because pretty is a method that simply exists on the cursor here. So, this is why find one does not support it because find one does not give us a cursor because it only gives us one document anyways. Find gives us multiple documents, therefore we don't get the data immediately, we get the cursor instead, the shell happens to take that cursor and give us the first 20 documents by default, but we can also control that with forEach or toArray and here.

This is what you need to take away here, for the other methods like insert, update, and delete, cursors do not exist because these methods don't fetch data, they manipulate it.

**Understanding Projection**



Now what is projection? Imagine you have this kind of data in your database, some information about a person, in this case me with name, age, job and that ID. Now imagine in your application, you don't need all this data, maybe you just need the name and age for a given page on your website. so you would kind of impact your bandwidth, you would fetch unnecessary data and you want to prevent this, so it would be better to filter this out on the mongodb server and this is exactly what projection can help you with. Projection is super easy to use, now let's say I'm only interested in the names of the passengers. Now you can do projection by using find, now you need to pass a first argument because I actually want to pass a second argument and that first argument now is an empty document because I want to find all passengers and I don't want to pass a filter and the second argument now allows us to project. And here, the projection is set up by passing another document, so another pair of curly braces and here you simply specify which key value pairs you want to get. For that, you simply name the field you want to get, so let's say in our passengers, we have a name and we want to get that name, so we type name here and then as a value, simply a one, this means include it in the data you're returning to me. If we now output that, we get all our fields with just a name and well still the ID. so the age is missing and the age is of course not missing in a database but only in the data we retrieved from it, so in the data we're showing here. So this is projection doing its job but what about the id? The id is a special field in your data, by default it's always included, so the default in general is everything is included but even if you use projection as we do it with the second argument, the id is always included too.

**>>> db.passengers.find({}, {name : 1}).pretty()**

[

{\_id: ObjectId("63aabcc2be806e3b8eec584e"), name: 'Max Schwarzmueller'

{ \_id: ObjectId("63aabcc2be806e3b8eec584f"), name: 'Manu Lorenz' },

{ \_id: ObjectId("63aabcc2be806e3b8eec5850"), name: 'Chris Hayton' },

{ \_id: ObjectId("63aabcc2be806e3b8eec5851"), name: 'Sandeep Kumar' },

{ \_id: ObjectId("63aabcc2be806e3b8eec5852"), name: 'Maria Jones' },

{\_id: ObjectId("63aabcc2be806e3b8eec5853"),name: 'Alexandra Maier'},

{ \_id: ObjectId("63aabcc2be806e3b8eec5854"), name: 'Dr. Phil Evans' },

{ \_id: ObjectId("63aabcc2be806e3b8eec5855"), name: 'Sandra Brugge' },

{ \_id: ObjectId("63aabcc2be806e3b8eec5856"), name: 'Elisabeth Mayr' },

{ \_id: ObjectId("63aabcc2be806e3b8eec5857"), name: 'Frank Cube' },

{ \_id: ObjectId("63aabcc2be806e3b8eec5858"), name: 'Karandeep Alun' },

{\_id: ObjectId("63aabcc2be806e3b8eec5859"),name: 'Michaela Drayer'},

{\_id: ObjectId("63aabcc2be806e3b8eec585a"),name: 'Bernd Hoftstadt'},

{ \_id: ObjectId("63aabcc2be806e3b8eec585b"), name: 'Scott Tolib' },

{ \_id: ObjectId("63aabcc2be806e3b8eec585c"), name: 'Freddy Melver' },

{ \_id: ObjectId("63aabcc2be806e3b8eec585d"), name: 'Alexis Bohed' },

{ \_id: ObjectId("63aabcc2be806e3b8eec585e"), name: 'Melanie Palace' },

{ \_id: ObjectId("63aabcc2be806e3b8eec585f"), name: 'Armin Glutch' },

{ \_id: ObjectId("63aabcc2be806e3b8eec5860"), name: 'Klaus Arber' },

{\_id: ObjectId("63aabcc2be806e3b8eec5861"), name: 'Albert Twostone'}

]

If you use projection like this normally, only what you add here, so name in this case is included but the id is also always included, you have to explicitly exclude it if you don't want to add it. Now you exclude something by simply specifying its name and then using 0 instead of 1, we could do the same for age but this is not required because the default for everything but the id is that it's not included if you don't add it in your projection configuration but for the ID, we have to be very clear about it. And now with that if I hit enter, we get only the objects with the names. So, this is projection doing its job and the important thing to understand here is that this filtering or this data transformation is happening on the mongodb server, so this happens before the data is shipped to you and this is of course what you want because then you don't get unnecessary data, you don't impact your bandwidth and therefore you can only work with what you need in this case.

**>>> db.passengers.find({}, {name : 1, \_id: 0}).pretty()**

[

{ name: 'Max Schwarzmueller' },

{ name: 'Manu Lorenz' },

{ name: 'Chris Hayton' },

{ name: 'Sandeep Kumar' },

{ name: 'Maria Jones' },

{ name: 'Alexandra Maier' },

{ name: 'Dr. Phil Evans' },

{ name: 'Sandra Brugge' },

{ name: 'Elisabeth Mayr' },

{ name: 'Frank Cube' },

{ name: 'Karandeep Alun' },

{ name: 'Michaela Drayer' },

{ name: 'Bernd Hoftstadt' },

{ name: 'Scott Tolib' },

{ name: 'Freddy Melver' },

{ name: 'Alexis Bohed' },

{ name: 'Melanie Palace' },

{ name: 'Armin Glutch' },

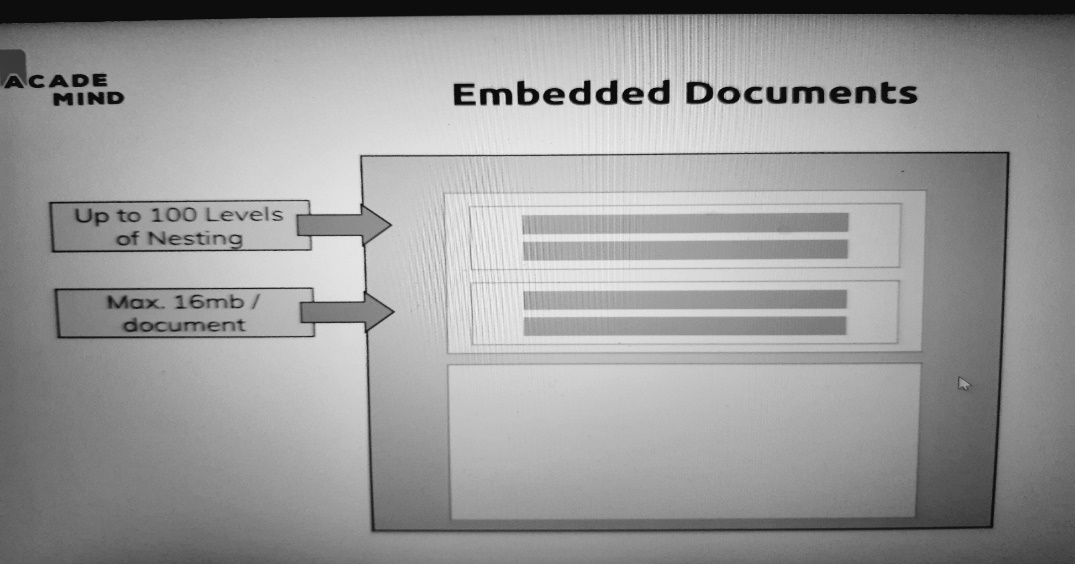
{ name: 'Klaus Arber' },

{ name: 'Albert Twostone' }

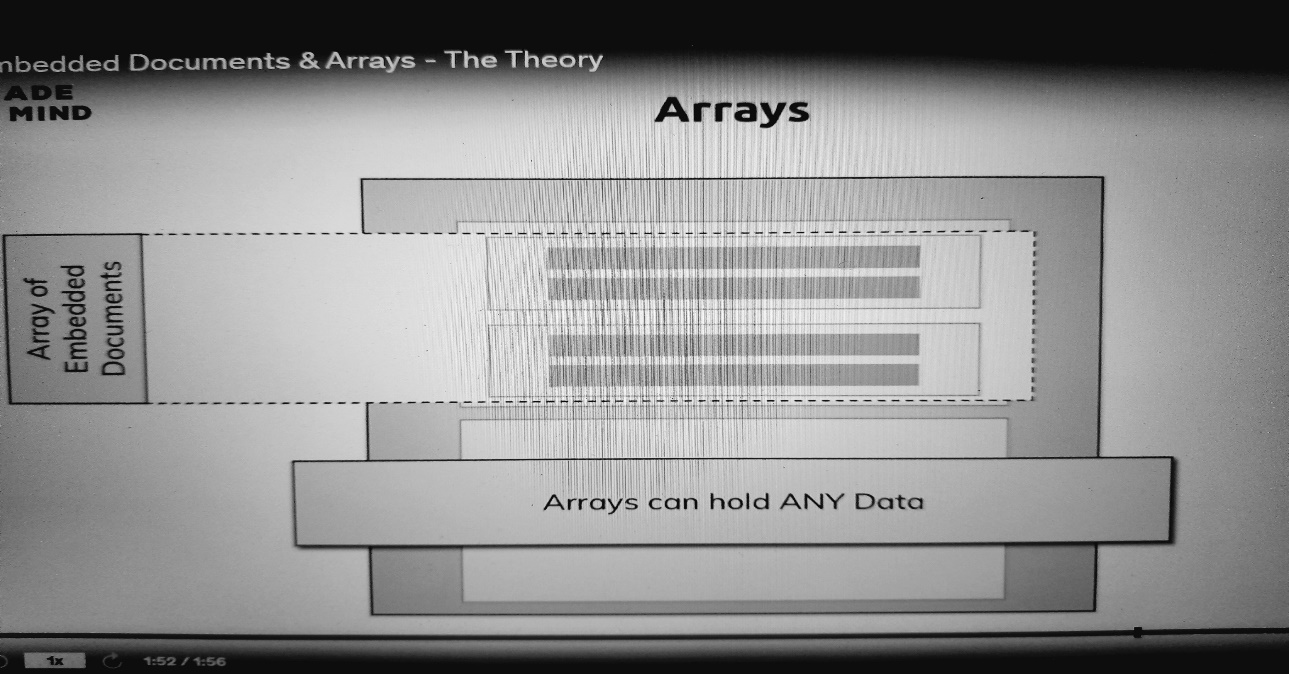
]

Type "it" for more

**Embedded Documents & Arrays**



We're almost done with this module. Now let me show you two other important pieces of data you can put into your documents. You can use embedded documents, and this is actually a core feature of mongodb and a core feature I'll come back to in my data schema and relations module in this course. Let's imagine this box is one document, embedded documents then simply means that you can have a field in your document like name but then this could be another document, so the value of the field could be another document and you can have multiple such documents and these documents can have other sub documents which can have other sub documents, so you can nest your documents all in one overarching document in one collection. So here we can have up to 100 levels of nesting in mongodb, this is a hard limit but you can have a lot of nesting and to be honest, you rarely need more than three or four levels and you always got another hard limit, the overall documents size has to be below 60 megabytes. That might sound small but since you only store text in there, you don't store files in mongodb for example, you would use your file storage for that, since it's only text, 16 megabytes is actually quite a lot.



another kind of data you can store are arrays and this is not strictly linked to embedded documents, though it can also be documents, you can have arrays in this case of embedded documents, but arrays can hold any data, this simply means that you have lists of data in a document.

**>>> db.flightData.find().pretty()**

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),

departureAirport: 'MUC',

arrivalAirport: 'SFO',

aircraft: 'Airbus A380',

distance: 12000,

intercontinental: true,

delayed: false,

flightName: 'Indigo'

},

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false,

flightName: 'Indigo'

}

]

Let's go back to our flight data, let's quickly find all the flights here, this is our flight data and now let's say we want to store the status for a flight. For this, I'll use my flight data and I'll update many elements, I'll pass an empty document as a filter to target them all and then here, we have $set and then how we want to change them and now let's say we want to add a status field but that status is not something like on time which you could use but let's say this is another document. So here another pair of curly braces, so this is now a value we set for status which we add to our flight data. And now here you can enter anything which you can have in a document, multiple key value pairs, even other nested documents if you want. So here we could have a description field which is on time and maybe last updated which is one hour ago, if we now hit enter here, it updated both documents because we had no filter and if I now output my flight data in a pretty way, we see these nested documents here and this is what's called embedded document, I use nested and embedded interchangeably here. So this is an embedded document, it's another json document within a wrapping json document. We got two documents in the collection and each of these documents has a child document and as I said, inside of the status, you could have another nested document.

**>>> db.flightData.updateMany({}, {$set: {status: {description: "on-time", lastUpdated: "1 hour ago"}}})**

{

acknowledged: true,

insertedId: null,

matchedCount: 2,

modifiedCount: 2,

upsertedCount: 0

}

**>>> db.flightData.find().pretty()**

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),

departureAirport: 'MUC',

arrivalAirport: 'SFO',

aircraft: 'Airbus A380',

distance: 12000,

intercontinental: true,

delayed: false,

flightName: 'Indigo',

status: { description: 'on-time', lastUpdated: '1 hour ago' }

},

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false,

flightName: 'Indigo',

status: { description: 'on-time', lastUpdated: '1 hour ago' }

}

]

So if I would repeat my command here and add another value with maybe details, that could be another nested document, so another pair of curly braces, maybe this has a responsible field and there, I put my name in there. If I now hit enter and I output all the flights again, now we see another nested document in there, so a nested document or an embedded document in an embedded document in a collection document. So these are embedded documents

**>>> db.flightData.updateMany({}, {$set: {status: {description: "on-time", lastUpdated: "1 hour ago", details: {responsible:"kaushal-singh"}}}})**

{

acknowledged: true,

insertedId: null,

matchedCount: 2,

modifiedCount: 2,

upsertedCount: 0

}

**flights> db.flightData.find().pretty()**

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),

departureAirport: 'MUC',

arrivalAirport: 'SFO',

aircraft: 'Airbus A380',

distance: 12000,

intercontinental: true,

delayed: false,

flightName: 'Indigo',

status: {

description: 'on-time',

lastUpdated: '1 hour ago',

details: { responsible: 'kaushal-singh' }

}

},

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false,

flightName: 'Indigo',

status: {

description: 'on-time',

lastUpdated: '1 hour ago',

details: { responsible: 'kaushal-singh' }

}

}

]

So, we saw embedded documents, let's now have a look at arrays of data and for that, let's go back to our passengers. So, remember we got our passengers collection, if we find all entries there, this is how a passenger looked like.

Now let's say we also record the hobbies of our passengers, for this I'll edit Albert Twostone here with updateOne, so by selecting the passenger with the name Albert Twostone, like this and I'll update it by setting a new field there, hobbies which is an array marked with square brackets, so this is important, now it's an array. And here we can have any kinds of data there, we could have multiple documents in here, that would be fine but we can all have multiple numbers or simply strings like sports and cooking, these are just strings not documents. So, any kind of data you can have in a document can also be in an array in a document. If I hit enter and I output my passengers, you see Albert now has hobbies, he's the only passenger with that because we filtered for him and hobbies now a list of data. It's not a nested or embedded document, it's a list of strings in this case and as I said, this could be numbers, this could also be other documents, so you can also have documents in an array and you can have arrays in documents, this is possible.

**flights> db.passengers.find()**

[

{

\_id: ObjectId("63aabcc2be806e3b8eec584e"),

name: 'Max Schwarzmueller',

age: 29

},

{

\_id: ObjectId("63aabcc2be806e3b8eec584f"),

name: 'Manu Lorenz',

age: 30

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5850"),

name: 'Chris Hayton',

age: 35

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5851"),

name: 'Sandeep Kumar',

age: 28

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5852"),

name: 'Maria Jones',

age: 30

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5853"),

name: 'Alexandra Maier',

age: 27

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5854"),

name: 'Dr. Phil Evans',

age: 47

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5855"),

name: 'Sandra Brugge',

age: 33

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5856"),

name: 'Elisabeth Mayr',

age: 29

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5857"),

name: 'Frank Cube',

age: 41

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5858"),

name: 'Karandeep Alun',

age: 48

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5859"),

name: 'Michaela Drayer',

age: 39

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585a"),

name: 'Bernd Hoftstadt',

age: 22

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585b"),

name: 'Scott Tolib',

age: 44

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585c"),

name: 'Freddy Melver',

age: 41

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585d"),

name: 'Alexis Bohed',

age: 35

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585e"),

name: 'Melanie Palace',

age: 27

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585f"),

name: 'Armin Glutch',

age: 35

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5860"),

name: 'Klaus Arber',

age: 53

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5861"),

name: 'Albert Twostone',

age: 68

}

]

Type "it" for more

**flights> db.passengers.updateOne({name: 'Albert Twostone'},{$set: {hobbies: ["Coding","Helping"]}})**

{

acknowledged: true,

insertedId: null,

matchedCount: 1,

modifiedCount: 1,

upsertedCount: 0

}

**flights> db.passengers.find().pretty()**

[

{

\_id: ObjectId("63aabcc2be806e3b8eec584e"),

name: 'Max Schwarzmueller',

age: 29

},

{

\_id: ObjectId("63aabcc2be806e3b8eec584f"),

name: 'Manu Lorenz',

age: 30

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5850"),

name: 'Chris Hayton',

age: 35

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5851"),

name: 'Sandeep Kumar',

age: 28

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5852"),

name: 'Maria Jones',

age: 30

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5853"),

name: 'Alexandra Maier',

age: 27

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5854"),

name: 'Dr. Phil Evans',

age: 47

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5855"),

name: 'Sandra Brugge',

age: 33

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5856"),

name: 'Elisabeth Mayr',

age: 29

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5857"),

name: 'Frank Cube',

age: 41

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5858"),

name: 'Karandeep Alun',

age: 48

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5859"),

name: 'Michaela Drayer',

age: 39

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585a"),

name: 'Bernd Hoftstadt',

age: 22

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585b"),

name: 'Scott Tolib',

age: 44

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585c"),

name: 'Freddy Melver',

age: 41

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585d"),

name: 'Alexis Bohed',

age: 35

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585e"),

name: 'Melanie Palace',

age: 27

},

{

\_id: ObjectId("63aabcc2be806e3b8eec585f"),

name: 'Armin Glutch',

age: 35

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5860"),

name: 'Klaus Arber',

age: 53

},

{

\_id: ObjectId("63aabcc2be806e3b8eec5861"),

name: 'Albert Twostone',

age: 68,

hobbies: [ 'Coding', 'Helping' ]

}

]

You can even have arrays in nested documents, this is all possible. information alert Schedule learning time Learning a little each day adds up. Research shows that students who make learning a habit are more likely to reach their goals. Set time aside to learn and get reminders using your learning scheduler.

**Accessing Structured Data:**

One thing I want to do before I close or finish this module is I want to show you how you could access such structured data and let's start with the hobbies here We can now access .hobbies here because find will give us that person and hobbies should give us access to the array here then, though one important thing, we should use findOne here so that mongodb knows we'll only have one element, so that we can directly access the fields of that element after fetching it, If we want to find all passengers with a hobby of sports, what we can do is we can access hobbies here which is an array and look for sports and I'd put that in a pretty way and here we find Albert too, the entire document not just the sports entry because we're looking for the entire passenger documents where hobbies is equal to sports and mongodb is clever enough to see that hobbies actually is an array so it will simply look if that array has one element named sports and then it gives us the entire document as a result.

**test> use flights**

switched to db flights

**flights> db.passengers.find().pretty()**

**flights> db.passengers.find({name: "Albert Twostone"}).pretty()**

[

{

\_id: ObjectId("63aabcc2be806e3b8eec5861"),

name: 'Albert Twostone',

age: 68,

hobbies: [ 'Coding', 'Helping' ]

}

]

**flights> db.passengers.findOne({name: "Albert Twostone"}).hobbies**

[ 'Coding', 'Helping' ]

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

[

{

\_id: ObjectId("63aabcc2be806e3b8eec5861"),

name: 'Albert Twostone',

age: 68,

hobbies: [ 'Coding', 'Helping' ]

}

]

**flights> db.passengers.findOne({hobbies: "Coding"}).pretty()**

TypeError: db.passengers ... ing"}).pretty is not a function

**flights> db.passengers.findOne({hobbies: "Coding"})**

{

\_id: ObjectId("63aabcc2be806e3b8eec5861"),

name: 'Albert Twostone',

age: 68,

hobbies: [ 'Coding', 'Helping' ]

}

Now how can we query objects?

Remember we had nested objects, nested documents in our flight data

**flights> db.flightData.find().pretty()**

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),

departureAirport: 'MUC',

arrivalAirport: 'SFO',

aircraft: 'Airbus A380',

distance: 12000,

intercontinental: true,

delayed: false,

flightName: 'Indigo',

status: {

description: 'on-time',

lastUpdated: '1 hour ago',

details: { responsible: 'kaushal-singh' }

}

},

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false,

flightName: 'Indigo',

status: {

description: 'on-time',

lastUpdated: '1 hour ago',

details: { responsible: 'kaushal-singh' }

}

}

]

So, you use dot notation to drill into your embedded document. so, description. Important when using that, so when having a dot in here, when using such a path to a nested field, you need to wrap the entire term with double quotation marks though otherwise it will fail and now we can check where is this equal to on time.

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

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),

departureAirport: 'MUC',

arrivalAirport: 'SFO',

aircraft: 'Airbus A380',

distance: 12000,

intercontinental: true,

delayed: false,

flightName: 'Indigo',

status: {

description: 'on-time',

lastUpdated: '1 hour ago',

details: { responsible: 'kaushal-singh' }

}

},

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false,

flightName: 'Indigo',

status: {

description: 'on-time',

lastUpdated: '1 hour ago',

details: { responsible: 'kaushal-singh' }

}

}

]

**flights> db.flightData.find({"status.details.responsible": "kaushal-singh"}).pretty()**

[

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1da"),

departureAirport: 'MUC',

arrivalAirport: 'SFO',

aircraft: 'Airbus A380',

distance: 12000,

intercontinental: true,

delayed: false,

flightName: 'Indigo',

status: {

description: 'on-time',

lastUpdated: '1 hour ago',

details: { responsible: 'kaushal-singh' }

}

},

{

\_id: ObjectId("63a9d09be5e6e4fb6c8bc1db"),

departureAirport: 'LHR',

arrivalAirport: 'TXL',

aircraft: 'Airbus A320',

distance: 950,

intercontinental: false,

flightName: 'Indigo',

status: {

description: 'on-time',

lastUpdated: '1 hour ago',

details: { responsible: 'kaushal-singh' }

}

}

]

**[======================== A S S I G N M E N T =======================]**

**# Patient Data**

1. **I need to insert 3 patients records with at least one history entry for our hospital db.**

**hospitals> db.patients.insertMany([{firstName: "xyz", lastName: "singh", age: 33, history: [{disease: "cold", treatment: 1}]}, {firstName: "qwe", lastName: "agrawal", age: 45, history:[{disease:"hungry", treatment: 56}]}, {firstName: "asd", lastName: "verma", age: 43, history: [{disease: "sad", treatment: 76}]}])**

**{**

**acknowledged: true,**

**insertedIds: {**

**'0': ObjectId("63abea92d149554fd9595c21"),**

**'1': ObjectId("63abea92d149554fd9595c22"),**

**'2': ObjectId("63abea92d149554fd9595c23")**

**}**

**}**

**hospitals> db.patients.find().pretty()**

**[**

**{**

**\_id: ObjectId("63abea92d149554fd9595c21"),**

**firstName: 'xyz',**

**lastName: 'singh',**

**age: 33,**

**history: [ { disease: 'cold', treatment: 1 } ]**

**},**

**{**

**\_id: ObjectId("63abea92d149554fd9595c22"),**

**firstName: 'qwe',**

**lastName: 'agrawal',**

**age: 45,**

**history: [ { disease: 'hungry', treatment: 56 } ]**

**},**

**{**

**\_id: ObjectId("63abea92d149554fd9595c23"),**

**firstName: 'asd',**

**lastName: 'verma',**

**age: 43,**

**history: [ { disease: 'sad', treatment: 76 } ]**

**}**

**]**

1. **Let’s update the data of one patient with the new name, age, and history entry.**

**hospitals> db.patients.updateOne({firstName: "asd"}, {$set: {lastName: "vermas", age: 44, history: [{disease: "very sad", treatment: 81}]}})**

{

acknowledged: true,

insertedId: null,

matchedCount: 1,

modifiedCount: 1,

upsertedCount: 0

}

**hospitals> db.patients.find().pretty()**

[

{

\_id: ObjectId("63abea92d149554fd9595c21"),

firstName: 'xyz',

lastName: 'singh',

age: 33,

history: [ { disease: 'cold', treatment: 1 } ]

},

{

\_id: ObjectId("63abea92d149554fd9595c22"),

firstName: 'qwe',

lastName: 'agrawal',

age: 45,

history: [ { disease: 'hungry', treatment: 56 } ]

},

{

\_id: ObjectId("63abea92d149554fd9595c23"),

firstName: 'asd',

lastName: 'vermas',

age: 44,

history: [ { disease: 'very sad', treatment: 81 } ]

}

]

1. **Find All patients who is older than 40.**

**hospitals> db.patients.find({age: {$gt: 40}}).pretty()**

[

{

\_id: ObjectId("63abea92d149554fd9595c22"),

firstName: 'qwe',

lastName: 'agrawal',

age: 45,

history: [ { disease: 'hungry', treatment: 56 } ]

},

{

\_id: ObjectId("63abea92d149554fd9595c23"),

firstName: 'asd',

lastName: 'vermas',

age: 44,

history: [ { disease: 'very sad', treatment: 81 } ]

}

]

1. **To delete all patients who got a cold as a disease.**

**hospitals> db.patients.deleteMany({"history.disease": "cold"})**

**{ acknowledged: true, deletedCount: 1 }**

**hospitals> db.patients.find().pretty()**

**[**

**{**

**\_id: ObjectId("63abea92d149554fd9595c22"),**

**firstName: 'qwe',**

**lastName: 'agrawal',**

**age: 45,**

**history: [ { disease: 'hungry', treatment: 56 } ]**

**},**

**{**

**\_id: ObjectId("63abea92d149554fd9595c23"),**

**firstName: 'asd',**

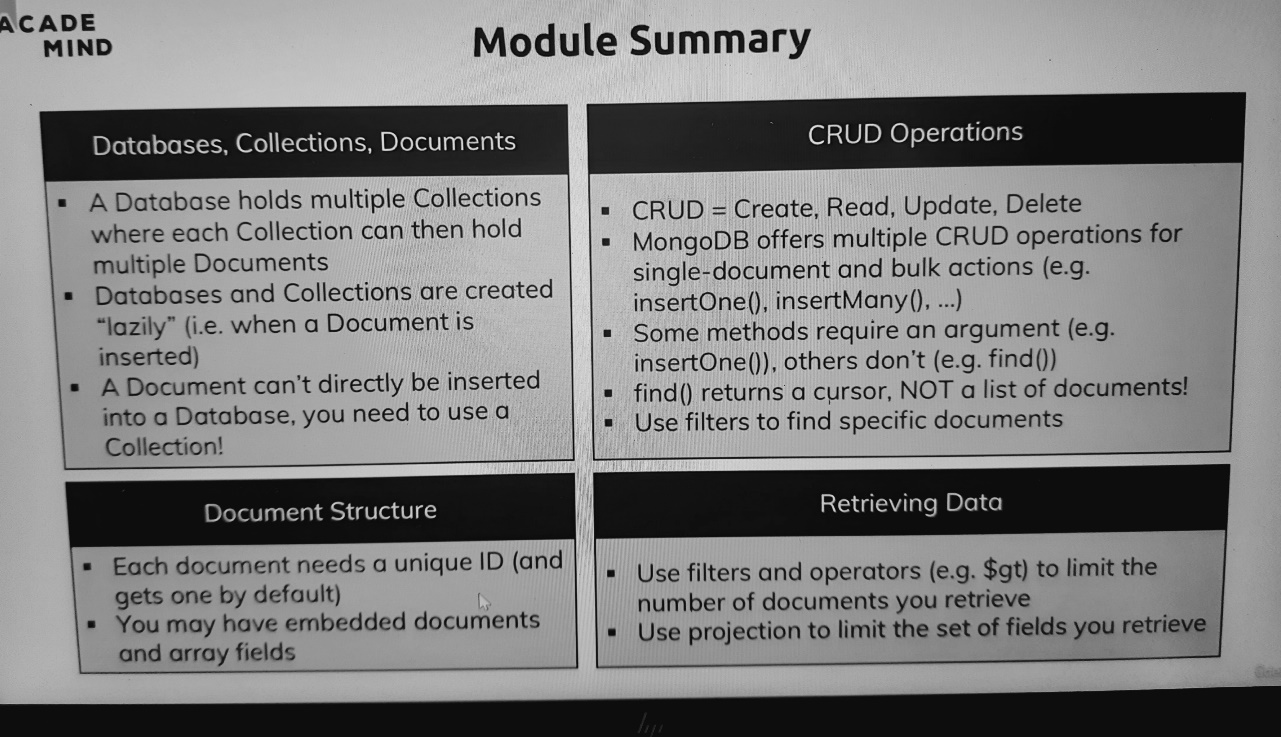
**lastName: 'vermas',**

**age: 44,**

**history: [ { disease: 'very sad', treatment: 81 } ]**

**}**

**]**



**Points to Remember: -**

1. **Databases, Collections, Documents**

* A Database holds multiple Collections where each Collection can then hold multiple Documents.
* Databases and Collections are created “lazily” (i.e when a Document is inserted)
* A Document can’t directly be inserted into a Database, you need to use a Collection!

1. **Document Structure**

* Each document needs a unique ID (and gets one by default)
* You may have embedded documents and array fields

1. **CRUD Operations**

* CRUD = Create, Read, Update, Delete
* MongoDB offers multiple CRUD operations for single-document and bulk actions (e.g. insertOne(), insertMany(), …)
* Some methods require an argument (e.g. insertOne()), others don’t (e.g. find())
* find()returns a cursor, NOT a list of documents!
* Use filters to find specific documents

1. **Retrieving Data**

* Use filters and operators(e.g. $gt) to limit the number of documents you retrieve
* Use projection to limit the set of fields you retrieve.

1. **Schemas & Relations: How to Structure Documents**

**============================================**

Important: We will regularly start with a clean database server (i.e. all data was purged) in this course. To get rid of your data, you can simply load the database you want to get rid of **(use databaseName**) and then execute **db.dropDatabase()**. Similarly, you could get rid of a single collection in a database via **db.myCollection.drop()**.

**>>> show dbs**



**>>> use flights**

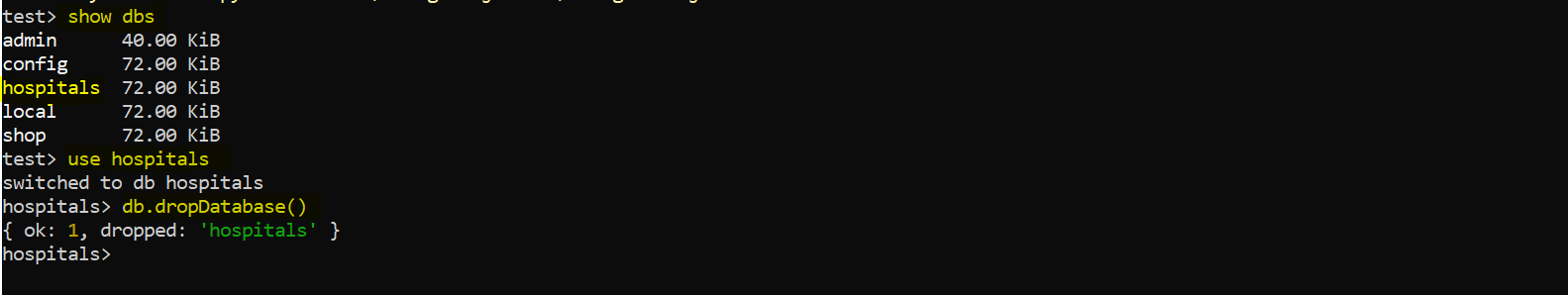
**>>> db.dropDatabase()**

**>> show dbs**



**>>> use hospitals**

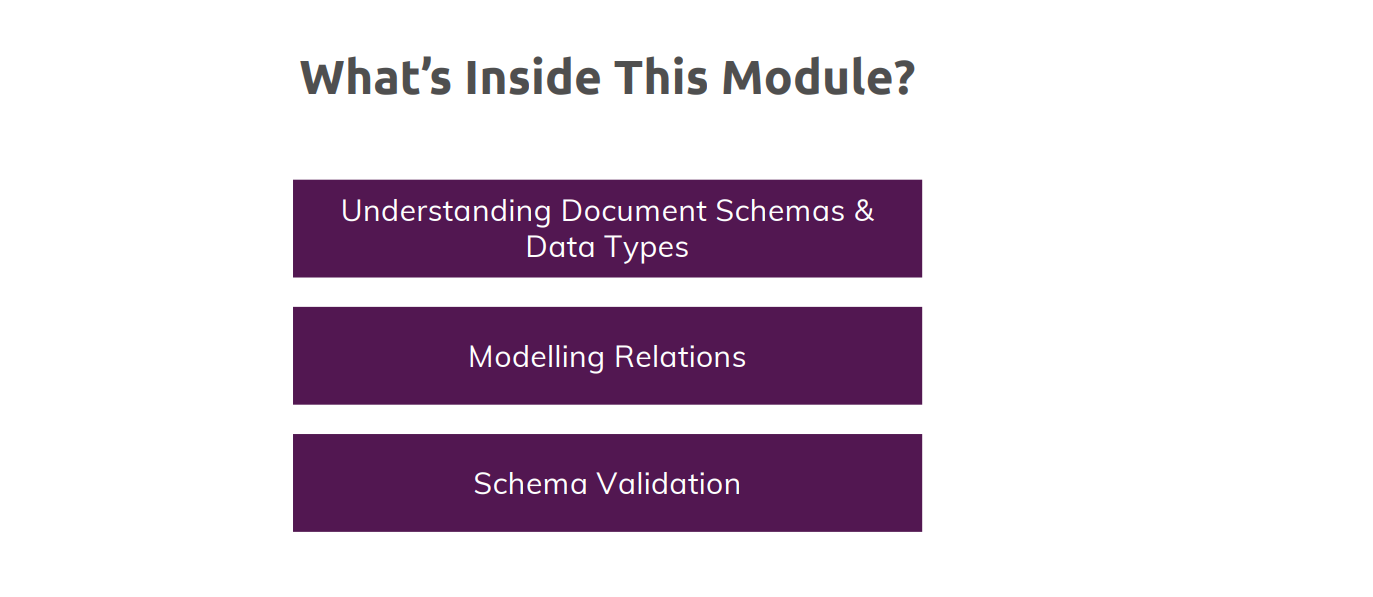
**>>> db.dropDatabase()**



**>>show dbs**



We cleaned up the Mongo Database Server.



**Why do we use Schemas ?**

**test> use shop**

**switched to db shop**

**shop> db.products.insertOne({name: "A book", price: 12.99})**

**{**

**acknowledged: true,**

**insertedId: ObjectId("63ac114616bac6ba91996672")**

**}**

**shop> db.products.insertOne({title: "T-Shirt", seller: {name: "Max", age: 29}})**

**{**

**acknowledged: true,**

**insertedId: ObjectId("63ac119716bac6ba91996673")**

**}**

**shop> db.products.find().pretty()**

**[**

**{**

**\_id: ObjectId("63ac114616bac6ba91996672"),**

**name: 'A book',**

**price: 12.99**

**},**

**{**

**\_id: ObjectId("63ac119716bac6ba91996673"),**

**title: 'T-Shirt',**

**seller: { name: 'Max', age: 29 }**

**}**

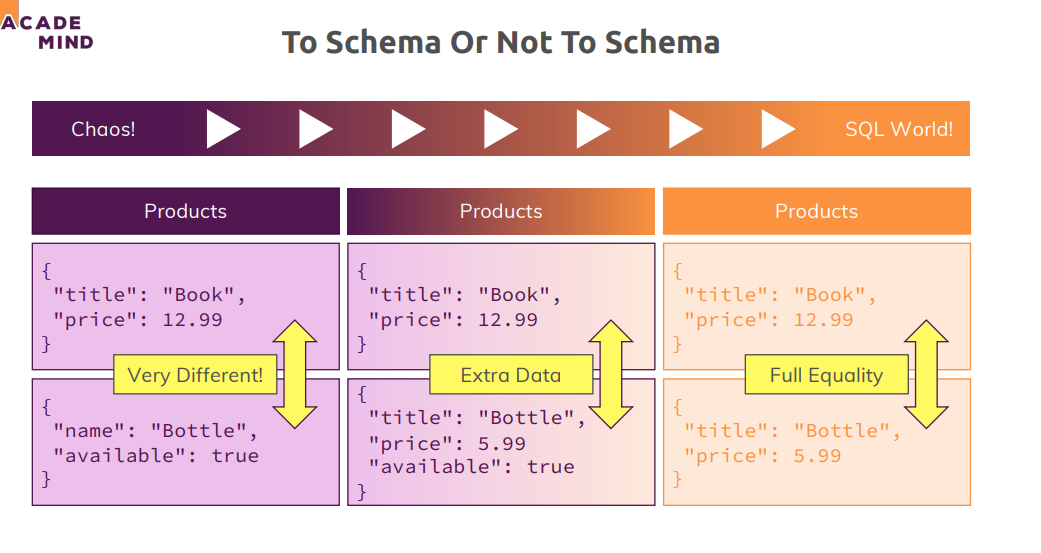
**]**

mongodb does not enforce any schemas, your documents can look the way you want, and you can have totally different documents in one and the same collection I will simply switch to a new database which as you learned will be created on the fly, on demand. So, I will switch to a new database, and I will simply name it shop because imagine we're building the backend for our online shop and we're now modeling the database for it. Now in that database, I'll probably have a product collection where I store my product data and there I will insert one product, a new document Now let's add another product and what we can do in mongodb is that this product looks totally different, so there I might have a title which is T-shirt and I might have a seller which is a nested document let's say, an embedded document with the name of Max and an age of 29. Now this is a totally different document.



you can mix totally different schemas and the schema simply means the structure of one document, so how does it look like, which fields does it have, which types of values do these fields have, that is what is a schema. So, we get totally different structures or schemas as you could call it in one and the same collection and that is fine. As I mentioned, mongodb enforces no schemas, you can mix them but in reality, you will probably have some kind of schema because if you are building the backend for your online shop, it's in your interest that all your products have a price you're not forced to have a schema, you will probably end up with some kind of structure and that is the important thing to understand. You've got the total freedom, total flexibility and if you need it, use it but in reality, you probably will have to think about how your data looks like because your data will need some kind of structure so that you could work with it efficiently.

**Structuring Documents**



we got maybe a products collection with totally different products in there, so that is possible in mongodb but it's probably not what you need or what you'll use in reality. Maybe you're somewhere in between, you got some kind of schema, for example here all my products have a title and a price but you got some extra information on some of your documents, so there is maybe some extra data on some documents but the general schema is the same and there probably are some core fields which exist on every document. Well or you got the other extreme where all documents have exactly the same structure, like here. Now anything is possible here, you can go with any of the three solutions or any step in-between depending on how you need it in your application but in reality, I guess either the approach on the right or most often the one in the middle is what you will see in reality

**let me clear all products first from our shop db**

**shop> db.products.deleteMany({})**

{ acknowledged: true, deletedCount: 2 }

**shop> db.products.find().pretty()**

B L A N K

let me again insert some products, insertOne and I want to show the two approaches, which are the most realistic ones. Let's go with the middle approach first, which might be the one you see most often.

**shop> db.products.insertOne({name: "A book", price: 12.99})**

{

acknowledged: true,

insertedId: ObjectId("63ac224616bac6ba91996674")

}

**shop> db.products.insertOne({name: "T - Shirt", price: 20.99})**

{

acknowledged: true,

insertedId: ObjectId("63ac226316bac6ba91996675")

}

**shop> db.products.find().pretty()**

[

{

\_id: ObjectId("63ac224616bac6ba91996674"),

name: 'A book',

price: 12.99

},

{

\_id: ObjectId("63ac226316bac6ba91996675"),

name: 'T - Shirt',

price: 20.99

}

]

**shop> db.products.insertOne({name: "T - Shirt", price: 12999, details: {cpu: "Intel Core I7"}})**

{

acknowledged: true,

insertedId: ObjectId("63ac234816bac6ba91996676")

}

shop> db.products.find().pretty()

[

{

\_id: ObjectId("63ac224616bac6ba91996674"),

name: 'A book',

price: 12.99

},

{

\_id: ObjectId("63ac226316bac6ba91996675"),

name: 'T - Shirt',

price: 20.99

},

{

\_id: ObjectId("63ac234816bac6ba91996676"),

name: 'T - Shirt',

price: 12999,

details: { cpu: 'Intel Core I7' }

}

]

**shop> db.products.deleteMany({})**

{ acknowledged: true, deletedCount: 3 }

**shop> db.products.insertOne({name: "A book", price: 12.99, details: null})**

{

acknowledged: true,

insertedId: ObjectId("63ac247216bac6ba91996677")

}

**shop> db.products.insertOne({name: "T - Shirt", price: 20.99, details: null})**

{

acknowledged: true,

insertedId: ObjectId("63ac248516bac6ba91996678")

}

**shop> db.products.insertOne({name: "Computers", price: 12999, details: {cpu: "Intel Core I7"}})**

{

acknowledged: true,

insertedId: ObjectId("63ac249416bac6ba91996679")

}

**shop> db.products.find().pretty()**

[

{

\_id: ObjectId("63ac247216bac6ba91996677"),

name: 'A book',

price: 12.99,

details: null

},

{

\_id: ObjectId("63ac248516bac6ba91996678"),

name: 'T - Shirt',

price: 20.99,

details: null

},

{

\_id: ObjectId("63ac249416bac6ba91996679"),

name: 'Computers',

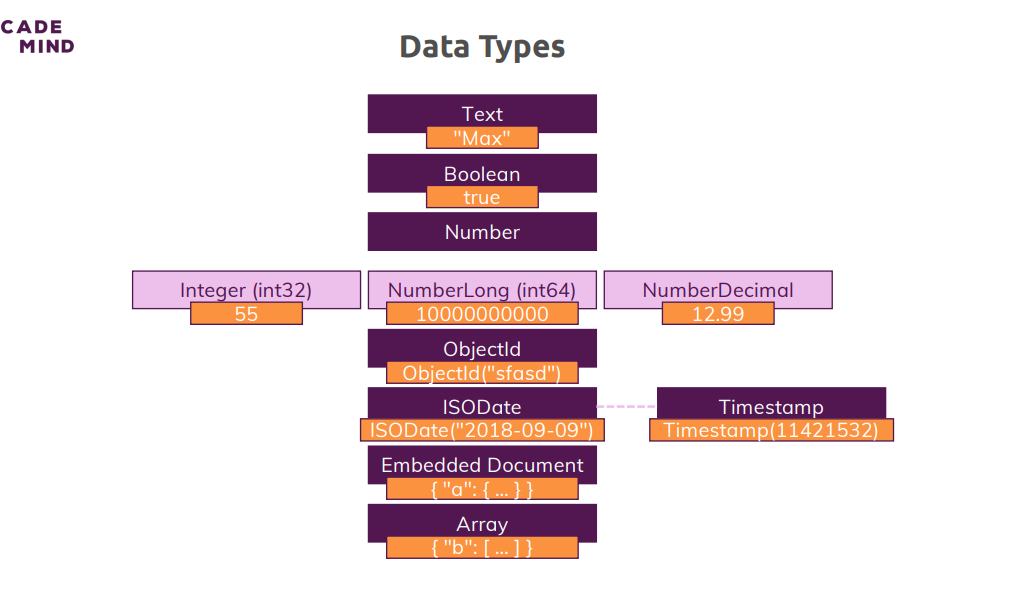
price: 12999,

details: { cpu: 'Intel Core I7' }

}

]

**Data Types: Overview**



I'm on my MongoDB server again and I dropped all existing databases db.dropdatabase and this will get rid of the database if you have it javascript doesn't differentiate between integers and floating point numbers

**>>> use companyData**

**companyData> db.companies.insertOne({name: "Anspro Tech Inc", isStartup: true, employees: 6, funding: 1234567890123456789, details: {ceo: "Kaushal Singh"}, tags: [{title: "super"}, {title: "perfect"}], foundingDate: new Date(), insertedAt: new Timestamp()})**

{

acknowledged: true,

insertedId: ObjectId("63ac3b5c7baeda143782baa6")

}

**companyData> db.companies.find().pretty()**

[

{

\_id: ObjectId("63ac3b5c7baeda143782baa6"),

name: 'Anspro Tech Inc',

isStartup: true,

employees: 6,

funding: 1234567890123456800,

details: { ceo: 'Kaushal Singh' },

tags: [ { title: 'super' }, { title: 'perfect' } ],

foundingDate: ISODate("2022-12-28T12:49:32.691Z"),

insertedAt: Timestamp({ t: 1672231772, i: 1 })

}

]

**companyData> db.numbers.insertOne({a: 1})**

{

acknowledged: true,

insertedId: ObjectId("63ac3d9e7baeda143782baa7")

}

**companyData> db.numbers.findOne()**

{ \_id: ObjectId("63ac3d9e7baeda143782baa7"), a: 1 }

**companyData> db.stats()**

{

db: 'companyData',

collections: 2,

views: 0,

objects: 2,

avgObjSize: 131.5,

dataSize: 263,

storageSize: 40960,

indexes: 2,

indexSize: 40960,

totalSize: 81920,

scaleFactor: 1,

fsUsedSize: 108171563008,

fsTotalSize: 255073447936,

ok: 1

}

**companyData> db.companies.drop()**

true

**companyData> db.stats()**

{

db: 'companyData',

collections: 1,

views: 0,

objects: 1,

avgObjSize: 29,

dataSize: 29,

storageSize: 20480,

indexes: 1,

indexSize: 20480,

totalSize: 40960,

scaleFactor: 1,

fsUsedSize: 108174311424,

fsTotalSize: 255073447936,

ok: 1

}

**companyData> db.numbers.deleteMany({})**

{ acknowledged: true, deletedCount: 1 }

**companyData> db.stats()**

{

db: 'companyData',

collections: 1,

views: 0,

objects: 0,

avgObjSize: 0,

dataSize: 0,

storageSize: 20480,

indexes: 1,

indexSize: 20480,

totalSize: 40960,

scaleFactor: 1,

fsUsedSize: 108176187392,

fsTotalSize: 255073447936,

ok: 1

}

**Data Types & Limits**

MongoDB has a couple of hard limits - most importantly, a single document in a collection (including all embedded documents it might have) must be <= **16mb**. Additionally, you may only have **100 levels of embedded documents**.

You can find all limits (in great detail) here: <https://docs.mongodb.com/manual/reference/limits/>

For the data types, MongoDB supports, you find a **detailed overview** on this page: <https://docs.mongodb.com/manual/reference/bson-types/>

**Important data type limits are:**

* Normal integers (int32) can hold a maximum value of +-2,147,483,647
* Long integers (int64) can hold a maximum value of +-9,223,372,036,854,775,807
* Text can be as long as you want - the limit is the 16mb restriction for the overall document

It's also important to understand the difference between int32 (NumberInt), int64 (NumberLong) and a normal number as you can enter it in the shell. The same goes for a normal double and NumberDecimal.

**NumberInt** creates a **int32** value => NumberInt(55)

**NumberLong** creates a **int64** value => NumberLong(7489729384792)

If you just use a number (e.g. insertOne({a: 1}), this will get added as a **normal double** into the database. The reason for this is that the shell is based on JS which only knows float/ double values and doesn't differ between integers and floats.

**NumberDecimal** creates a high-precision double value => NumberDecimal("12.99") => This can be helpful for cases where you need (many) exact decimal places for calculations.

When not working with the shell but a MongoDB driver for your app programming language (e.g. PHP, .NET, Node.js, ...), you can use the driver to create these specific numbers.

Example for Node.js: <http://mongodb.github.io/node-mongodb-native/3.1/api/Long.html>

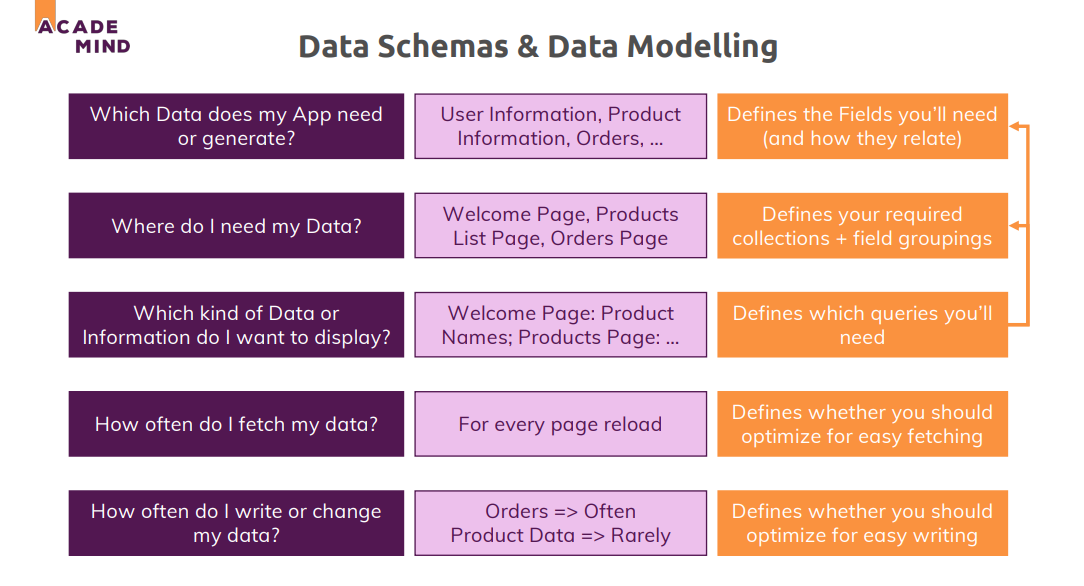
This will allow you to build a NumberLong value like this:

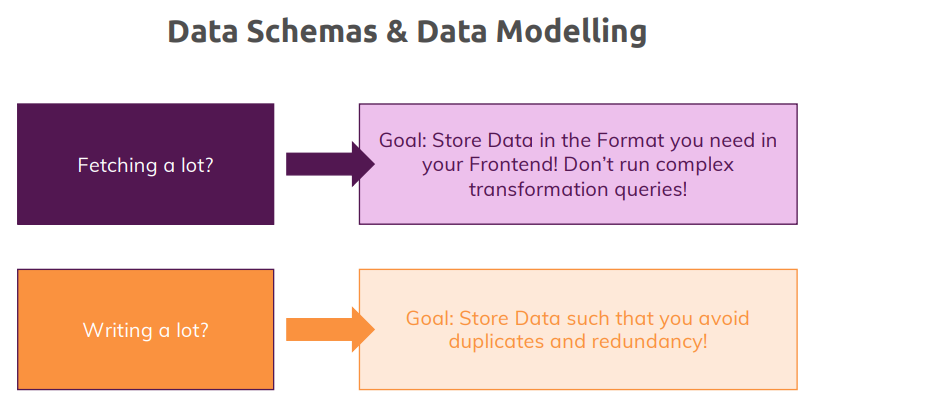
1. const Long = require('mongodb').Long;
3. db.collection('wealth').insert( {
4. value: Long.fromString("121949898291")
5. });

By browsing the API docs for the driver you're using, you'll be able to identify the methods for building int32s, int64s etc.

**Data Schemas & Modelling**

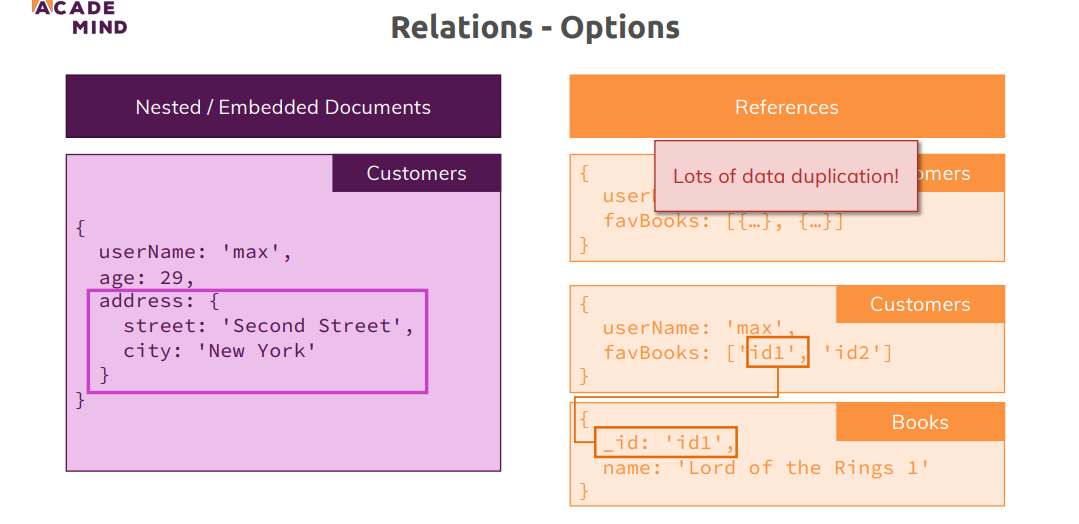
Now that we understand that we can use schema and that we probably will use them because our application typically works with certain types of data, certain schemas and that we had a look at the data types you can use, the different integers, text and so on let's see how we kind of think about modeling our schemas and I want to give you some guidelines or hints that you can keep in mind when you think about how should I structure my data.





**Understanding Relations:**

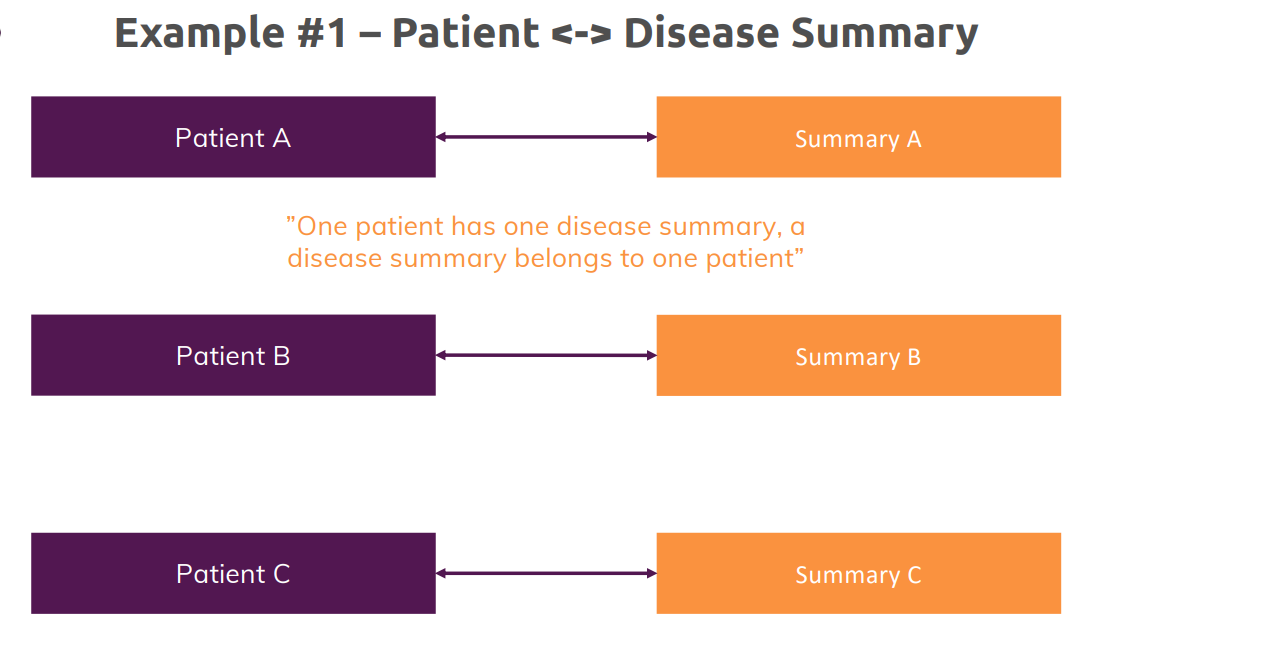
So, we had a look at the general data structure but always only for one collection and the documents in that the one collection. Typically, you have multiple collections though, you might have users and products and orders. So if you have multiple collections that still are related of course or where the documents in these relations are related, you obviously have to think about how do you store related data, do you use embedded documents because that of course is one way of reflecting a relation, if a user has an address and that address consists of a street and a city, then you might have a nested address document in the user document or do you work with references? And here, my customer has an address, and that address could totally be stored in a new collection but we can also embed it into our user or into our customer here because it's strongly related to that customers and we might want to fetch both at the same time or at least have an easy way of fetching both if we need to. The alternative is that you use references.



let's say we still have customers, and each customer has a list of his favorite books, and we could absolutely go with nested or embedded data. So here we have an array with embedded documents and each document is shortened here but each document could be a book with a title, a price, an author and so on. Now the problem here is that we will have a lot of duplicate data because different customers can have the same favorite books and if we then change something about that favorite book, we'll have to change it on all customers, so instead here, we might go for references. We might have a customer’s collection and the books collection and, in the customer, we still have that array of favorite books but there, we only store the IDs of the favorite books. That means that when we fetch the customers and we also want to fetch the favorite books for the customer, we have to run two queries, that is correct but on the other hand if we change a book, we don't have to change it on 10 different customers but only in the books collection and the ID will never change, so we never have to change or touch the favorite books array in the customers document, so we have that relation through a reference.

**One To One Relations- Embedded**

So, every patient has one disease summary which belongs to that patient only and there is only one summary per patient. So, we're not talking about the history we had earlier in this course, it's just a summary, something like is ill, had previous diseases and a list of previous diseases, things like that. So, one summary per patient and of course this is a one-to-one relation because the summary of patient A can never belong to patient B and the other way around, how could we model that



Back in a brand new cleaned up database server…

**companyData> db.dropDatabase()**



**>>> hospitals> use hospitals**

**>>> db.patients.insertOne({name: "Max", age: 29, diseaseSummary: "summary-max-1"})**

**>>> db.patients.findOne()**

{

\_id: ObjectId("63ac65e71e2d47753eb471a0"),

name: 'Max',

age: 29,

diseaseSummary: 'summary-max-1'

**}**

**>>> db.diseaseSummaries.insertOne({\_id: "summary-max-1", diseases:["cold", "broken leg"]})**

**hospitals> db.diseaseSummaries.findOne()**

{ \_id: 'summary-max-1', diseases: [ 'cold', 'broken leg' ] }

**hospitals> db.patients.findOne().diseaseSummary**

summary-max-1

**hospitals> var dsid = db.patients.findOne().diseaseSummary**

hospitals> dsid

summary-max-1

**hospitals> db.diseaseSummaries.findOne({\_id: dsid})**

{ \_id: 'summary-max-1', diseases: [ 'cold', 'broken leg' ] }

Now let's say a patient simply has a name Max, an age and then we have that diseaseSummary, now let's go for the reference approach first, diseaseSummary could be summary-max-1, any identifier you want, so here I'll go for that ID approach the bad thing here is that this took two steps and also from a performance perspective, this will actually be super fast but if you have a very big database, a very big collection of patients and a big collection of diseaseSummaries, this might not be the optimal way of doing that, splitting that across two collections feels a bit unnecessary here

the better approach in such a case where we have a strong one-to-one relation would be to use embedded document.

**hospitals> db.patients.insertOne({name: "Max", age: 29, diseaseSummary: {diseases: ["cold", "broken-leg"]}})**

{

acknowledged: true,

insertedId: ObjectId("63ac7f761e2d47753eb471a1")

}

**hospitals> db.patients.findOne()**

{

\_id: ObjectId("63ac7f761e2d47753eb471a1"),

name: 'Max',

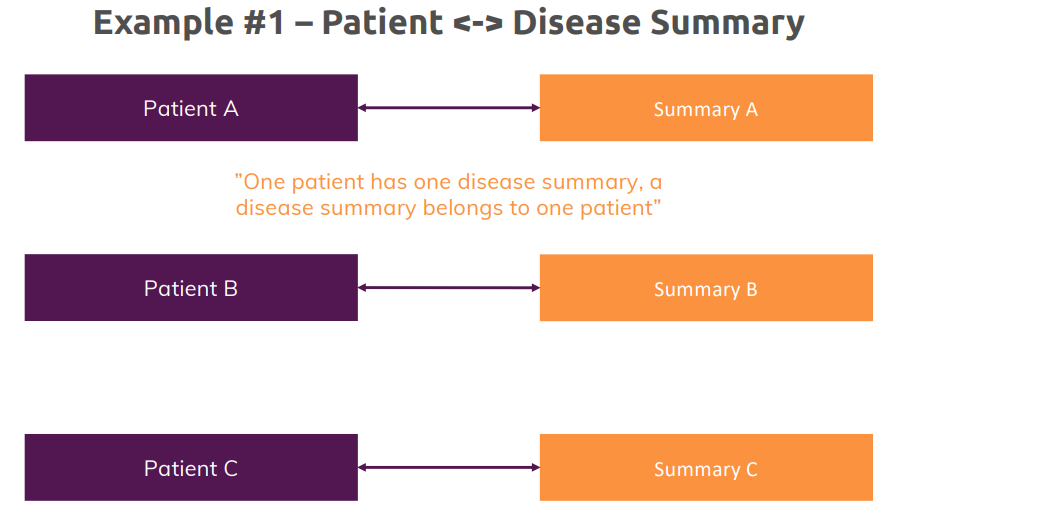
age: 29,

diseaseSummary: { diseases: [ 'cold', 'broken-leg' ] }

}

And now the huge advantage of course is that in the application we're writing with whichever language we use, I can simply write my query to find my patient by ID, by name or in this case as a shortcut, simply with findOne and I will get all the data I need, the patient data and the diseaseSummary data. So, this is an example for when you would use an embed document, you have a strong one-to-one relationship.

**One To One – Using References**



**cardData> db.persons.insertOne({name: "max", age: 29, salary: 6789})**

{

acknowledged: true,

insertedId: ObjectId("63ad2293c5de4581c33472f4")

}

**cardData> db.cars.insertOne({model: "BMW", price: 40000, owner: ObjectId("63ad2293c5de4581c33472f4")})**

{

acknowledged: true,

insertedId: ObjectId("63ad235cc5de4581c33472f5")

}

**cardData> db.cars.find().pretty()**

[

{

\_id: ObjectId("63ad235cc5de4581c33472f5"),

model: 'BMW',

price: 40000,

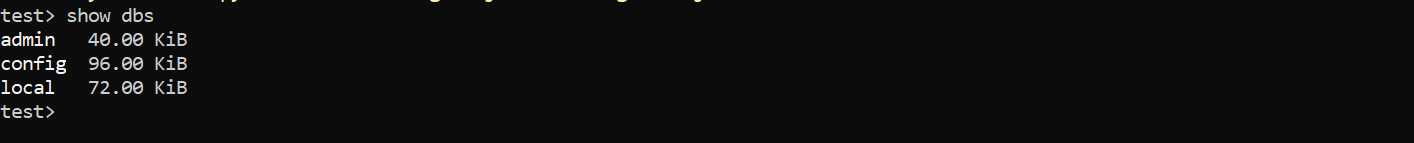
owner: ObjectId("63ad2293c5de4581c33472f4")

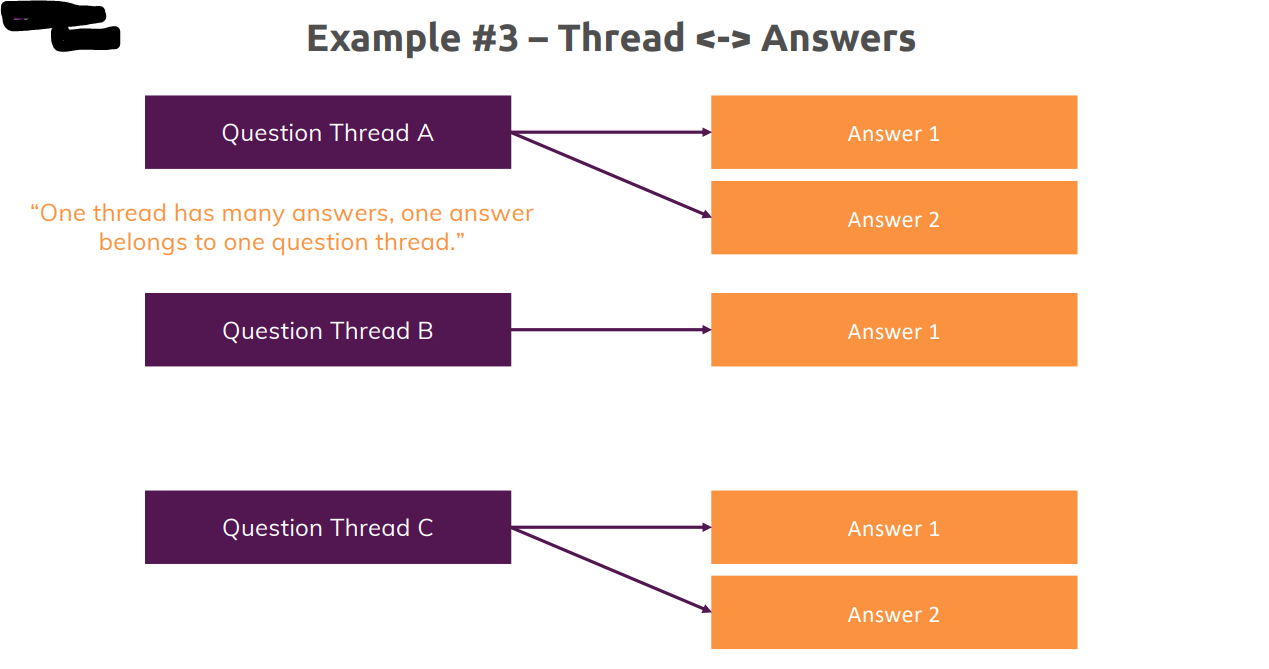
}

]

**One To Many – Embedded**

let's now have a look at one-to-many relations. Let's say like in the Q&A section of Udemy, we have question threads, and each question thread then has a couple of answers, so an answer only belongs to one thread, but a thread can have multiple answers, that's a typical one-to-many relationship. How would you model that in MongoDB?





**test> use support**

switched to db support

**support> db.questionThreads.insertOne({creator: "Max", question: "How does that all works?", answer: ["q1a1","q1a2"]})**

{

acknowledged: true,

insertedId: ObjectId("63ad2ab8e778fac55e9ba716")

}

**support> db.questionThreads.findOne()**

{

\_id: ObjectId("63ad2ab8e778fac55e9ba716"),

creator: 'Max',

question: 'How does that all works?',

answer: [ 'q1a1', 'q1a2' ]

}

**support> db.answers.insertMany([{\_id: "q1a1", text:"It works like that!"}, {\_id: "q1a2", text: "Thanks!"}])**

{ acknowledged: true, insertedIds: { '0': 'q1a1', '1': 'q1a2' } }

**support> db.answers.find()**

[

{ \_id: 'q1a1', text: 'It works like that!' },

{ \_id: 'q1a2', text: 'Thanks!' }

]

Here again embedding might be the better solution

**support> db.questionThreads.deleteMany({})**

{ acknowledged: true, deletedCount: 1 }

well then we see now we have that embedded approach, now a list of embedded documents because it's a one-to-many relationship, so we have many documents but still it's embedded documents which we're using here you need to fetch the questions along with the answers, so from application requirement perspective, you want to fetch the merged data and we also don't have thousands of answers on one thread typically, so we also don't have the danger of bloating our document and maybe reaching that 16mb limit we have for a document. So that is not a problem here and therefore this might be a perfect use case for embedding documents in a one-to-many relationship.

**support> db.questionThreads.insertOne({creator: "Max", question: "How does It works?", answers: [{test: "Like That."},{text: "Thanks!"}]})**

{

acknowledged: true,

insertedId: ObjectId("63ad2d69e778fac55e9ba717")

}

**support> db.questionThreads.find().pretty()**

[

{

\_id: ObjectId("63ad2d69e778fac55e9ba717"),

creator: 'Max',

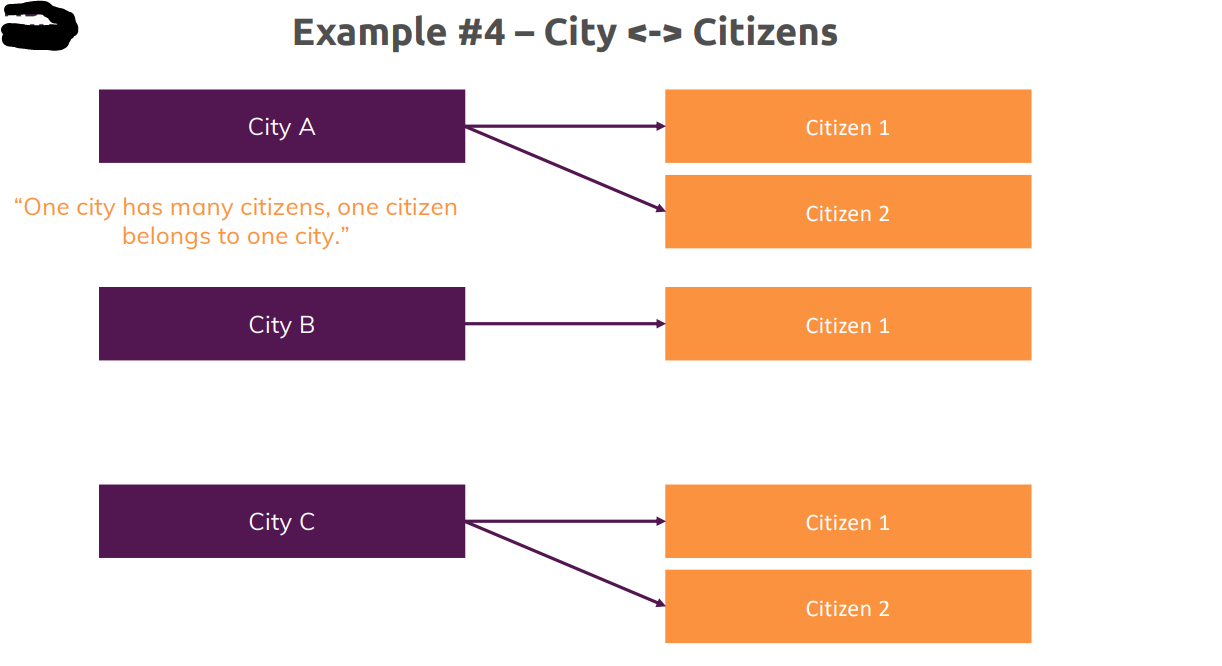
question: 'How does It works?',

answers: [ { test: 'Like That.' }, { text: 'Thanks!' } ]

}

]

**One To many – using References**



**test> use cityData**

**switched to db cityData**

**cityData> db.cities.insertOne({name: "New York City", coordinates: {lat:21, lng: 55}})**

**{**

**acknowledged: true,**

**insertedId: ObjectId("63ad32093a1583c14925f3c3")**

**}**

**cityData> db.cities.find()**

**[**

**{**

**\_id: ObjectId("63ad32093a1583c14925f3c3"),**

**name: 'New York City',**

**coordinates: { lat: 21, lng: 55 }**

**}**

**]**

**cityData> db.citizens.insertMany([{name: "Max Schwartz", cityId: ObjectId("63ad32093a1583c14925f3c3")}, {name: "Manuel Loren", cityId: ObjectId("63ad32093a1583c14925f3c3")}])**

**{**

**acknowledged: true,**

**insertedIds: {**

**'0': ObjectId("63ad32f23a1583c14925f3c4"),**

**'1': ObjectId("63ad32f23a1583c14925f3c5")**

**}**

**}**

**cityData> db.citizens.find()**

**[**

**{**

**\_id: ObjectId("63ad32f23a1583c14925f3c4"),**

**name: 'Max Schwartz',**

**cityId: ObjectId("63ad32093a1583c14925f3c3")**

**},**

**{**

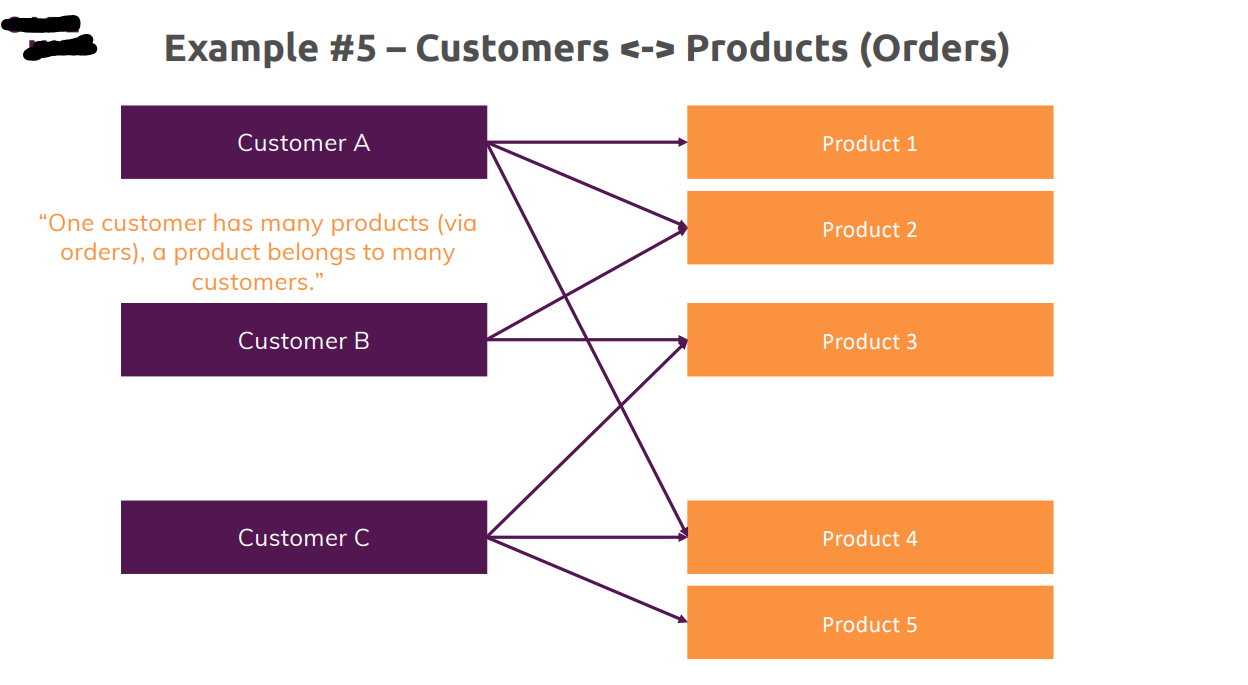
**\_id: ObjectId("63ad32f23a1583c14925f3c5"),**

**name: 'Manuel Loren',**

**cityId: ObjectId("63ad32093a1583c14925f3c3")**

**}**

**Many To many- Embedded**



**test> use shop**

switched to db shop

**shop> db.products.insertOne({title: "A book", price: 12.99})**

{

acknowledged: true,

insertedId: ObjectId("63ad35c172a1353512bcb0e3")

}

**shop> db.customers.insertOne({name: "Max", age: 29})**

{

acknowledged: true,

insertedId: ObjectId("63ad35e272a1353512bcb0e4")

}

**shop> db.orders.insertOne({productId: ObjectId("63ad35c172a1353512bcb0e3"), customerId: ObjectId("63ad35e272a1353512bcb0e4")})**

{

acknowledged: true,

insertedId: ObjectId("63ad366972a1353512bcb0e5")}

Now this would be the SQL world approach where we have three tables in SQL, and we have that join table in the middle which matches products and customers. Now we can do a bit better in quotation marks in mongodb, we can make it work with two tables only, so I can actually drop my orders collection, so db.orders can be dropped

**shop> db.orders.drop()**

true

**shop> db.orders.find()**

I still have my products by the way, so we still have the products and we still have the customers, so we have these two tables

**shop> db.products.find()**

[

{

\_id: ObjectId("63ad35c172a1353512bcb0e3"),

title: 'A book',

price: 12.99

}

]

**shop> db.customers.find()**

[ { \_id: ObjectId("63ad35e272a1353512bcb0e4"), name: 'Max', age: 29 } ]

I can now add orders without adding a third table and for that, I'll go to my customers

**shop> db.customers.updateOne({}, {$set: {orders: [{productId: ObjectId("63ad366972a1353512bcb0e5"), quantity: 2}]}})**

{

acknowledged: true,

insertedId: null,

matchedCount: 1,

modifiedCount: 1,

upsertedCount: 0

}

**shop> db.customers.findOne()**

{

\_id: ObjectId("63ad35e272a1353512bcb0e4"),

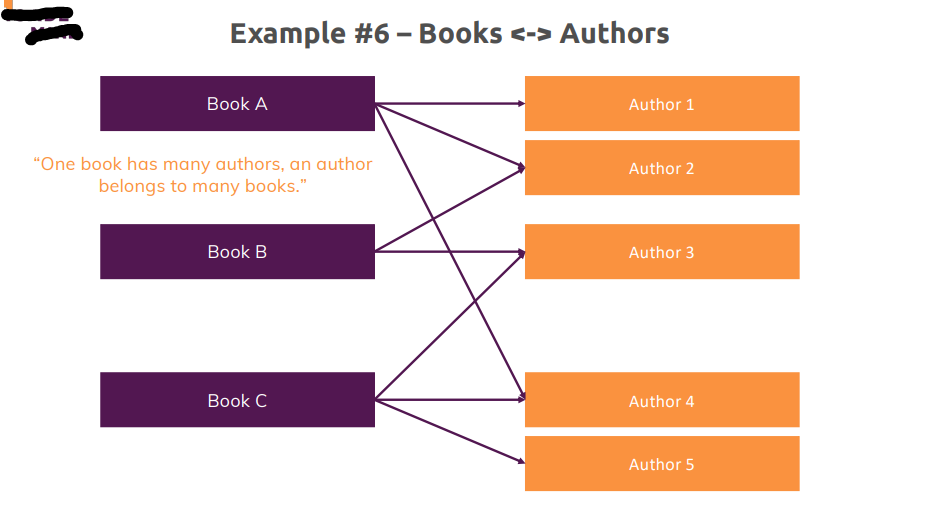
name: 'Max',

age: 29,

orders: [ { productId: ObjectId("63ad366972a1353512bcb0e5"), quantity: 2 } ]

}

**Many To many- Using References**





**Now let's have a look at a many-to-many relationship where splitting it up and using references might be better.**

Here we got a couple of books and a couple of authors and one book can be written by multiple authors

and one author will probably write more than one book,

so it's a typical many-to-many relationship

now let's see how we could model that and how we could store that in our database.

**test> use bookRegistry**

**switched to db bookRegistry**

**bookRegistry> db.books.insertOne({name: "My Favourite Book", authors: [{name: "Max Schwartz", age: 29}, {name: "Miller Toffy", age: 32}]})**

**{**

**acknowledged: true,**

**insertedId: ObjectId("63ad40e41b0196a2c5330ed9")**

**}**

**bookRegistry> db.books.find().pretty()**

**[**

**{**

**\_id: ObjectId("63ad40e41b0196a2c5330ed9"),**

**name: 'My Favourite Book',**

**authors: [**

**{ name: 'Max Schwartz', age: 29 },**

**{ name: 'Miller Toffy', age: 32 }**

**]**

**}**

**]**

**bookRegistry> db.authors.insertMany([{name: "Max Schwartz", age: 29, address: {street: "Main"}}, {name: "Miller", age: 30, address: {street: "Tree"}}])**

**{**

**acknowledged: true,**

**insertedIds: {**

**'0': ObjectId("63ad43af1b0196a2c5330eda"),**

**'1': ObjectId("63ad43af1b0196a2c5330edb")**

**}**

**}**

**bookRegistry> db.authors.find().pretty()**

**[**

**{**

**\_id: ObjectId("63ad43af1b0196a2c5330eda"),**

**name: 'Max Schwartz',**

**age: 29,**

**address: { street: 'Main' }**

**},**

**{**

**\_id: ObjectId("63ad43af1b0196a2c5330edb"),**

**name: 'Miller',**

**age: 30,**

**address: { street: 'Tree' }**

**}**

**]**

Now that would be a typical use case, we got our books with embedded author data and that could be all or some of the data we have in the authors collection and we get just the authors collection, the problem is if one author changes, maybe I got older, maybe I moved, then I need to update that everywhere where I use that and that's not just the authors collection but that would be all my books and that is especially important let's say if I changed my name because I married, then I can't say now I don't care it's an old snapshot like we did for the orders. For the books and for the authors, I want to have the most recent data of course and if I'm displaying the age in the book, I might say that is a snapshot data, that is ok, for the name it's probably not and in my registry in the database even the age is not ok, that should be all up-to-date for future books, for future prints, let's say. and therefore, this is probably not the best approach for this scenario because in our application, if the data does change, it should change everywhere

**bookRegistry> db.books.updateOne({}, {$set: {authors: [ObjectId("63ad43af1b0196a2c5330eda"), ObjectId("63ad43af1b0196a2c5330edb")]}})**

**{**

**acknowledged: true,**

**insertedId: null,**

**matchedCount: 1,**

**modifiedCount: 1,**

**upsertedCount: 0**

**}**

**bookRegistry> db.books.findOne()**

**{**

**\_id: ObjectId("63ad40e41b0196a2c5330ed9"),**

**name: 'My Favourite Book',**

**authors: [**

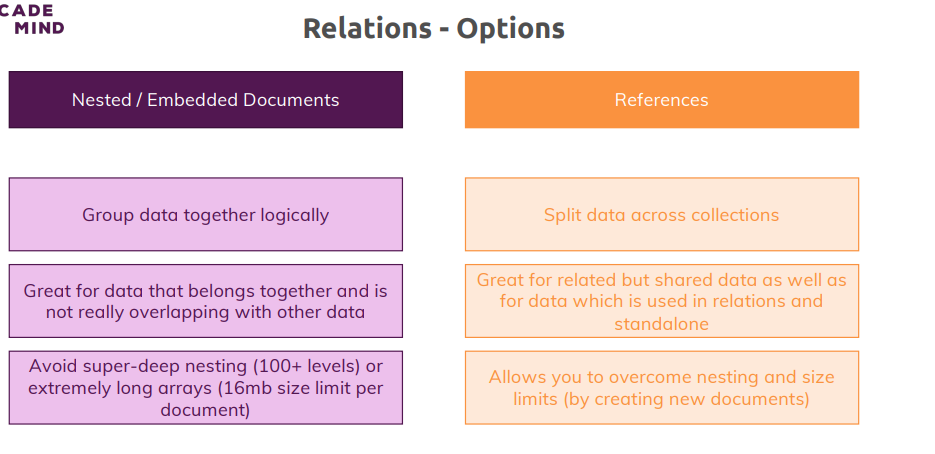
**ObjectId("63ad43af1b0196a2c5330eda"),**

**ObjectId("63ad43af1b0196a2c5330edb")**

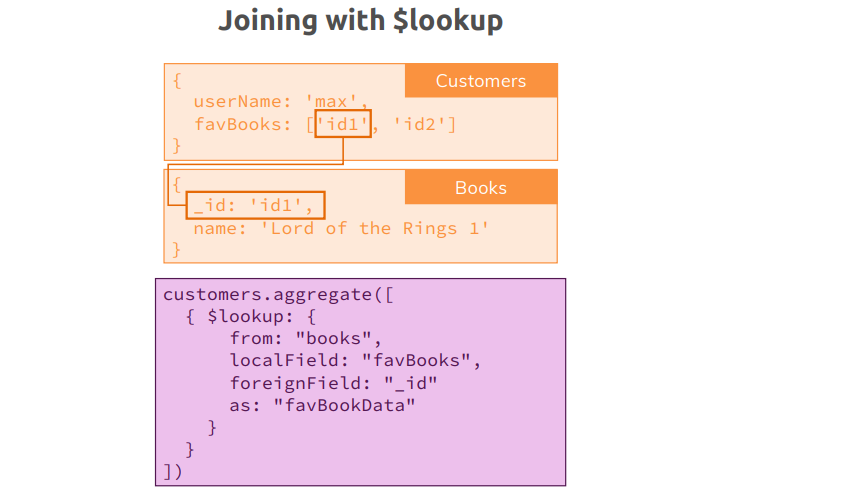
**]**

**}**

So instead of using embedded documents here, we should go for the reference approach, so we should take the references of our authors and then go to our books collection, update the book in our case here, the one book we got and update it by setting the authors to an array of our objectIds, so of the objectId of Max and the objectId of Manuel. And now if we output our books here with findOne, now we only got the references in there and this is better because now if we do fetch all the books, yeah we will have to merge it manually with the author data but therefore, the author data is guaranteed to be up-to-date and if we change the author data, we don't have to do that in hundreds or thousands of documents.



**Using “lookup()” for Merging Reference Relations**



lookup is essentially a helpful tool that allows you to fetch two related documents merged together in one document. in one step instead of having to do two steps and this mitigates some of the disadvantages of splitting your documents across collections because now you can at least merge them in one go.

**bookRegistry> db.books.aggregate([{$lookup: {from: "authors", localField: "authors", foreignField: "\_id", as: "creators"}}]).pretty()**

[

{

\_id: ObjectId("63ad40e41b0196a2c5330ed9"),

name: 'My Favourite Book',

authors: [

ObjectId("63ad43af1b0196a2c5330eda"),

ObjectId("63ad43af1b0196a2c5330edb")

],

creators: [

{

\_id: ObjectId("63ad43af1b0196a2c5330eda"),

name: 'Max Schwartz',

age: 29,

address: { street: 'Main' }

},

{

\_id: ObjectId("63ad43af1b0196a2c5330edb"),

name: 'Miller',

age: 30,

address: { street: 'Tree' }

}

]

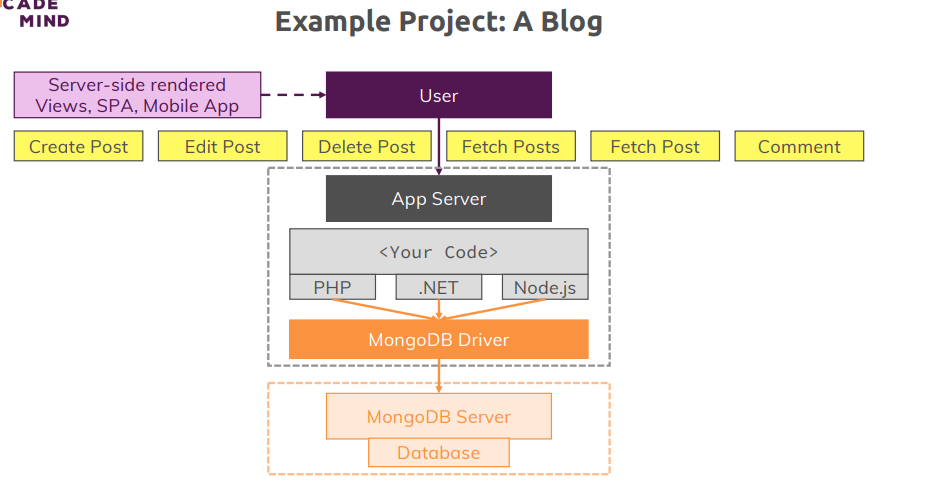
}

]

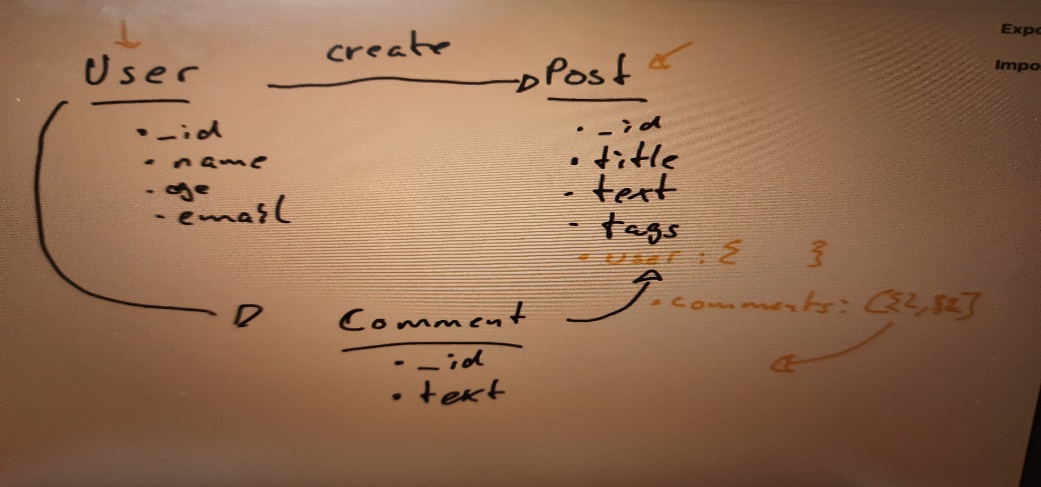
So here we're running aggregate on the books collection, so we want to pull in other documents from the authors collection, so that is the name of the collection where your related documents live in. You then define the local field, so in the collection we're running this on, so the books collection, where can the references to the other collection be found in? And for books, we stored that in the authors field, authors simply is an array of objectIds. Now we need to tell MongoDB ok I know where your keys are stored, which field are you relating to in your target collection, so in the author collection here, the authors collection and you define that with the foreign field key and there we know it's an \_ID. Please note that all these values are enclosed in double quotation marks, authors, authors and here, my \_ID field. So, I know that in authors, in books I have objectId stored where the matching equivalence can be found in \_ID in the authors collection. Last but not least, you give this an alias under which this will be merged, and I'll name this creators, the name is up to you here.

So, if you can and if your application needs it, go for an embedded document. If you have to use references or if you want to use references, well then, this lookup step in the aggregate method can at least help you get the data you need.

**=======================ASSIGNMENTS=====================**



we have our user in our application and that can be a server side rendered website, a single page application, a mobile app, anything like that. We've got our app server and in that app server or to that app server, we send a couple of different requests because let's say we're building a blog as the title says, so users are able to create posts, edit posts, delete posts, fetch all posts so that we can display a list of posts on a page, fetch a single post so we can read that post and also comment posts, so we can write comments and we can also fetch these comments of course to display them. Now we got our application server with our code being written in node js and .NET, PHP. We won't write the code here, this is not a web development course, we won't write any code here, we'll just structure the data for that and see how such a dummy request, how such a write or such an edit could look like, our code in a real app would of course use the mongo driver, we will use the shell here and in the end it will send the data the requests to the server. So, our goal here now in this exercise is to simply define a schema, identify the core entities we have in this world and how the schemas for these entities could look like, how they are related and how we would model these relations and then we'll just play around with some queries so that we can see how our application, how the mongodb driver would interact with our database.



this would be the only collection we then have, posts with embedded users in a single post or the embedded user and embedded comments. Now is that a good approach? Well parts of it are good, I'm fine with embedding my comments here because the comments are a one-to-many relationship, a post can have many comments but a comment always belongs to one post.

**Implementing Use-Case**

**>>> use blog**

**blog> db.users.insertMany({name:"asd", age:29, email: "abc@gmail.com"},{name:"ard", age:39, email: "adc@gmail.com"})**

**MongoInvalidArgumentError: Argument "docs" must be an array of documents**

**blog> db.users.insertMany([{name:"asd", age:29, email: "asd@gmail.com"},{name:"qwe", age:39, email: "qwe@gmail.com"}])**

**{**

**acknowledged: true,**

**insertedIds: {**

**'0': ObjectId("63ad6a02aaae44dd2c92f298"),**

**'1': ObjectId("63ad6a02aaae44dd2c92f299")**

**}**

**}**

**blog> db.users.find().pretty()**

**[**

**{**

**\_id: ObjectId("63ad6a02aaae44dd2c92f298"),**

**name: 'asd',**

**age: 29,**

**email: 'asd@gmail.com'**

**},**

**{**

**\_id: ObjectId("63ad6a02aaae44dd2c92f299"),**

**name: 'qwe',**

**age: 39,**

**email: 'qwe@gmail.com'**

**}**

**]**

**blog> db.posts.insertOne({title: "My First Post!", text: "This is my first post, I hope you like it!", tags: ["new", "tech"], creator: ObjectId("63ad6a02aaae44dd2c92f299"), comments: [{text: "I like this post!", author: ObjectId("63ad6a02aablog>**

**{**

**acknowledged: true,**

**insertedId: ObjectId("63ad6bf3aaae44dd2c92f29a")**

**insertedId: ObjectId("63ad6bf3aaae44dd2c92f29a")**

**blog> db.posts.findOne()**

**{**

**\_id: ObjectId("63ad6bf3aaae44dd2c92f29a"),**

**title: 'My First Post!',**

**text: 'This is my first post, I hope you like it!',**

**tags: [ 'new', 'tech' ],**

**creator: ObjectId("63ad6a02aaae44dd2c92f299"),**

**comments: [**

**{**

**text: 'I like this post!',**

**author: ObjectId("63ad6a02aaae44dd2c92f298")**

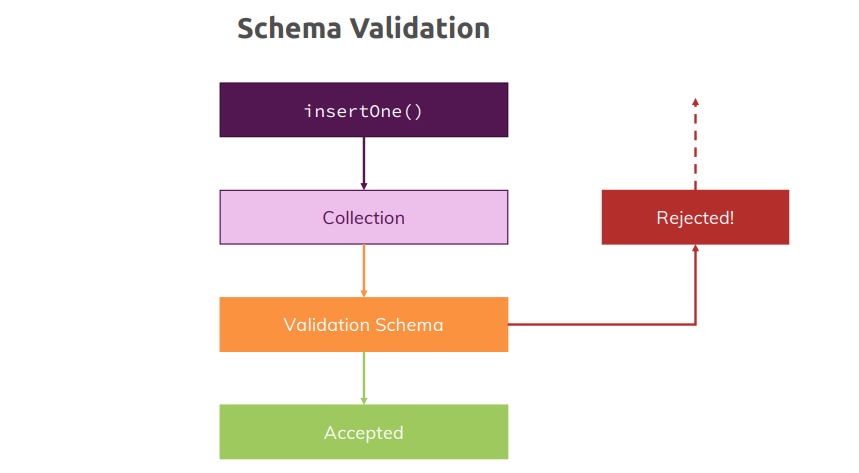
**}**

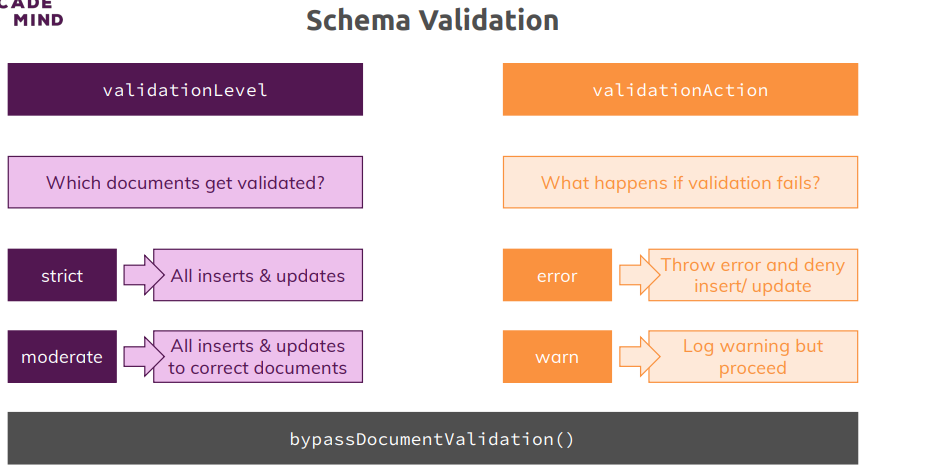
**]**

**}**

**Now my relation here is that this post is written by a user, let's name it creator and this could be a list if you have multiple authors in your application, let's say in this application and that is totally up to you of course, we only have one author per host. So my creator or author, however you want to name it here should be one of my users and let's say this was written by Manuel, so I'll copy that objectId, we could match it by any key of course but we should insure it's unique and the objectId isn't sure to be unique and I'll therefore set this as a value for creator. Now we also might have comments here and let's say we get a couple of comments there and each comment then is another document, so here I got a list of documents where I have some text, I like this post and as another key here, I don't just have the text but also the author of this comment and that again would be the ID of a user, so let's grab Max here maybe, grab that objectId here and insert this as the author. So here again, I'm also using not an embedded user for the same reason why I'm not using it on a post, I'm not using it on a comment either instead here, I also use a reference. Let's hit enter, this worked and let's have a look at our post, this is how it looks like. And this post has the fields I sketched out and most importantly, we have a reference relation to the person who created the post but we got an embedded document or even multiple documents for the comments where each comment in turn also has a reference relation to the author. This is how our blog could look like in code, so in the database and I also explained why I would structure it like this and this example together with the previous lectures hopefully gets you into the right mental model on how to think about this and hopefully helps you with your next projects to structure your data and your relations in a way that makes sense for your application to your size of data, the amount of data, the frequency with which you change it and your overall structure or relation that is imposed on to you by your application.**

**Understanding Schema Validation-**





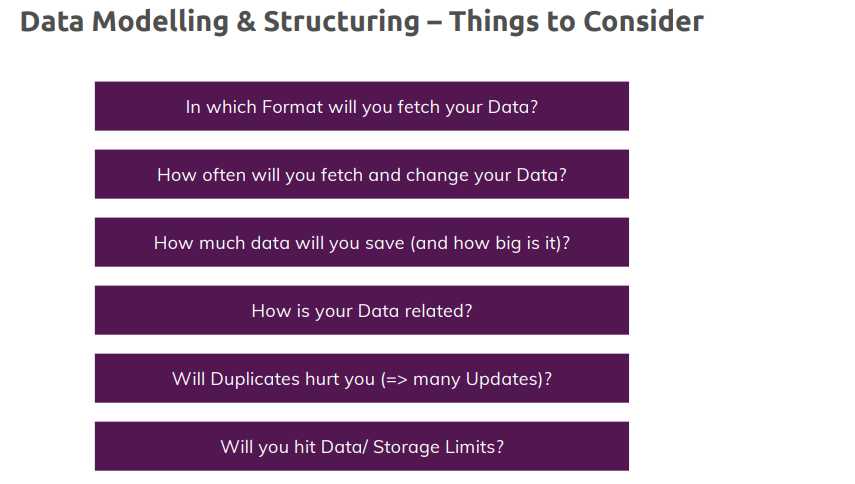
Now let's talk about schema validation you learned that mongodb is very flexible, you can have totally different schemas or documents in one and the same collection and that flexibility is a huge plus but sometimes you want to lock down your flexibility here, you want to get rid of it. Sometimes you need a strict schema because you know your application is going to fetch posts and it is going to access the title on each post and it does expect that each title is a string and for cases like this, schema validation can help you. Now what is schema validation?

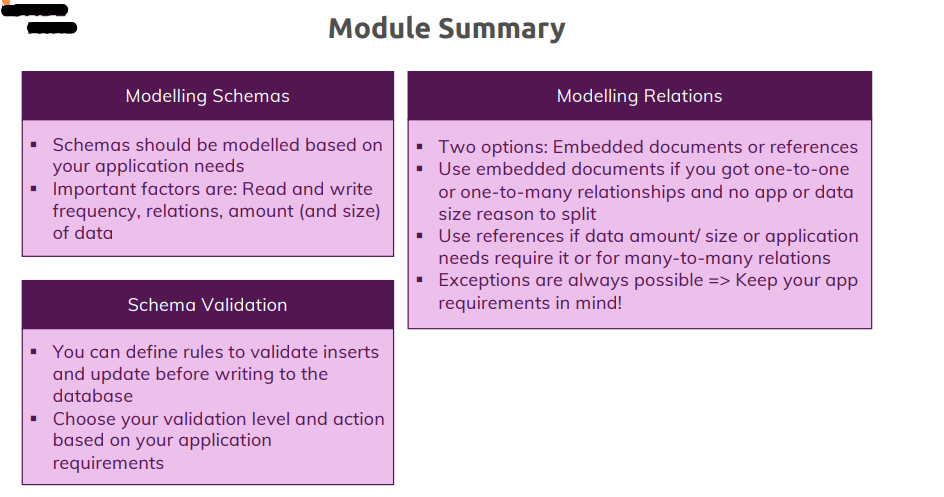
Let's say we have an insert operation going on on our collection, on our posts collection maybe and the same would be true for update or insertMany operations by the way, so we're trying to add or edit some data in a collection. If we add a validation schema and I will show you how you add one in the next lecture, if we added one, then the schema will validate or the mongodb will validate the incoming data based on the schema we defined and either it accepts it and then allows the write to the database or it rejects the incoming data, hence your database is not touched and is not changed and the user gets an error.

which kinds of operations you want to validate and what you want to do if validation fails with two settings you can add.

what happens if it fails, you can either set the validation level to strict which means all inserts and updates are checked or to moderate which means all inserts are checked but updates are only checked for documents which were valid before. So if you had some invalid data in there because it existed before you set up a schema, then you could still change these documents even if they don't fit your schema. For validation action, you can decide whether you want to throw an error and not go on with the insert or update, so it will not change your database data or if you only want to log a warning and proceed.

**Wrap Up-**



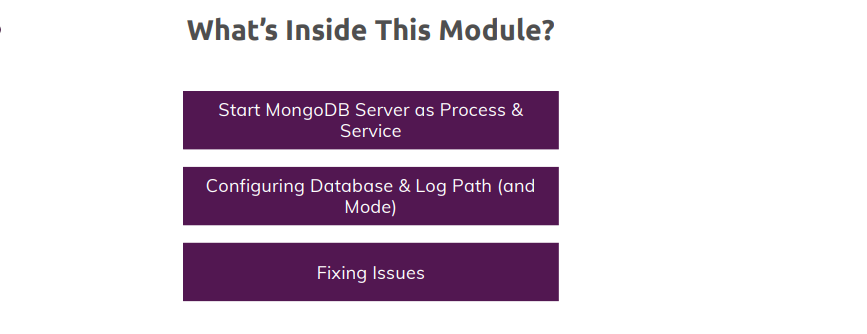


**Helpful Articles/ Docs:**

* **The MongoDB Limits**: <https://docs.mongodb.com/manual/reference/limits/>
* **The MongoDB Data Types**: <https://docs.mongodb.com/manual/reference/bson-types/>
* **More on Schema Validation**: <https://docs.mongodb.com/manual/core/schema-validation/>

1. **Exploring The Shell & The Server**

**========================================**



**Finding Available Options**

**For MongoDB documentation please follow below links:-**

* <https://www.mongodb.com/docs/>
* <https://www.mongodb.com/docs/manual/>

**test> help**

Shell Help:

**use** Set current database

show **'show databases'/'show dbs'**: Print a list of all available databases.

**'show collections'/'show** tables': Print a list of all collections for current database.

**'show profile'**: Prints system.profile information.

**'show users'**: Print a list of all users for current database.

**'show roles'**: Print a list of all roles for current database.

**'show log** <type>': log for current connection, if type is not set uses 'global'

**'show logs'**: Print all logs.

test>

**quit** Quit the MongoDB shell with **quit/quit()**

**Mongo** Create a new connection and return the Mongo object. Usage: new Mongo(URI, options [optional])

connect Create a new connection and return the **Database object. Usage: connect(URI, username [optional], password [optional])**

**i**t result of the last line evaluated; use to further iterate

**versio**n Shell version

**load** Loads and runs a JavaScript file into the current shell environment

enableTelemetry Enables collection of anonymous usage data to improve the mongosh CLI

disableTelemetry Disables collection of anonymous usage data to improve the mongosh CLI

passwordPrompt Prompts the user for a password

**sleep** Sleep for the specified number of milliseconds

**print** Prints the contents of an object to the output

**printjson** Alias for print()

convertShardKeyToHashed Returns the hashed value for the input using the same hashing function as a hashed index.

**cls** Clears the screen like console.clear()

isInteractive Returns whether the shell will enter or has entered interactive mode

For more information on usage: https://docs.mongodb.com/manual/reference/method

test>

**Useful Resources & Links**

Helpful Articles/ Docs:

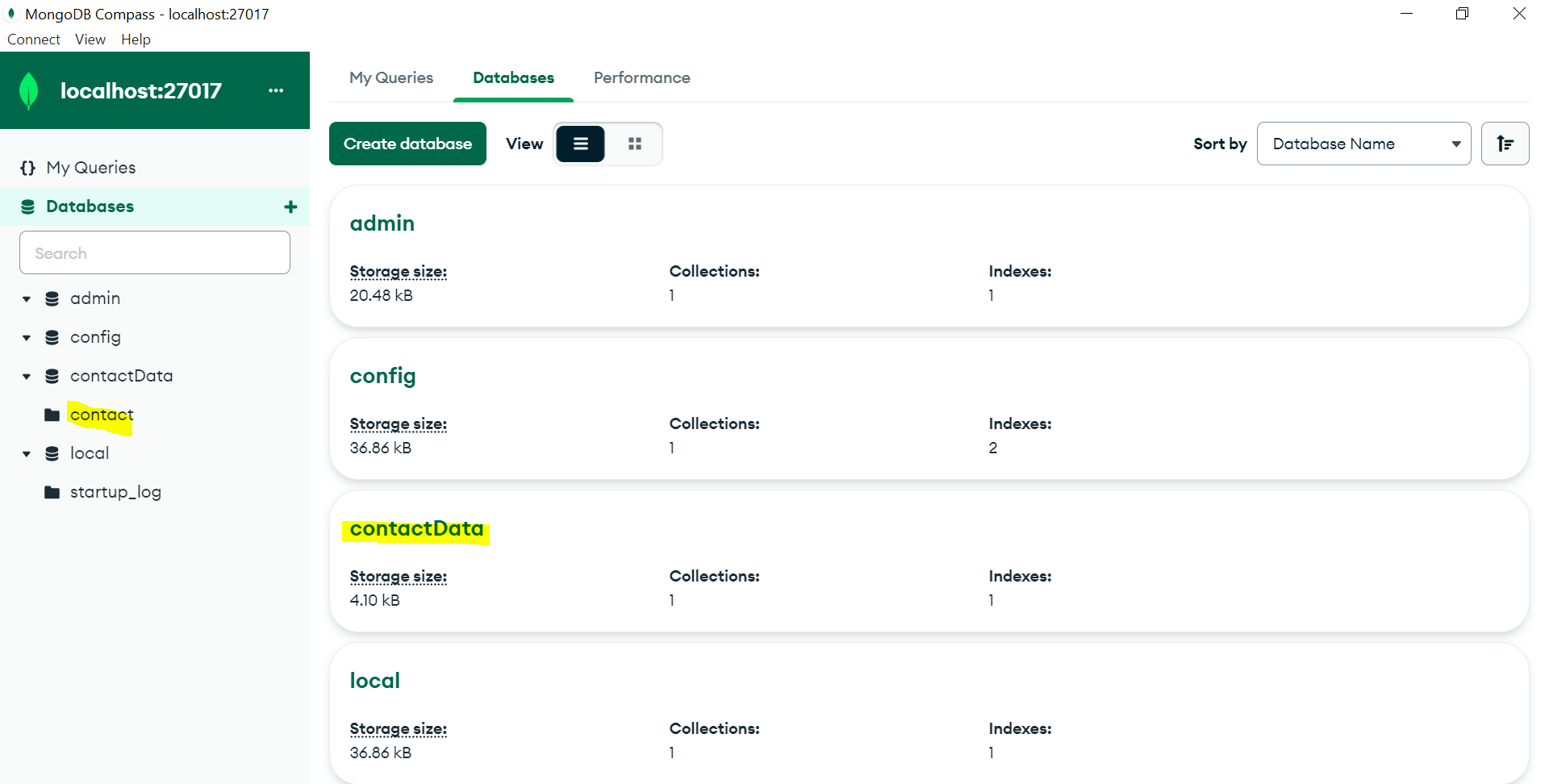
* More Details about Config Files: <https://docs.mongodb.com/manual/reference/configuration-options/>
* More Details about the Shell (mongo) Options: <https://docs.mongodb.com/manual/reference/program/mongo/>

1. **Using the MongoDB Compass to Explore Data Visually**

**==================================================**

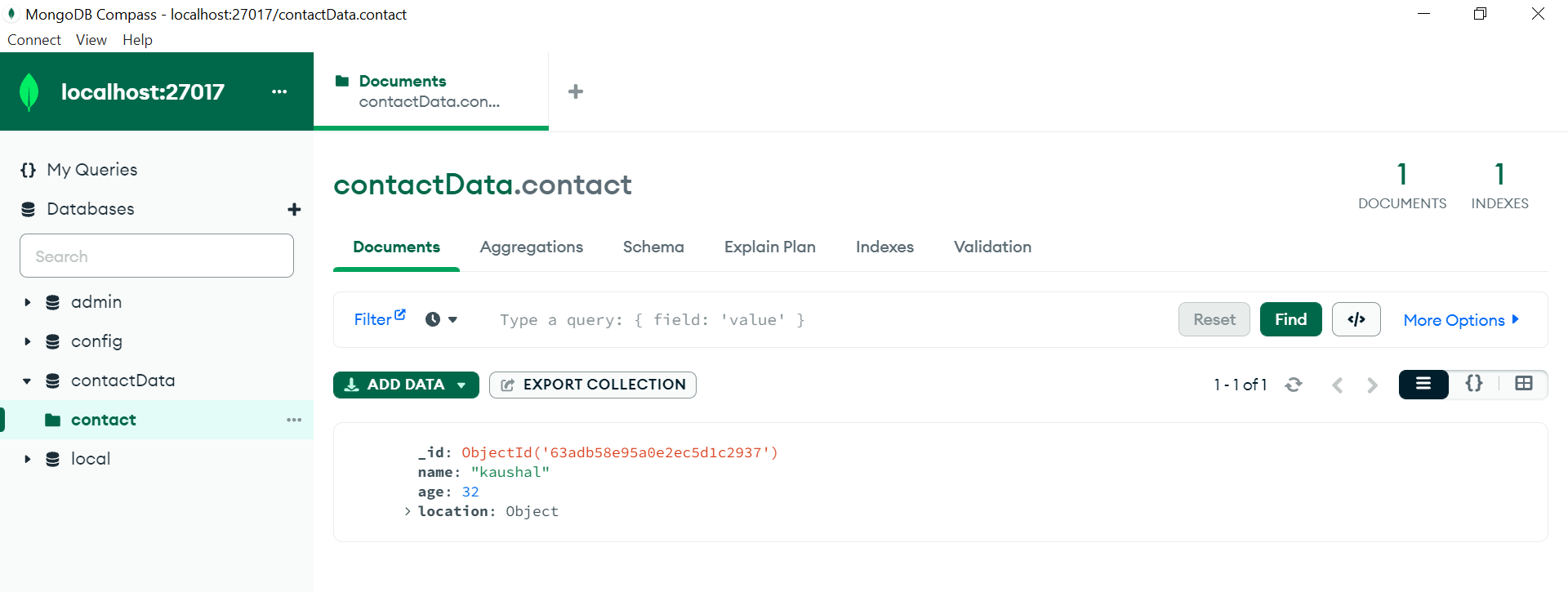
Install MongoDB Compass

<https://www.mongodb.com/try/download/compass>

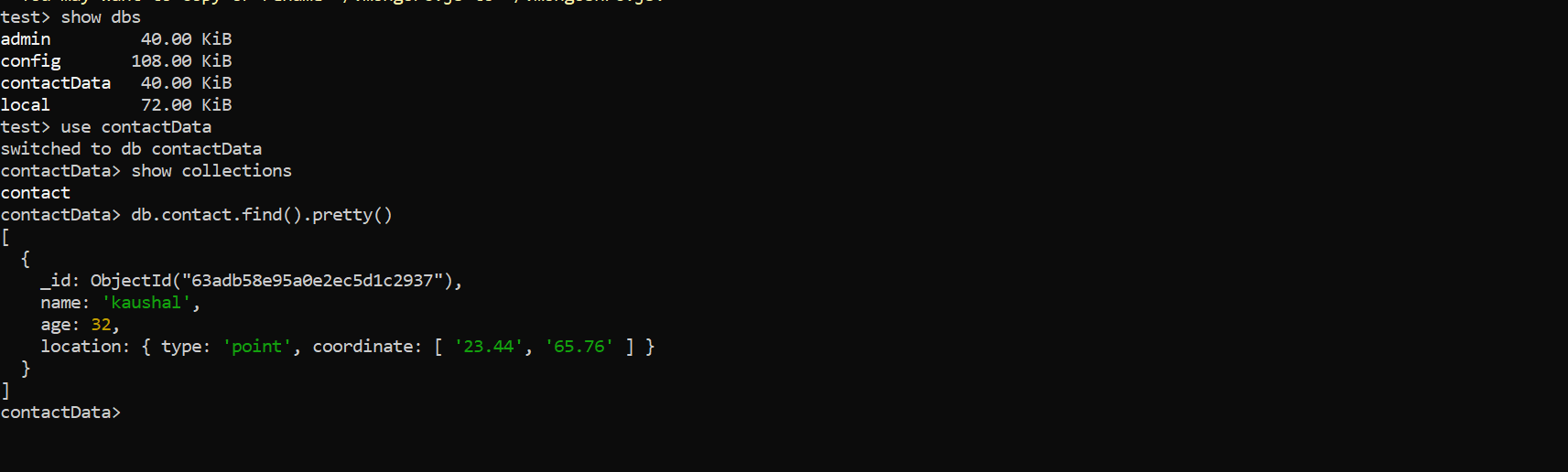


Let’s create a Database **contactData** and collection **contact**

Let’s create a document in our collection **contact**



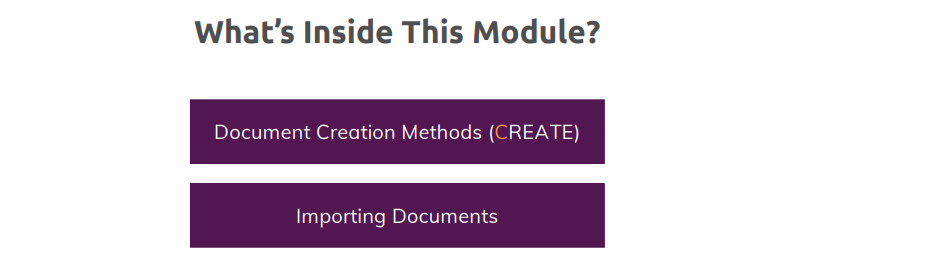
After creating the db, collection and document we can see through or mongodb Shell.



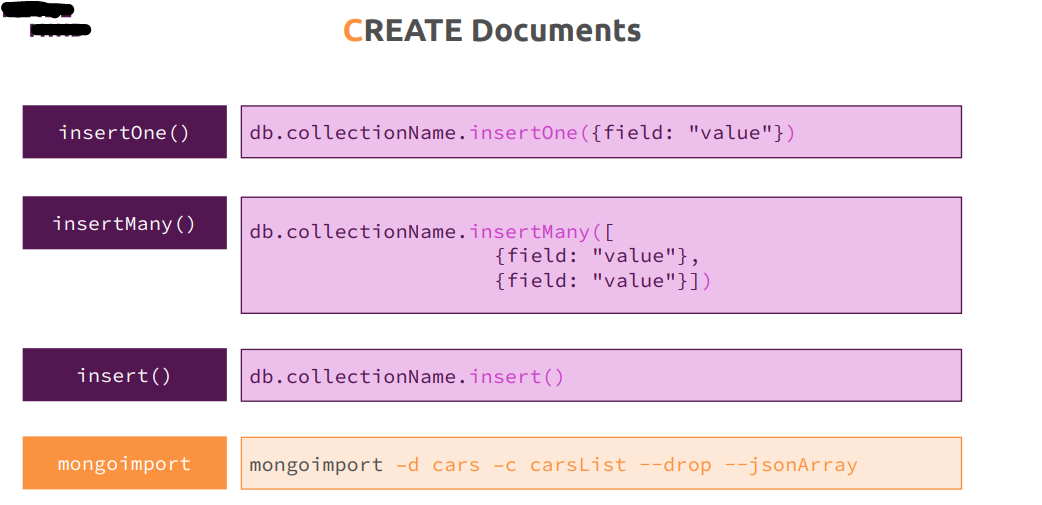
1. **Diving Deeper into Create Operations**

**==============================================**

A Closer Look at Creating & Importing Documents



The important thing to recognize here is that insert one takes a single document with the values you want to insert and you can but don't have to specify an ID because if you don't specify one, you will get one automatically and insert many does the same but with an array of documents, so with a list of documents that gets inserted. Now there also is a third method, insert, it's a bit more flexible because it takes both a single document or an array of documents but I will not bother diving into that because insert was used in the past, insert one and insert many were introduced on purpose so that you have to be more clear about what you will do because with insert, your code if you are writing an application could have been a bit harder to understand because it was not instantly obvious if you were to insert one or multiple documents and it therefore might have also been a bit more error prone because you could have ended up in a situation where you expected it to insert multiple documents.



**The Full insertMany() Docs:**

<https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/>

**Understanding “insert()” Methods-**

If u have some databases and if u want to get rid of from your database and all the data which is there like test…then use

* **Use test**
* **db.dropDatabase()**

let’s insert some person there…

we could name it **contactData** and then there, let's use the **persons** collection and let's insert one person. Now you learned this before but in the end here, you enter one document, a document always is marked by the curly braces that's around it and then a document is made up of the key value pairs

**test> use contactData**

switched to db contactData

**contactData> db.persons.insertOne({name: "Max", age: 30, hobbies: ["sports", "cooking"]})**

{

acknowledged: true,

insertedId: ObjectId("63aea7185bdeea5c04d405f6")

}

**contactData> db.persons.insertOne({name: "Manuel", age: 31, hobbies: ["cars", "cooking"]})**

{

acknowledged: true,

insertedId: ObjectId("63aee1d35bdeea5c04d405f7")

}

**now let's insert many and for this I'll reach out to the same collection and there, I'll use insert many like this and now the important thing here is you don't enter multiple documents separated by a comma because you don't pass them as arguments to this method. Insert many does indeed have a way where it accepts some data after this comma, so a second argument and I will dive into that second argument in a second but if you want to insert multiple documents, you just have to pass an array of these documents, so a list of these documents, a comma separated list.**

**contactData> db.persons.insertMany([{name: "Anna", age: 29, hobbies: ["sports", "yoga"]}])**

{

acknowledged: true,

insertedIds: { '0': ObjectId("63aef8395bdeea5c04d405f8") }

}

**contactData> db.persons.insertMany([{name: "Maria", age: 31}, {name: "Chris", age: 25}])**

{

acknowledged: true,

insertedIds: {

'0': ObjectId("63aefa585bdeea5c04d405f9"),

'1': ObjectId("63aefa585bdeea5c04d405fa")

}

}

as you learned in the schema module, you don't have to have the same structure on all documents in the same collection, this is totally optional so it would be totally fine if you inserted Anna without hobbies but I'll insert Anna with the hobbies of sports and yoga and as you see, that worked too even though it was only one person but of course the main idea behind insert many is that you do enter multiple persons or multiple documents but you don't have to as you just saw, it's perfectly fine if you use insert many with one document only, you still need to put it into an array, it does not work otherwise. here you see I inserted two documents separated by a comma also enclosed in square brackets, hence it's an array and this also worked

**contactData> db.persons.find().pretty()**

[

{

\_id: ObjectId("63aea7185bdeea5c04d405f6"),

name: 'Max',

age: 30,

hobbies: [ 'sports', 'cooking' ]

},

{

\_id: ObjectId("63aee1d35bdeea5c04d405f7"),

name: 'Manuel',

age: 31,

hobbies: [ 'cars', 'cooking' ]

},

{

\_id: ObjectId("63aef8395bdeea5c04d405f8"),

name: 'Anna',

age: 29,

hobbies: [ 'sports', 'yoga' ]

},

{ \_id: ObjectId("63aefa585bdeea5c04d405f9"), name: 'Maria', age: 31 },

{ \_id: ObjectId("63aefa585bdeea5c04d405fa"), name: 'Chris', age: 25 },

{ \_id: ObjectId("63aefbfb5bdeea5c04d405fb"), name: 'Phil', age: 35 }

]

Now as I mentioned, there also would be the insert method, the insert method can still be used, it's just not recommended.

**Working with Ordered Inserts**

I also briefly want to mention here that there are some other ways of inserting, one important alternative can be seen in the update module because there will have an option to either update a document or insert it if it doesn't exist yet but this is it from the insert method world for now. Now the more interesting part is now what you can do with these insert methods and there, you can actually define or specifies some additional information.

**contactData> db.hobibes.insertMany([{\_id: "sports", name: "Sports"},{\_id: "cooking", name: "Cooking"},{\_id: "cars", name: "Cars"}])**

{

acknowledged: true,

insertedIds: { '0': 'sports', '1': 'cooking', '2': 'cars' }

}

**contactData> db.hobibes.find().pretty()**

[

{ \_id: 'sports', name: 'Sports' },

{ \_id: 'cooking', name: 'Cooking' },

{ \_id: 'cars', name: 'Cars' }

]

**contactData> db.hobibes.insertMany([{\_id: "yoga", name: "Yoga"},{\_id: "cooking", name: "Cooking"},{\_id: "hiking", name: "Hiking"}])**

Uncaught:

MongoBulkWriteError: E11000 duplicate key error collection: contactData.hobibes index: \_id\_ dup key: { \_id: "cooking" }

Result: BulkWriteResult {

result: {

ok: 1,

writeErrors: [

WriteError {

err: {

index: 1,

code: 11000,

errmsg: 'E11000 duplicate key error collection: contactData.hobibes index: \_id\_ dup key: { \_id: "cooking" }',

errInfo: undefined,

op: { \_id: 'cooking', name: 'Cooking' }

}

}

],

writeConcernErrors: [],

insertedIds: [

{ index: 0, \_id: 'yoga' },

{ index: 1, \_id: 'cooking' },

{ index: 2, \_id: 'hiking' }

],

nInserted: 1,

nUpserted: 0,

nMatched: 0,

nModified: 0,

nRemoved: 0,

upserted: []

}

}

**contactData> db.hobibes.insertMany([{\_id: "yoga", name: "Yoga"},{\_id: "cooking", name: "Cooking"},{\_id: "hiking", name: "Hiking"}], {ordered: true})**

Uncaught:

MongoBulkWriteError: E11000 duplicate key error collection: contactData.hobibes index: \_id\_ dup key: { \_id: "yoga" }

Result: BulkWriteResult {

result: {

ok: 1,

writeErrors: [

WriteError {

err: {

index: 0,

code: 11000,

errmsg: 'E11000 duplicate key error collection: contactData.hobibes index: \_id\_ dup key: { \_id: "yoga" }',

errInfo: undefined,

op: { \_id: 'yoga', name: 'Yoga' }

}

}

],

writeConcernErrors: [],

insertedIds: [

{ index: 0, \_id: 'yoga' },

{ index: 1, \_id: 'cooking' },

{ index: 2, \_id: 'hiking' }

],

nInserted: 0,

nUpserted: 0,

nMatched: 0,

nModified: 0,

nRemoved: 0,

upserted: []

}

}

**contactData> db.hobibes.insertMany([{\_id: "yoga", name: "Yoga"},{\_id: "cooking", name: "Cooking"},{\_id: "hiking", name: "Hiking"}], {ordered: false})**

Uncaught:

MongoBulkWriteError: E11000 duplicate key error collection: contactData.hobibes index: \_id\_ dup key: { \_id: "yoga" }

Result: BulkWriteResult {

result: {

ok: 1,

writeErrors: [

WriteError {

err: {

index: 0,

code: 11000,

errmsg: 'E11000 duplicate key error collection: contactData.hobibes index: \_id\_ dup key: { \_id: "yoga" }',

errInfo: undefined,

op: { \_id: 'yoga', name: 'Yoga' }

}

},

WriteError {

err: {

index: 1,

code: 11000,

errmsg: 'E11000 duplicate key error collection: contactData.hobibes index: \_id\_ dup key: { \_id: "cooking" }',

errInfo: undefined,

op: { \_id: 'cooking', name: 'Cooking' }

}

}

],

writeConcernErrors: [],

insertedIds: [

{ index: 0, \_id: 'yoga' },

{ index: 1, \_id: 'cooking' },

{ index: 2, \_id: 'hiking' }

],

nInserted: 1,

nUpserted: 0,

nMatched: 0,

nModified: 0,

nRemoved: 0,

upserted: []

}

}

**contactData> db.hobibes.find().pretty()**

[

{ \_id: 'sports', name: 'Sports' },

{ \_id: 'cooking', name: 'Cooking' },

{ \_id: 'cars', name: 'Cars' },

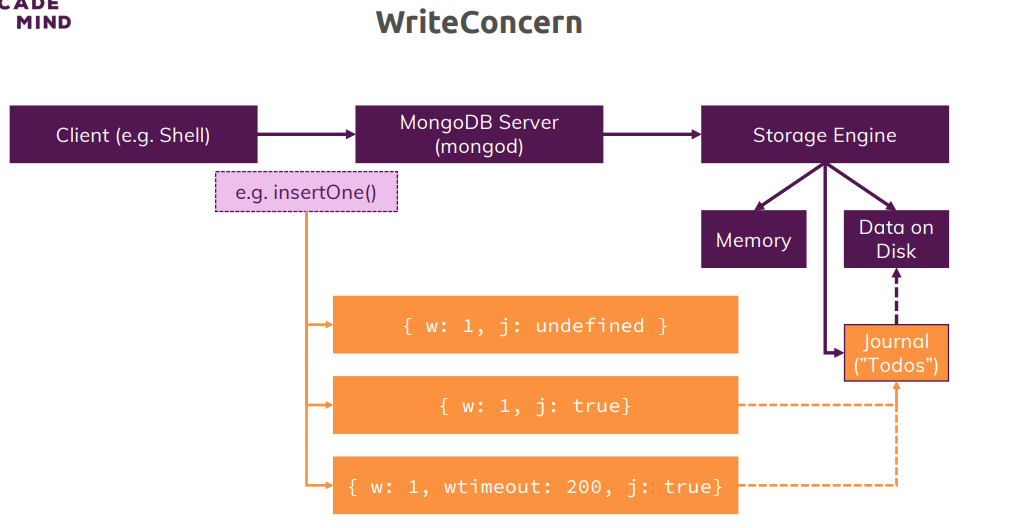
{ \_id: 'yoga', name: 'Yoga' },

{ \_id: 'hiking', name: 'Hiking' }

]

Often you want to use that ID but it also is the case that you want to use your own ID, maybe because the data is fetched from some other database where you already had an ID or maybe because you need a shorter ID, here for example you could say the name of the sport also makes up for a great ID. So we could have \_id sports here because we know that this will be unique, we will not have sports twice in this collection, so we can absolutely use sports as an ID here and the same here for cooking, we can of course also use cooking and I just name it all lowercase, you could absolutely use a capital C in there, that would be accepted, so this is not required to be lowercase but I'll use it like this and the same for my last document, for the cars, I'll set \_id to cars. Now important, It has to be \_id, just ID will not work, it has to be \_id. If I now hit enter, you'll see these were the inserted IDs and indeed when we look into the database, we'll see no other IDs, so no object ID and this is perfectly fine and this is not bad to do, absolutely not, bring your own IDs if you have them, so now I got these three entries in there and I got my own ID. Now let's say I repeat that operation, so I add more hobbies, I add let's say yoga here and then let's leave the middle one untouched, I also add let's say hiking here. Now the middle one, cooking is untouched because let's say the process where we add this, maybe it's user added data doesn't guarantee that we check whether this item exists in the database before, so I try to enter data with an ID that already exists in the database. What happens if I now hit enter? Well if I do this, I get an error, now let's see what we got. I get an error, a write error at item 1, arrays start with index 0, so it tells us that the element at index 0, the first element succeeded but at index 1, so the second element in the array it failed. So we failed in that bulk operation and it gives us some more information, it tells us again where it failed and that it failed because of a duplicate key error collection. It could have also failed due to a different error, maybe our server went down during our operation, that could have been a thing, now here it failed due to duplicate key problems, it also tells us the ID, cooking, so it gives us the exact document on which it failed. So this is of course helpful but you also see here that one element was inserted and indeed if I find, whoops, if I reach out to my hobbies and I find them all and pretty print them, you see sports was added, cooking was added, cars was added, that was my first ride but in the last ride where I got the error, yoga which was the first document was added to the collection, so it failed after this element. Now this is the default behavior of mongodb and this is called an ordered insert and ordered inserts simply means that every element you insert is processed standalone but if one fails, it cancels the entire insert operation but it does not rollback the elements it already inserted, that's important but it cancels the operation and does not continue with the next document which as we know would have succeeded. So this is the default behavior, often you want that behavior, sometimes you don't and for these cases, you can override the behavior. Now if I repeat that same operation, it will again fail but now already on the first document because yoga was inserted in the last step, now it will of course fail because now it's already in the database. So how can I change this behavior? Well we can pass a second argument separated with a comma to insert many, that argument is a document and this is now not a document that will get inserted instead it is a document that configures this operation. And here we can set some options and the option that helps us in this case here is the ordered option, the ordered option allows us to specify whether mongodb should perform an ordered insert which is the default, you can set this to true but that is redundant because data is the default setting, so now it will behave in exactly the way we saw it before but we can also set this to false and now it will not be an ordered insert Now if I now hit enter, I still get that error but you'll see I now get a list of all the right errors, yoga and cooking which already tells us that it continued after failing on yoga and we also see that one element was inserted and that has to be hiking since we try to insert three and two of them failed. And indeed, if I now look into my database with find pretty, we see hiking in there. So by setting ordered to false here, we changed the default behavior and it is up to you what you require or what you want in your application. Be aware that it will never rollback the entire operation if something failed, that is something I'll cover towards the end of the course in the transactions module because we'll need some additional knowledge to work with such transactions. A rollback will not happen here but you can control whether it continues with the other documents and tries to insert everything which is fine or if it does not and being able to set this is very useful because you might have cases where you can't really control whether the data you are trying to insert already is in the database. And then an unordered insert with ordered false can be your solution because you don't care about any documents that fail, they already are in the database, that's fine for you and you can add everything which is not in there yet. So this is how you can use ordered and unordered inserts to finetune the way documents get inserted to your needs.

**Understanding the “writeConcern”**



We learned about ordered and unordered inserts now there is a second option we can specify on insertOone and insertMany and that is the write concern option We have a client and we have our mongodb server, the client would be the shell or your application using a mongodb driver, the mongodb server is what you start with the mongod executable. Now let's say you want to insert one document in there and it would also be true for insert many and for update and so on, so for all the write operations because it's the write concern option we're talking about. So we're trying to insert one document, now as you learned in the first module of this course, on the mongodb server, we have a so-called storage engine, this is the engine being responsible for really writing our data onto the disk and also for managing it in memory. So your write might first end up in memory here and there, it manages the data which it needs to access with high frequency because that is faster than working with the disk. Of course your write is also scheduled to then end up on the disk, that is not the thing, so it will also eventually store your data on the disk. Now you can configure a so-called write concern on all the write operations like insert one with an additional argument, the write concern argument which is in turn a document where you can set settings like that, now what does this mean? The w simply means write and the number here indicates to how many instances, in case you're using multiple instances on one server, that is also something we'll cover towards the end of the course so let's ignore it for now, so on how many instances you want this write to be acknowledged. With W1 which is the default, you're basically saying hey my mongodb server should have accepted that write, so the storage engine is aware of it and will eventually write it to the disk. The J stands for the journal, the journal is an additional file which the storage engine manages which is like a To-Do file. The journal can be kept to then for example save operations that the storage engine needs to-do that have not been completed yet, like the write. Now it is aware of the write and that it needs to store that data on disk just by the write being acknowledged and being there in memory, it doesn't need to keep a journal for that. The idea of that journal file which is a real file on the disk is just that it is aware of this and if the server should go down for some reason or anything else happens, that file is still there and if you restart the server or if it recovers basically, it can look into that file and see what it needs to-do and that is of course a nice back up because the memory might have been wiped by then. So your write could be lost if it's not written to the journal, if it hasn't been written to the real data files yet, that is the idea of the journal, it's like a back up to-do list. Now the question is why do we write it in the journal and not directly into the database files? Because writing into the database files simply is more performance heavy, the journal is like a single line which describes the write operation, writing into the database files is of course a more complex task because there you need to find the right position to insert it, if you have indexes, something we'll cover later in this course, you also need to update these, so it simply takes longer, adding a to-do in a journal is faster. Still that also takes longer than not using the journal and the default is that the journal is not getting used with j undefined and that does simply mean that the storage engine will eventually handle this write and also write it to the journal but you don't have that information yet, you don't know if it has been stored in the journal yet, if the write succeeded yet, if the write has been done on the disk, you don't know any of that, you just know that the server is aware of your write. So if the server should go down in that exact moment, it might indeed not have done the write because it hasn't been added to the journal yet, it hasn't been saved to the database files yet. Now you can set a different option and set J to true, what you're now saying is hey please only report a success for this write to me after it has been both acknowledged and been saved to the journal, so now I have a greater security that this will happen and succeed even if the server should face issues right now. Now there also is a third option not directly related to the journal but it's a W timeout option, now this simply means which timeframe do you give your server to report a success for this write before you cancel it. So if you have some issues with the network connection or anything like that, you might simply timeout here, obviously setting this too low might timeout even though it would have perfectly succeeded normally and therefore you should know what you do when you set this timeout value because if you set it to a very low number, you might fail a lot even though there is no actual problem, just some small latency. So this is the write concern and how you can control this, obviously enabling the journal confirmation means that your writes will take longer because you don't just tell the server about them but you also need to wait for the server to store that write operation in the journal but you get higher security that the write also succeeded.

**The “writeConcern” in Practice**

**contactData> db.persons.insertOne({name: "Chrissy", age: 41}, {writeConcern: {w:0}})**

{

acknowledged: false,

insertedId: ObjectId("63b049cf029426c13971064a")

}

**contactData> db.persons.find().pretty()**

[

{

\_id: ObjectId("63aea7185bdeea5c04d405f6"),

name: 'Max',

age: 30,

hobbies: [ 'sports', 'cooking' ]

},

{

\_id: ObjectId("63aee1d35bdeea5c04d405f7"),

name: 'Manuel',

age: 31,

hobbies: [ 'cars', 'cooking' ]

},

{

\_id: ObjectId("63aef8395bdeea5c04d405f8"),

name: 'Anna',

age: 29,

hobbies: [ 'sports', 'yoga' ]

},

{ \_id: ObjectId("63aefa585bdeea5c04d405f9"), name: 'Maria', age: 31 },

{ \_id: ObjectId("63aefa585bdeea5c04d405fa"), name: 'Chris', age: 25 },

{ \_id: ObjectId("63aefbfb5bdeea5c04d405fb"), name: 'Phil', age: 35 },

{ \_id: ObjectId("63aefd7d5bdeea5c04d405fc"), name: 'Hansh', age: 28 },

{

\_id: ObjectId("63aefd7d5bdeea5c04d405fd"),

name: 'Sandeep',

age: 38

},

{

\_id: ObjectId("63b049cf029426c13971064a"),

name: 'Chrissy',

age: 41

}

]

**contactData> db.persons.insertOne({name: "Alex", age: 36}, {writeConcern: {w:1}})**

{

acknowledged: true,

insertedId: ObjectId("63b04e62029426c13971064b")

}

**contactData> db.persons.insertOne({name: "Michael", age: 51}, {writeConcern: {w:1, j: false}})**

{

acknowledged: true,

insertedId: ObjectId("63b04ec1029426c13971064c")

}

**contactData> db.persons.insertOne({name: "Michaela", age: 51}, {writeConcern: {w:1, j: true}})**

{

acknowledged: true,

insertedId: ObjectId("63b04eee029426c13971064d")

}

**contactData> db.persons.insertOne({name: "Alia", age: 22}, {writeConcern: {w:1, j: true, wtimeout: 200}})**

{

acknowledged: true,

insertedId: ObjectId("63b04f4a029426c13971064e")

}

**contactData> db.persons.insertOne({name: "Alia", age: 22}, {writeConcern: {w:1, j: true, wtimeout: 1}})**

{

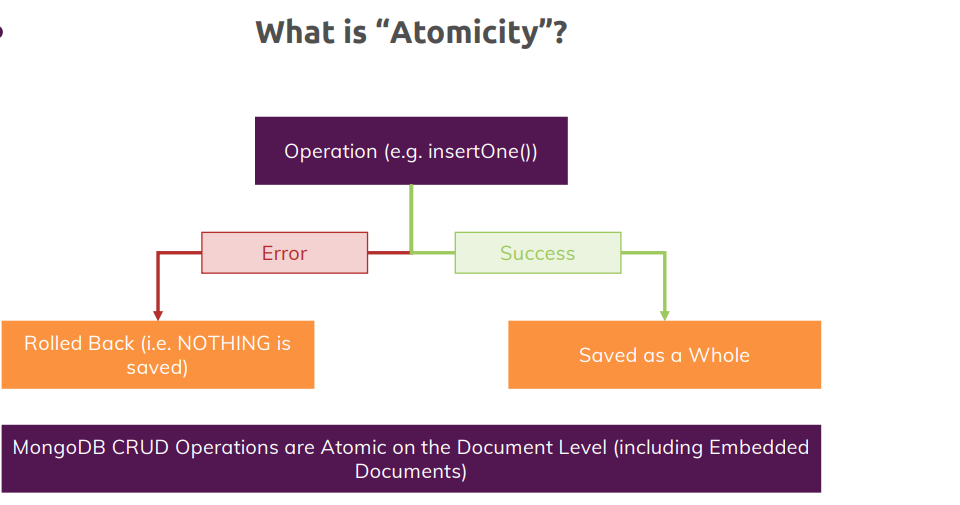
acknowledged: true,

insertedId: ObjectId("63b04f5e029426c13971064f")

}

I will now set my write concern. I do this by setting the write concern, named like this, the casing is important, so the write concern key, I set it to a value which again is a document which has the shape you saw on the slides. W 1 is the default, it simply means I need to be sure that the server acknowledge this, you can set this to 0. If you do this, you get back acknowledged false but if I find everything, you see that Chrissy was inserted. So you get back a different result, also without an objectID because it can't give you one, the server hasn't really registered this write yet, you just sent the request and you immediately return, you don't wait for a response of this request, so to say. So the storage engine had no chance to store it in memory and generate that objectId and therefore, you get back acknowledged false because you sent the request, you don't even know if it reached the server. This is of course super fast because you don't have to wait for any response here, for any ID generation but obviously, it tells you nothing about whether this succeeded or not. Could still be a valid option for data where it's ok for you, if some data does not end up in a database, so if you log some value every second about an application and you don't care if a couple of seconds get lost, you could do that. So that is w 0, the default with Alex who is 36, the default is w 1 of course, this gives you the output you saw before, acknowledge true and the inserted ID. Now let's go for Michael and let's play around with the journal, the journal can be set to true, the default is undefined or false, so if I set it to false, I have the same result as before. Now if I change it to McKayla and we set the journal to true now, the output for us does not change and . it also was super-fast here because everything runs locally and it's not like the journaling will take four hours but it will have been a little bit slower because the entry will have been added to the journal and we waited for that journal editing to finish here. So here, we have higher security because we can also be sure that it ended up in this to do list of the storage engine which will eventually lead to the writes happen to database files. So now we got this, now let me add another person, Aliya 22 and let me also add a third option, w timeout. If I add this like this, it succeeded. Now if I repeat it with a very small value, it also succeeded because this is super-fast here but it is an option which you can set in case you get shaky, a shaky connection or speed really matters and your connection is generally good but you can't rule out that once in a year, it's kind of shaky and you would then rather have that write fail and you recognize this in your client application of course because you'll get back an error here too. So you'll have that write fail and therefore you can try again later but you don't wait unnecessarily long.

**What is Atomicity ?**



**==========================ASSIGNMENTS===========================**

1. Insert multiple companies (company data of your choice) into a collection – both with insertOne() and insertMany()

**test> use companyData**

switched to db companyData

**companyData> db.companies.insertOne({name: "Fresh Food Inc", stock: 100, \_id: 1})**

{ acknowledged: true, insertedId: 1 }

**companyData> db.companies.insertMany([{name: "Smart Teddy Bear Inc", stock: 200, \_id: 2},{name: "Heavy Beef Inc", stock: 50, \_id: 3}])**

{ acknowledged: true, insertedIds: { '0': 2, '1': 3 } }

**companyData> db.companies.find().pretty()**

[

{ \_id: 1, name: 'Fresh Food Inc', stock: 100 },

{ \_id: 2, name: 'Smart Teddy Bear Inc', stock: 200 },

{ \_id: 3, name: 'Heavy Beef Inc', stock: 50 }

1. Deliberately insert duplicate ID data and “fix” failing additions with unordered inserts

**companyData> db.companies.insertMany([{name: "Fresh Beveragess Inc", stock: 88, \_id: 1},{name: "Cool Burgers Inc", stock: 12, \_id: 4}])**

Uncaught:

MongoBulkWriteError: E11000 duplicate key error collection: companyData.companies index: \_id\_ dup key: { \_id: 1 }

Result: BulkWriteResult {

result: {

ok: 1,

writeErrors: [

WriteError {

err: {

index: 0,

code: 11000,

errmsg: 'E11000 duplicate key error collection: companyData.companies index: \_id\_ dup key: { \_id: 1 }',

errInfo: undefined,

op: { name: 'Fresh Beveragess Inc', stock: 88, \_id: 1 }

}

}

],

writeConcernErrors: [],

insertedIds: [ { index: 0, \_id: 1 }, { index: 1, \_id: 4 } ],

nInserted: 0,

nUpserted: 0,

nMatched: 0,

nModified: 0,

nRemoved: 0,

upserted: []

}

}

Here the id : 1 which is already exists in our database and the id : 4 which doesn’t exist into our database. we got an error and u can see nothing was inserted and we see here it got failed because of a duplicate key.

Now we learned that we can change this behavior by adding a second argument and now this is out side of the array because it’s not a document we are going to insert, it’s just a document which configure insertMany(). We can add **ordered: false,** the default ordered is true.

**companyData> db.companies.insertMany([{name: "Fresh Beveragess Inc", stock: 88, \_id: 1},{name: "Cool Burgers Inc", stock: 12, \_id: 4}], {ordered: false})**

Uncaught:

MongoBulkWriteError: E11000 duplicate key error collection: companyData.companies index: \_id\_ dup key: { \_id: 1 }

Result: BulkWriteResult {

result: {

ok: 1,

writeErrors: [

WriteError {

err: {

index: 0,

code: 11000,

errmsg: 'E11000 duplicate key error collection: companyData.companies index: \_id\_ dup key: { \_id: 1 }',

errInfo: undefined,

op: { name: 'Fresh Beveragess Inc', stock: 88, \_id: 1 }

}

}

],

writeConcernErrors: [],

insertedIds: [ { index: 0, \_id: 1 }, { index: 1, \_id: 4 } ],

nInserted: 1,

nUpserted: 0,

nMatched: 0,

nModified: 0,

nRemoved: 0,

upserted: []

}

}

We still got the error but our document also get inserted

**companyData> db.companies.find().pretty()**

[

{ \_id: 1, name: 'Fresh Food Inc', stock: 100 },

{ \_id: 2, name: 'Smart Teddy Bear Inc', stock: 200 },

{ \_id: 3, name: 'Heavy Beef Inc', stock: 50 },

{ \_id: 4, name: 'Cool Burgers Inc', stock: 12 }

]

So, after duplicate entry error also our document got inserted.

1. Write data for a new company with both journaling being guaranteed and not being guaranteed

This test will need to play around **writeConcern**

So**, id : 5** does not exist yet.

Now I will add my document to configure my document to insert request again and set a writeConcern like this. we specify w: 1 which is the default setting.

**W: 0** which mean that basically we don’t wait for the server to acknowledge this it will still probably succeed if we don’t know of it if the server was aware of it, so let’s keep of it. Let’s play around the journaling.

If I set it to false or undefined or don’t set it at all, I have the default behavior. Where we don’t wait for the journal to do anything.

**test> use companyData**

switched to db companyData

**companyData> db.companies.insertOne({name: "Touchy Tables Inc", stock: 212, \_id: 5}, {writeConcern: {w: 1, j: false}})**

{ acknowledged: true, insertedId: 5 }

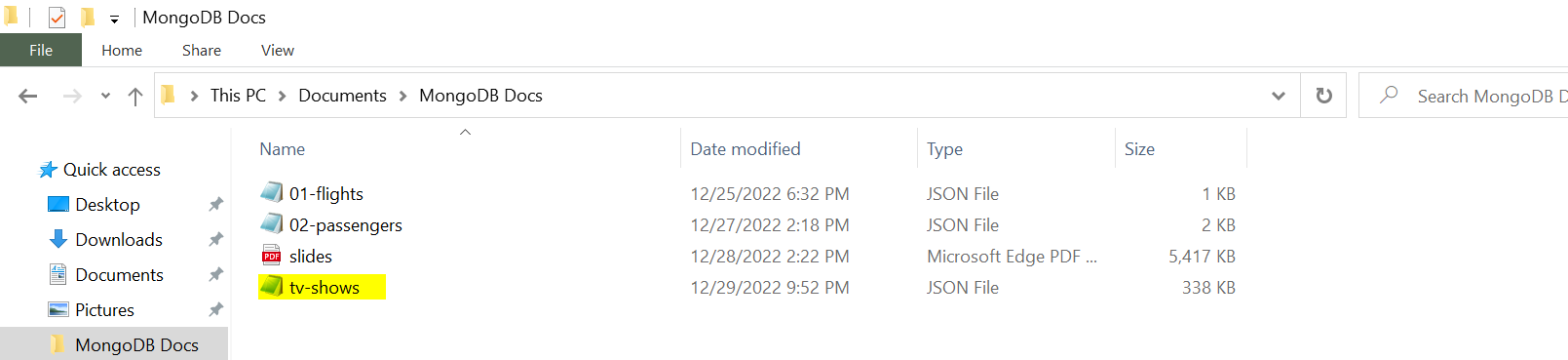
Now to repeat this and for a simplicity one let’s have a same company name and different id and I set here the journal parameter to true, then I do wait this to be written to the journal. It still super-fast here. but now we will wait for this to be written to the journal. Therefore, here we can be sure that even if we now have some server issue this write will eventually ends up in our database file. So, with that our assignment got solved.

**companyData> db.companies.insertOne({name: "Touchy Tables Inc", stock: 212, \_id: 6}, {writeConcern: {w: 1, j: true}})**

{ acknowledged: true, insertedId: 6 }

**Importing Data**

To have an older data collection in our db let’s import the json doc in our db.



We have downloaded a json document for mongo db in our MongoDB Docs folder.

Let’s open a command prompt there.

**C:\Users\kaukumar3\Documents\MongoDB Docs>dir**

Volume in drive C is OSDisk

Volume Serial Number is 427C-6DE8

Directory of C:\Users\kaukumar3\Documents\MongoDB Docs

01/01/2023 07:45 PM <DIR> .

01/01/2023 07:45 PM <DIR> ..

12/25/2022 06:32 PM 306 01-flights.json

12/27/2022 02:18 PM 1,097 02-passengers.json

12/28/2022 02:22 PM 5,546,320 slides.pdf

12/29/2022 09:52 PM 345,911 **tv-shows.json**

4 File(s) 5,893,634 bytes

2 Dir(s) 146,035,814,400 bytes free

**C:\Users\kaukumar3\Documents\MongoDB Docs>mongoimport tv-shows.json -d movieData -c movies --jsonArray --drop**

2023-01-01T19:53:21.316+0530 connected to: mongodb://localhost/

2023-01-01T19:53:21.323+0530 dropping: movieData.movies

2023-01-01T19:53:21.410+0530 240 document(s) imported successfully. 0 document(s) failed to import.

Now already we imported successfully in our mongodb, let’s verify this…

**test> show dbs**

admin 40.00 KiB

companyData 72.00 KiB

config 108.00 KiB

contactData 144.00 KiB

local 72.00 KiB

movieData 192.00 KiB

**test> use movieData**

switched to db movieData

**movieData> show collections**

movies

**movieData> db.movies.find().pretty()**

movieData>

{

\_id: ObjectId("63b197590059243887582810"),

id: 2,

url: 'http://www.tvmaze.com/shows/2/person-of-interest',

name: 'Person of Interest',

type: 'Scripted',

language: 'English',

genres: [ 'Drama', 'Action', 'Crime' ],

status: 'Ended',

runtime: 60,

premiered: '2011-09-22',

officialSite: 'http://www.cbs.com/shows/person\_of\_interest/',

schedule: { time: '22:00', days: [ 'Tuesday' ] },

rating: { average: 9 },

weight: 96,

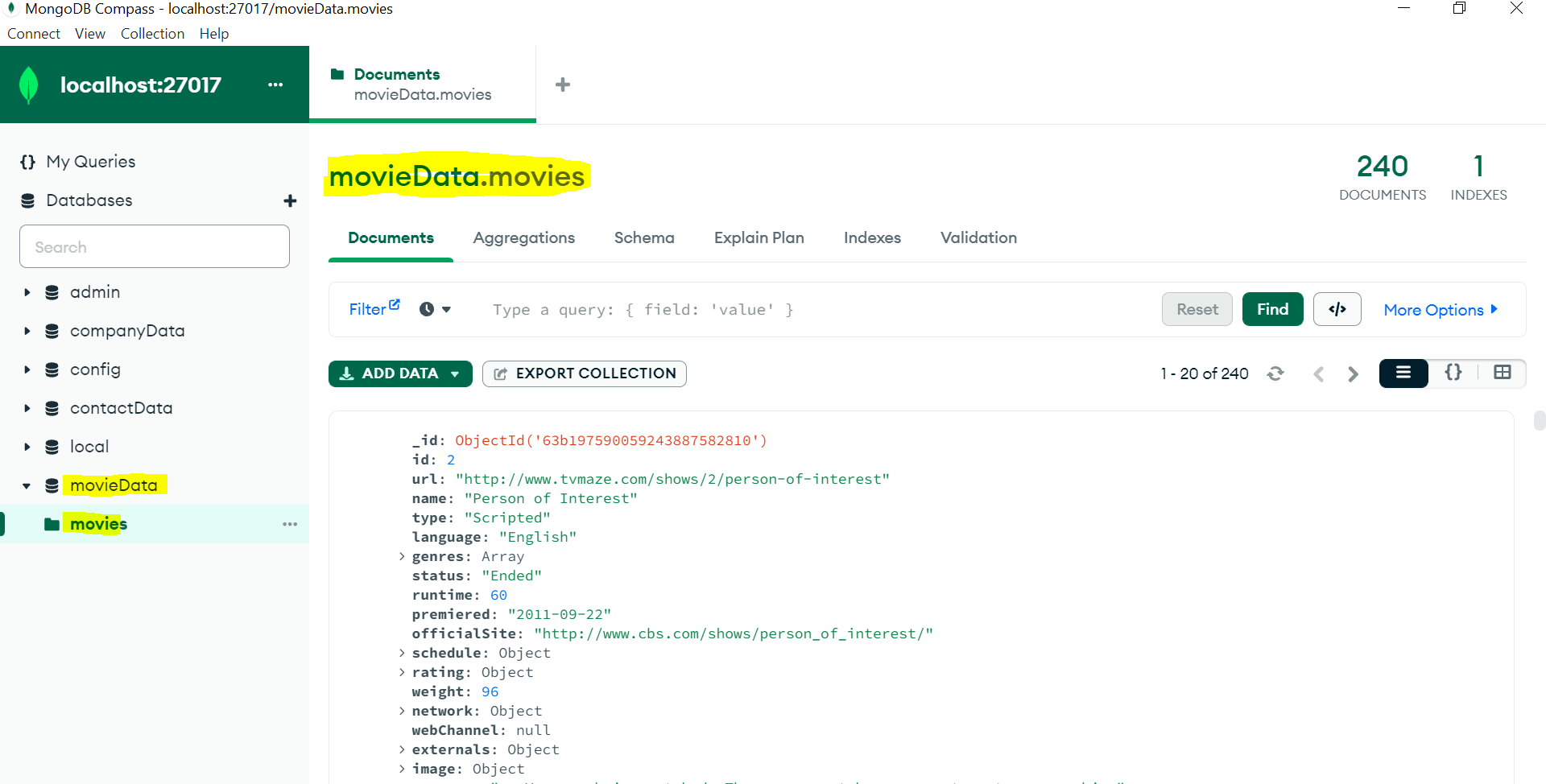
network: {

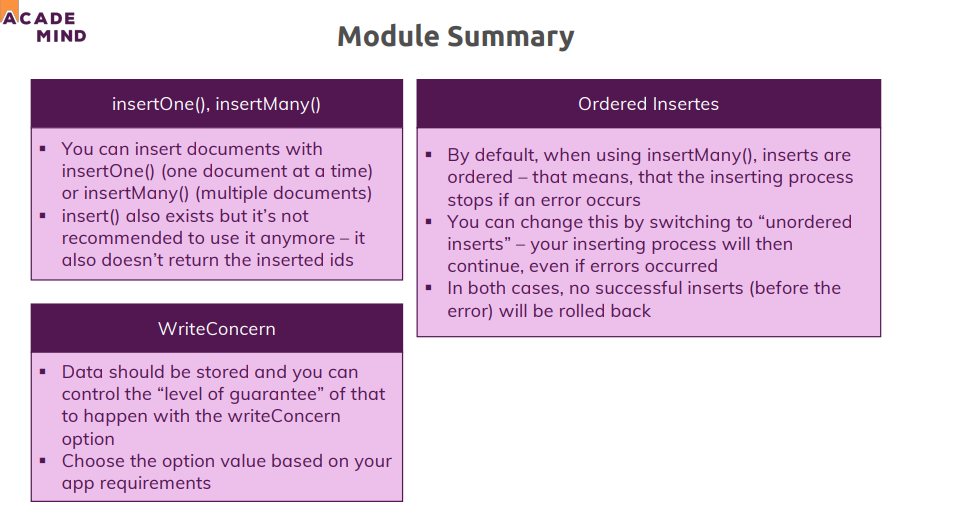
id: 2,

name: 'CBS',

-----------------------------------------

-------------------------------------------------------





**Helpful Articles/ Docs:**

* insertOne(): <https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/>
* insertMany(): <https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/>