
MongoDB

Contents

Introduction to MongoDB	8
Introduction	8
What is MongoDB.....	8
How MongoDB is different than other Databases / How it works	8
The Key MongoDB Characteristics.....	9
No Schema	10
No / Less Relations.....	10
Understanding MongoDB Ecosystem.....	11
Overview of MongoDB	12
Features of mongoDB.....	13
Documents, BSON and Embedding	14
Setting up MongoDB on Windows.....	15
Installing MongoDB	15
Create folder to store the MongoDB database.....	16
Starting the MongoDB server and Connecting using MondoDB Shell	16
Set System Environment Variables.....	18
MongoDB Compass.....	18
Creating Remote Database Hosted on MongoDB Atlas	19
Connecting to the Remote Hosted Database.....	19
Connecting Mongo Shell to the Remote Cluster	23
Shell vs Driver.....	25
Working with MongoDB.....	25
A closer look	26
Understanding the Basics and CRUD Operations.....	27

Understanding Databases, Collections and Documents	27
Default Databases	27
Creating a Local Database.....	27
ObjectId Object	28
JSON vs BSON	29
_id field in the document.....	29
Embedded Documents.....	29
CRUD operations on Documents	30
Creating Documents.....	30
Querying/Reading Documents	31
Updating Documents	33
Deleting Documents.....	34
Summary.....	34
Schemas and Relations: How to Structure Documents	35
Why do we use Schemas	35
Structuring Documents	35
Data Types.....	36
Overview.....	36
Important Notes.....	38
Data Modelling Guidelines	39
MongoDB Data Modelling.....	40
Types of Relationships between Data.....	41
Referencing and Embedding two datasets	42
When to Embed and When to Reference.....	43
Types of Referencing.....	45
Summary	46
Understanding Relations	47
Using "lookup()" for merging reference relations.....	48
Understanding Schema Validation	48
Configuring a Collection	49

Exploring the Shell and the Server	50
MongoDB Server: Finding Available Options	50
Stopping the background MongoDB Server.....	50
MongoDB Server (mongod) Configuration File.....	51
Mongo Shell (mongo.exe) options.....	51
Options After connecting to Shell.....	51
Diving into Create Operations	52
Overview	52
Working with Ordered Inserts	52
Understanding the “writeConcern”	53
Atomicity.....	55
Importing Data	55
Summary.....	56
Read Operations – A Closer Look.....	57
Methods, Filters and Operators	57
Operators – An Overview	57
Query Selectors and Projection Operators	58
Understanding findOne() and find()	58
Working with Comparison operators	58
Querying Embedded Fields and Arrays.....	59
Working with Logical operators.....	59
Working with Element Query Operators	60
Working with Evaluation Query Operators.....	60
Working with Array Query Operators.....	60
Understanding Cursors.....	61
Applying Cursors.....	61
Sorting Cursor Results.....	62
Skipping and Limiting Cursor Results.....	62
Using Projection to shape our Results	62
Update Operations.....	64

Overview	64
updateOne(), updateMany() and using \$set	64
Update Field Operators.....	64
Understanding upsert	64
Updating Arrays	65
Understanding Delete Operations	65
Working with Indexes.....	66
Indexes introduction	66
Adding a Single Field Index.....	67
Indexes Behind the Scenes	68
Understanding Index Restrictions.....	69
Creating Compound Index.....	69
Using Indexes for sorting	70
Understanding the Default index	70
Understanding Partial Index.....	70
Understanding Time-To-Live (TTL) index	71
Query Diagnosis & Query Planning.....	72
Understanding Covered Queries	73
How MongoDB Rejects a Plan.....	73
Clearing the Winning Plan from Cache	74
Using multi-key indexes.....	75
Understanding Text Index.....	75
Text index and sorting	76
Setting Default Language and using Weights.....	76
Building indexes	77
Working with Geospatial Data	78
Adding GeoJSON Data	78
Running GeoJSON Queries.....	78
Adding Geospatial Index to Track the Distance.....	78
Geospatial Query Operators.....	79

Summary.....	79
Understanding the Aggregation Framework.....	80
What is Aggregation Framework.....	80
Docs.....	80
Using the Aggregate Framework	81
Match stage	81
Group stage	81
Sort Stage	81
Transform Stage	81
\$group vs \$project	82
Unwind Stage	82
Bucket Stage.....	82
Using \$sort, \$skip and \$limit stages.....	82
Writing pipeline results into a new Collection with \$out stage	83
Working with Geo Data using \$geoNear stage	83
How MongoDB Optimizes Your Aggregation Pipelines.....	83
Working with Numeric Data	84
Number Types – An Overview	84
MongoDB Shell & Data Types	84
Understanding Programming Language Defaults.....	85
Working with int32	85
Working with int64	85
Doing Maths with Floats int32s and int54s.....	85
What’s wrong with normal doubles	86
Working with Decimal 128.....	86
Important.....	86
MongoDB and Security.....	87
Security Checklist.....	87
Authentication and Authorization.....	87
Transport Encryption.....	87

Encryption at Rest.....	87
Auditing.....	87
Server and Network Config and Setup.....	87
Backups and Software Update.....	88
Understanding Role based Access Control	88
Roles	89
Creating/Update User.....	90
Localhost Exception	91
Built-in Roles	91
User Management Methods Docs:	92
Transport Encryption.....	92
Encryption at Rest	93
Performance, Fault Tolerance and Deployment.....	94
What influences Performance?.....	94
Capped Collections.....	94
Replica Sets.....	95
Why do we replicate Data?	96
Read Performance	97
Howe to create Replica Set	97
Sharding (Horizontal Scaling)	98
How Sharding works?.....	99
Querying with Sharding.....	100
Deploying MongoDB Server	100
MongoDB Atlas.....	101
Transactions	102
What are Transactions	102
How does a Transaction Work?	102
From Shell to Driver	103
Splitting work between Shell and Driver.....	103
MongoDB Drivers.....	103

Course Project.....	103
Introducing Stitch / MongoDB Realm.....	104
What is Stitch	104
Idea of using Stitch.....	106
MongoDB Realm Cloud	106
Tips and Tricks.....	107

Introduction to MongoDB

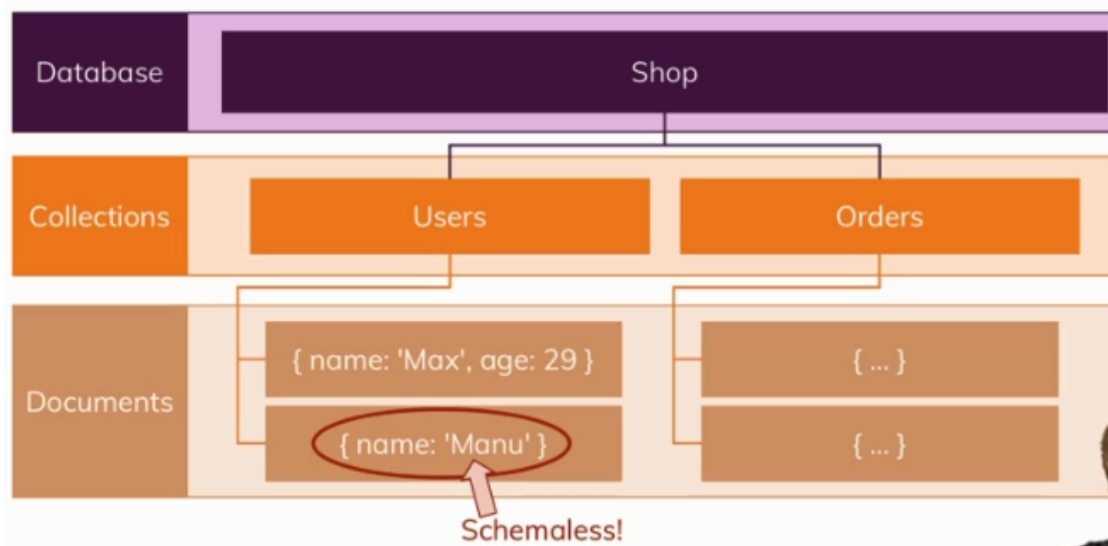
Introduction

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

What is MongoDB

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

How MongoDB is different than other Databases / How it works



- mongodb is most importantly a database server that allows you to then run different databases on it.
- Relational DBs (MySQL, Oracle) have tables, in mongodb we have so-called collections.
- We can have multiple databases and multiple collections per database.
- Now inside of a collection, you have so-called documents which look like JSON objects and indeed this is basically how you store data in mongodb.

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

JSON (BSON) Data Format

```
{
  "name": "Max",
  "age": 29,
  "address":
    {
      "city": "Munich"
    },
  "hobbies": [
    { "name": "Cooking" },
    { "name": "Sports" }
  ]
}
```

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

The Key MongoDB Characteristics

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

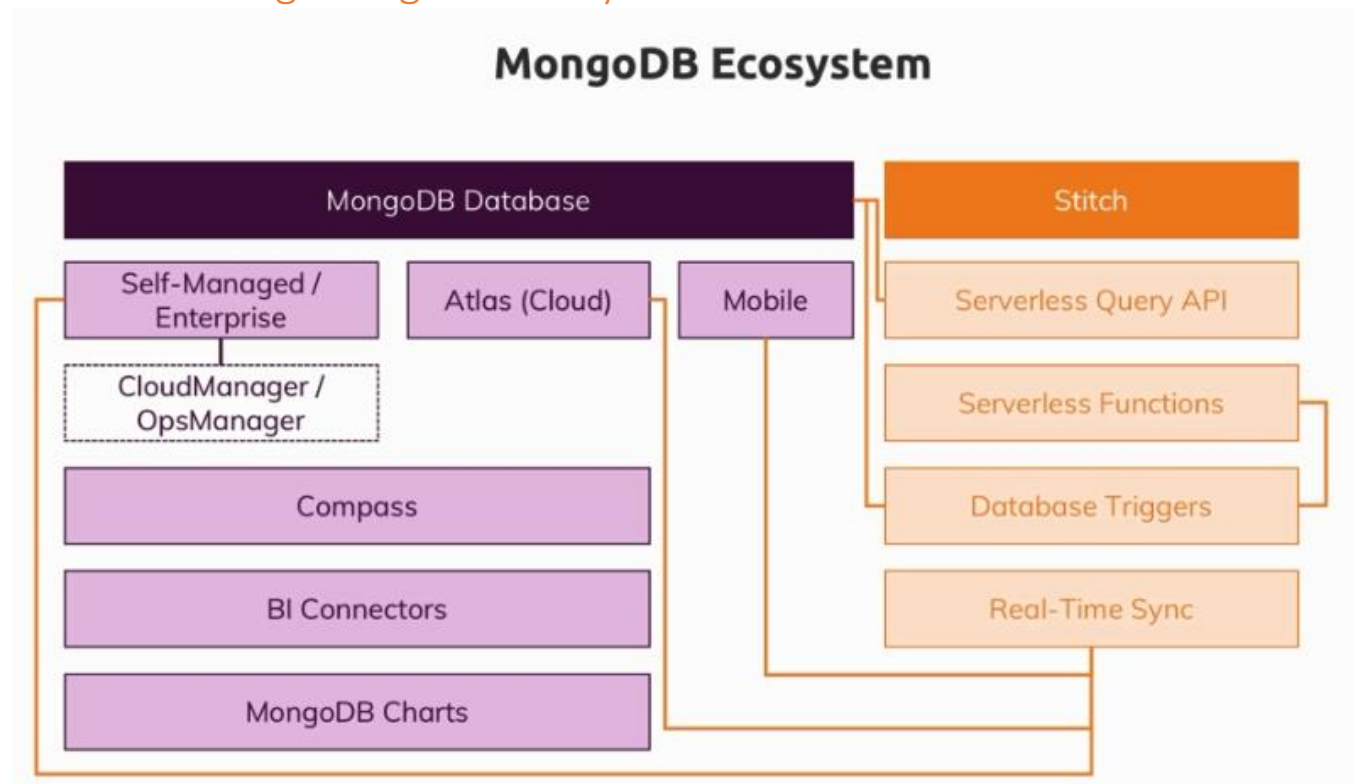
No Schema

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

No / Less Relations

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

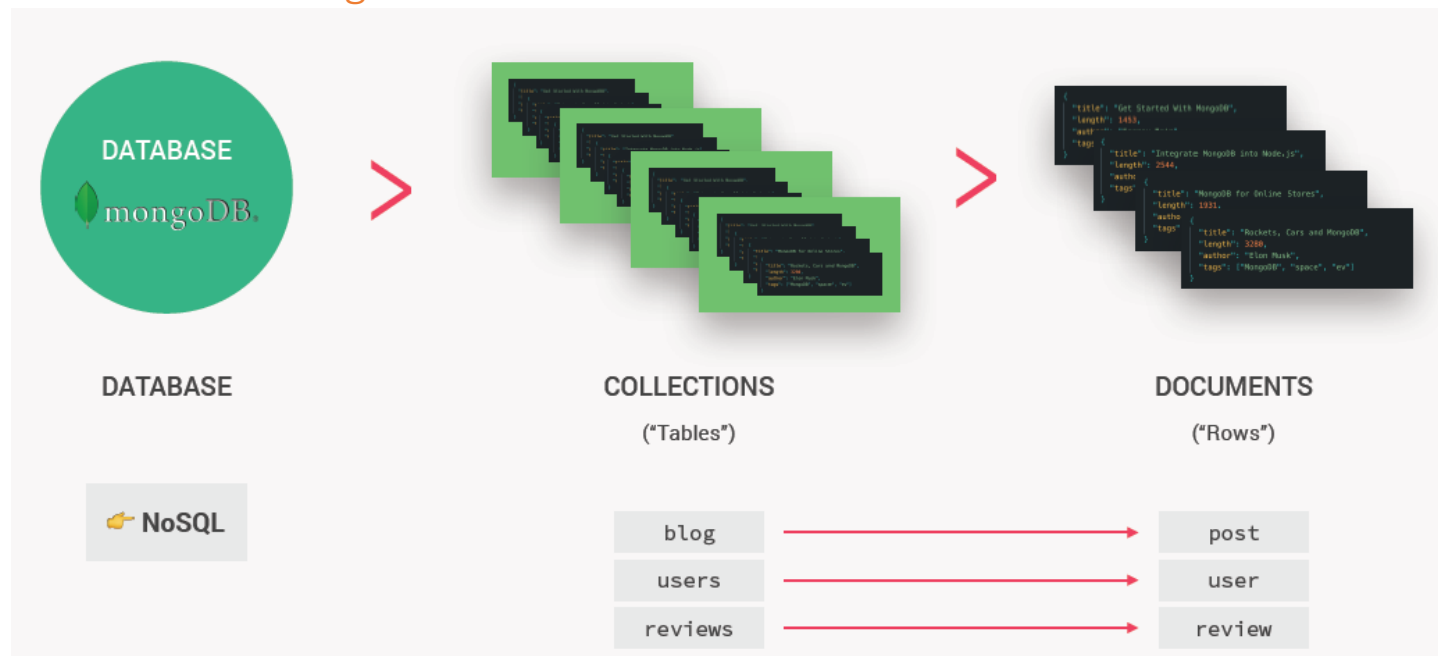
Understanding MongoDB Ecosystem



- The company behind the mongodb database solution is also called mongodb and actually, they have a couple of other products too.
- MongoDB Ecosystem has below systems/features.
- Obviously we got the **mongodb database** which is the core asset, the core feature. There you have a **self-managed and an enterprise solution**.
- Then they have some tools (**CloudManager/OpsManager**) that are related to managing that database that is more of a system admin, database admin task.
- And they have a **cloud solution (Atlas)**, which is an amazing solution for you as a developer, because all the managing a system administrator would have to do is done for you there and you can focus on the data and your logic there.
- They also have a **mobile solution** which basically means you can install mongodb on a mobile device to store data there directly and even work **without an internet connection**.
- They got a graphical user interface (**compass**) which allows you to connect to your database and look into it with a nice user interface.
- And we also get some other tools like BI connectors or Mongodb charts if you're more on the data science area. This is not really something where you will write a lot of code. These are more tools that allow you to connect different analytics tools if you need that.

- Now this is the entire database world and this is of course their core feature hence they have a lot of products there as you can see.
- Now they also got a relatively new offering which is called **stitch** and stitch is basically their **serverless backend solution**. So this is a bit decoupled from the database, still strongly connected but also it offers you more than just data storage. Stitch for example gives you a serverless query API. This is essentially a toolset, a tool which you can use to efficiently query your database directly from inside your client side apps, so from inside your react app or anything like that.
- You also have **serverless functions** which allows you to execute code in the cloud on demand. So now this is totally unrelated to the database, you can execute any javascript code in the cloud there and if you know something like AWS Lambda or Google cloud functions, this is your equivalent.
- We also have **database triggers** which is basically a service that allows you to listen to events in a database, like document was inserted and then execute a function in response to that.
- There is a feature called **real time sync** which basically is built to synchronize a database in a cloud with that mobile offline supporting database you might have on mobile devices.

Overview of MongoDB



- In MongoDB (or just "mongo"), each database can contain one or more collections. A collection is like a table of data.

- Each collection can contain one or more data structures called documents. A document is like a row in a table. So each document contains the data about one single entity, for example, one blog post or one user or one review, etc.
- The collection is like the parent structure that contains all these entities. For example, a blog collection for all posts, users collection or a reviews collection.
- The document has a data format like JSON.

Features of mongoDB

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



MONGODB

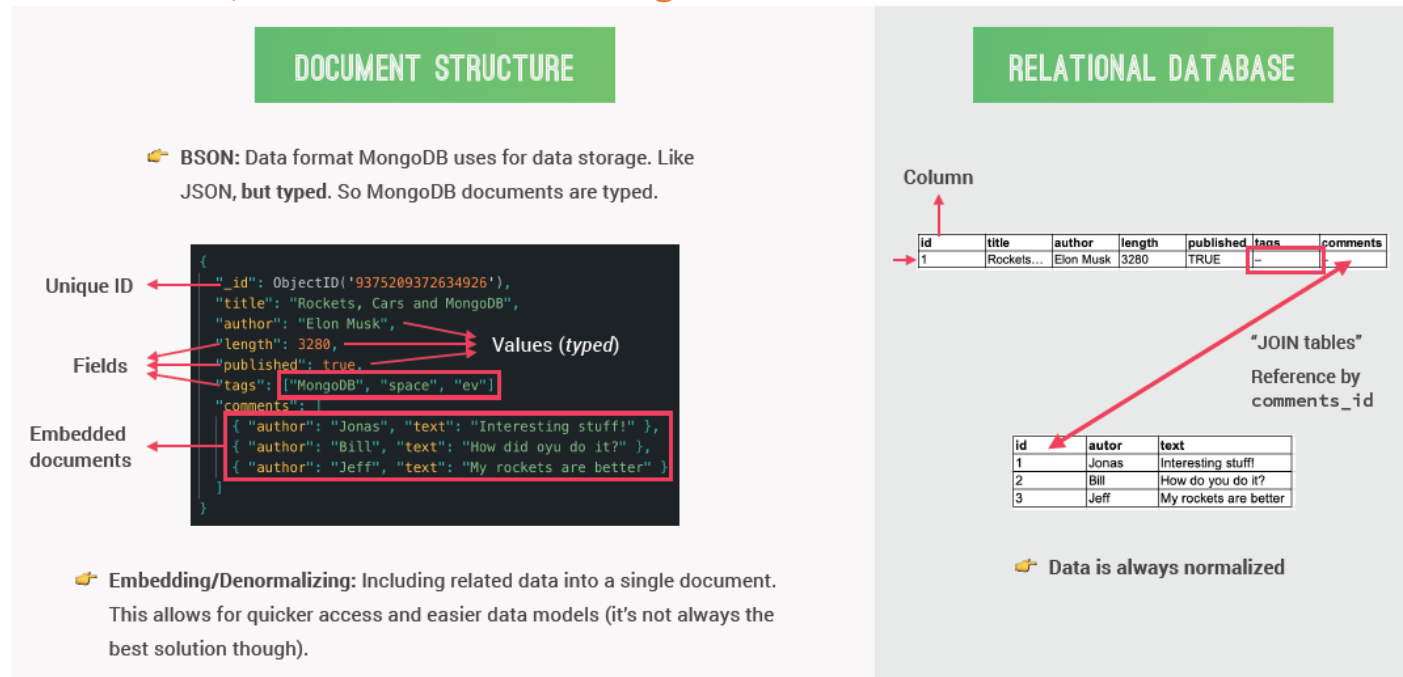
"MongoDB is a document database with the scalability and flexibility that you want with the querying and indexing that you need"

KEY MONGODB FEATURES:



- 👉 **Document based:** MongoDB stores data in documents (field-value pair data structures, NoSQL);
- 👉 **Scalable:** Very easy to distribute data across multiple machines as your users and amount of data grows;
- 👉 **Flexible:** No document data schema required, so each document can have different number and type of fields;
- 👉 **Performant:** Embedded data models, indexing, sharding, flexible documents, native duplication, etc.
- 👉 Free and open-source, published under the SSPL License.

Documents, BSON and Embedding



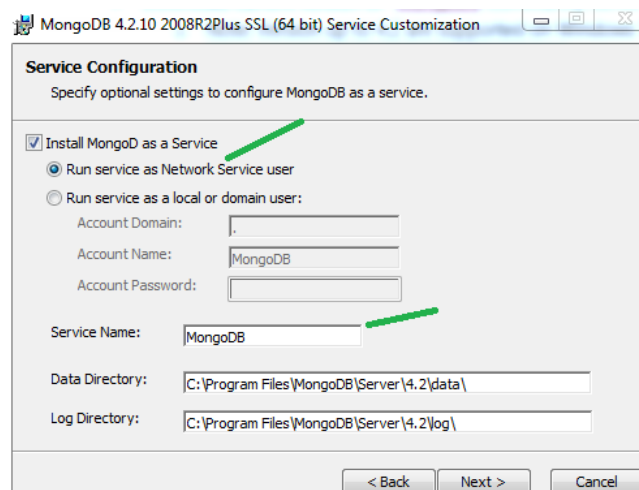
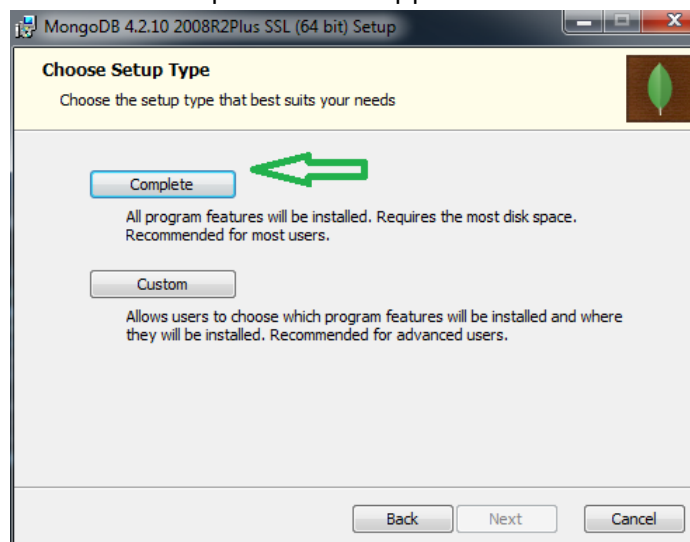
- MongoDB uses a data format similar to JSON for data storage called BSON. It looks basically the same as JSON, but it's **typed**, meaning that all values will have a data type such as string, Boolean, date, integer, double, object or more.
- These BSON documents will also have fields and data is stored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

Setting up MongoDB on Windows

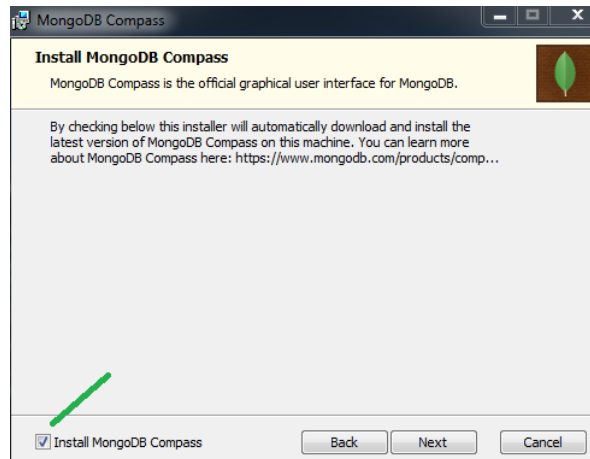
Here we will install the MongoDB server and optionally MongoDB Compass (UI tool), then will create a directory to store the mongodb databases, then will see how to start and connect to the mongodbg server and finally adding the mongodb in System Path Variable so that it can be run from anywhere.

Installing MongoDB

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



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



Create folder to store the MongoDB database

- Now once the MongoDB is installed, we need to create a directory in which MongoDB will store our data.
 - Go to the disk where MongoDB is installed.
 - If let's say it is C: drive, then create folder 'data' under C drive and another folder 'db' under the 'data' folder like this –
C:\data
C:\data\db – This is the place where MongoDB will store our databases.
 - By **default**, MongoDB expects the data and data\db folders directly under root (For windows, it should be directly under C: drive where MongoDB is installed)
 - If you don't have these default folders at this exact same path(C:\data\db), the mongodb server will fail to start.
 - If you want to provide a different path, e.g. say C:\Users\Sameer\data\db, then while starting the MongoDB Server, we need to specify this path as –
mongod -dbpath "C:\Users\Sameer\data\db"
- Few files at MongoDB installed location - C:\Program Files\MongoDB\Server\4.2\bin
 - mongod.exe – To start the MongoDB server.
 - mongo.exe – This is MongoDB Shell through which we can interact with our MongoDB databases and perform DB operations.
 - mongoimport.exe – used for importing data.

Starting the MongoDB server and Connecting using MongoDB Shell

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

- After you run this file, you will see an message in the end – “[listener] waiting for connections on port 27017”

```
C:\Program Files\MongoDB\Server\4.2\bin>mongod.exe
2020-11-04T09:41:05.449+0530 I CONTROL [main] Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --ssl
2020-11-04T09:41:05.914+0530 W ASIO [main] No TransportLayer configured during NetworkInterface startup
2020-11-04T09:41:05.914+0530 I CONTROL [initandlisten] MongoDB starting : pid=5948 port=27017 dbpath=C:\data\db\ 64
2020-11-04T09:41:05.914+0530 I CONTROL [initandlisten] targetMinOS: Windows 7/Windows Server 2008 R2
2020-11-04T09:41:05.914+0530 I CONTROL [initandlisten] db version v4.2.10
2020-11-04T09:41:05.914+0530 I CONTROL [initandlisten] git version: 882762286-97b47a0a6442621342a5217eebd720
2020-11-04T09:41:08.271+0530 I NETWORK [listener] Listening on 127.0.0.1
2020-11-04T09:41:08.272+0530 I NETWORK [listener] waiting for connections on port 27017
2020-11-04T09:41:09.001+0530 I SHARDING [initandlisten] marking collection local.oplog.rs as collection version: <unsharded>
```

- MongoDB by default runs on port 27017. If you want to change this, you need to start the server with ‘port’ flag like this –
`>mongod --port 27018`
- So we’ve basically started a server and now we need a shell to connect to the server to be able to manipulate our databases (Create, Delete, Query, etc.).
- Note:
 - On windows, you may not have to start the mongodb server (mongod) manually because on windows it is running as a “Service” so it is anyways running in the background. However if you get error while connecting to Shell (below), then you can manually start the server.
 - To stop the background MongoDB server which is running as a service, execute –
`net stop MongoDB`
 To start the MongoDB server as a service in background, execute this –
`net start MongoDB`

```
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> net stop MongoDB
The MongoDB Server (MongoDB) service is stopping.
The MongoDB Server (MongoDB) service was stopped successfully.

PS C:\Windows\system32>
PS C:\Windows\system32> net start MongoDB
The MongoDB Server (MongoDB) service is starting..
The MongoDB Server (MongoDB) service was started successfully.

PS C:\Windows\system32>
```

- To run the **Shell**, we need to run the mongo.exe file
`C:\Program Files\MongoDB\Server\4.2\bin>mongo`
- After running this command, you can see that we’ve automatically connected to the default port (27017). And you will see the shell to interact with the MongoDB database.

```
C:\Program Files\MongoDB\Server\4.2\bin>mongo.exe
MongoDB shell version v4.2.10
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : "UUI0C687431ca-f398-47e6-819a-123277767d54" }
MongoDB server version: 4.2.10
Welcome to the MongoDB shell.
```

- If you have used a different port to run MongoDB server, then you have to specify same port explicitly while running the Shell. E.g. `>mongo --port 27018`

- To 'double' test the connection, we can run the db command and you should see output as 'test' which is the default database we get.

```
> db
```

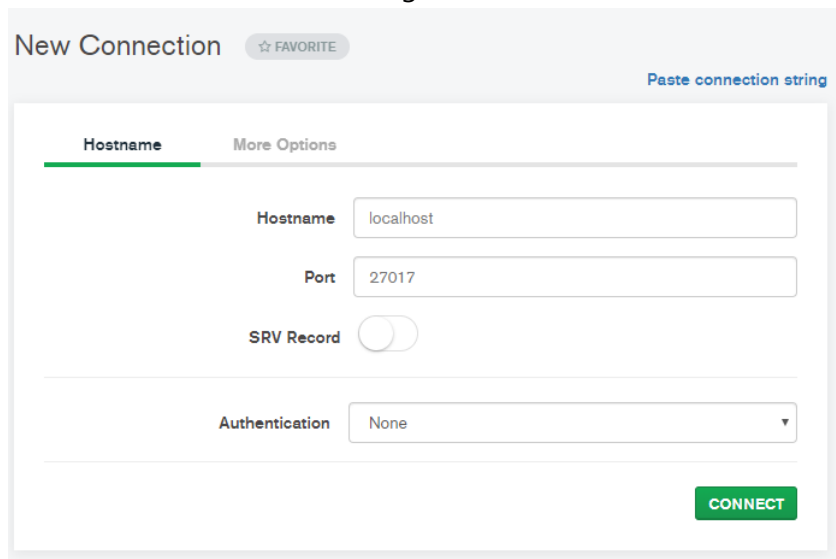
```
Test
```

Set System Environment Variables

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

MongoDB Compass

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



The screenshot shows the 'New Connection' dialog in MongoDB Compass. At the top, there's a 'New Connection' title and a '☆ FAVORITE' button. A link 'Paste connection string' is on the right. Below is a tabbed interface with 'Hostname' selected. The 'Hostname' tab contains fields for 'Hostname' (set to 'localhost'), 'Port' (set to '27017'), and a toggle for 'SRV Record' (currently off). Below these is an 'Authentication' dropdown menu set to 'None'. A green 'CONNECT' button is at the bottom right.

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

Creating Remote Database Hosted on MongoDB Atlas

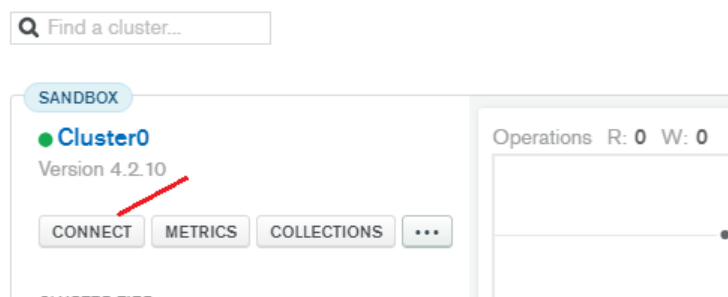
- We can either use local database (hosted on our machine) or use a remote database hosted on a service called Atlas, which is actually owned by the same company that involves MongoDB.
- Go to MongoDB website (<https://www.mongodb.com>) and select product MongoDB Atlas or you can use this link directly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

Connecting to the Remote Hosted Database

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

[SAMEER'S INDIVIDUAL ACCOUNT > NATOURS](#)

Clusters



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

1 Add a connection IP address

Add Your Current IP Address

Add a Different IP Address

Allow Access from Anywhere

1 Add a connection IP address

IP Address	Description (Optional)
<input type="text" value="49.35.215.155"/>	<input type="text" value="Sameer's Windows 7 Personal Laptop"/>
<div>Cancel Add IP Address</div>	

- Then, we need to create a database user.

2 Create a Database User

This first user will have [atlasAdmin](#) permissions for this project.
Keep your credentials handy, you'll need them for the next step.

Username	Password
<input type="text" value="sameer"/>	<div><input type="text" value="6hq8UJ5xORPHxCWE"/> Autogenerate Secure Password HIDE</div>
<div>Create Database User</div>	

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

```
DATABASE_PASSWORD=6hq8UJ5xORPHxCWE
```

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

- Next, we need to choose a connection method.




Choose a connection method

Select 'Connect using MongoDB Compass'

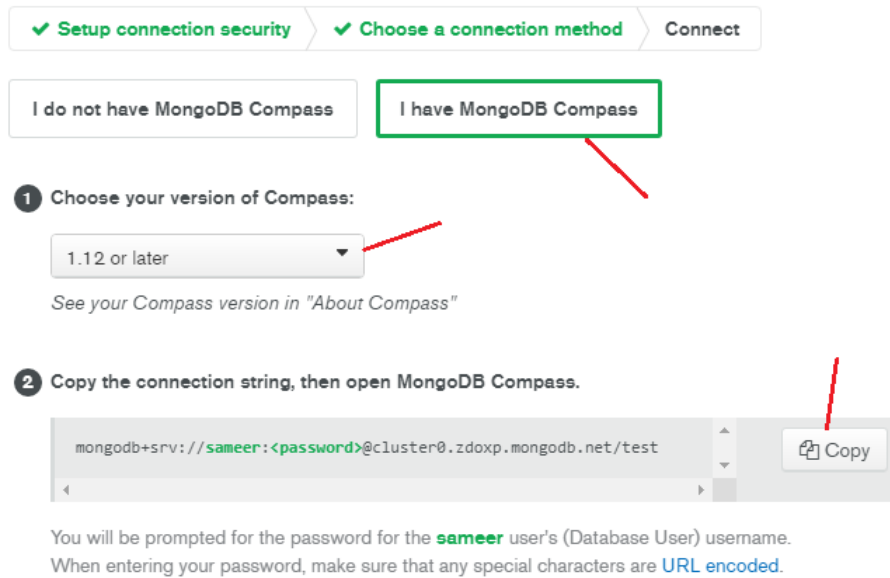
✓ Setup connection security Choose a connection method Connect

Choose a connection method [View documentation](#)

Get your pre-formatted connection string by selecting your tool below.

	Connect with the mongo shell Interact with your cluster using MongoDB's interactive Javascript interface	>
	Connect your application Connect your application to your cluster using MongoDB's native drivers	>
	Connect using MongoDB Compass Explore, modify, and visualize your data with MongoDB's GUI	>

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



✓ Setup connection security ✓ Choose a connection method Connect

I do not have MongoDB Compass I have MongoDB Compass

1 Choose your version of Compass:

1.12 or later

See your Compass version in "About Compass"

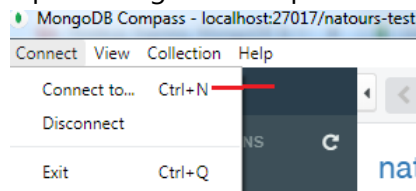
2 Copy the connection string, then open MongoDB Compass.

mongodb+srv://sameer:<password>@cluster0.zdoxp.mongodb.net/test

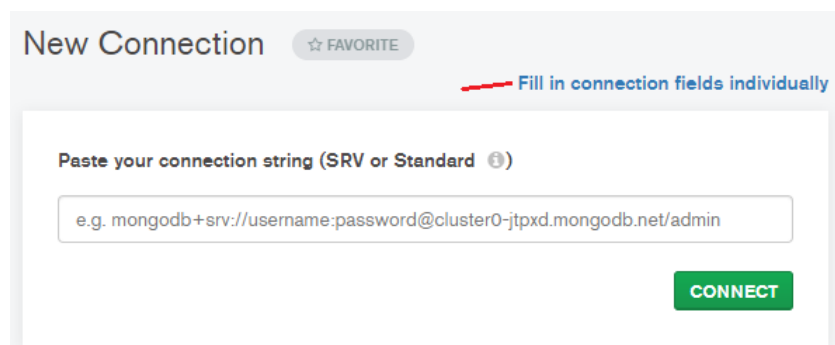
Copy

You will be prompted for the password for the sameer user's (Database User) username.
When entering your password, make sure that any special characters are URL encoded.

- Open MongoDB Compass and navigate to



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



New Connection ☆ FAVORITE

Fill in connection fields individually

Paste your connection string (SRV or Standard ⓘ)

e.g. mongodb+srv://username:password@cluster0-jtpxd.mongodb.net/admin

CONNECT

Hostname

More Options

Hostname

cluster0.zdoxp.mongodb.net

SRV Record

☒

Authentication

Username / Password

Username

sameer

Password

.....

Authentication Database

admin

CONNECT

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

Atlas

Realm

Charts

Cluster0

Overview

Real Time

Metrics

Collections

Profiler

Performance Ad

DATABASES: 1 COLLECTIONS: 1

+ Create Database

Q NAMESPACES

natours

tours

natours.tours

COLLECTION SIZE: 80B TOTAL DOCUMENTS: 1 INDEXES TOTA

Find Indexes Schema Anti-Patterns

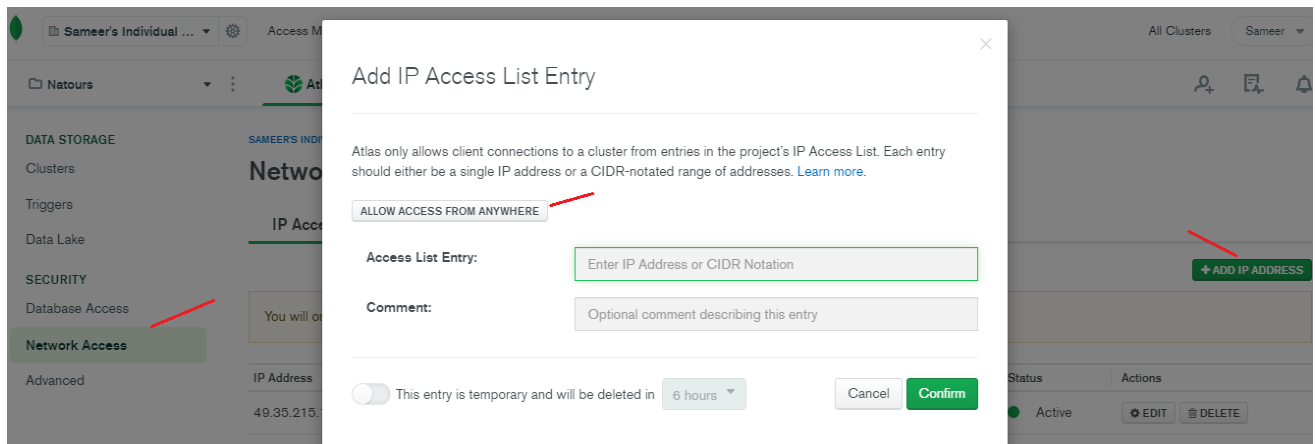
FILTER {"filter": "example"}

QUERY RESULTS 1-1 OF 1

_id: ObjectId("5fa2b672d0e06c9e10c2bcd8")
name: "The Forest Hiker"
price: 297
rating: 4.7

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

- For this, do this –



- We can update this setting later on, if we want

SAMEER'S INDIVIDUAL ACCOUNT > NATOURS

Network Access

IP Access List Peering Private Endpoint

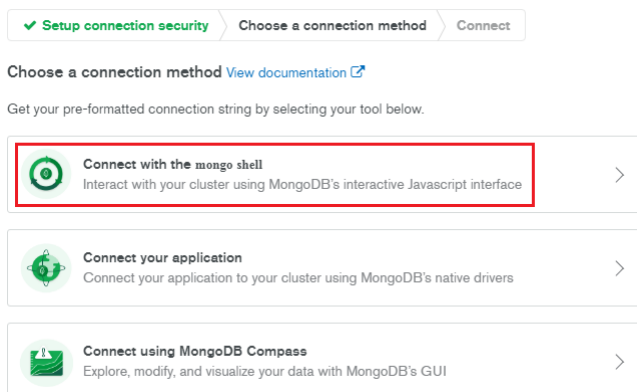
+ ADD IP ADDRESS

You will only be able to connect to your cluster from the following list of IP Addresses:

IP Address	Comment	Status	Actions
49.35.215.155/32 (includes your current IP address)	Sameer's Windows 7 Personal Laptop	Active	EDIT DELETE
0.0.0.0/0 (includes your current IP address)		Active	EDIT DELETE

Connecting Mongo Shell to the Remote Cluster

- Go to Cluster -> Connect -> Select 'Connect using the mongo shell'



- Select appropriate option and copy the connection string

✓ Setup connection security ✓ Choose a connection method Connect

I do not have the mongo shell installed **I have the mongo shell installed**

1 Select your mongo shell version

4.2

(To check your shell version, run `mongo --version`)

2 Run your connection string in your command line

Use this connection string in your application:

```
mongo "mongodb+srv://cluster0.zdoxp.mongodb.net/<dbname>" --username
```

Copy

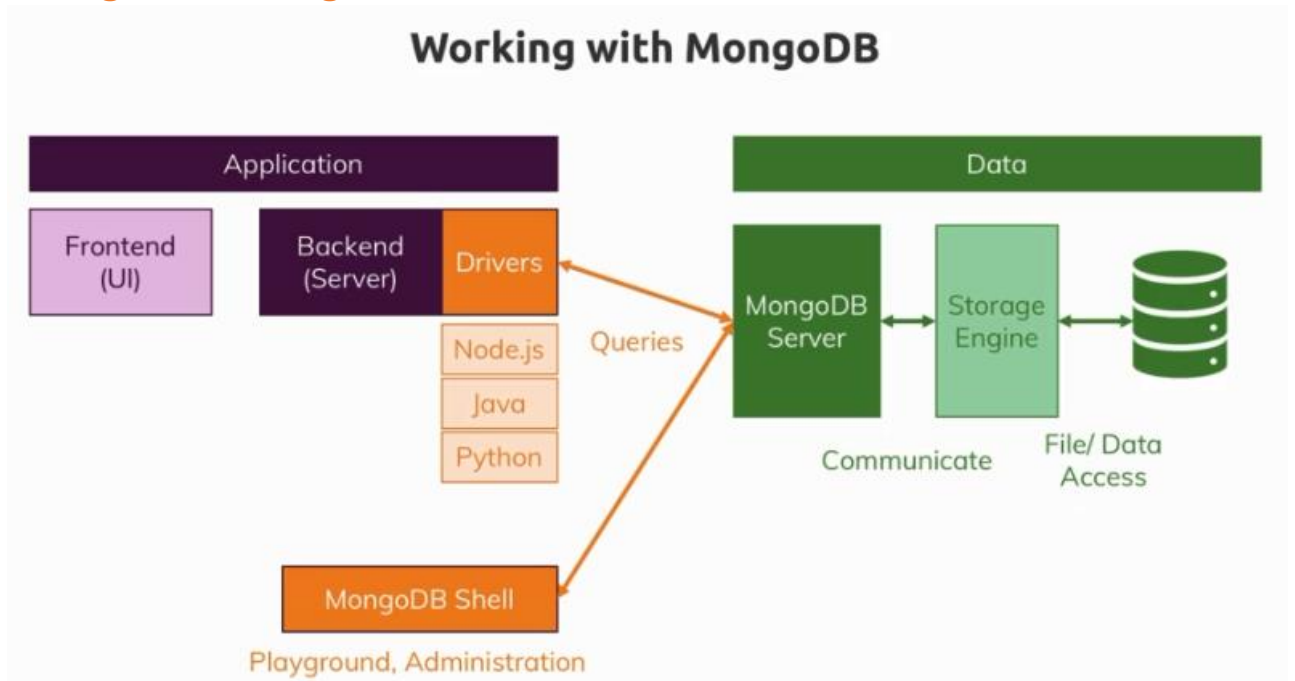
- Open New command prompt and paste the copied text. This text is the complete command to connect to our cluster.
- Now replace the text `<dbname>` with your actual database name .e.g `natours`. And hit enter.
E.g.

```
>mongo "mongodb+srv://cluster0.zdoxp.mongodb.net/natours" --username sameer
```
- Then it will ask for the password. Copy the database password which we created above and paste it and hit enter again.
- You should be connected now. To confirm it, enter command `show dbs`. You should see your DB.
MongoDB Enterprise atlas-kcxepu-shard-0:PRIMARY> show dbs
admin 0.000GB
local 1.252GB
natours 0.000GB
- Now we are all set to go ahead.

Shell vs Driver

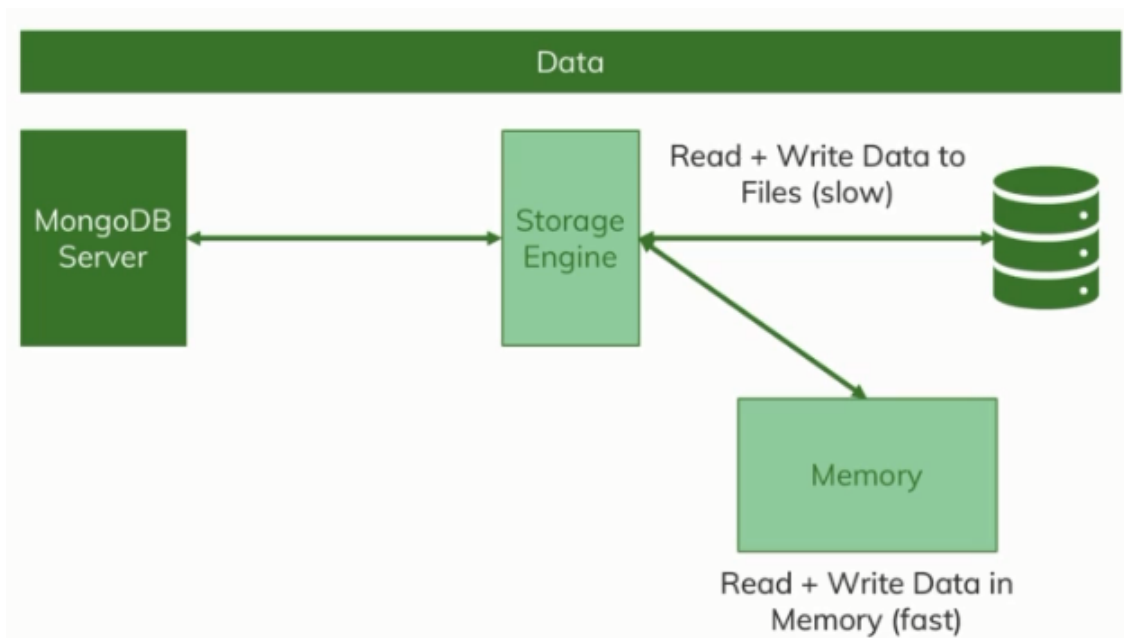
- Use Shell, to see all the commands and the different ways of using these commands because the shell is a great neutral ground for working with mongodb.
- Drivers are packages that you install/use for the different programming languages your app might be written in, and these drivers are then your bridges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

Working with MongoDB



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

A closer look



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

Understanding the Basics and CRUD Operations

Understanding Databases, Collections and Documents

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

Default Databases

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

Creating a Local Database

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

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

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

```
      "insertedId" : ObjectId("5fa234442873acd05244dc3b")
    }
  }
```

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

```
> db.tours.find()
{ "_id" : ObjectId("5fa234442873acd05244dc3b"), "name" : "The Forest Hiker", "price" : 297, "rating" : 4.7 }
```
- As you can see, mongodb works with the Javascript syntax and JSON, which is one of the main reasons why MongoDB is so popular for Node JS applications.
- To see all the databases in mongodb, use command `show dbs`.

```
> show dbs
admin                0.000GB
config               0.000GB
local                0.000GB
natours-test        0.000GB
```
- To see collections inside a database, use `show collections`.

```
tours
```
- To exit/quit the mongo shell, use `quit()` operator.

ObjectId Object

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

JSON vs BSON

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

_id field in the document

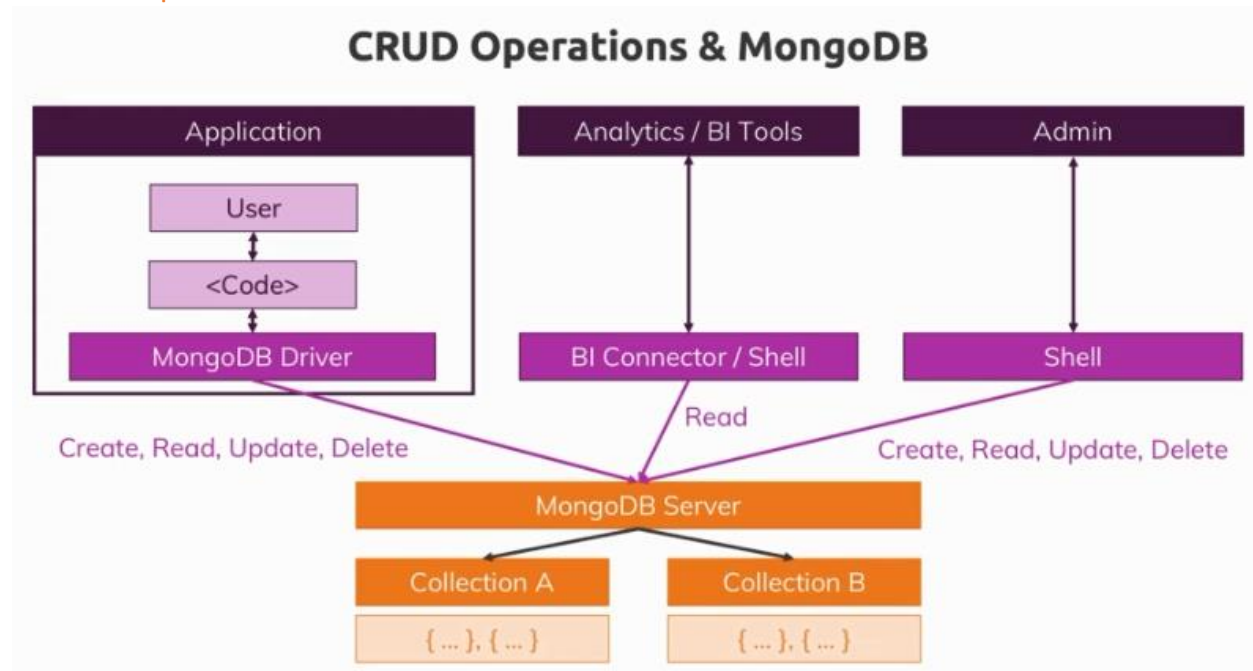
- When we insert documents, we get "_id" as an auto-generated field.
- We don't have to use the auto-generated ID, we just have to ensure that we have a unique ID and if we can ensure this, we can also assign IDs on our own.
- Each document must have unique key.
- E.g.

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

Embedded Documents

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

CRUD operations on Documents



Creating Documents

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

E.g.

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

```
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("5fa294d58251e593f8ea0992"),
    ObjectId("5fa294d58251e593f8ea0993")
  ]
}
```

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

Querying/Reading Documents

Theory

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

Examples

- To query for all the documents in a certain collection, use `find()` operator without any arguments.
E.g.
`> db.tours.find()`
- To query for a certain document with some filtering criteria, just pass the filtering criteria as a BSON object (filter object) to the `find()` operator.
E.g.
`> db.tours.find({ name: "The Forest Hiker" })`
`> db.tours.find({ difficulty: "easy" })`
- We can also use some special query operators for more granular filtering. E.g. less than (`$lt`), greater than (`$gt`), less than or equal to (`$lte`), greater than or equal to (`$gte`).
E.g.
`> db.tours.find({ price: { $lte: 500 } })`
Here we want to get all documents whose price is less than or equal to 500.

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

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


Here we want to get all documents whose price is less than to 500 AND rating greater than or equal to 4.8.
- We can also use OR criteria to query the documents, i.e. to get documents satisfying either of the conditions. For this, we use \$or operator, which takes an array of conditions.
E.g.

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


Here we want to get documents having price less than 500 OR having rating greater than or equal to 4.8
- Besides our filter object, we can also pass in an object for **projection**. Projection means we simply want to select some of the fields in the output.
- For projections, we need to 0 or 1 for the document fields. **0 means exclusion, 1 means inclusion**. We cannot mix both 0 and 1 for different fields in the same query.
- E.g. If we want only name from the above query,

```
> db.tours.find({ $or: [ { price: { $lt: 500 } }, { rating: { $gte: 4.8 } } ] }, { name: 1 })
```


Here we want to get documents (with names only) having price less than 500 OR having rating greater than or equal to 4.8
- E.g. 2) If you want all fields except 'name'

```
> db.tours.find({ $or: [ { price: { $lt: 500 } }, { rating: { $gte: 4.8 } } ] }, { name: 0 })
```
- E.g. 2) If you want all fields except 'name' and 'price'

```
> db.tours.find({ $or: [ { price: { $lt: 500 } }, { rating: { $gte: 4.8 } } ] }, { name: 0, price: 0 })
```
- We cannot mix both 0 and 1 for different fields in the same query.
E.g.

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


Error: error: {
 "ok" : 0,
 "errmsg" : "Projection cannot have a mix of inclusion and exclusion.",
 "code" : 2,
 "codeName" : "BadValue"
}
- By default, "_id" field is always returned even with projections. We have to explicitly mention in the projection if we don't want it by setting "_id": 0

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


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

```
> db.flightData.find({"status.description": "on-time"}).pretty()
{
  "_id" : ObjectId("5ff856d1c1bd95b8458e8602"),
  "departureAirport" : "MUC",
  "arrivalAirport" : "SFO",
  "aircraft" : "Airbus A380",
  "status" : {
    "description" : "on-time",
    "lastUpdated" : "1 hr ago"
  }
}
```

- **Querying arrays.**

```
> db.passengers.find({"hobbies": "sports"}).pretty()
{
  "_id" : ObjectId("5ff9723ef82ed11a0231da5b"),
  "name" : "Albert Twostone",
  "age" : 68,
  "hobbies" : [
    "sports",
    "cooking"
  ]
}
```

Updating Documents

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

```
> db.tours.updateOne({ name: "The Snow Adventurer" }, { $set: { price: 597 } })
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

 Here the first argument is filter object ({ name: "The Snow Adventurer" }, and the second argument is to set price of 597 using the \$set operator.
- In case of updateOne(), if the filter query returns more than one documents, then only first document will be updated. So in order to update multiple documents, use updateMany()
- E.g.

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

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

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

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

Deleting Documents

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

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

Here we delete all documents having rating less than 4.8
- To delete all the documents in a collection, pass empty filtering object like this

```
> db.tours.deleteMany({})
```

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

Summary

Module Summary

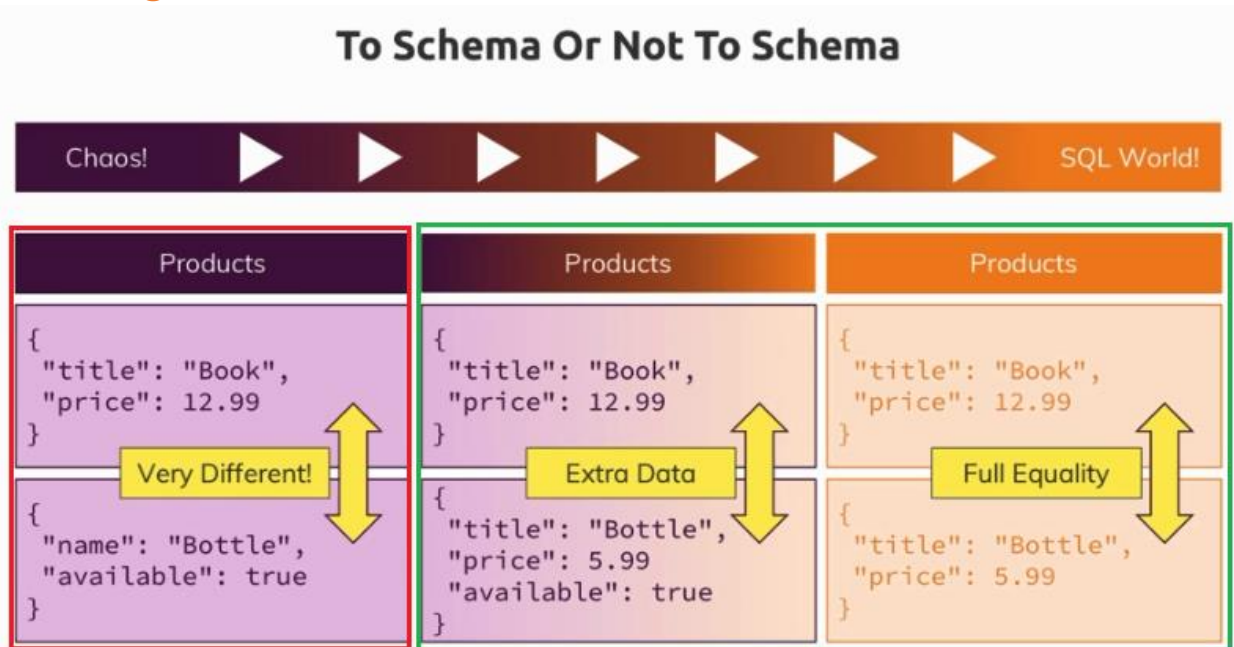
Databases, Collections, Documents	CRUD Operations
<ul style="list-style-type: none">▪ A Database holds multiple Collections where each Collection can then hold multiple Documents▪ Databases and Collections are created "lazily" (i.e. when a Document is inserted)▪ A Document can't directly be inserted into a Database, you need to use a Collection!	<ul style="list-style-type: none">▪ CRUD = Create, Read, Update, Delete▪ MongoDB offers multiple CRUD operations for single-document and bulk actions (e.g. <code>insertOne()</code>, <code>insertMany()</code>, ...)▪ Some methods require an argument (e.g. <code>insertOne()</code>), others don't (e.g. <code>find()</code>)▪ <code>find()</code> returns a cursor, NOT a list of documents!▪ Use filters to find specific documents
Document Structure	Retrieving Data
<ul style="list-style-type: none">▪ Each document needs a unique ID (and gets one by default)▪ You may have embedded documents and array fields	<ul style="list-style-type: none">▪ Use filters and operators (e.g. <code>\$gt</code>) to limit the number of documents you retrieve▪ Use projection to limit the set of fields you retrieve

Schemas and Relations: How to Structure Documents

Why do we use Schemas

- MongoDB does not enforce any schemas. Documents don't have to use the same Schema inside of one Collection. So you can mix different documents in a collection but in reality, you will probably have some kind of schema.
- Schema simply means the structure of a 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.

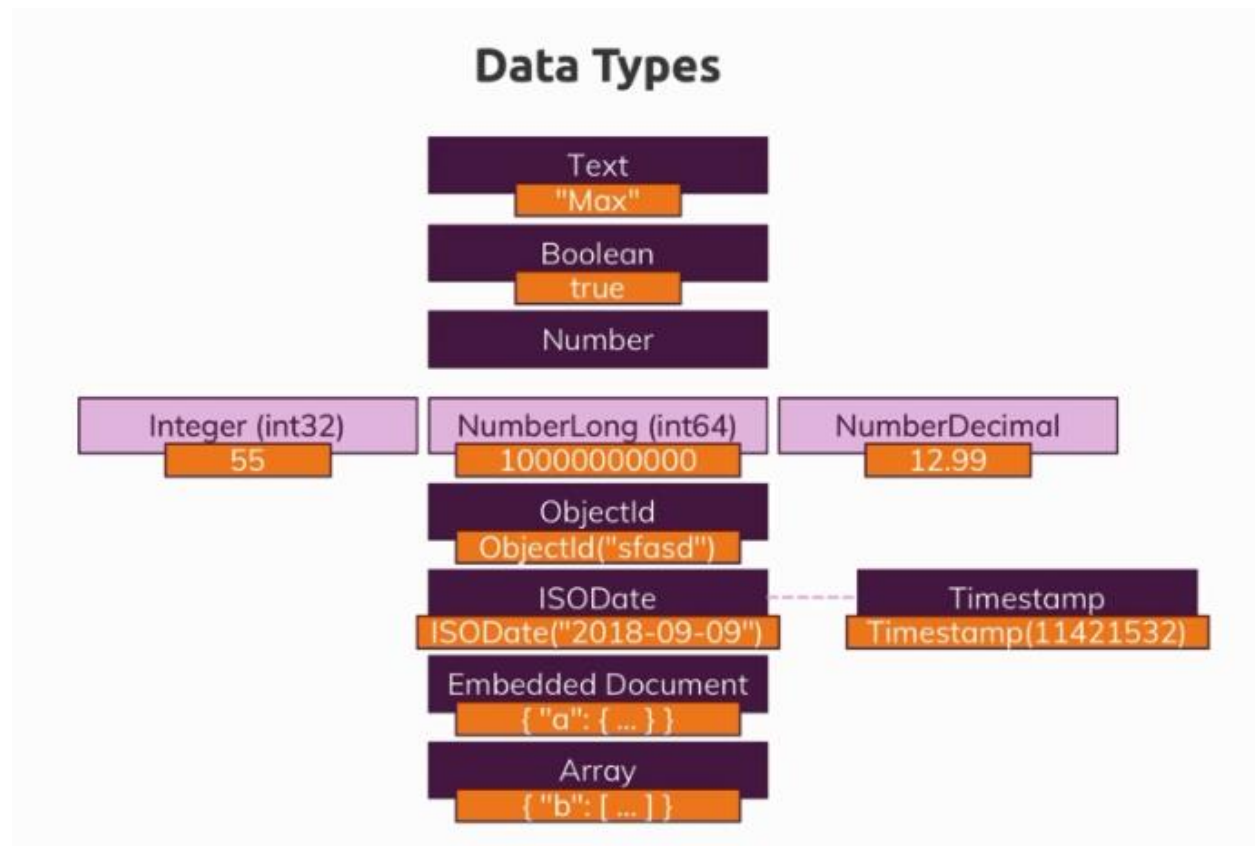
Structuring Documents



- In reality, I guess either the approach on the right or most often the one in the middle is what you will see because there, you use the best of both worlds. You have some structure because your usecase or application probably needs it but you also take advantage of the flexibility mongodb gives you so that you can store extra information.

Data Types

Overview



Text

- Text always uses quotation marks around the value
- There is no hard limitation here, the only limitation you have regarding the maximum amount of characters in your text are the 16 MB overall size for the whole document.

Boolean

- true or false

Number

- Actually, we got a couple of different numbers in mongodb.
- We got **NumberInt** (int32). int32 integers are integers that are 32bits long and therefore if you try to store values that are longer than that, they will overflow that range and you will end up with a different number.
- For longer numbers, mongodb has a an int64 integer (**NumberLong**) which you can also assign to store values.
- In the shell by the way if you just enter a normal value, it is treated as a float value, that is because the normal shell is based on javascript and javascript doesn't differentiate

between integers and floating point values, therefore everything will be stored as a 64bit float value in the shell, that is the default value.

- Be aware that mongodb is able to store smaller integers and bigger integers, the solution you choose here of course kind of defines how much space will be allocated and therefore how much space will be eaten up.
- And you also can store decimal numbers, you also got a special type, **NumberDecimal** and that is provided by mongodb **to store high precision floating point values** because normal floating point values also called doubles are basically rounded, so they're not super precise after their decimal place.
- For a lot of use cases, normal floats are enough though, but if you're doing scientific calculations or anything like that, you might need a very high precision. This NumberDecimal offers this, a very high decimal place precision where you got 34 decimal places, so after the comma which are not rounded but guaranteed to be correct and precise.
- It's simply a computing limitation that we got no 100% precision all the time but that rounding occurs.

ObjectId

- The ObjectId object is a special object automatically generated by mongodb to give you a unique ID which is not just a unique random string but also a string that contains a temporal component, so that if you create two elements after each other, two documents after each other, you are guaranteed to have the right order due to that ID because the older element will have an ID that comes prior to the other one, so there is this sorting built into the ObjectId because it respects a timestamp.

Date and Timestamp

- ISODate – to store Date. E.g. ISODate("2021-01-11T06:05:49.188Z")
- Timestamp – The timestamp is mostly used internally, you can create it automatically, mostly you let mongodb create that for you and that is guaranteed to be unique too. So even if you create two documents at the same time, they will not have exactly the same timestamp because it will basically take into account the current time and then also add an ordinal value, so that two documents at the same time still don't get the same time stamp but respect the order in which the insert command was issued.
- E.g. Timestamp(1610345149, 1) => here first number is current timestamp in milliseconds and second argument is ordinal value so that two documents at the same time still don't get the same timestamp.
- ObjectId is the kind of based on the timestamp and then just uses some algorithm to spit out a seemingly random string.

Embedded documents

- You can embed documents and these documents in turn can embed other documents.

Arrays

- Arrays are simply lists of values,
- You can have a list of strings, a list of booleans, a list of numbers or a list of other embedded documents or even a list of lists, a list of arrays.

Important Notes

- For the data types, MongoDB supports
Refer – <https://docs.mongodb.com/manual/reference/bson-types/>
- MongoDB Limits and Thresholds
Refer – <https://docs.mongodb.com/manual/reference/limits/>
- 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.

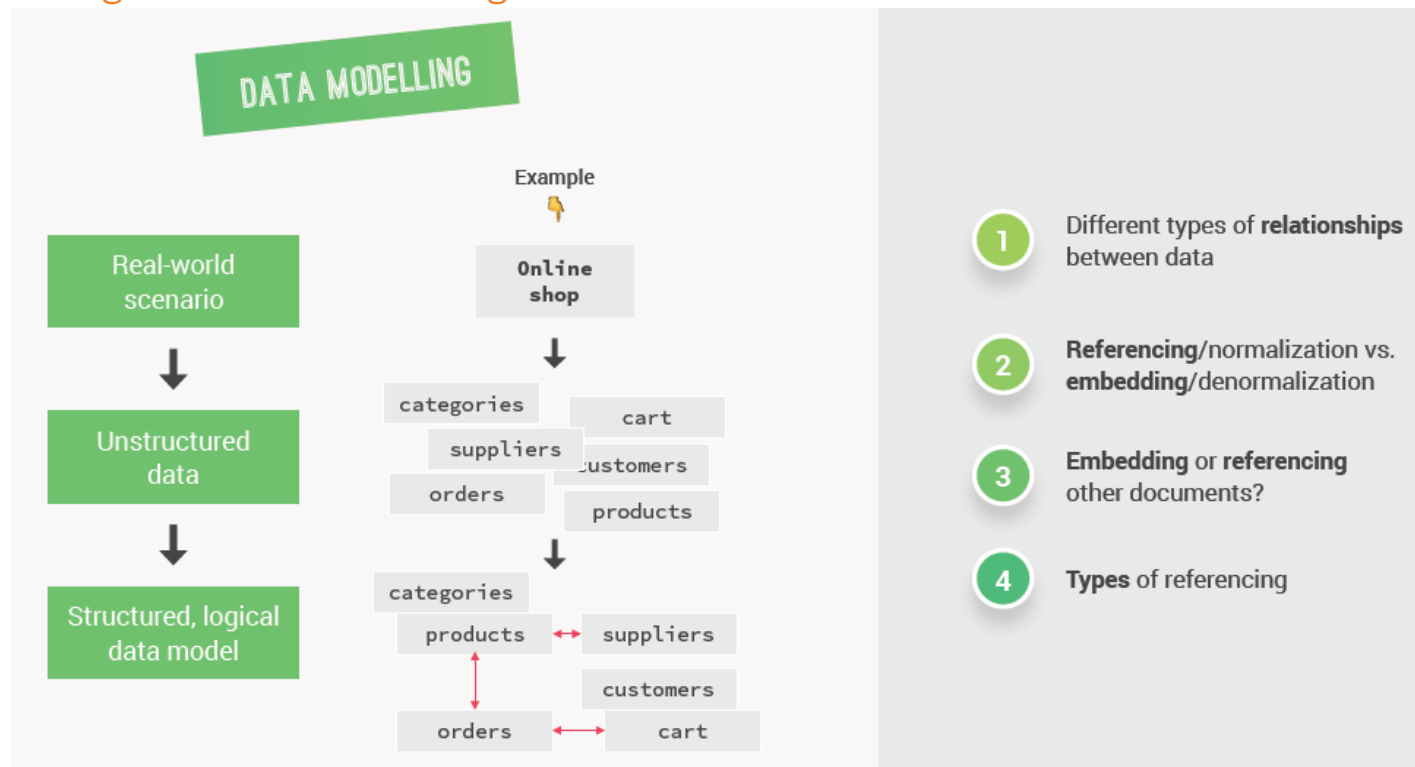
Data Modelling Guidelines

Data Schemas & Data Modelling

Which Data does my App need or generate?	User Information, Product Information, Orders, ...	Defines the Fields you'll need (and how they relate)
Where do I need my Data?	Welcome Page, Products List Page, Orders Page	Defines your required collections + field groupings
Which kind of Data or Information do I want to display?	Welcome Page: Product Names; Products Page: ...	Defines which queries you'll need
How often do I fetch my data?	For every page reload	Defines whether you should optimize for easy fetching
How often do I write or change my data?	Orders => Often Product Data => Rarely	Defines whether you should optimize for easy writing

- mongodb really embraces that idea of planning your data structure based on the way you'll retrieve your data, so that you don't have to do complex joins but that you can retrieve your data in the format or almost in the format you need it in your application.

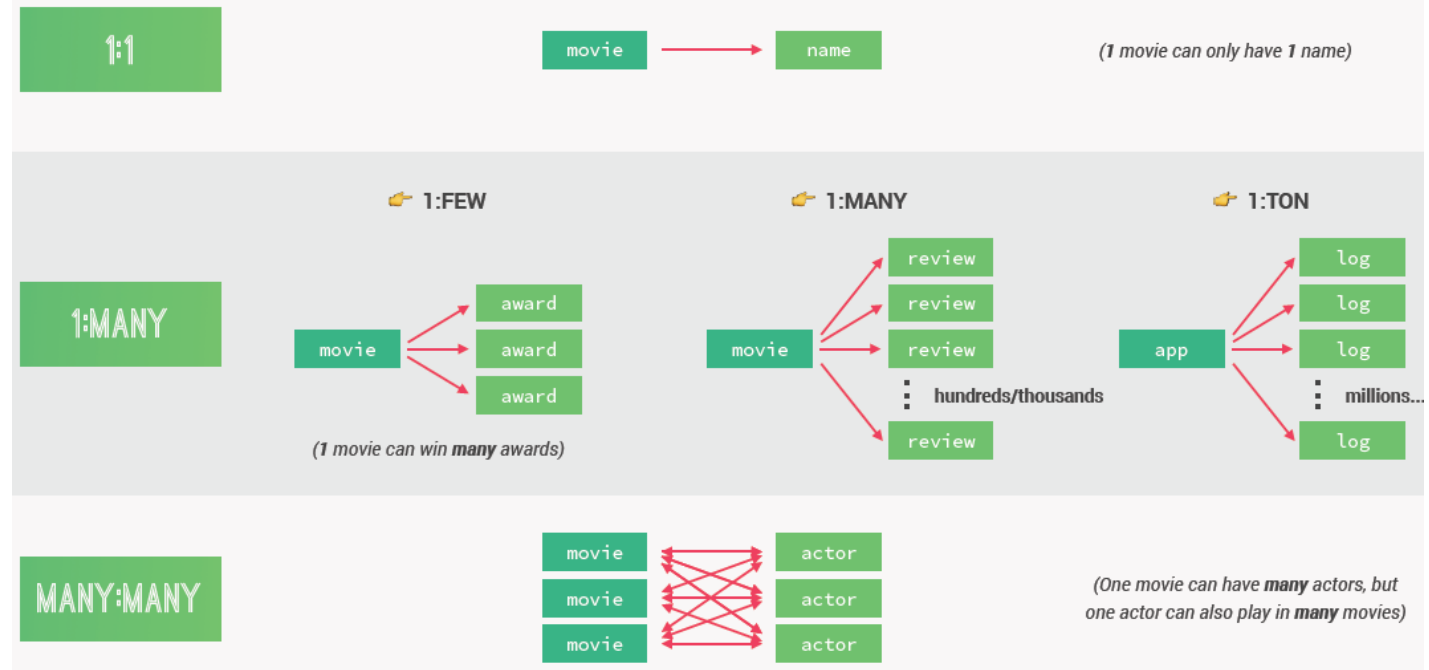
MongoDB Data Modelling



- One of the most important steps in building data intensive apps is to actually model all this data in MongoDB.
- **Data Modeling** is the process of taking unstructured data generated by a real world scenario and then structure it into a logical data model in a database.
- It's the most demanding part of building an entire application. Because it really is not always straight-forward. And sometimes there are simply no right answers. So not just one unique correct way of structuring the data.
- There are 4 steps in the process of doing data modelling
 1. Identify different types of relationships between data.
 2. Understand the difference between referencing or normalization and embedding or denormalization.
 3. Deciding whether we should embed documents or reference to other documents based on a couple of different factors.
 4. Types of referencing.
- The way we design data so the way we model our data can make or break our entire application.
- Let's go through the steps one by one

Types of Relationships between Data

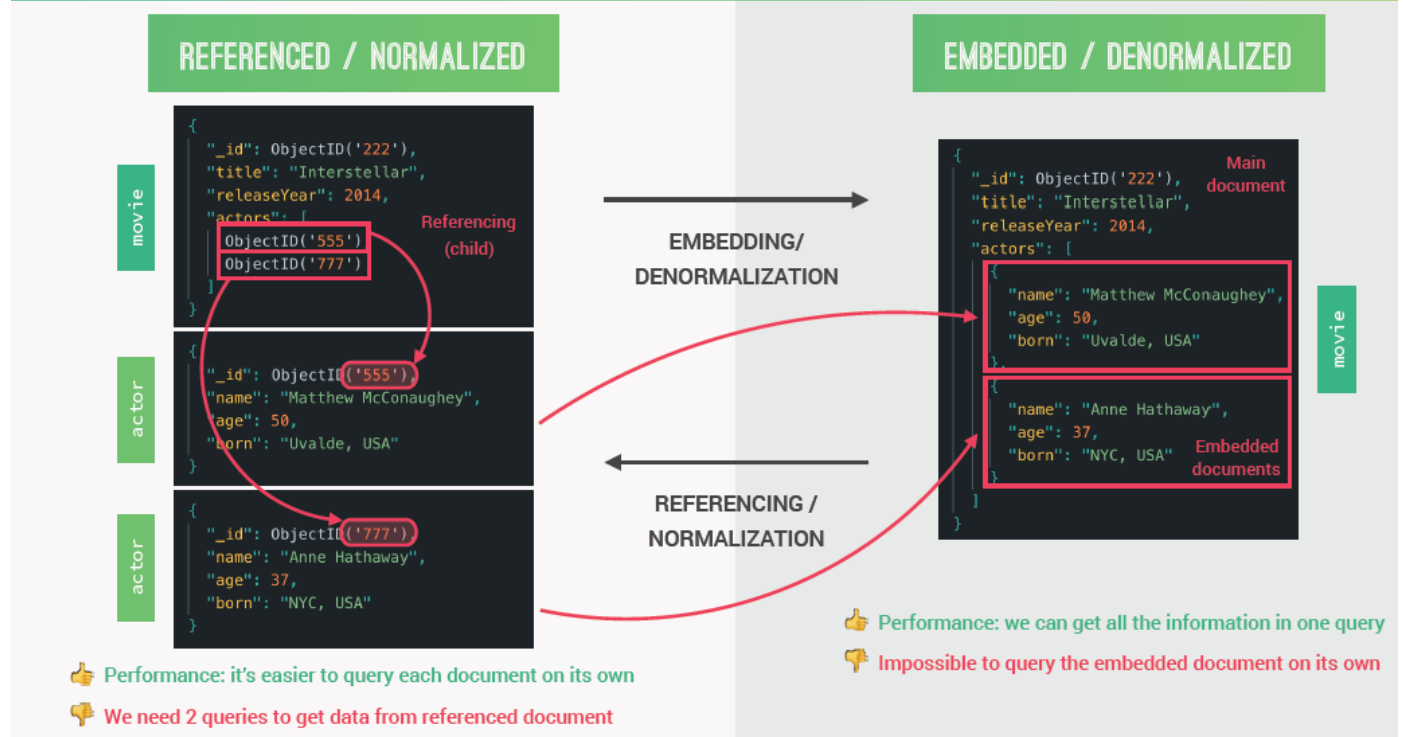
1. TYPES OF RELATIONSHIPS BETWEEN DATA



- The one to many relationships are the most important ones to know.
- The difference between 'many' and a 'ton' is of course a bit fuzzy but just think that if something can grow almost to infinity then it's definitely a one to a ton relationship.
- In Relational databases, there is just one to many without quantifying how much that many actually is. However with NoSQL databases like MongoDB, this difference (one-to-few, one-to-many, one-to-ton) is extremely important because it's one of the factors that we're going to use to decide if we should denormalize or normalize data.

Referencing and Embedding two datasets

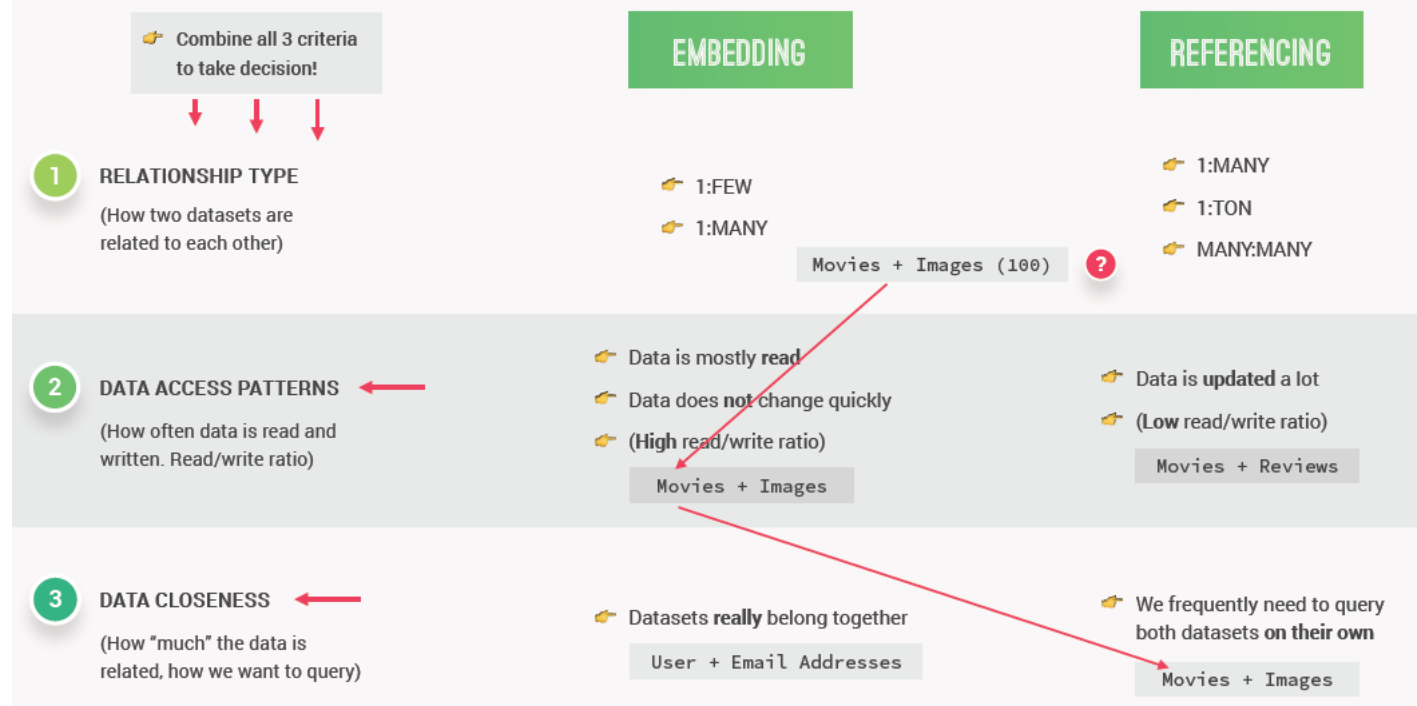
2. REFERENCING VS. EMBEDDING



- Each time we have two related datasets we can either represent that related data in a **referenced or normalized** form or in an **embedded or denormalized** form.
- In the **referenced** form, we keep two related datasets and all the documents separated. So all the data is nicely separated which is exactly what normalized means.
- Child referencing is when the parent (e.g. Movie) references its children (e.g. Actors).
- In relational databases, all data is always represented in normalized form like this. But in a NoSQL database like MongoDB we can denormalize data into a denormalized form simply by **embedding** the related document right into the main document.
- If we choose to **denormalize** or to **embed** our data, we will have one main document containing all the main data as well as the related data.
- The result of embedding is that our application will need to fewer queries to the database. Because we can get all the data about movies and actors all at the same time which will of course increase our performance.
- The downside of embedding is of course that we can't really query the embedded data on its own. And so if that's a requirement for the application, you would have to choose a normalized design.
- Note: We could begin our thought process with denormalized data and then come to the conclusion that it's best to actually normalize the data.

When to Embed and When to Reference

3. WHEN TO EMBED AND WHEN TO REFERENCE? A PRACTICAL FRAMEWORK

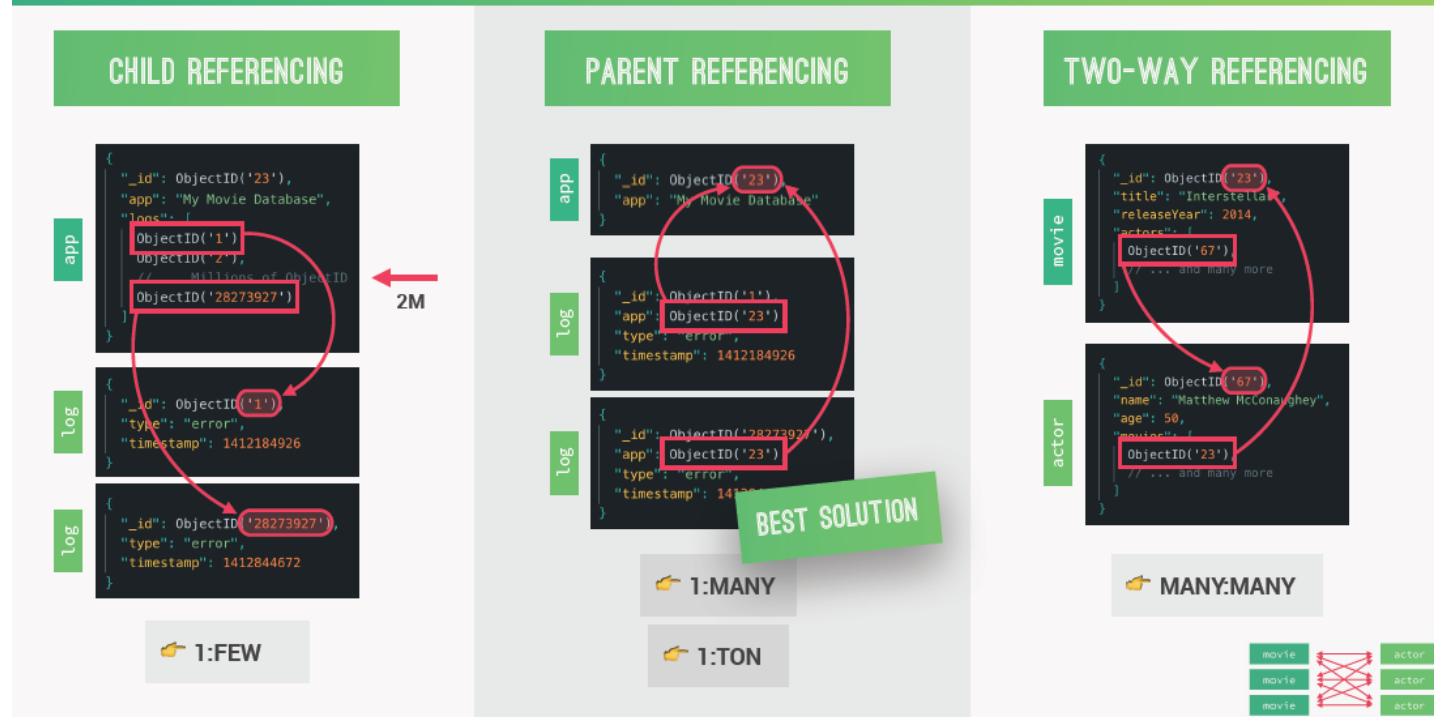


- When we have two related datasets; we have to decide if we're going to embed the datasets or if we're going to keep them separated and reference them from one dataset to the other.
- How do we actually decide if we should normalize or denormalize the data? Three criteria –
 - First look at the type of relationships that exists between datasets.
 - Second try to determine the data access pattern of the dataset that we want to either embed or reference. This just means to analyze how often data is read and written in that dataset.
 - Last, look at data closeness. Data closeness means how much the data is really related and how we want to query the data from the database.
- Now to actually take the decision; we need to combine all of these three criteria and not just use one of them in isolation.
- Usually when we have one to few relationship we will always embed the related dataset into the main dataset.
- Now in a one to many relationship; things are a bit fuzzy so it's okay to either embed or reference. In that case we will have to decide according to the other two criteria.

- On a one to a ton or a many to many relationship we usually always reference the data. That's because if we actually did embed in this case we could quickly create way too large document. Even potentially surpassing the maximum of 16 MB.
- Data access patterns is just a fancy description for evaluating whether a certain dataset is mostly written to or mostly read from.
- So if the dataset that we're deciding about is **mostly read** and the data is not updated a lot then we should probably **embed** that dataset. The reason for this is that by embedding we only need one trip to the database per query.
- On the other hand, if our data is updated a lot then we should consider referencing or normalizing the data. That's because it's more work for the database engine to update and embed a document than a simpler standalone document. And since our main goal is performance, we just normalize the dataset.
- Data closeness is just like a measure for how much the data is related.
- If the two datasets really intrinsically belong together then they should probably be embedded into one another.
- Now if we frequently need to query both of datasets on their own then that's a very good reason to normalize the data into two separate datasets even if they are closely related.
- There are not really completely right or completely wrong ways of modeling our data. There are no hard rules; these are just like guidelines that you can follow to find the probably most correct way of structuring your data.

Types of Referencing

4. TYPES OF REFERENCING



- If we have chosen to normalize/reference our datasets, then after that we still have to choose between three different types of referencing – Child referencing, parent referencing and two-way referencing.
- In child referencing, we basically keep references to the related child documents in a parent document, usually stored in an array.
- However, the problem with child referencing is that this array of IDs can become very large if there are lots of children. And this is an **anti-pattern** in MongoDB, so something that we should avoid at all costs. Also, child referencing makes it so that parents and children are very tightly coupled which is not always ideal. But that's exactly why we have parent referencing.
- In parent referencing, it actually works the other way around. So here in each child document we keep a reference to the parent element. The child always knows its parent but the parent actually knows nothing about the children.
- The **conclusion** of all this is that in general child referencing is best used for one to a few relationships, where we know before hand that the array of child documents won't grow that much. On the other hand, parent referencing is best used for one to many and one to a ton relationships.

- Note: The most important principals of MongoDB data modeling is that array should never be allowed to grow indefinitely. In order to never break that 16 MB limit.
- Two-way referencing is for many-to-many relationships.
 - Considering many to many relationship of Movie and Actor (one movie can have many actors, one actor can act in many movies), tow-way referencing works like this – in each movie we will keep references to all the actors that star in that movie. So a bit like in child referencing. However and at the same time in each actor we also keep references to all the movies that the actor played in. So movies and actors are connected in both directions. In therefore the name two-way referencing.
 - This makes it really easy to search for both movies and actors completely independently. While also making it easy to find the actors associated to each movie and the movies associated to each actor.
 - In case of **few-to-few** relationships, we can use embedding for intrinsic dataset.

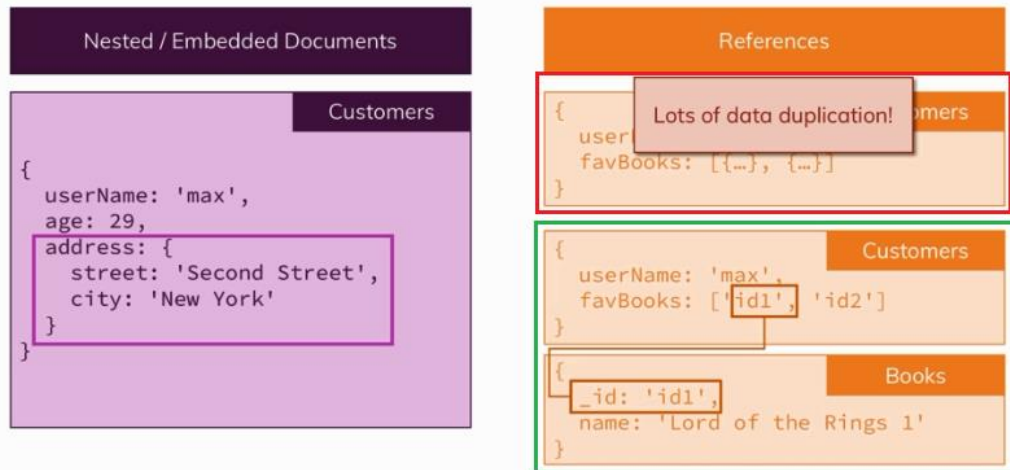
Summary

- 👉 The most important principle is: Structure your data to **match the ways that your application queries and updates data**;
- 👉 In other words: Identify the questions that arise from your **application's use cases** first, and then model your data so that the **questions can get answered** in the most efficient way;
- 👉 In general, **always favor embedding**, unless there is a good reason not to embed. Especially on 1:FEW and 1:MANY relationships;
- 👉 A 1:TON or a MANY:MANY relationship is usually a good reason to **reference** instead of embedding;
- 👉 Also, favor **referencing** when data is updated a lot and if you need to frequently access a dataset on its own;
- 👉 Use **embedding** when data is mostly read but rarely updated, and when two datasets belong intrinsically together;
- 👉 Don't allow arrays to grow indefinitely. Therefore, if you need to normalize, use **child referencing** for 1:MANY relationships, and **parent referencing** for 1:TON relationships;
- 👉 Use **two-way referencing** for MANY:MANY relationships.

Understanding Relations

- How do you store related data?
- Do you use embedded documents? Or do you work with references?

Relations - Options



Relations - Options

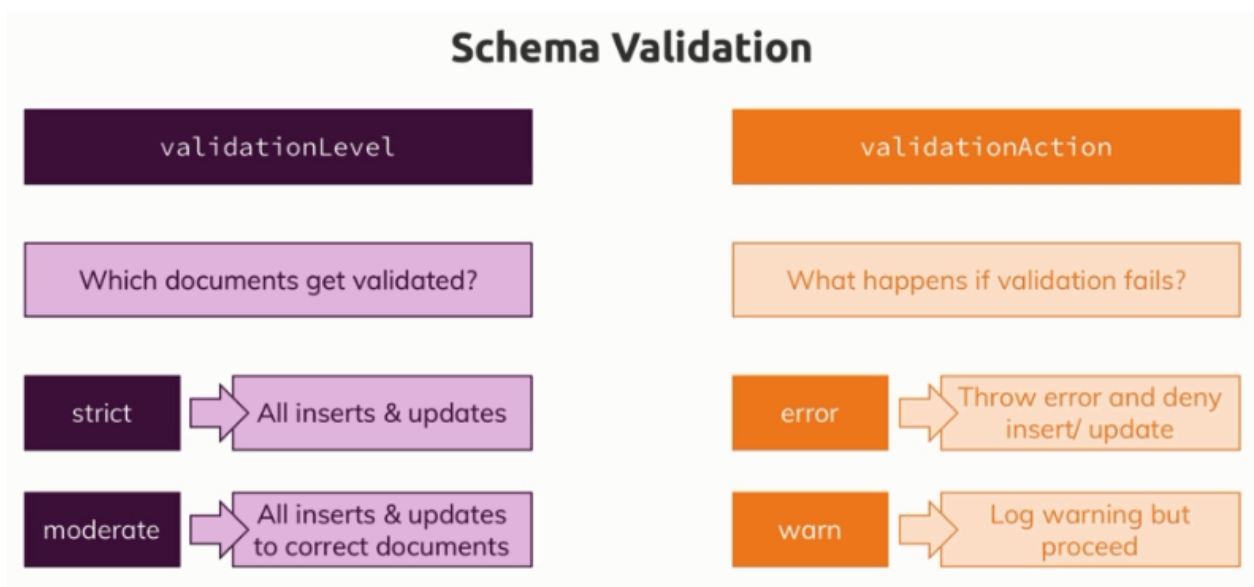
Nested / Embedded Documents	References
Group data together logically	Split data across collections
Great for data that belongs together and is not really overlapping with other data	Great for related but shared data as well as for data which is used in relations and standalone
Avoid super-deep nesting (100+ levels) or extremely long arrays (16mb size limit per document)	Allows you to overcome nesting and size limits (by creating new documents)

Using “lookup()” for merging reference relations

- \$lookup operator is essentially a helpful tool that allows you to fetch two related documents, merge together in one document in one step instead of having to do two steps.
- This mitigates some of the disadvantages of splitting your documents across collections because now you can at least merge them in one go.
- \$lookup operator / function is very useful for merging data in one step and it allows you to get the best of both worlds, having it split up and still fetching it in one go. Now this still is not an excuse for always using references because obviously this costs more performance than having an embedded document. 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.

Understanding Schema Validation

- 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.
- If we add a validation schema, 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.



- JSON Schema (using \$jsonSchema operator) is the recommended means of performing schema validation.

- Validation occurs during updates and inserts. When you add validation to a collection, existing documents do not undergo validation checks until modification.
- The `validationLevel` option determines which operations MongoDB applies the validation rules:
 - `strict` (the default), MongoDB applies validation rules to all inserts and updates.
 - `moderate`, MongoDB applies validation rules to inserts and to updates to existing documents that already fulfill the validation criteria. With the moderate level, updates to existing documents that do not fulfill the validation criteria are not checked for validity.
 - To disable validation entirely, you can set `validationLevel` to `off`.
- More one Schema Validation (Recommended):
<https://docs.mongodb.com/manual/core/schema-validation/>

Configuring a Collection

- In MongoDB, a collection is automatically created when we insert at least one document.
- If you want to configure your collection in a special way, well then you can use the `db.createCollection()` command.
- `createCollection()` command takes two parameters. 1st arg is – name of the collection and 2nd arg is a document where you can configure that collection.
- You can define that validation schema when you create the collection using `createCollection()` command or also thereafter with the help of `runCommand()`.

Exploring the Shell and the Server

MongoDB Server: Finding Available Options

- One of the options to seek help is look for official documentation
- Refer - <https://docs.mongodb.com/manual/reference/program/mongod/>
- Options available with mongod command (MongoDB Server)
 - help – to see all available options with mongod command
e.g. mongod --help
 - port – to override default port
 - dbpath – to use specified DB path to store data in the database.
 - logpath – this is where the logs will be written to
 - quiet – to change the way things get logged or output by the server (talk less)
 - repair – to run if you have any issues connecting and see warnings or errors related to database files being corrupted or anything like that.
 - directoryperdb – each database (collections, indexes, etc.) will be stored in a separate directory.
 - storageEngine – default is WiredTiger which is very high performant. However we can change it if there is any other good reason. Mostly we never this it.
 - SSL related configuration is related to Security and Authentication.
 - fork – only works on Mac and Linux in order to run the MongoDB server as a background process. Because on windows it can be run as Service (provided that you have selected the “Service” option during installation) directly so that it will run in background.

Stopping the background MongoDB Server

- For All OS –
 - Connect using Mongo shell (mongo.exe).
 - Run command >use admin
 - Run command >db.shutdownServer()
- For windows –
 - Above approach also works for Windows.
 - Another approach is run this command to stop background Service
net stop mongod
 - To start MongoDB server as a Service (background process) on windows
net start mongod

MongoDB Server (mongod) Configuration File

- Official docs: <https://docs.mongodb.com/manual/reference/configuration-options/>
- One file mongod.cfg is added as part of MongoDB installation which has some settings for MongoDB Server (mongod.exe).
Path: C:\Program Files\MongoDB\Server\4.2\bin\mongod.cfg
- To use this file, we need to pass this as an arg when starting the MongoDB Server (mongod)
> mongod --config C:\Program Files\MongoDB\Server\4.2\bin\mongod.cfg
- Using a configuration file makes managing mongod and mongos options easier, especially for large-scale deployments. You can also add comments to the configuration file to explain the server's settings.

Mongo Shell (mongo.exe) options

- Refer – <https://docs.mongodb.com/manual/reference/program/mongo/>
- To see all available options, use flag --help
> mongo --help
- Some options for mongo command are –
 - nodb – You can for example run it without connecting to a database because the shell is based on javascript so if you just want to execute some javascript code, you could do that.
 - quiet – to output less information.
 - port and host – you can define a port and a host for the server to which it should try to connect, by default it takes localhost 27017. We can use this to connect to say a cloud server.
 - We can also add authentication information.

Options After connecting to Shell

- Once you connect o MongoDB Shell, type command help.
> help
- Help command will output a short list of some important help information, some important commands you can execute and you can even dive into detailed help like for example what can I do as an admin by typing help admin and now you see a couple of commands that might be useful when it comes to administrating the database.
- Some useful commands –
 - db.help() – I see commands that I can run here on this database.
 - db.test.help() – to see the commands which can be executed at collection level. (assuming 'test' is a database). Shows all the commands I can execute directly on the collection.

Diving into Create Operations

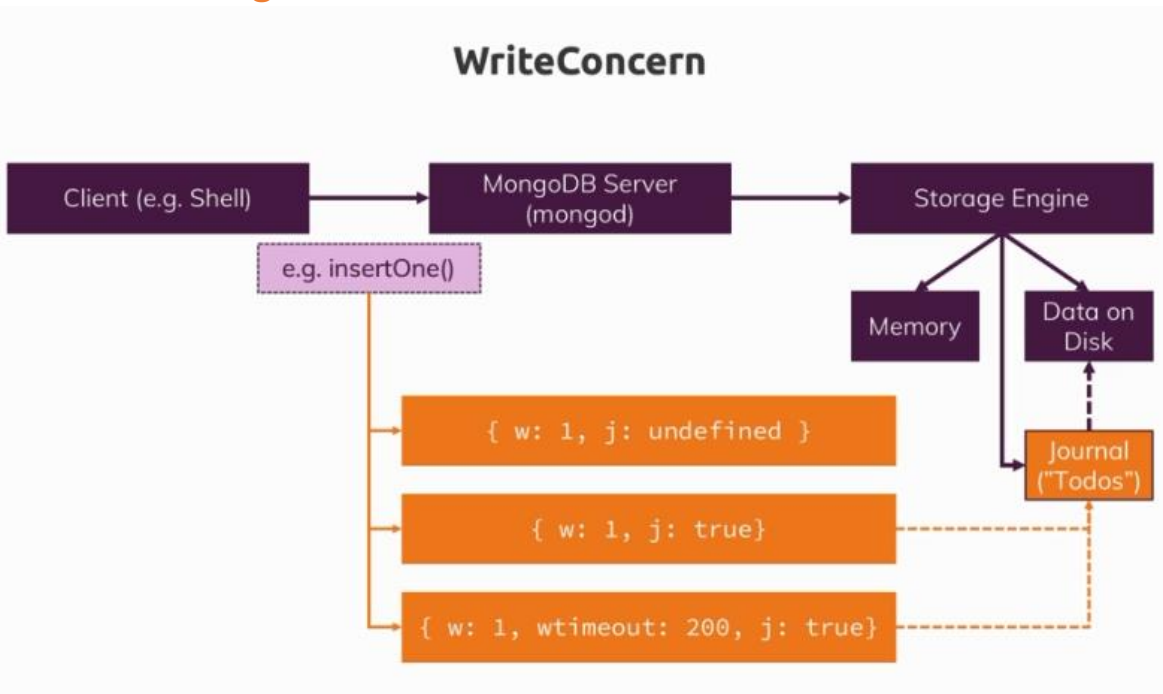
Overview

- `insertOne()` – to insert one document.
- `insertMany()` – to insert multiple documents. Takes an array of documents.
- `insert()` – it's a bit more flexible because it takes both a single document or an array of documents.
 - `insert()` was used in the past, `insertOne()` and `insertMany()` were introduced on purpose so that you have to be more clear about what you will do.
 - Using `insert()` method is NOT recommended.
 - `insert()` takes either an object to array of objects(documents) as input.
 - Unlike `insertOne()` and `insertMany()`, `insert()` does not return ID of the document that was just inserted. This is one of the downsides of `insert()` as sometimes we immediately need the ID in order for the frontend to work properly.
- `mongoimport` – is a tool to import data.

Working with Ordered Inserts

- **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. It cancels the operation and does not continue with the next document which could have succeeded. So this is the default behavior.
- You can reproduce this by using `insertMany()` operation by having a duplicate record (record with same "_id"). You will see that the document with duplicate _id will fail and all documents after this document will not be inserted into database (even though they could be unique). But documents inserted before the failure will NOT be rollback.
- We can override this default behavior. We can pass another argument to the `insertMany()` operation which will have our configuration for this `insertMany()` operation.
- So we can pass the "ordered" option which allows us to specify whether mongodb should perform an "ordered insert" or not. To override the default behavior, set "ordered" to false.
- So with "ordered" set to false, mongodb will continue inserting next documents in the input even if a document fails to be inserted into database.

Understanding the “writeConcern”



- Docs: <https://docs.mongodb.com/manual/reference/write-concern/>
- Now you can configure a so-called writeConcern on all the write operations like insert one with an additional argument the writeConcern argument which is in turn a document where you can set settings like above shown.
- The **w simply means write**, with a number which indicates to how many instances, in case you're using multiple instances on one server, you want this write to be acknowledged.
 - 1 is the default value. With w: 1, which is the default, you're basically saying that my mongodb server should have accepted that write, so the storage engine is aware of it and will eventually write it to the disk. And once storage engine writes this to memory first, it will return the ObjectId and will acknowledge it.
 - With w: 0, you just sent the request and you immediately return, you don't wait for a response of this request. 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.
- The **j stands for 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 operation and that it needs to store that data on disk just by the write being acknowledged and being there in memory. It does not 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 write operation and if the server goes 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**.
- 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, you also need to update these, so it simply takes longer, adding a to-do in a journal is faster.
- The default is that the journal is not getting used with "j" set to undefined.
- Setting "j" to false has the same effects as that of undefined.
- So The default option is **{w: 1, j: undefined}**
- There is a second option we can specify on insertOne and insertMany and that is the writeConcern option.
- With **{w: 1, j: 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.
- Third option not directly related to the journal but it's a W timeout option, which is **{w: 1, wtimeout: 200, j: true}** 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.
- This is the writeConcern 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.

- Again this is a decision you have to make depending on your application needs, what you need.

- E.g.

```
>db.persons.insertOne({name: "Sam", age: 32},  
                      {writeConcern: {w: 1, j: true}});
```

Atomicity

- MongoDB guarantees you an atomic transaction which means the transaction either succeeds as a whole or it fails as a whole.
- If it fails during the write, everything is rolled back for this document you inserted and that is important. E.g. in one document, suppose you have 3 fields name, age and hobbies, then it will never happen that the partial data like just name and age without hobbies is saved in database because of any reasons or server failures.
- It's on a per document level, that document means the top level document so it includes all embedded documents, all arrays so that is all included.
- Docs: <https://docs.mongodb.com/manual/core/write-operations-atomicity/#atomicity>

Importing Data

- Docs: <https://docs.mongodb.com/manual/reference/program/mongoimport/index.html>
- Go to the path where you have the json file which you want to import the data from. Or you can pass the full file path as first arg to mongoimport command.
- D:\MongoDB\workspace\02-import-data
> **mongoimport** tv-shows.json -d movieData -c movies --jsonArray --drop
- -d means which database this imported data should be added to
- -c means which collection inside given database the data should be added to
- --jsonArray – means we are just telling that the file contains more than one document. Can be ignored if there is only one document in the file.
- --drop means if this collection should already exist, it will be dropped and then re-added. If --drop is not used, it we'll append the data to the existing collection if one already exists.

```
PS D:\MongoDB\workspace\02-import-data> mongoimport tv-shows.json -d movieData -c movies --jsonArray --drop
2021-01-12T19:40:43.169+0530   connected to: mongodb://localhost/
2021-01-12T19:40:43.170+0530   dropping: movieData.movies
2021-01-12T19:40:43.382+0530   240 document(s) imported successfully. 0 document(s) failed to import.
PS D:\MongoDB\workspace\02-import-data>
```

Summary

Module Summary

insertOne(), insertMany()

- You can insert documents with insertOne() (one document at a time) or insertMany() (multiple documents)
- insert() also exists but it's not recommended to use it anymore – it also doesn't return the inserted ids

WriteConcern

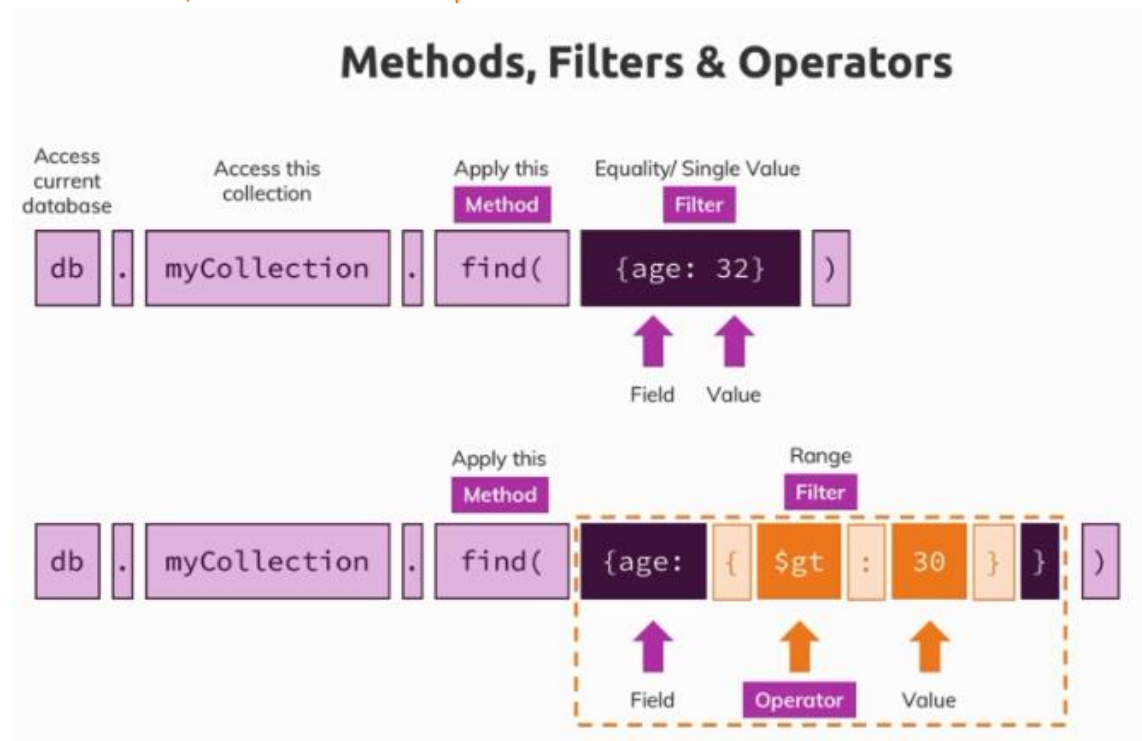
- Data should be stored and you can control the "level of guarantee" of that to happen with the writeConcern option
- Choose the option value based on your app requirements

Ordered Inserts

- By default, when using insertMany(), inserts are ordered – that means, that the inserting process stops if an error occurs
- You can change this by switching to "unordered inserts" – your inserting process will then continue, even if errors occurred
- In both cases, no successful inserts (before the error) will be rolled back

Read Operations – A Closer Look

Methods, Filters and Operators



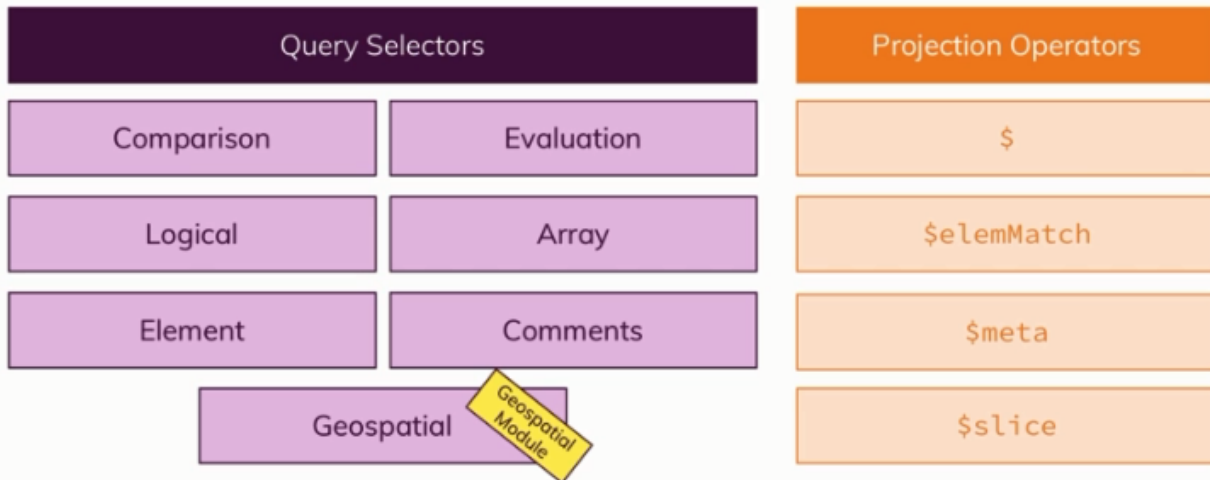
Operators – An Overview

How Operators Impact our Data

Type	Purpose	Changes Data?	Example
Query Operator	Locate Data	⊘	<code>\$eq</code>
Projection Operator	Modify data presentation	⊘	<code>\$</code>
Update Operator	Modify + add additional data		

Query Selectors and Projection Operators

Query Selectors & Projection Operators



Understanding findOne() and find()

- findOne() – always fetches first matching document. findOne() does not give you back a cursor but only one document which it instantly loads.
- find() method does not give us collection of data, but it gives us a so called **cursor** object.
- The filter by default checks for equality, but we can use other operators as per our needs. E.g. gt, lt, le, gte, etc.
- E.g.
`db.movies.find({runtime: 60 })`
IS SAME AS
`db.movies.find({runtime: { $eq: 60 }})`

Working with Comparison operators

- Docs for Operators: <https://docs.mongodb.com/manual/reference/operator/query/>
- Comparison Operators: <https://docs.mongodb.com/manual/reference/operator/query-comparison/>

Querying Embedded Fields and Arrays

- Use quotes around the embedded field and use dot (.) to traverse to that field.
E.g.

```
"rating" : {  
  "average" : 6.7  
}
```

```
db.movies.find({"rating.average": 6.7}).pretty()
```
- In case of arrays, it will check if given element **exists** in the array of elements.
E.g.

```
"genres" : [  
  "Action",  
  "Science-Fiction"  
],
```

```
> db.movies.find({genres: "Action"}).pretty()
```
- In case of arrays, to check for **exact** equality, we need to use brackets ([]) over that value. E.g.

```
> db.movies.find({genres: ["Action"]}).pretty()
```

Working with Logical operators

- Docs for Operators: <https://docs.mongodb.com/manual/reference/operator/query/>
- Docs for Logical Operators: <https://docs.mongodb.com/manual/reference/operator/query-logical/>
- AND is the default concatenation mechanism of mongodb.
- There is a \$and operator because you might have multiple conditions on the same field.
- In Mongo Shell, if you have repeated/duplicate fields in the filter, the last field-value will be used.
E.g.

```
> db.movies.find({genres: "Drama", genres: "Horror" }).count()
```


Here genres: "Horror" will be used and genres: "Drama" is ignored.

Working with Element Query Operators

- Docs for Operators: <https://docs.mongodb.com/manual/reference/operator/query/>
- Docs for Element Query Operators: <https://docs.mongodb.com/manual/reference/operator/query-element/>
- \$exists – Matches documents that have the specified field.
 - If we want documents that have the specified field which could also be null
E.g. `>db.users.find({age: {$exists: true}})`
This will return all documents where the 'age' field exists (documents where age field is explicitly set to null are also returned).
 - If we want documents that have the specified field which are NOT null, then
E.g. `>db.users.find({age: {$exists: true, $ne: null}})`
- \$type – Selects documents if a field is of the specified type like number, string, etc. This is helpful when same field's value is of different types in different documents.

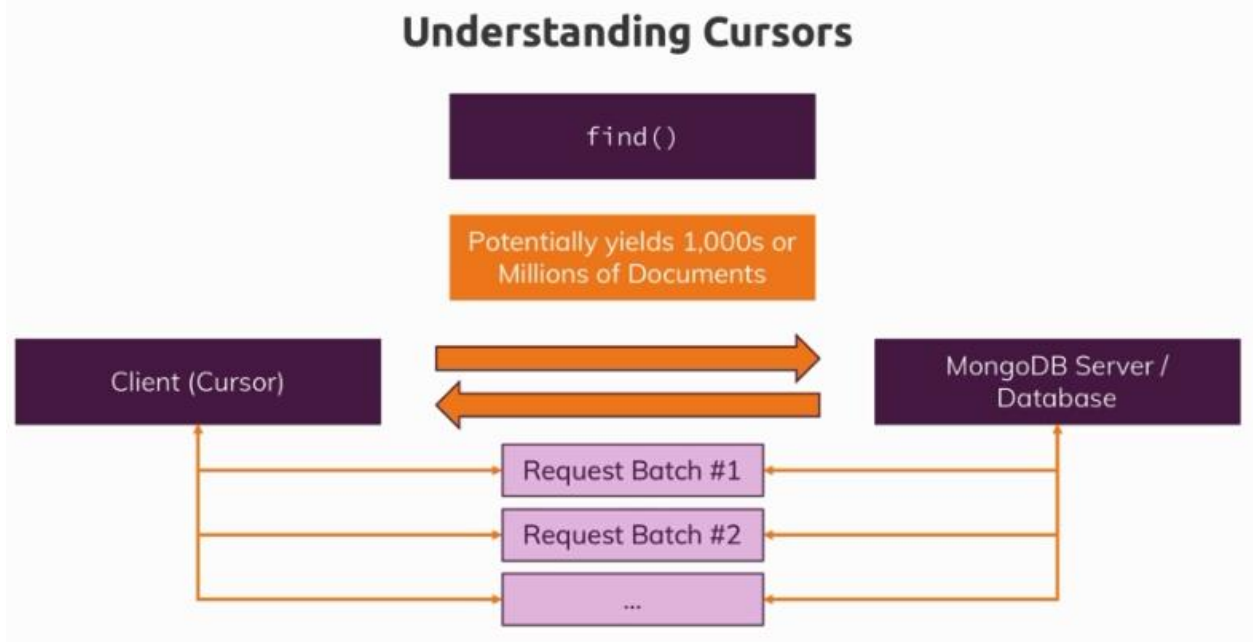
Working with Evaluation Query Operators

- Docs for Operators: <https://docs.mongodb.com/manual/reference/operator/query/>
- Docs for Evaluation Query Operators: <https://docs.mongodb.com/manual/reference/operator/query-evaluation/>
- Some of the operators are – \$expr, \$jsonSchema, \$mod, \$regex, \$text, \$where
- \$where is deprecated and replaced with \$expr.
- \$regex selects documents where values match a specified regular expression. Useful for example to search based on a word within a long text string. (Ideally for text based search, text indexing should be used).
- \$expr is useful if you want to compare two fields inside of one document and then find all documents where this comparison returns a certain result.
- We can use \$cond along with \$expr in some complex cases.

Working with Array Query Operators

- Docs for Operators: <https://docs.mongodb.com/manual/reference/operator/query/>
- Docs for Array Query Operators: <https://docs.mongodb.com/manual/reference/operator/query-array/>
- You can use the path embedded approach (e.g. "hobbies.title") not only on directly embedded documents so if you have one embedded document in a field but also if you have an **array** of embedded documents.
- If you want to use multiple fields of the embedded document within an array as a filter criteria, we need to use \$elemMatch operator.

Understanding Cursors



- A cursor is basically a pointer which has the query you wrote stored and which can therefore quickly go to the database and say hey, give me the next batch, give me the next batch.
- You fetch the data one by one, so one document by a time and it really is only transferred over the wire when you request the next one.
- Now in the shell, we get 20 by default because the shell automatically basically takes the cursor and gets us the first 20 documents before we can get more.
- If you write an application with a mongodb driver, then you have to control that cursor manually and make sure you get back your results. And that cursor approach is great because it saves resources.

Applying Cursors

- Iterate a Cursor in the mongo Shell
<https://docs.mongodb.com/manual/tutorial/iterate-a-cursor/>
- APIs available on cursor object: e.g. sort(), skip(), etc.
<https://docs.mongodb.com/manual/reference/method/js-cursor/index.html>

Sorting Cursor Results

- Just like `pretty()` function, `sort()` function is available on cursor only. (so not available on `findOne()`)
- E.g. To sort by average rating asc.
`> db.movies.find().sort({"rating.average": 1})`
- 1 means ascending, -1 means descending.
- E.g. To sort by average rating asc first and then by runtime desc, then
`> db.movies.find().sort({"rating.average": 1, runtime: -1})`

Skipping and Limiting Cursor Results

- Skipping and limiting is useful when we have pagination.
E.g. To skip 10 documents
`> db.movies.find().sort({"rating.average": 1}).skip(10)`
- Limit allows you to limit the amount of elements the cursor should retrieve at a time
E.g. Skip 10 documents and return next 2 documents
`> db.movies.find().sort({"rating.average": 1}).skip(10).limit(2)`
- Generally for skip and limit to work, we would expect that the elements must be sorted first. However the good thing about mongodb cursor is that the order of sort, skip, limit does not matter. mongodb and the cursor will automatically do it in the right order – it will sort first, then skip and then limit. (However it is a good practice to follow the order sort – skip – limit for readability)

Using Projection to shape our Results

- With projection, we can control which data is returned. There may be hundreds of fields and embedded documents but with projection, we can choose which fields we want.
- E.g. To only return name, genres and rating, pass such data as second parameter to the `find()` method.
`> db.movies.find({}, {name: 1, genres: 1, rating: 1}).pretty()`
- 1 means include, 0 means exclude.
- `_id` field is by default always returned even if you use projection object. To exclude `_id` field, we have to explicitly use `_id: 0`.
E.g.
`> db.movies.find({}, {name: 1, genres: 1, rating: 1, _id: 0}).pretty()`
- You cannot mix inclusion (1) and exclusion (0) criteria (except for `_id` field).
E.g. Below does NOT work
`> db.movies.find({}, {name: 1, genres: 1, runtime: 1, rating: 0, _id: 0}).pretty()`
Error: error: {

```
    "ok" : 0,
    "errmsg" : "Projection cannot have a mix of inclusion and
exclusion.",
    "code" : 2,
    "codeName" : "BadValue"
}
```

- Projection can also be used on **embedded documents**.

E.g.

```
> db.movies.find({}, {name: 1, genres: 1, "schedule.time": 1}).pretty()
```

- Projection can also be used on **Arrays**. We can use **\$slice** operator when using projection on arrays.
- Docs on using \$slice in projection:

<https://docs.mongodb.com/manual/reference/operator/projection/slice/index.html>

Update Operations

Overview

- Update Documents
<https://docs.mongodb.com/manual/tutorial/update-documents/>
- Update Operators
<https://docs.mongodb.com/manual/reference/operator/update/>

updateOne(), updateMany() and using \$set

- updateOne() and updateMany() are used to update one more many documents resp.
- \$set is generally used along with above functions to perform the updates (add new fields or update existing fields within the document) to the data.

Update Field Operators

- Field Update Operators
<https://docs.mongodb.com/manual/reference/operator/update-field/>
- \$currentTime – Sets the value of a field to current date, either as a Date or a Timestamp.
- \$inc – Increments the value of the field by the specified amount.
- \$min – Only updates the field if the specified value is less than the existing field value.
- \$max – Only updates the field if the specified value is greater than the existing field value.
- \$mul – Multiplies the value of the field by the specified amount.
- \$rename – Renames a field.
- \$set – Sets the value of a field in a document.
- \$setOnInsert – Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.
- \$unset – Removes the specified field from a document.

Understanding upsert

- upsert – (**U**psert or **I**nsert) we want to update some document where we are not sure whether it exists in the collection or not. In this case, if the document does not exist, it will be created otherwise it will be updated. It will also set the fields which are part of filter criteria.
- E.g. Here "Sameer" does not exist in DB so it will insert a documents with name and totalAge fields.

```
> db.users.updateOne({name: "Sameer"}, {$set: {totalAge: 32}}, {upsert: true})
```


Updating Arrays

- Array Update Operators
<https://docs.mongodb.com/manual/reference/operator/update-array/>
- E.g. Updating matched Array Elements

```
> db.users.updateMany({ hobbies: {$elemMatch: {title: "Sports", frequency: {$gte: 3} }} }, {$set: {"hobbies.$.highFrequency": true} } )
```

`hobbies.$` will automatically refer to the first matched document in our array (for each filtered document).
- The positional `$` operator acts as a placeholder for the first element that matches the query document.
- `$` – Acts as a placeholder to update the first element that matches the query condition.
- `[$]` – Acts as a placeholder to update all elements in an array for the documents that match the query condition.
- `[$<identifier>]` – Acts as a placeholder to update all elements that match the `arrayFilters` condition for the documents that match the query condition.
- `$addToSet` – Adds elements to an array only if they do not already exist in the set.
- `$pop` – Removes the first or last item of an array.
- `$pull` – Removes all array elements that match a specified query.
- `$push` – Adds an item to an array.
- `$pullAll` – Removes all matching values from an array.

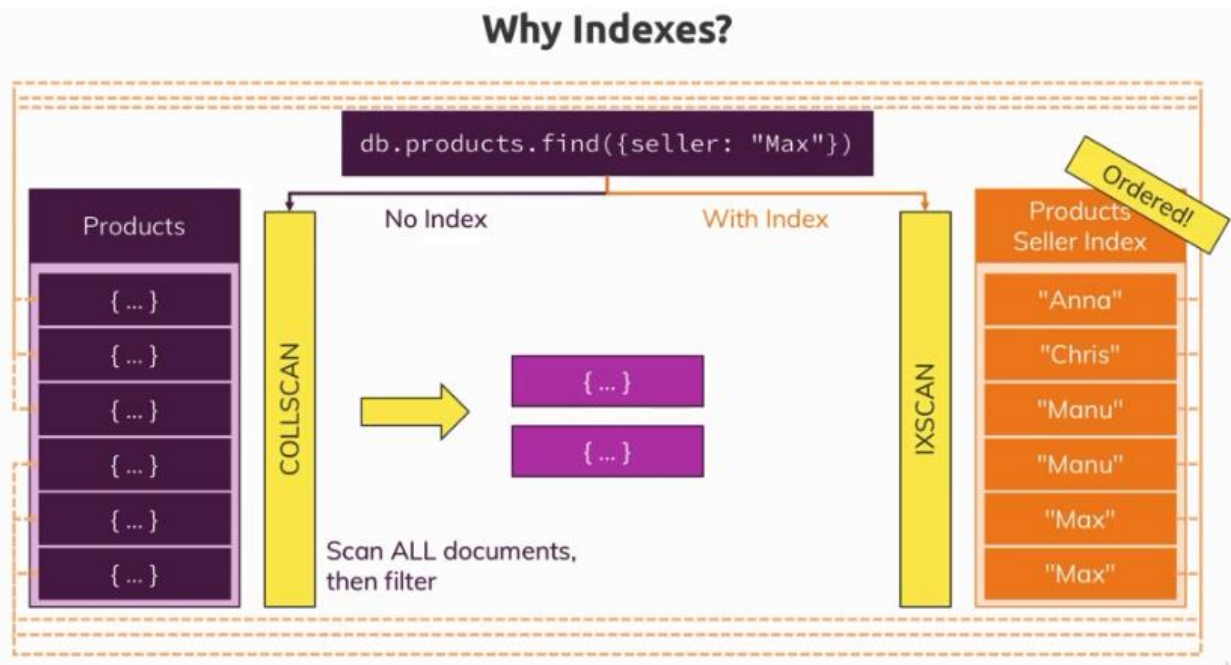
Understanding Delete Operations

- Delete Documents
<https://docs.mongodb.com/manual/tutorial/remove-documents/>
- Delete All Entries in a Collection
 - `db.users.deleteMany({})`
 - `drop()` should be used by admin only, never through code
`db.users.drop()`
- To drop entire database,
`db.dropDatabase()`

Working with Indexes

- Retrieve data **efficiently**.
- Indexes are a feature that can drastically speed up your queries, though if used incorrectly, they can also slow down some of your operations.
- Index Docs: <https://docs.mongodb.com/manual/indexes/>

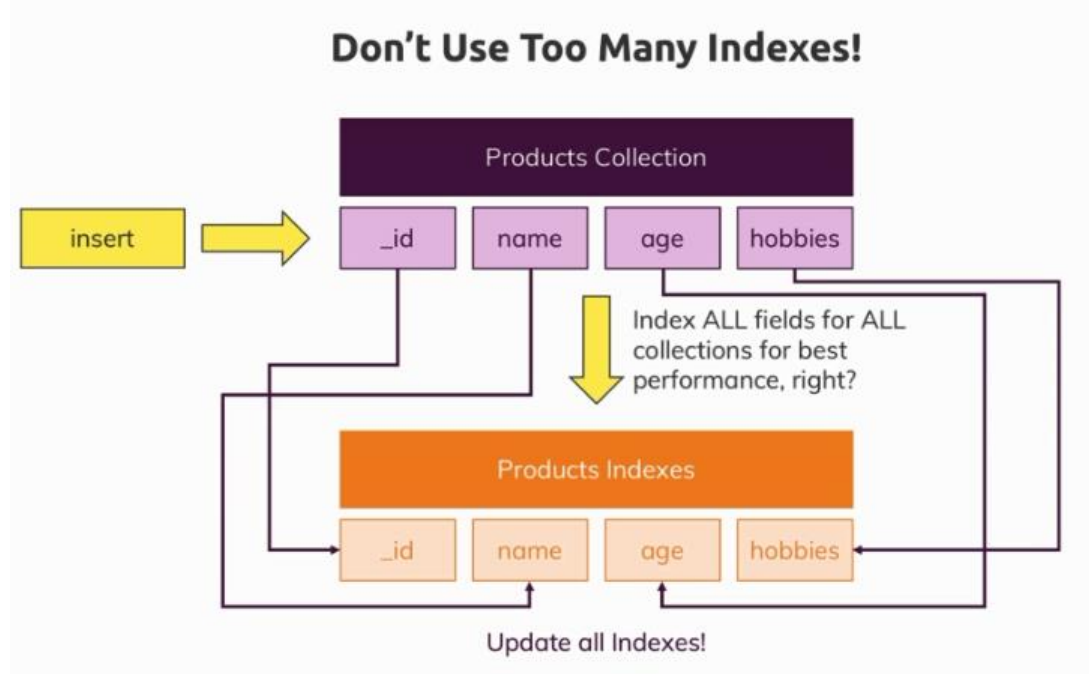
Indexes introduction



- An index can speed up our find, update or delete queries, so all the queries where we are looking for certain documents that should match some criteria.
- Index is essentially an ordered list of all the values that are placed or stored in the specified key (e.g. key for field name "Name") for all the documents. And it's not just an ordered list of the values, of course every value, every item in the index has a pointer to the full document it belongs to.
- Without index, mongodb performs collection scan (COLLSCAN). If there is index available on given field, then it performs the index scan (IXSCAN).
- Since indexes are stored in an ordered way (sorted), mongo can very efficiently go through that index and then find the matching documents because of that ordering and because of that pointer, every element in this index has. So mongodb finds the value for this query and then finds the related documents it can return. So it's this direct access that mongodb can use here and that speeds up your queries. So this is how an index works and it also answers the question why you would use one because creating such indexes can drastically speed up your queries. However you also shouldn't overdo it with

the indexes because an index does not come for free, you will pay some performance cost on inserts because that extra index that has to be maintained needs to be updated with every insert.

- You really have to find out which indexes makes sense and which indexes don't.



Adding a Single Field Index

- mongodb gives us a nice tool (`explain()`) that we can use to analyze how it executed the query.

E.g.

```
db.contacts.explain().find(...);
```

- `explain()` works for find, update, delete and not for insert.
- This is how **`explain()`** works –
 - `explain()` gives us the detailed description of what mongodb did and how it derived our results. Mongodb thinks in so-called plans and plans are simply alternatives it considers for executing that query and in the end it will find a winning plan and that winning plan is essentially what it did to get our results. We could also have rejected plans but for this, we would need alternatives and without indexes, a full scan is always the only thing mongodb can do.
- Now we can get even more detailed output by passing an argument to `explain()` and that argument is a string where we control the verbosity of our command.
- By passing 'executionStats' as that argument, you find a detailed output for the query and how the results were returned.

E.g. `db.contacts.explain('executionStats').find(...);`

It will show below object in the output (if there is no index) –

```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 1222,
  "executionTimeMillis" : 10,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 5000,
  "executionStages" : {
    "stage" : "COLLSCAN",
    "filter" : {
      "dob.age" : {
        "$gt" : 60
      }
    },
    "nReturned" : 1222,
    "executionTimeMillisEstimate" : 0,
    "works" : 5002,
    "advanced" : 1222,
    "needTime" : 3779,
    "needYield" : 0,
    "saveState" : 39,
    "restoreState" : 39,
    "isEOF" : 1,
    "direction" : "forward",
    "docsExamined" : 5000
  },
  "nReturned" : 1222,
  "executionTimeMillisEstimate" : 0,
  "works" : 5002,
  "advanced" : 1222,
  "needTime" : 3779,
  "needYield" : 0,
  "saveState" : 39,
  "restoreState" : 39,
  "isEOF" : 1,
  "direction" : "forward",
  "docsExamined" : 5000
},
```

- Please see official docs for creating indexes and other methods.

E.g. `> db.contacts.createIndex({"dob.age": 1})`

Indexes Behind the Scenes

- What does `createIndex()` do in detail?
- Whilst we can't really see the index, you can think of the index as a simple list of values + pointers to the original document. Something like this (for the "age" field):
(29, "address in memory/ collection a1")
(30, "address in memory/ collection a2")
(33, "address in memory/ collection a3")
- The documents in the collection would be at the "addresses" a1, a2 and a3. The order does not have to match the order in the index (and most likely, it indeed won't).
- The important thing is that the index items are ordered (ascending or descending - depending on how you created the index). `createIndex({age: 1})` creates an index with ascending sorting, `createIndex({age: -1})` creates one with descending sorting.

- MongoDB is now able to quickly find a fitting document when you filter for its age as it has a sorted list. Sorted lists are way quicker to search because you can skip entire ranges (and don't have to look at every single document).
- Additionally, sorting (via `sort(...)`) will also be sped up because you already have a sorted list. Of course this is only true when sorting for the age.

Understanding Index Restrictions

- If you have a query that will return a **large portion or the majority** of your documents, an index can actually be **slower** because you then just have an extra step to go through your almost entire index list and then you have to go to the collection and get all these documents.
- So if you have queries that regularly return the majority or all of your documents, an index will not really help you there, it might even slow down the execution and that is important to keep in mind as a first restriction that you need to know when planning your queries.
- If you have a dataset where your queries typically only return fractions, like 10 or 20 percent or lower than that of the documents, then indexes will almost certainly always speed it up.
- Logically, that makes sense because the idea of index is to quickly let you get to a narrow subset of your document list and not to the majority of that.

Creating Compound Index

- You can create indexes on fields that have some text or number. Text is also something you regularly search for and it can perfectly be sorted. **Indexes on boolean fields don't make much sense** because since you only got two kinds of values, true or false, chances are that the index will not speed up your queries too much.
- We can create a compound index which is index based on multiple fields. You can use upto 31 fields.
- Order of the fields in the compound index does matter. It doesn't matter in case of single field index.
- You can **utilize compound index from left to right**.

Suppose you create compound index using fields age and gender (with age as first) like this – `db.contacts.createIndex({"dob.age": 1, gender: 1})`

Then index scan (IXSCAN) will be used for queries involving both age and gender, and also for queries involving age (first key in the compound index). But it will not be used if the query involves only second part of the key like gender in this case.

This makes sense because for our example the index values will be created like this 32-male, 33-male, 34-female, etc. So they are anyway sorted by age, hence can be used for

both age+gender and just age. It can't be used only for gender because of above sorting (33-male, 34-female, etc.)

Using Indexes for sorting

- It's also important to understand that indexes are not just there for finding but they can also help you with sorting because we have a sorted list of elements of the index, mongodb can utilize that in case you want to sort in the same way that index list is sorted.
- If you are not using indexes and you do a sort on a large amount of documents, you can actually timeout because mongodb has this threshold of 32 MB which it reserves in memory for the fetched documents and sorting them. And if you have no index, mongodb will essentially fetch all your documents into memory and do the sort there and for large collections and large amounts of fetched documents, this can simply be too much to then sort.
- So sometimes, you also need an index not just to speed up the query which always makes sense but also to be able to sort at all.

Understanding the Default index

- To see all indexes on a collection
`db.contacts.getIndexes()`
- Mongodb creates a default index on the "_id" field automatically.
- This is helpful when we are fetching documents by ID (_id).
- _id is unique by default.
- Unique indexes can help you as a developer ensure data consistency and help you avoid duplicate data for fields that you need to have unique.

Understanding Partial Index

- If you know that certain values will not be looked at or only very rarely and you would be fine using a collection scan if that happens, you can actually create a partial index where you only add the values you're regularly going to look at.
- Docs: <https://docs.mongodb.com/manual/core/index-properties/>
- Docs Partial Index: <https://docs.mongodb.com/manual/core/index-partial/>
- What's the **difference between a partial index and a compound index**?
The difference is that for the partial index, the overall index simply is smaller. The index size is smaller leading to a lower impact on your hard drive and also your write queries are of course also sped up because the index is also updated only when the `partialFilterExpression` matches.
- Whenever mongodb has the impression that your find request would yield more than what's in your index, it will not use that index but if you typically run queries where you

are within your partial index, well then mongodb will take advantage of it and then you benefit from having a smaller index and having less impact with writes.

- **Use case of using Partial Index**

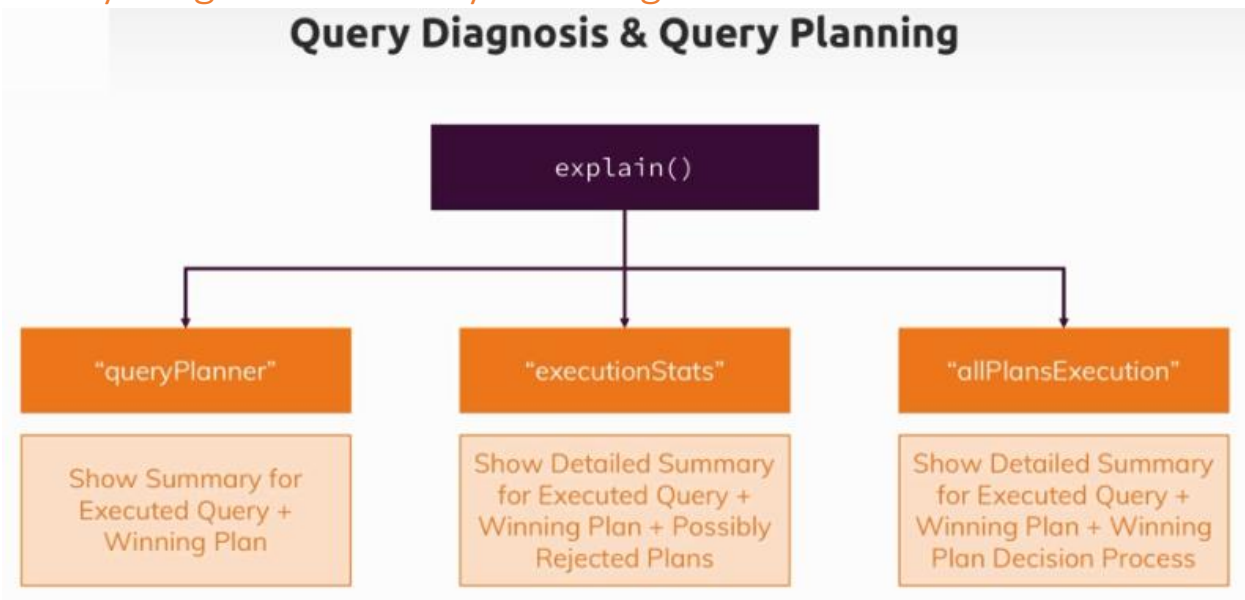
- Suppose we have a unique index on a field and in one of the documents that field simply does not exist. So now if you try to insert another document without this field, then you will get an error – duplicate key. This is because mongodb allows one null or non existing field if you have unique index and hence it will not allow other document having this field as null or non existent. The solution in this case is to use unique AND partial index, that is creating unique index and passing `partialFilterExpression` on that field.
- E.g.

```
>db.users.createIndex({email: 1},  
  {unique: true, partialFilterExpression: {email: {$exists: true}} })
```

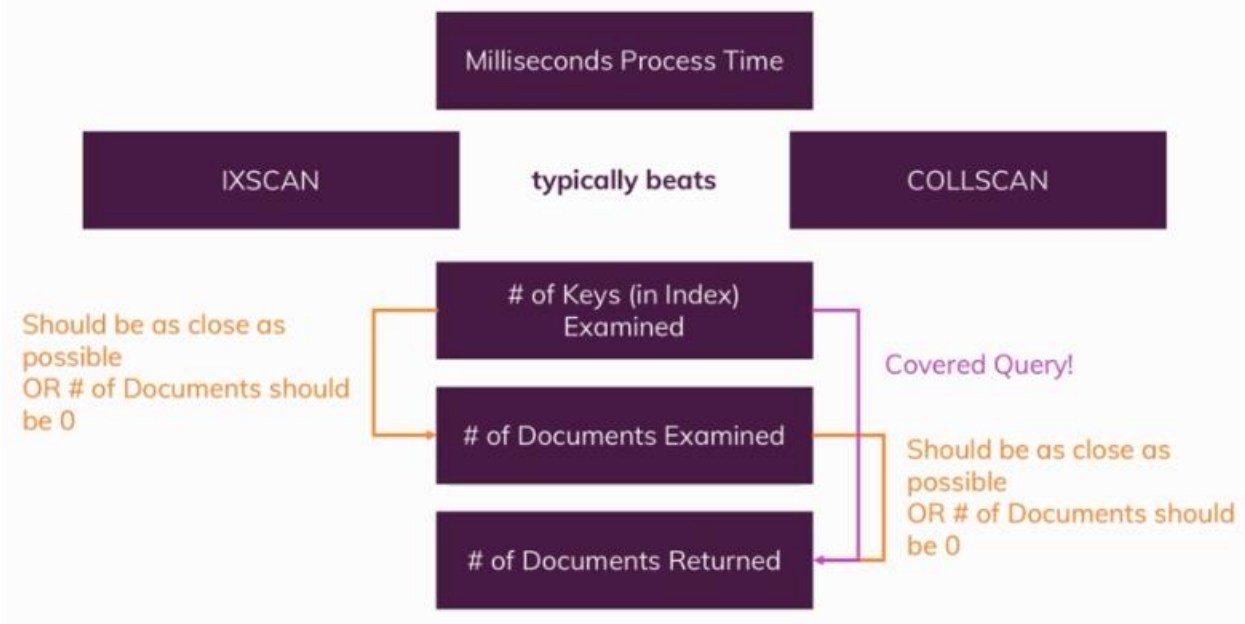
Understanding Time-To-Live (TTL) index

- Docs: <https://docs.mongodb.com/manual/core/index-properties/>
- Docs Partial Index: <https://docs.mongodb.com/manual/core/index-ttl/>
- TTL Index can be very helpful for a lot of applications where you have self-destructing data, let's say sessions of users where you want to clear their data after some duration or anything like that.
- We can add `expireAfterSeconds` field while creating an index. This is a special feature mongodb offers and that only works on date indexes or on date fields, on other fields it will just be ignored if you add it.
- TTL indexes are special single-field indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time or at a specific clock time. Data expiration is useful for certain types of information like machine generated event data, logs, and session information that only need to persist in a database for a finite amount of time.
- TTL indexes only work with single field and that too Date field.

Query Diagnosis & Query Planning



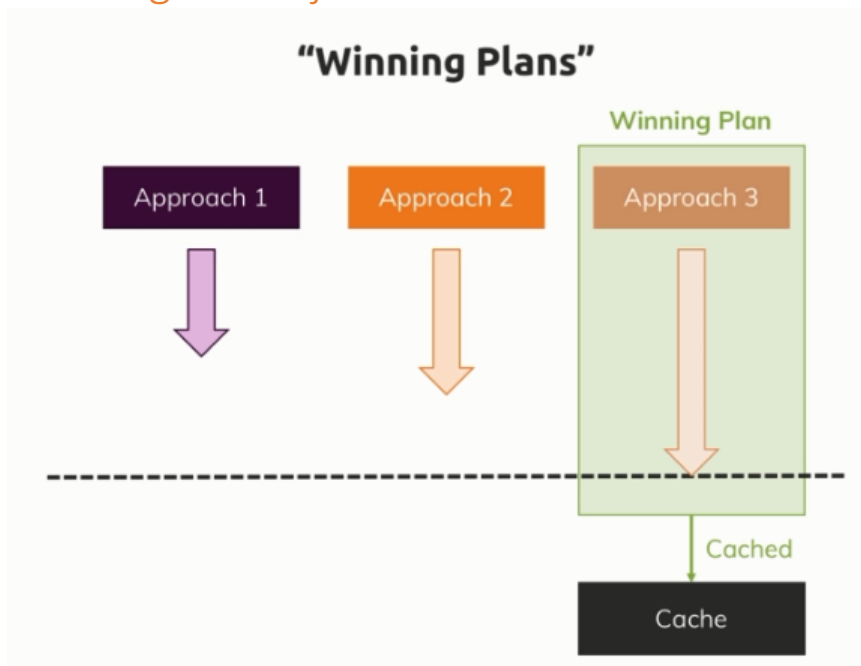
Efficient Queries & Covered Queries



Understanding Covered Queries

- Query Optimization: <https://docs.mongodb.com/manual/core/query-optimization/>
- Docs: <https://docs.mongodb.com/manual/core/query-optimization/#covered-query>
- A covered query is a query that can be satisfied entirely using an index and does not have to examine any documents.
- An index covers a query when all of the following apply:
 - All the fields in the query are part of an index, and
 - All the fields returned in the results are in the same index.
 - No fields in the query are equal to null (i.e. {"field" : null} or {"field" : {\$eq : null}}).
- You will not always be able to reach that state but if you can, if you have some query where you can optimize your index for that, to reach that **covered query state** as it is called because the query is fully covered by the index, then you will of course have a very efficient query because you skipped that stage of reaching out to the collection getting the documents and that obviously speeds up your query.

How MongoDB Rejects a Plan



- For this, mongodb uses an approach where it simply first of all looks for indexes that could help you with the query at hand.
- Let's say based on number of indexes, mongodb comes up with three approaches.
- mongodb then simply let's those approaches race against each other but not for the full dataset but it sets a certain winning condition, currently that should be 100 documents.

- So it looks who's the first to find 100 documents and of course one of the approaches will be the fastest to reach that threshold, mongodb will then basically use that approach for the real query.
- And of course that would be cumbersome if it would have to do this for every find method, for every query you send to the database because that obviously costs a little bit of performance. Therefore mongodb **caches this winning plan** for this exact query, so for exactly the query you send with the fields you were looking for and the values for the fields you were looking for, so it caches this plan as the winner plan for this kind of query.
- And for future queries that are looking exactly equal, it uses this winning plan. For future queries that look different, that use different values or different keys, it will of course race again and find a winning plan for that type of query.

Clearing the Winning Plan from Cache



- Now of course this cache is not there forever, it is cleared after a certain amount of inserts or a db restart. To be precise, instead of being stored forever –
 - This winning plan is removed from cache after you wrote a certain amount of documents to that collection, currently there should be one thousand. So after you added that many documents, mongodb says I don't know if the current winning plan will still win because the collection changed a lot, so I should reconsider.
 - The same happens if you rebuild the index, so if you drop it and recreate it.
 - It also gets deleted if you add other indexes because that new index could be better, so the cache for indexes or winning plans I should say gets cleared when you add new indexes
 - Or when you restart the mongodb server, then this also gets reset.

Using multi-key indexes

- Docs: <https://docs.mongodb.com/manual/core/index-multikey/>
- To index a field that holds an array value, MongoDB creates an index key for each element in the array. These multikey indexes support efficient queries against array fields. Multikey indexes can be constructed over arrays that hold both scalar values (e.g. strings, numbers) and nested documents.
- Technically, multikey indexes are working like normal indexes but they are stored up differently.
- What mongodb does is it pulls out all the values in your index key, so it pulls out all the values in the array and stores them as separate elements in an index.
- This of course means that multikey indexes for a lot of documents are bigger than single field indexes because if every document has an array with let's say four values on average and you have a thousand documents and that array field is what you index, you would store four thousand elements because four times one thousand.
- Multikey indexes are possible but typically are also bigger, doesn't mean you shouldn't use them.
- You can also use an index on a field in an embedded document which is part of an array with that multikey feature.
- Restriction when using multi-key indexes – We cannot create compound index on more than one array field. You will get error – cannot index parallel arrays. The reason for that is simple, mongodb would have to store the cartesian product of the values of both indexes, of both arrays, so it would have to pull out all the addresses and for every address, it would have to store all the hobbies. So if you have two addresses and five hobbies, you already have to store 10 values and that of course becomes even worse the more values you have in addresses, so that is why this is not possible.
- Compound indexes with multikey indexes are possible but only with one multikey index.

Understanding Text Index

- Docs: <https://docs.mongodb.com/manual/core/index-text/>
- Text index is a special kind of index supported by mongodb which will essentially turn this text into an array of single words and it will store it as such.
- So it stores it essentially as if you had an array of these single words, one extra thing it does for you is it removes all the stop words and it stems all words, so that you have an array of keywords essentially and things like "is", "or", "the" or "a" are not stored there because that is typically something you don't search for because it's all over the place.
- To create text index – we don't use 1 or -1, but we use "text" as a keyword while creating text index. E.g. `db.products.createIndex({description: "text"})`
- You may have only one text index per collection.

- To query the collection having a text index, we need to use **\$text** along with **\$search** operators. Other operators include **\$caseSensitive**, **\$language**, etc.
- To search for documents having both "red" and "book" texts,
E.g. `db.products.find({$text: {$search: "red book"}})`
- To search for documents having **exact** phrase "red book" as a part of description,
E.g. `db.products.find({$text: {$search: "\"red book\""}})`
- Text indexes are very powerful but expensive as well.
- We can create a **combined text index** of more than one fields.
There is still will only be one text index but it will contain the keywords of both the fields.
E.g. `db.products.createIndex({title: "text", description: "text"})`
- We can **use text indexes to exclude words** as well.
E.g. This query will look for documents having word "red" but exclude documents which has the word "book" `db.products.find({$text: {$search: "red -book"}})`

Text index and sorting

- mongodb does something interesting or special when managing such a text index or when searching for text in a text index, we can find out how it scores its results.
- "textScore" is a meta field added or managed by mongodb for text searches with the \$text operator on a text index.
- We can use this score to sort results based on relevance of the search string.
E.g. `db.products.find({$text: {$search: "red book"}}, {score: {$meta: "textScore"}})`

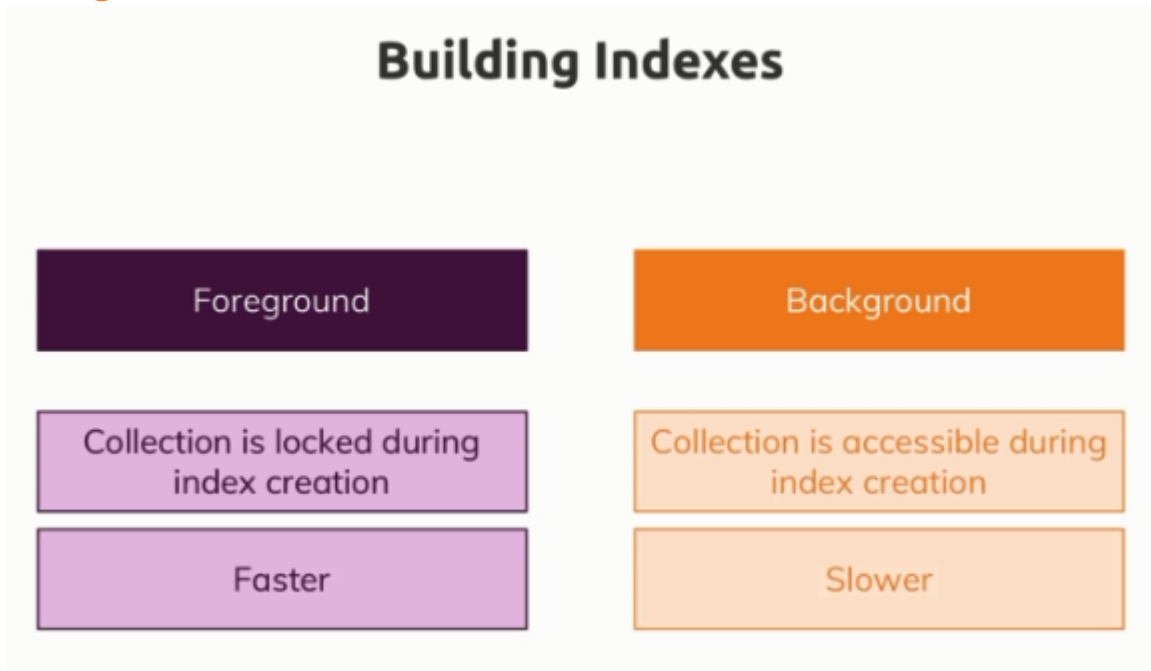
Setting Default Language and using Weights

- Docs: <https://docs.mongodb.com/manual/tutorial/specify-language-for-text-index/>
- We can set language for our text index using the **default_language** key in the config object while creating the index.
- Allowed values in default_language are at
<https://docs.mongodb.com/manual/reference/text-search-languages/#text-search-languages>
- Now the field here is not just some cosmetics, we can later use it when searching for text but most importantly what it will do here is it will define how words are stemmed, so how prefixes and so on are removed and it will also define what stop words are removed, so words like "is" or "a" are removed in English, in German it would be "ist" and "die". So the language here is something you should specify when you know which kind of language or which language will be stored in your text.
- Now another cool value you can set here is that you can define different **weights** for the different fields you're merging together (in case of combined text index). So when

merging together title and description, now maybe I want to merge them but I want to say the description should be a higher weight and the weights will become important when mongodb calculates the **score** of the result because you might have a string "awesome" in the title but that should not count as much as in the description.

- We can set the weights using the "weights" key in the config object while creating the index.
- **Put the weight on the field that holds your most important key words.**

Building indexes



- If you have millions of documents in your collection and you want to create an index, by default the index creation will lock the entire collection till the index is created. This might take some time and hence may not be feasible for production database.
- We have an option to create indexes in the **background** so that the collection is not locked.

E.g.

```
>db.ratings.createIndex({age: 1}, {background: true})
```

Working with Geospatial Data

Adding GeoJSON Data

- GeoJSON official site:
- GeoJSON Docs from MongoDB: <https://docs.mongodb.com/manual/reference/geojson/>
- **Coordinates** are always expressed as latitude and longitude or longitude and latitude pairs because the **longitude** essentially describes a position on a **vertical axis** if you draw it onto our globe, and the **latitude** describes a **horizontal axis** on the earth globe and with that coordinate system, we can map any point onto our earth.
- To specify GeoJSON data, use an embedded document with:
 - A field named **type** that specifies the GeoJSON object type and
 - A field named **coordinates** that specifies the object's coordinates.
 - If specifying latitude and longitude coordinates, list the longitude first and then latitude:
 - Valid longitude values are between -180 and 180, both inclusive.
 - Valid latitude values are between -90 and 90, both inclusive.
- Allowed GeoJSON objects with MongoDB
(<https://docs.mongodb.com/manual/reference/geojson>) –
 - Overview
 - Point
 - LineString
 - Polygon
 - MultiPoint
 - MultiLineString
 - MultiPolygon
 - GeometryCollection

Running GeoJSON Queries

- In general, "location APIs" will always return coordinates in the form of latitude and longitude, this is the standard format.
- Not all geospatial queries require index but some do.

Adding Geospatial Index to Track the Distance

- E.g. To create a geospatial index on a field say location,
> `db.places.createIndex({location: "2dsphere"})`

Geospatial Query Operators

- Docs: <https://docs.mongodb.com/manual/reference/operator/query-geospatial/>
- Query Selectors –
 - \$geoIntersects
 - \$geoWithin
 - \$near
 - \$nearSphere
- Geometry Specifiers –
 - \$box
 - \$center\$centerSphere
 - \$geometry
 - \$maxDistance
 - \$minDistance
 - \$polygon
 - \$uniqueDocs
- The important **difference** between the \$near query which also allows us to find places within a certain min and max distance and the (\$geoWithin and \$centreSphere) is that with (**\$geoWithin and \$centreSphere**) query we get back an unsorted list, this keeps the order of the elements in the database or we could of course manually sort them with sort method. The **\$near** method gives us the elements in a certain radius (based on min and max distance) and **sorts them by proximity**.

Summary

Module Summary

Storing Geospatial Data

- You store geospatial data next to your other data in your documents
- Geospatial data has to follow the special GeoJSON format – and respect the types supported by MongoDB
- Don't forget that the coordinates are [longitude, latitude], not the other way around!

Geospatial Queries

- \$near, \$geoWithin and \$geoIntersects get you very far
- Geospatial queries work with GeoJSON data

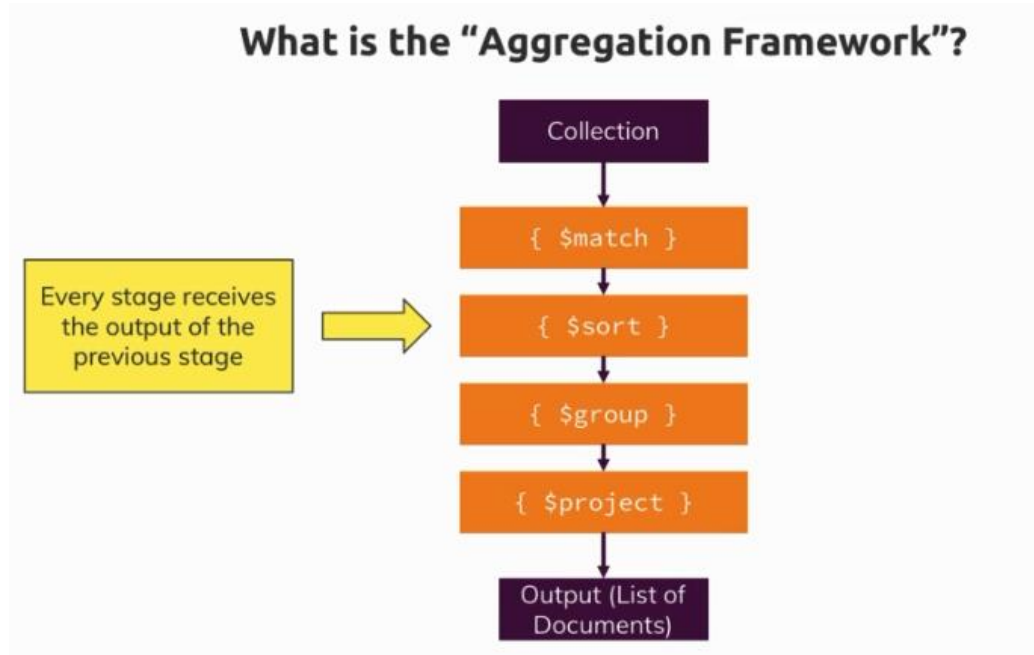
Geospatial Indexes

- You can add an index to geospatial data: "2dsphere"
- Some operations (\$near) require such an index

Understanding the Aggregation Framework

- Retrieving data efficiently and in a structured way.

What is Aggregation Framework



- The aggregation framework in its core is an advanced alternative to the find method you could say.
- You have your collection and now the aggregation framework is all about building a pipeline of steps/stages that runs on the data that is retrieved from your collection and then gives you the output in the form you needed.
- You can reuse certain stages.
- You have a very powerful way of modeling your data transformation.
- Every stage in the pipeline receives the output of the previous stage and therefore, you can have a very structured way of having some input data and then slowly modifying it in the way you need it in the end.

Docs

- Aggregation Pipeline Stages Docs:
<https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/>
- Aggregation Pipeline Operators
<https://docs.mongodb.com/manual/reference/operator/aggregation/>

Using the Aggregate Framework

- The first step will receive the entire data right from the collection you could say and the next step can then do something with the data returned by the first step and so on.
- Aggregate does not go ahead, fetch all the data from the database and then give it to you and then do something on it.
- Of course the first step runs on the database and can take advantage of indexes, so if you filter in the first step or you sort there, you can take advantage of your indexes, so you don't have to fetch all the documents just because you're using aggregate. Aggregate as `find()` executes on the mongodb server and therefore can take advantages of things like indexes.

Match stage

- **\$match** essentially is just a filtering step. The way add filter criteria for `find()` is the same way we use here for `$match`.

Group stage

- **\$group** stage allows you to group your data by a certain field or by multiple fields.

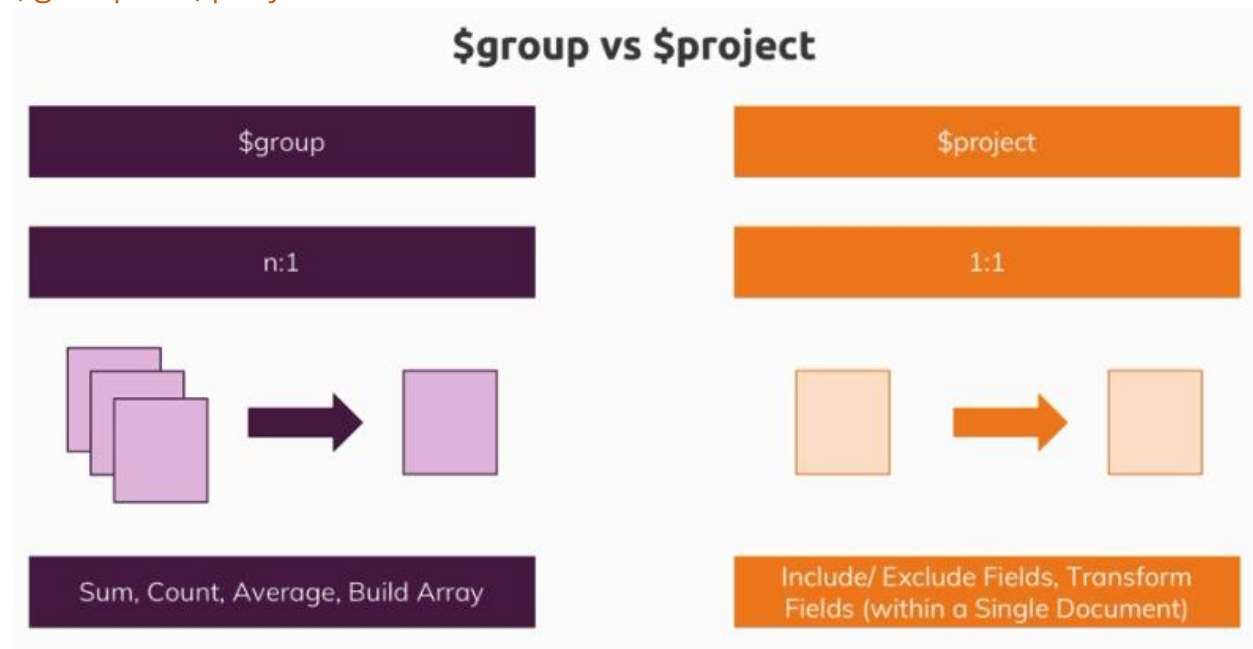
Sort Stage

- **\$sort** stage takes a document as an input to define how the sorting should happen.

Transform Stage

- Docs: <https://docs.mongodb.com/manual/reference/operator/aggregation/project/>
- `$project` stage allows us to transform every document instead of grouping multiple together.
- In its most simple form, `$project` works in the same way as the projection works in the `find` method.
- We cannot just include and exclude data but that we can even add new fields with hardcoded values if we want to do or in our case, with a derived value derived from the data that was in the document before.
- `$filter` allows you to filter arrays in documents inside of the projection phase.

\$group vs \$project



Unwind Stage

- The **\$unwind** stage is always a great stage when you have an array of which you want to pull out the elements.
- \$group merges multiple documents into one, but \$unwind takes one document and spits out multiple documents.
- \$unwind deconstructs an array field from the input documents to output a document for each element. Each output document is the input document with the value of the array field replaced by the element.

Bucket Stage

- The \$bucket stage allows you to output your data in buckets for which you can calculate certain summary statistics.
- \$bucket takes a groupBy parameter where you define by which field do you want to put your data into buckets.
- \$bucketAuto does the bucketing algorithm for you. \$bucketAuto can be an even quicker way for getting a feeling for your data.

Using \$sort, \$skip and \$limit stages

- To implement pagination, the order of the \$sort, \$skip and \$limit stages matters. (It doesn't matter in the find() function, but matters in the aggregation pipeline because output of one stage is input to the next stage).

Writing pipeline results into a new Collection with \$out stage

- We actually have a pipeline stage for working with geo data but that has to always come first in order to take advantage of indexes.
- You can also take the result of a pipeline and write it into a new collection, of course not just if you have geo data in there, but you can always do that. For that you do specify another pipeline stage, the \$out stage for output and this will take the result of your operation and write it into a collection, either a new one which is created on the fly or an existing one.
- The \$out operator is great if you have a pipeline where you want to funnel your results right into a new collection, often you want to use it and just fetch it but if you need to store it, you can do that with the out stage.

Working with Geo Data using \$geoNear stage

- We have a special stage which is built for working with geo data, the \$geoNear stage.
- \$geoNear allows us to simply find elements in our collection, documents in our collection which are close to our current position, that is what geoNear does.
- If you want to use \$geoNear stage, it has to be the **first stage** in the pipeline because it needs to use that index and the first pipeline element is the only element with direct access to the collection, other pipeline stages just get the output of the previous pipeline stage.

How MongoDB Optimizes Your Aggregation Pipelines

- MongoDB actually tries its best to optimize your Aggregation Pipelines without interfering with your logic.
- Aggregation Pipeline Optimization Docs:
<https://docs.mongodb.com/manual/core/aggregation-pipeline-optimization/>

Working with Numeric Data

Number Types – An Overview

Integers, Longs, Doubles			
Integers (int32)	Longs (int64)	Doubles (64bit)	"High Precision Doubles" (128bit)
Only full Numbers	Only full Numbers	Numbers with Decimal Places	Numbers with Decimal Places
-2,147,483,648 to 2,147,483,647	-9,223,372,036,854, 775,808 to 9,223,372,036,854, 775,807	Decimal values are approximated	Decimal values are stored with high precision (34 decimal digits)
Use for "normal" integers	Use for large integers	Use for floats where high precision is not required	Use for floats where high precision is required

- 64 bit double is the default value type mongodb uses if you pass a number with no extra information, no matter, if that number is theoretically an integer and has no decimal place or not, it will be stored as a 64 bit double when passing in the number through the shell (because shell is based on JavaScript).
- Doubles for some technical reasons are not guaranteed, so the decimal places there are not guaranteed to be correct, they are approximated.
- With 128 bit doubles, you have a guarantee for up to 34 decimal digits.
- 64 bit double will not have the same range of that as the 64 bit integer has. And the reason for that is obvious, the 64 bit double also has to handle decimal places.

MongoDB Shell & Data Types

- There's one important thing you have to keep in mind about the MongoDB Shell (which we're using via the mongo command): It is based on JavaScript, it's running on JavaScript.
- Hence you can use JavaScript syntax in there and hence the default data types are the default JavaScript data types.
- That matters especially for the numbers. JavaScript does NOT differentiate between integers and floating point numbers => Every number is a 64bit float instead.
- So 12 and 12.0 are exactly the same number in JavaScript and therefore also in the Shell.

Understanding Programming Language Defaults

- **The default which gets written into mongodb does not depend on mongodb but on the client you're using.**
- In the shell it's based on JavaScript, so the default is the 64 bit double.

Working with int32

- To store 32 bit integers
E.g. `db.persons.insertOne({name: "Sam", age: NumberInt(29)})`

Working with int64

- If you put a very high value than what can accommodate in let's say int32 (refer above picture for the allowed ranges), then you won't get any error, but in the end a totally wrong value will be stored in the database.
- So be careful while storing huge numbers.
- To store 64 bit integers
E.g. `db.persons.insertOne({valuation: NumberLong(1234567890)})`
- To store largest possible number 9223372036854775807
E.g. `db.companies.insertOne({valuation: NumberLong(9223372036854775807)})`
Here you will get **ERROR**. The problem is that this value which is still a number here is simply too big because it's a **double 64** (default shell/JS) which then well gets wrapped by a number long. So in order for this to work, we should wrap this in quotation marks instead.
`db.companies.insertOne({valuation: NumberLong("9223372036854775807")})`
- This is really **important** to understand, NumberInt and NumberLong can be used with a number passed as a value and with quotation marks and **you should always use quotation marks**, so basically pass a string because mongodb will then internally kind of convert that string you could say or simply store it appropriately but if you pass a number, well the number you pass still faces the javascript limitations in the shell which is based on Javascript.

Doing Maths with Floats int32s and int54s

- If you add a double to an int, the result will be double. So pay attention while doing calculations on your int fields, they will get converted to doubles if you do calculations in wrong way.

What's wrong with normal doubles

- E.g.

```
> db.science.insertOne({a: 0.3, b: 0.1})
> db.science.findOne()
{ "_id" : ObjectId("600aaf7dd2c588a97d423247"), "a" : 0.3, "b" : 0.1 }
>
> db.science.aggregate([{$project: {result: {$subtract: ["$a",
"$b"]}}}}])
{ "_id" : ObjectId("600aaf7dd2c588a97d423247"), "result" :
0.19999999999999998 }
```
- In above example, subtracting two doubles did not give exact output of **0.2**.
- This is the behavior with normal doubles.

Working with Decimal 128

- Above example with 128 bit decimals and **without quotation marks**

```
> db.science.insertOne({a: NumberDecimal(0.3), b: NumberDecimal(0.1)})
> db.science.aggregate([{$project: {result: {$subtract: ["$a",
"$b"]}}}}])
{ "_id" : ObjectId("600ab136d2c588a97d423248"), "result" :
NumberDecimal("0.2000000000000000") }
```
- Above example with 128 bit decimals and **with quotation marks**

```
> db.science.insertOne({a: NumberDecimal("0.3"), b:
NumberDecimal("0.1")})
> db.science.aggregate([{$project: {result: {$subtract: ["$a",
"$b"]}}}}])
{ "_id" : ObjectId("600ab180d2c588a97d423249"), "result" :
NumberDecimal("0.2") }
```

E.g. `db.science.insertOne({a: NumberDecimal("0.3"), b: NumberDecimal("0.1")})`
- **Always use quotation marks to avoid precision issues.** ☺
- Of course for high precision more memory is required to store the data.

Important

- Modelling Monetary Data
<https://docs.mongodb.com/manual/tutorial/model-monetary-data/>
- Difference between decimal, float and double in .NET?
<https://stackoverflow.com/questions/618535/difference-between-decimal-float-and-double-in-net>

MongoDB and Security

Security Checklist

- Authentication and Authorization, Transport Encryption and Encryption at Rest are the main building blocks of MongoDB Security because these three blocks kind of affect your work as a developer.
- Docs: <https://docs.mongodb.com/manual/administration/security-checklist/>

Authentication and Authorization

- Authentication and Authorization mainly means that the database you're using to store data will know users and you will have to authenticate, so you means you as a developer, so your code will have to authenticate with that database in order to insert data, get data or do all kinds of stuff.
- So authentication and authorization is one super important building block when it comes to securing your mongodb environment.

Transport Encryption

- Another important building block is transport encryption, this means that data that is sent from your app to the server should be encrypted so that no one can sit in the middle and spoof your connection and read that data.

Encryption at Rest

- Additionally encryption at rest is an important topic and that means that the data in the database also should be encrypted otherwise if someone somehow gets access to your database servers, well they can read plain text information and that will make getting the data out of there easier.
- If you do encrypt your data in the database, you are still protected a bit even if people get access to your database server because then they might have access but the data is unreadable.

Auditing

- This is a pure server admin task but of course mongodb does provide features to audit your service, so to see who did what, which actions occurred and you can do a lot of stuff there to make sure that you always control what's happening and you always are aware of what's happening in your database.

Server and Network Config and Setup

- Additionally the server on which you run your database server and now I'm talking about the physical machine which is running somewhere or if you're using a cloud provider like

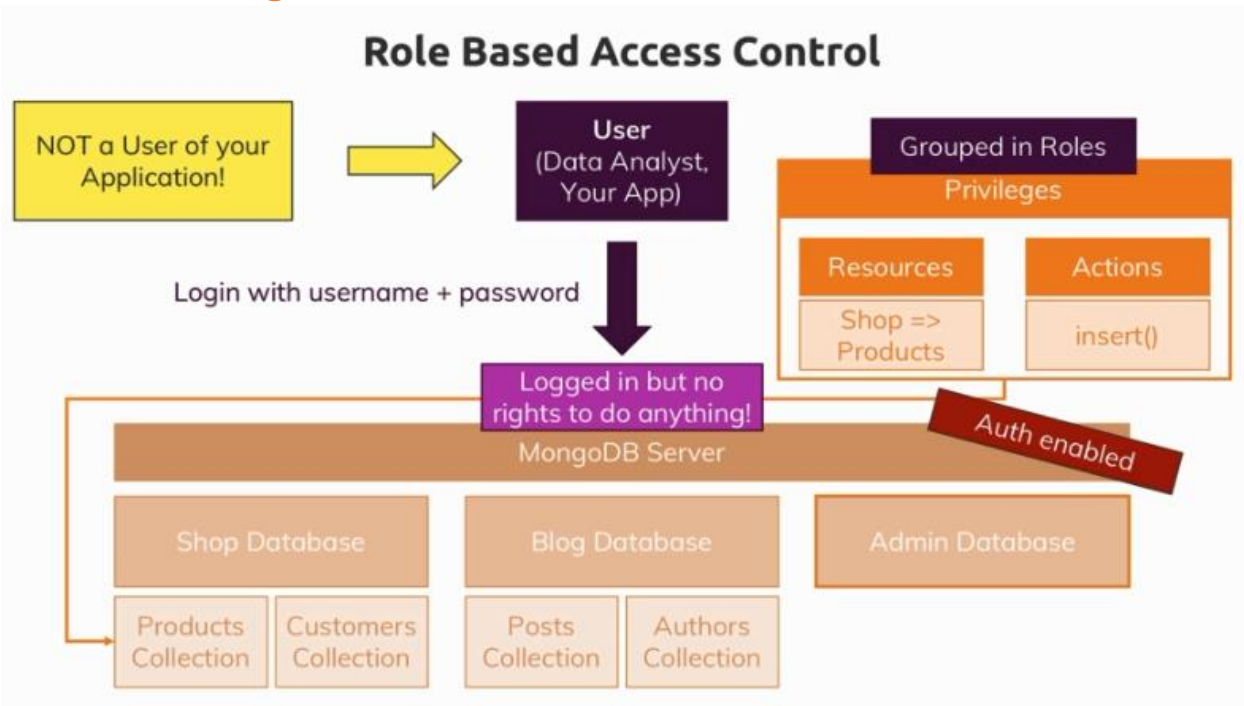
AWS, I'm talking about the instances you booked there, the network you are using to host your mongodb environment of course also should be secured.

- So if you are administrating and setting up the entire mongodb environment, this is also something you will have to take care about.

Backups and Software Update

- You as the owner of a mongodb environment should regularly backup your data and of course you should make sure that all the software you're running is up to date to fix any security holes that might be in there.

Understanding Role based Access Control



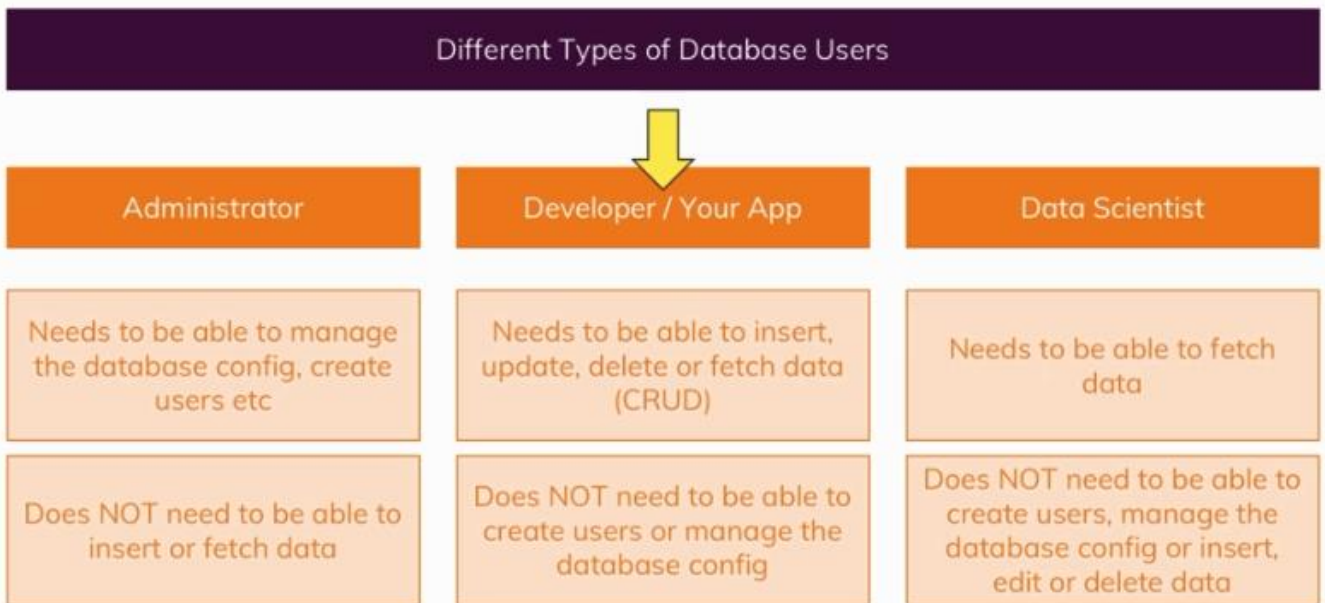
- **Authentication** is all about identifying users (users of the mongodb server) in your database.
- **Authorization** on the other hand is all about identifying what these users (users of the mongodb server) may then actually do in the database.
- MongoDB uses these two concepts to control who is able to connect to the database server and then what these people or apps who are able to connect can do.
- MongoDB employs a Role based Access Control system.
- Users in mongodb are not just entities that are made up of a username and a password but they also are assigned some roles and these roles are essentially groups of privileges.
- A **privilege** is a combination of a resource and an action. A resource would be something like the products collection in our shop database and an action would be an

insert command. So actions are essentially the kinds of tasks, the commands we can execute, we can do in our mongo database and the resource is the one on which we can execute this command.

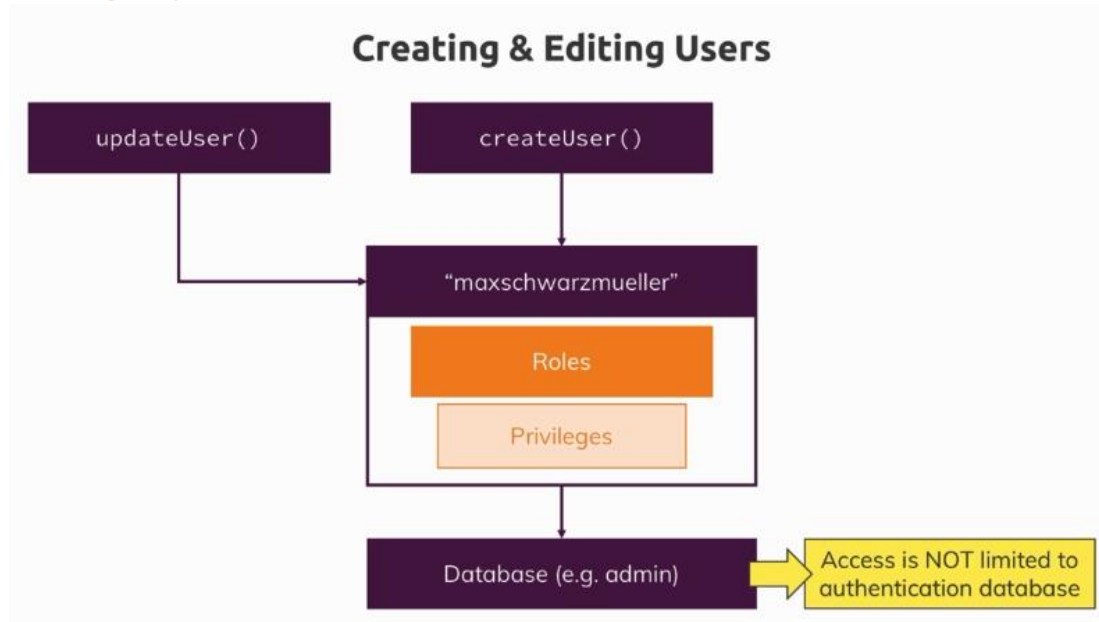
- Typically you don't just have one privilege but instead multiple privileges are grouped into so-called **Roles** and therefore a user has a role and that role includes all the privileges.
- This Role based Access Control model is a very flexible model because it allows us to create multiple users where we as a database owner or administrator can give every user exactly the rights the user needs.

Roles

Why Roles?



Creating/Update User



- Users are created by a user with sufficient permissions with the createUser command.
- You then create a user with a username and a password and this user will have a couple of roles or at least one role and then each role will typically contain a bunch of privileges.
- Now a user is created on a database, so a user is attached to a database.
- Now this does not actually limit the access of the user to that database even though you might think it does but this is the database against which the user will have to authenticate, the exact rights the user then has depend on the roles you assigned to that user and even if a user authenticates against the shop database, the user might still have the role to work with all databases that exist in your mongodb environment.
- **Important:** In order to allow the user based access to mongodb, we need to start the mongodb server (mongod.exe) with a special parameter named --auth.
> **mongod --auth**
- **To login to mongo shell with user credentials**, there are 2 ways.
 - Login during connecting to mongo shell
>mongo -u <username> -p <password> --authenticationDatabase <dbName>
 - Login after connecting to mongo shell
>mongo
> switch to the database against which this user was registered.
>db.auth('username','password')
>db.logout() // this will logout current user

Localhost Exception

- When connecting to your database in this state where you haven't added any users, you are allowed to add one user and that one user should of course be a user who is then allowed to create more users.
- And to do this, you first of all should switch to the admin database and then run db.createUser() command.

```
> show dbs
> use admin
switched to db admin
> db.createUser({user: "sameer", pwd: "sameer", roles:
["userAdminAnyDatabase"]})
```
- The role "userAdminAnyDatabase" is a special built-in role which will grant this user the right to administrate any database in this mongodb environment.

Built-in Roles

Built-in Roles		
Database User	Database Admin	All Database Roles
read readWrite	dbAdmin userAdmin dbOwner	readAnyDatabase readWriteAnyDatabase userAdminAnyDatabase dbAdminAnyDatabase
Cluster Admin	Backup/ Restore	Superuser
clusterManager clusterMonitor hostManager clusterAdmin	backup restore	dbOwner (admin) userAdmin (admin) userAdminAnyDatabase root

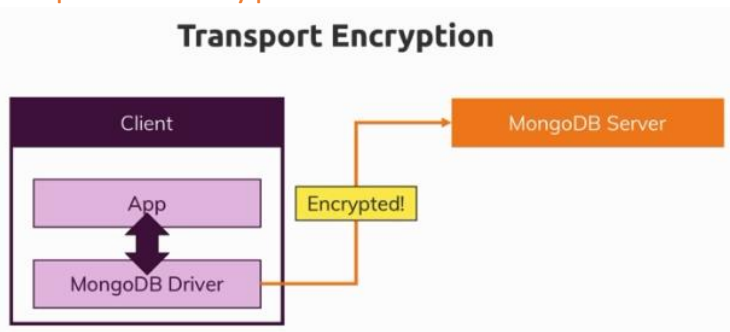
- Mongodb ships with a bunch of built-in roles that basically cover all the typical use cases you have.
- You can also create your own roles.
- dbAdmin is for example a role for people who have that role are able to manage the database obviously, to create collections.
- userAdmin is more geared towards creating and managing users as the name suggests.
- These roles (Database User and Database Admin) will grant the user access to you for example readWrite data but only inside the database to which you assigned that role.

- We can also give access to all databases.
- We also have cluster administration roles.
- **Clusters** are essentially constructs where you have multiple mongodb servers working together, this is used for scaling so that you can actually well have multiple machines running mongodb servers and store your data which then work together.
- You also have special backup and restore roles in case you have some users who are only responsible for that, so these really are roles that just allow you to do that, backup the database or restore it.
- **Super User** – If you assign the dbOwner or a user Admin role to the admin database, then this will kind of be a special case because the admin database is a special database and these users are then able to create new users and also change their own role and that's why it's a super user.
- The **root** role is basically the most powerful role, if you assign this to a user, this user can do everything.
- Built-In Roles Docs: <https://docs.mongodb.com/manual/reference/built-in-roles/>

User Management Methods Docs:

- Docs: <https://docs.mongodb.com/manual/reference/method/js-user-management/>

Transport Encryption

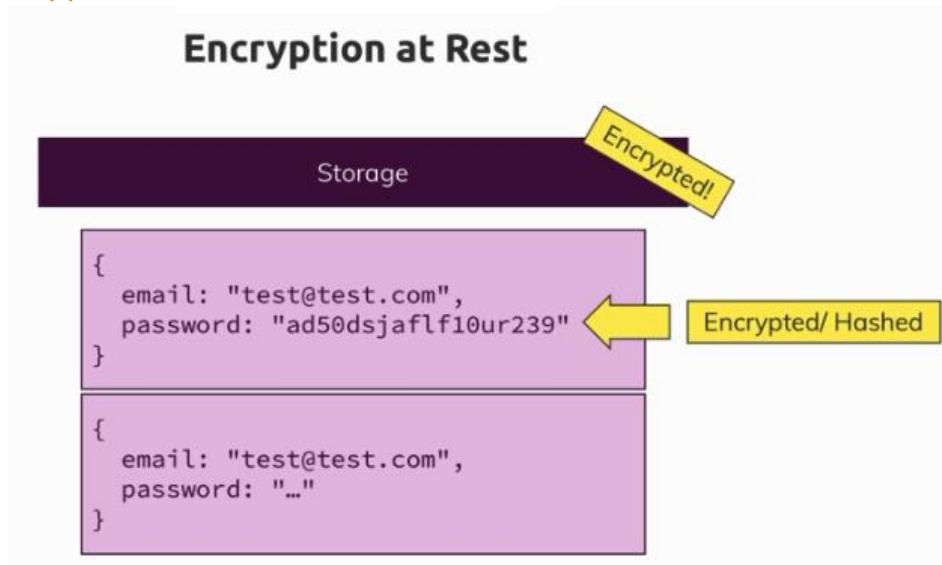


- Another important building block is transport encryption, this means that data that is sent from your app to the server should be encrypted so that no one can sit in the middle and spoof your connection and read that data.
- MongoDB uses TLS/SSL to encrypt our data in transport.
- SSL in the end is just a way of encrypting our data whilst it's on its way and we'll use a public private key pair to decrypt this information on the server and to prove that we as a client are who we well make the server think we are. So essentially it's a secure way of encrypting our data and decrypting it on the server and whilst it's on its way, it's consistently encrypted.
- Configure mongod and mongos for TLS/SSL Docs: <https://docs.mongodb.com/manual/tutorial/configure-ssl/>

- What is SSL/ TLS?

<https://www.acunetix.com/blog/articles/tls-security-what-is-tls-ssl-part-1/>

Encryption at Rest

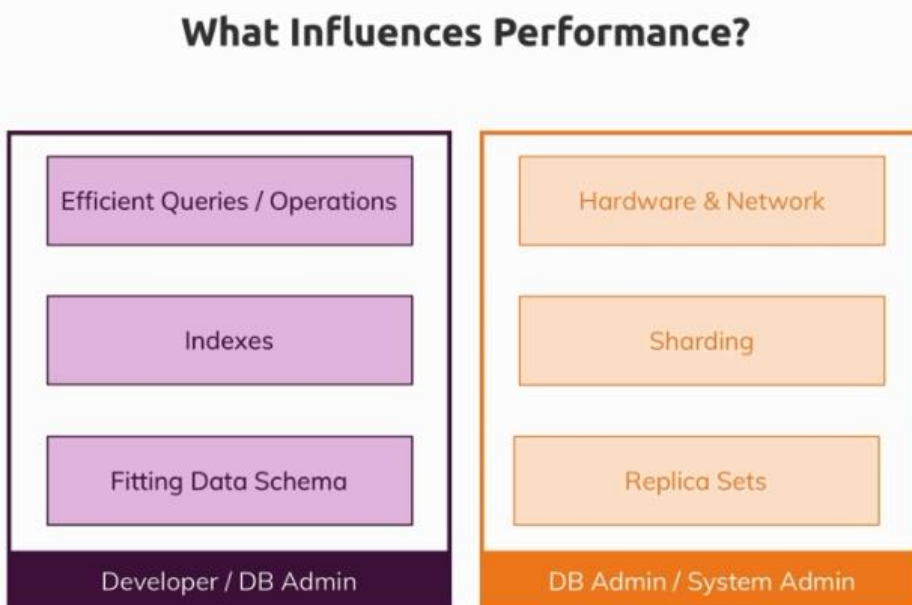


- Encryption at rest simply means that the data we store, so the data which is stored on our mongodb server in a file, that this of course might also be encrypted, so all the documents we have in there.
- There we can encrypt two different things.
- We can encrypt the overall storage, so the files themselves. And this is a feature which is built into mongodb for enterprise. There you can easily enable it.
- The thing you as a developer can and should always do is that you encrypt at least certain values in your code, so for example if you're storing a user password, you should hash that password and not store the plain text and that is a good pattern for any sensible data. You could even go so far to do it for all data, so that you always have a way of encrypting that.
- So you can encrypt and hash both your data as well as the overall file to have the most security possible.
- Docs: <https://docs.mongodb.com/manual/core/security-encryption-at-rest/>

Performance, Fault Tolerance and Deployment

- These things are not directly your job as a developer but still important for you to understand and especially important to you if you're working alone on a project.
- You need to understand what impacts performance of your mongodb server or your mongodb solution, how you can improve the fault tolerance of it and how you can deploy your local mongodb database and server into the web, so on a cloud machine or on a server in the web and not on your localhost of course because your web application, your mobile application or whatever you're building will most likely also run well on some server or on some smart device, whatever it is and will communicate to your server in the web and not to your local machine you are developing on.

What influences Performance?

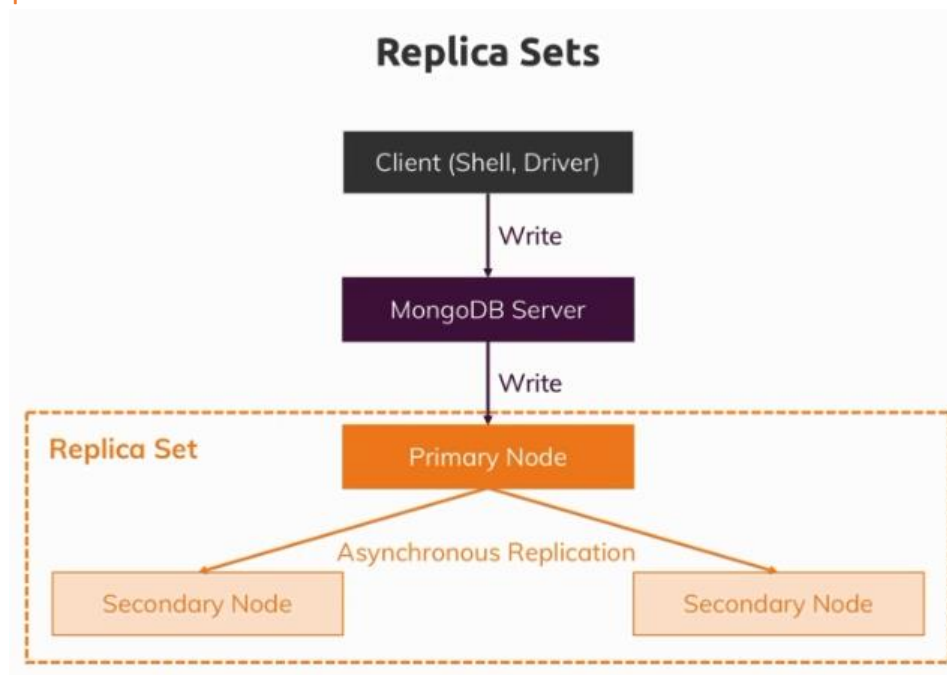


Capped Collections

- Capped collections are a special type of collection which you have to create explicitly where you limit the amount of data or documents that can be stored in there and old documents will simply be deleted when well this size is exceeded.
- So it's basically a store where oldest data is automatically deleted when new data comes in.
- This can be efficient for high throughput application logs where you only need the most recent logs or as a caching service where you cache some data and if the data then was deleted because it hasn't been used in a while, well then you're fine with that and you can just re-add it.

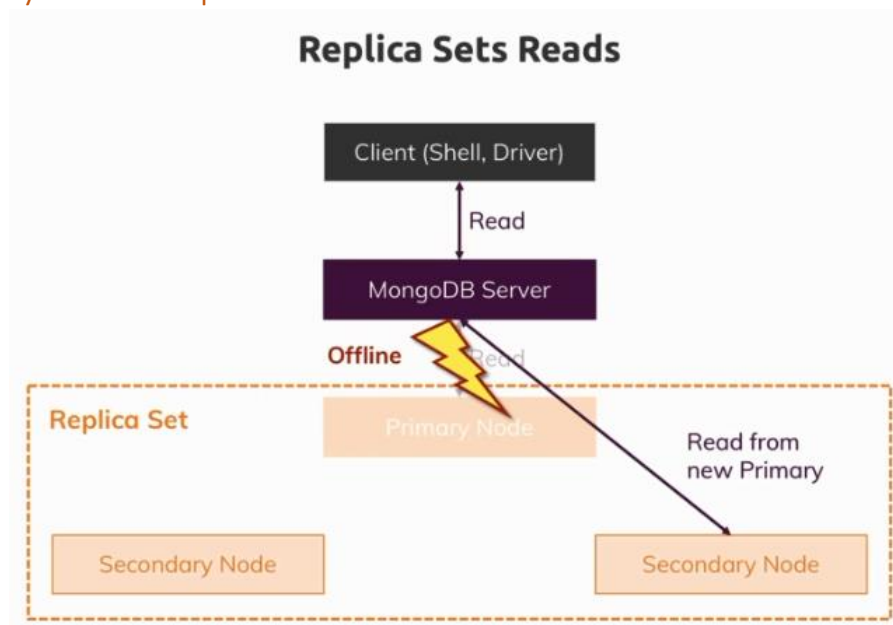
- To create a Capped collection ,
E.g. Here we are using the collection name as "capped"
> use performance
switched to db performance
> db.createCollection("capped", {capped: true, size: 1000, max: 3})
- The important thing is for a capped collection, the order in which we retrieve the documents is always the order in which they were inserted. For a normal collection, that may be the case but it's not guaranteed.
- If you want to change the order and sort and reverse, there is a special key you can use and that is **\$natural**, the natural order by which it is sorted.
E.g.
> db.capped.find().sort({\$natural: -1}).pretty()
- Why would you use a capped collection instead of a normal collection?
Because of the **automatic clear up**. You can keep this collection fairly small and therefore, efficient to work with and you don't have to worry about manually deleting old data. So for use cases where you need to get rid of old data anyways with new data coming in or where you need high throughput and it's ok if you lose old data at some point, like for caching, then the capped collection is something you should keep in mind as a tool to improve performance.
- Capped Collections Docs: <https://docs.mongodb.com/manual/core/capped-collections/>

Replica Sets



- Replica sets are something you would create and manage as a database or system administrator.
- A node is a single database server. Typically you may have a mongodb server which is connected to multiple nodes.
- So typically we have a primary node. And you can add more nodes, so-called secondary nodes, so these are additional database servers which are all tied together though in a so-called replica set.
- Now the idea here is that you always communicate with your primary node automatically. You don't need to do that manually, this happens automatically, If you send an insert command to your connected mongo server, it will automatically talk to the primary node but behind the scenes, the primary node will asynchronously replicate the data on the secondary nodes and asynchronously simply means that if you insert data, it's not immediately written to the secondary nodes but relatively soon. So you have this replication of data.
- Docs: <https://docs.mongodb.com/manual/replication/>

Why do we replicate Data?

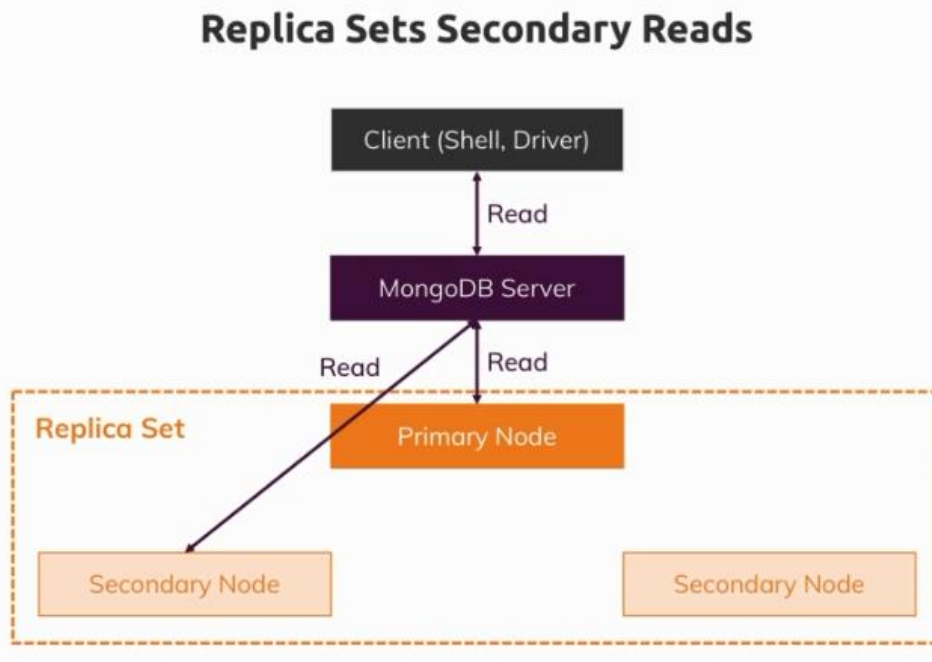


- If we read data and for some reason, our primary node should be offline, that we can reach out to a secondary node that will be then the elected new primary node, the secondary nodes in a replica set hold a so-called election when the primary node goes down to elect and select a new primary node and then we talk to that new primary node until our entire replica set is restored.
- So we get some fault tolerance in here because even if one of our servers you could say goes down, we can talk to another instance another node in that server network, in that

cluster so to say to still read data and as a new primary, we can then also not just to read but also write data.

- So this is why we use replica sets, we have the **back up** and **fault tolerance** and we get **better read performance** as well.

Read Performance

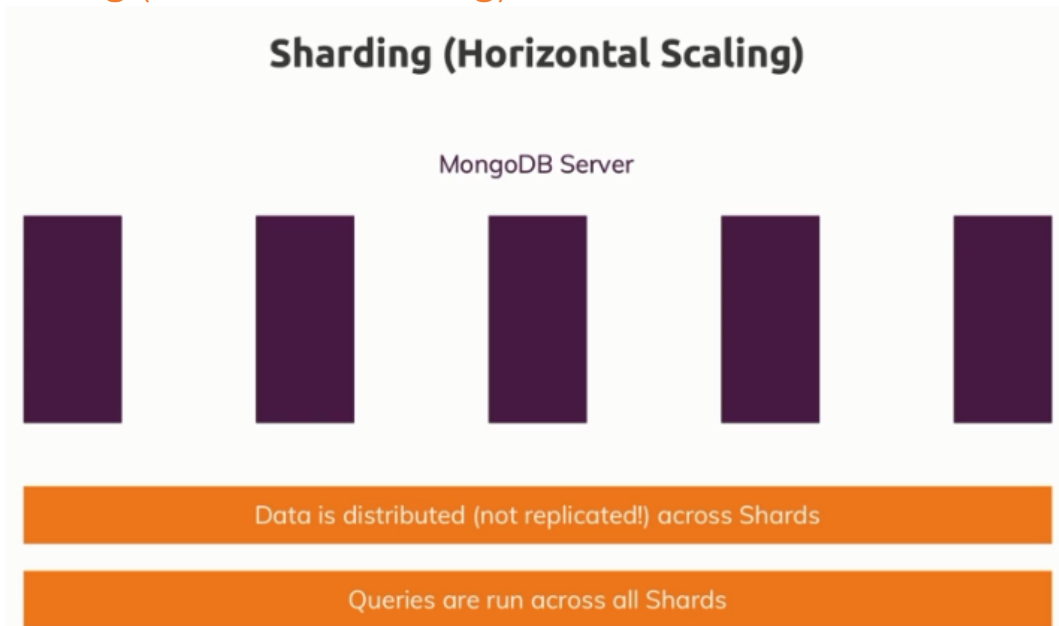


- This is of course fine if the primary node is online but even if it does not go offline, you can configure everything such that your backend will automatically **distribute incoming read requests across all nodes**.
- The writes will always go to the primary node but read requests can be if the server is configured appropriately and that is a task of your system or database admin and that the reads can also talk to secondary nodes.
- And the idea here that you want to ensure that you can read your data as fast as possible and if you have an application where you have thousands of read requests per second, then it is awesome if you can read not just from one node which is still one computer who has to handle all of that but if you can read from multiple computers and therefore, you kind of **split the load evenly**.

How to create Replica Set

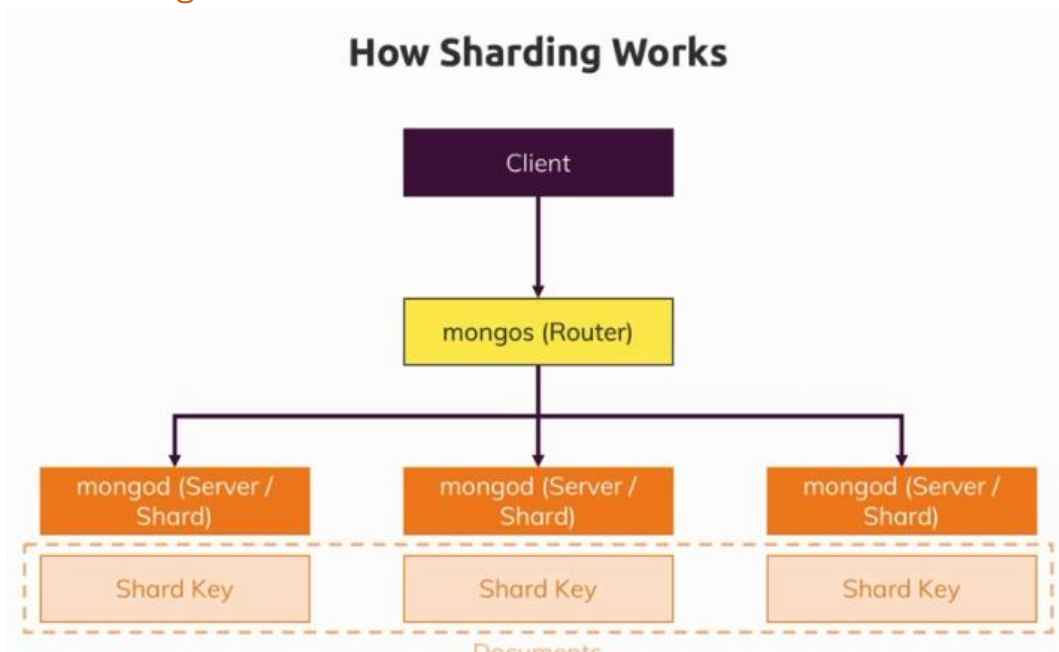
- This is an administrative task.

Sharding (Horizontal Scaling)



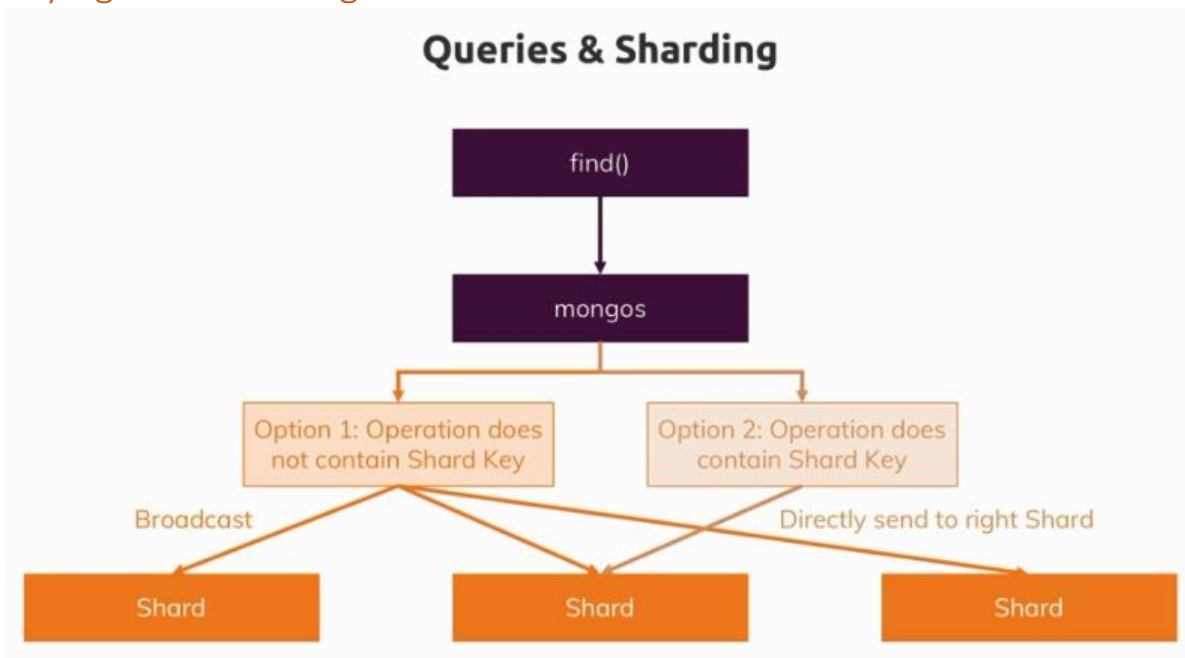
- Sharding is all about horizontal scaling.
- If you have a mongodb server (just a computer) which runs your mongodb server and where your database files are then stored on, if you have that server and you need more power because your application started and everything is going good and you've got more requests coming in, you've got more users, you've got more read and write operations, what can you do? Well you can upgrade your server, you can buy more CPU, more memory and put that into the server and if you use a cloud provider, you can typically upgrade this with a single click.
- Now that is of course a solution but it only gets you so far because at some point, you can't squeeze more CPU and more memory into a single machine and at that point, you need horizontal scaling which means you need more servers.
- The servers in Horizontal scaling don't duplicate the data, they are not backups, they split the data.
- With sharding, you have multiple computers who all run mongodb servers but these servers don't work standalone but work together and split up the available data, so the data is **distributed** across your shards, **not replicated**.
- And queries where you find data but also insert, update and delete operations therefore have to be run against all the servers or the right server because each chunk manages its data and its range of the data.
- Sharding is all about distributing data across servers and then setting up everything such that it can be queried and used efficiently.

How Sharding works?



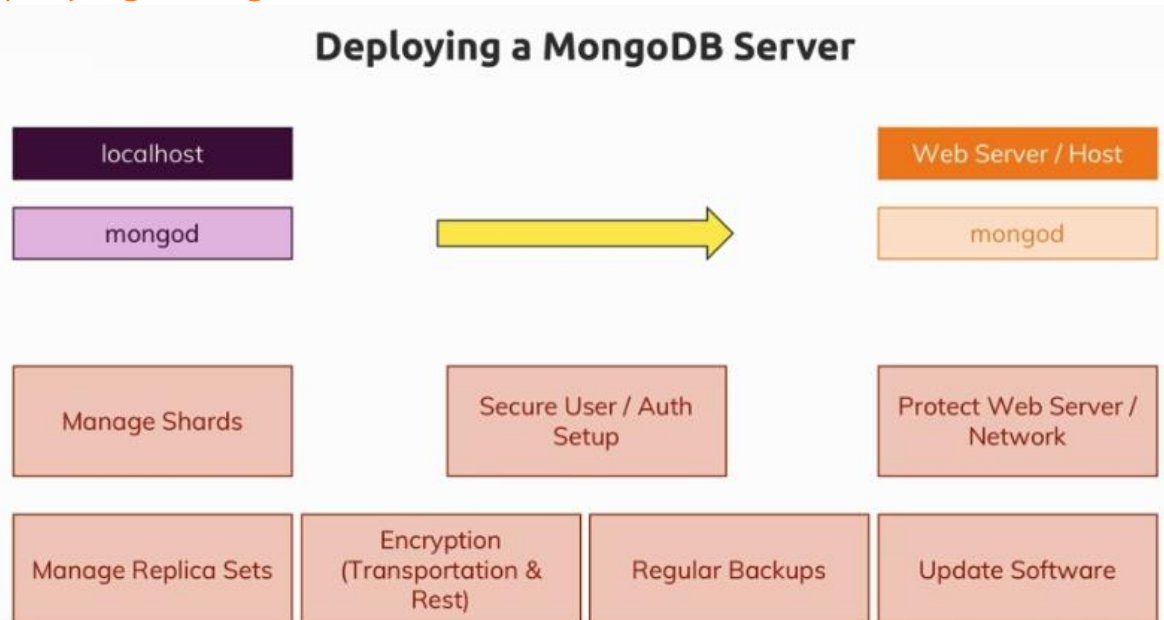
- This **mongos router** is responsible for forwarding your operations, inserts, reads and so on to the right shards.
- So the router has to find out which shard is responsible for the data you're inserting, so where this should be stored and which shard will hold the data you want to retrieve.
- For this splitting, you'll use a so-called **shard key**. Now a shard key is essentially just a field that's added to every document which kind of is important for the server to understand where this document belongs to.
- Now this shard key configuration is actually something which is not trivial because you want to ensure that you have a shard key that is evenly distributed. You can assign your own values here, you could say the shard key is the name of my user but then you should ensure that your usernames are roughly evenly distributed and not are old clunked at let's say names starting with A and then you've got no names starting with M, N, O, P, Q, R, S and so on because then you would put them all on one server which is not great.
- So choosing the shard key wisely is important and also something where you can read a lot in the official docs which is a job of your admin but this is then an important part of mongos (the router), finding out where an operation should go, so which server is responsible for storing the incoming data and so on.

Querying with Sharding



- That is the part that then matters for you as a developer again.
- So if you know you are using sharding, you should sit together with your administrators and choose a wise shard key keep based on your application needs that is evenly distributed so that you can write queries that use that shard key as often as possible.

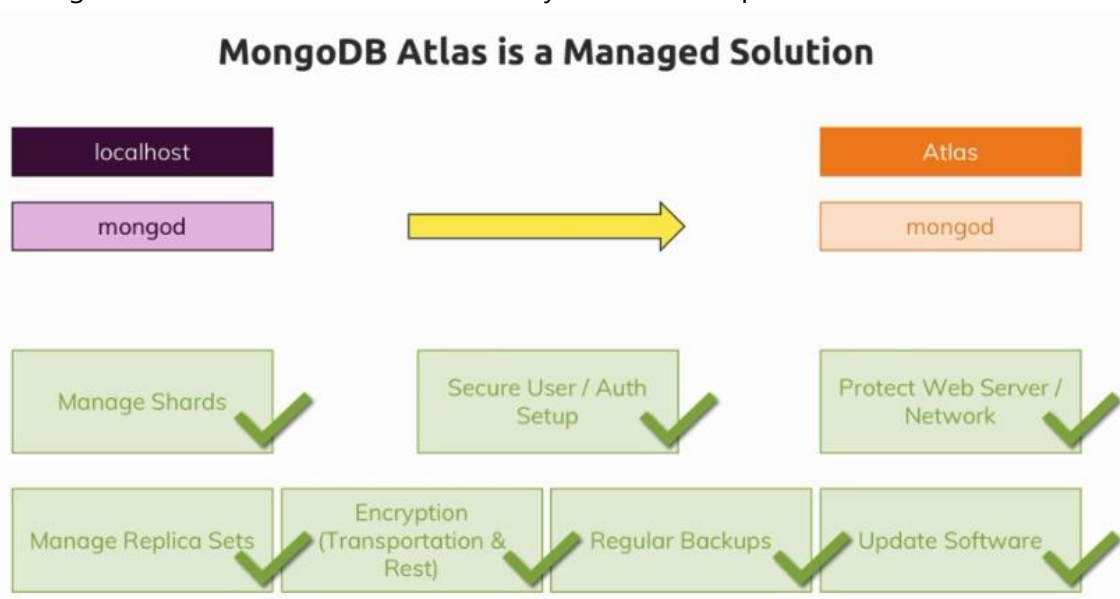
Deploying MongoDB Server



- We want to get our localhost mongod instance into the web, so we want to get it on a server that we can reach from anywhere and not only from our computer.
- Now deploying a mongodb server is a complex task, it's definitely more complex than just deploying a website because you need to do a lot of things, a lot of configuration.
- You have to manage all parts shown above, but luckily there is a managed solution for this from the MongoDB company called MongoDB Atlas.

MongoDB Atlas

- **Mongodb Atlas** is a managed solution, service provided by MongoDB that gives you a scalable and best practice mongodb server running in the cloud which you can configure through a convenient interface and which you can scale up, scale down.

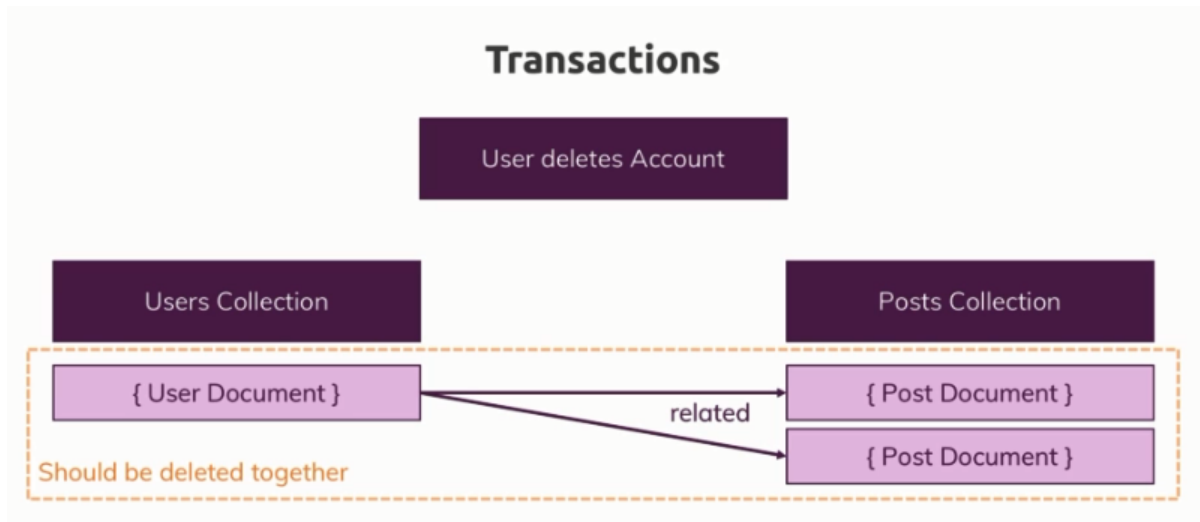


- A cluster contains all your shards, all your replica sets, it's basically what you deploy, it's your deployed mongodb server.

Transactions

- You need a replica set environment and MongoDB 4.0 for transactions to work.
- Transaction Docs: <https://docs.mongodb.com/manual/core/transactions/>

What are Transactions



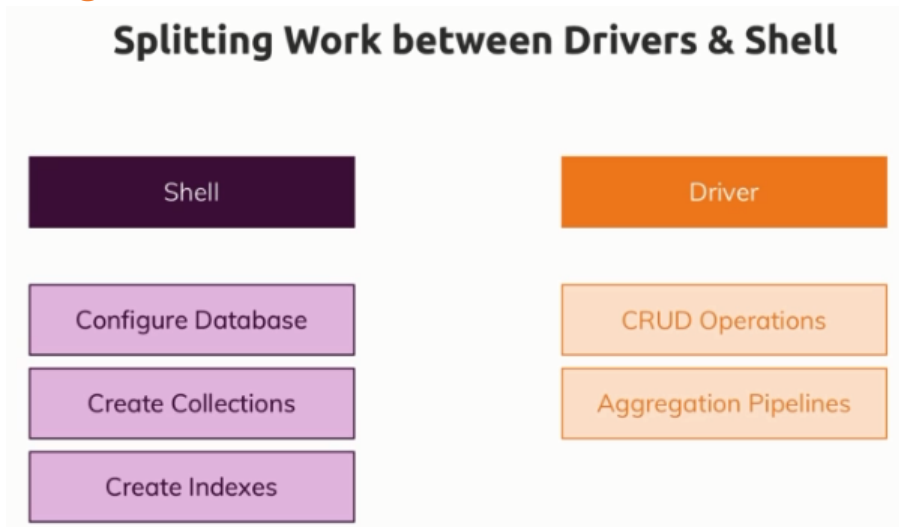
- With a transaction, we can basically tell mongodb hey these operations as many as you want either succeed together or they fail together and you roll back, so you restore the database in the state it was in before the transaction.

How does a Transaction Work?

- Now for a transaction, we need a so-called session.
- A session basically means that all our requests are grouped together logically.
- Mongodb is atomic on a document level, so if you write a document, it either is written entirely or not written at all.
- Transactions basically gives you atomicity across multiple operations, so across multiple steps or even something like deleteMany. Normally each operation on its own would be guaranteed to work but if you delete two posts here, you're not guaranteed that both deletes will work. So with a session, you could even just wrap delete many or insert many to guarantee that either all documents are deleted or inserted or none at all. So you get atomicity on an operation level so to say and not just on a document level.
- That is the idea behind transactions and that is a useful feature for situations where you know that some operations depend on each other.
- You should definitely not overuse it though, if you don't need it don't use it because obviously this takes a bit more performance than a normal delete or insert, so only use it if you know you need that cross operation consistency.

From Shell to Driver

Splitting work between Shell and Driver



- The initial database setup is not something your application deals with, it assumes that the database is there and that you can then communicate with the database. So this whole setup stuff will be done from inside a shell or basically up front.

MongoDB Drivers

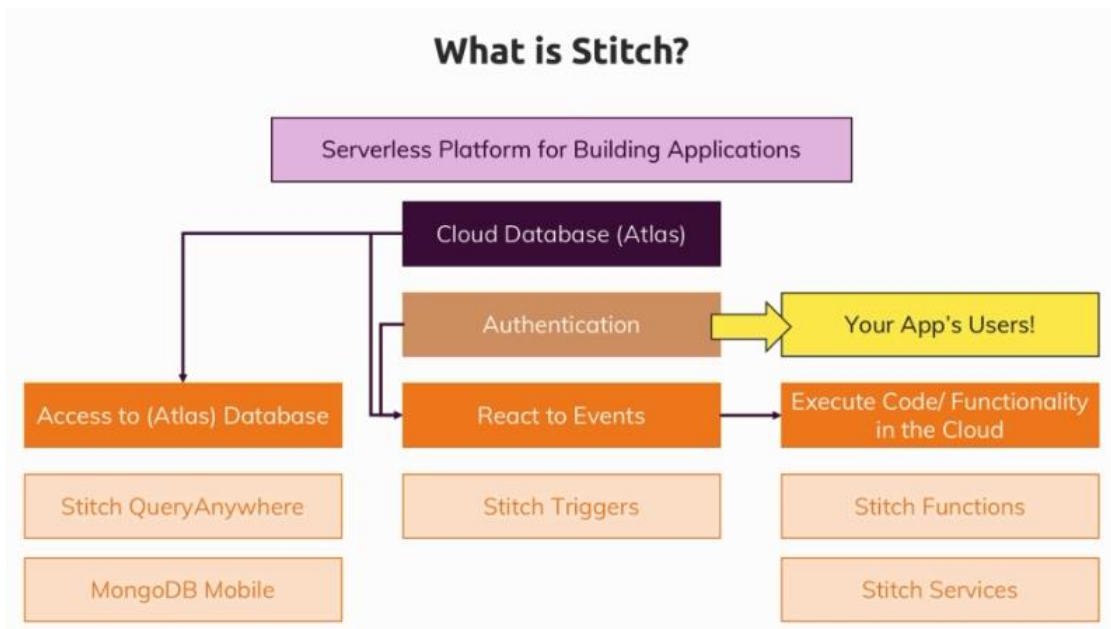
- Docs: <https://docs.mongodb.com/drivers/>
 - This page has list of drivers for most of the languages (e.g. Node, Python, etc.)
- MongoDB Node Driver: <https://docs.mongodb.com/drivers/node/>

Course Project

- <https://github.com/sameerbhilare/MongoDB/tree/main/workspace/05-shell-to-driver-project>

Introducing Stitch / MongoDB Realm

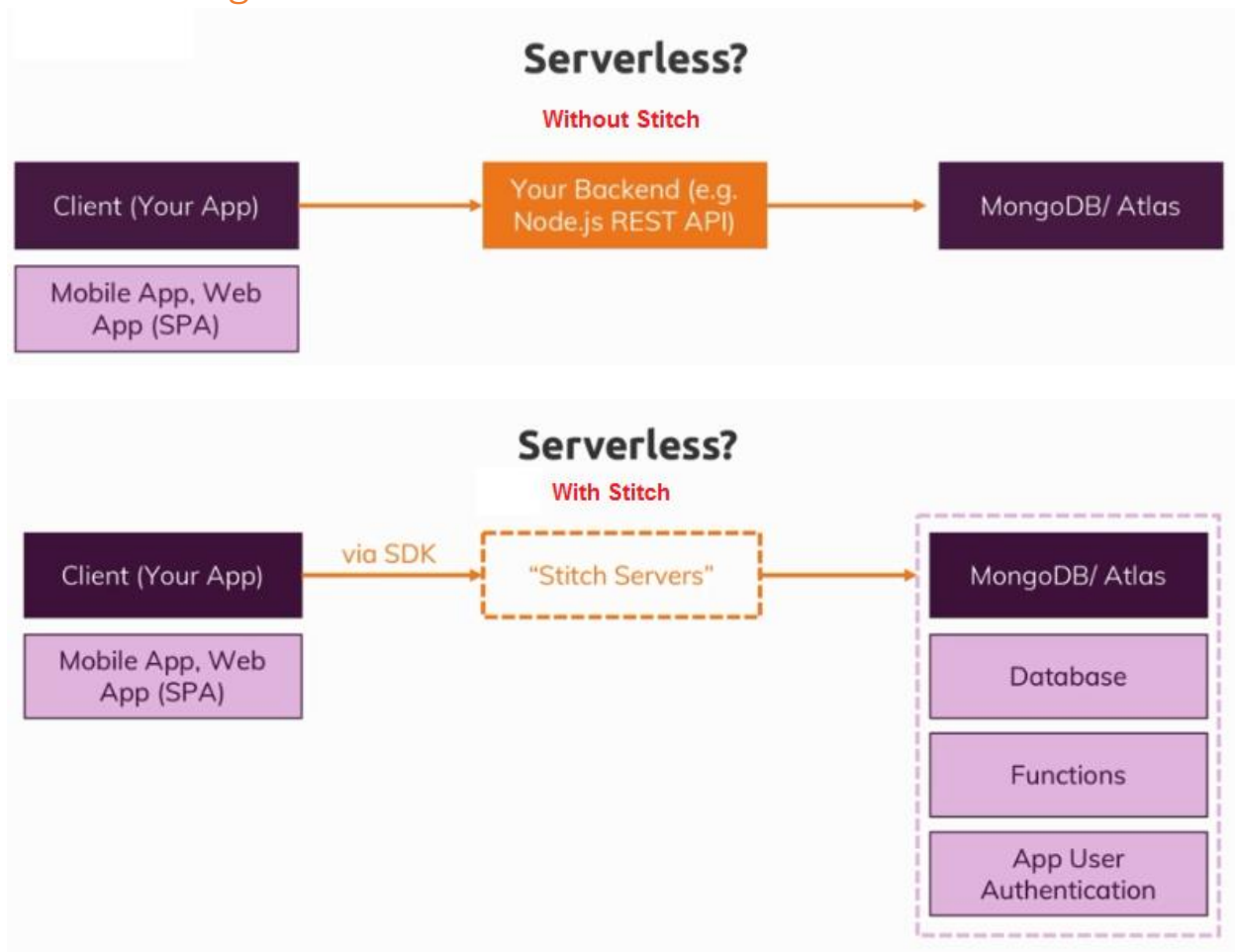
What is Stitch



- MongoDB Stitch is now labeled "MongoDB Realm".
- MongoDB Realm Docs: <https://docs.mongodb.com/realm/>
- It's a serverless platform for building applications you could say.
- It's a collection of services offered by the mongodb company that you can use when building an app (web app or a mobile app or also a desktop app), so that you get to focus on your user interface and your core logic even the logic which should run on a server so on a backend but you don't have to write the entire boilerplate for setting up the server, for managing the server or for example for creating a restful API on your own, that is all handled by stitch.
- It all is kind of built around Atlas as a cloud database or at least integration with that is particularly easy and one core feature for example is **authentication (for app users not db users)**.
- Stitch gives us access to our mongodb Atlas database and this access will then be available from inside our client side application, for example, it will be available in the react app.
- Now the idea behind stitch's solution here is that it doesn't expose our database credentials, instead our application users can sign up and log in through stitch's service and the user will get temporary credentials for finegrained access to the database and we can lock down what a single user can do.

- So stitch manages this behind the scenes for us, that it takes our rules into account and make sure that users do have access to the database but only to exactly the things they have to do.
- We can also **react to events** in stitch, so for example if something gets inserted into a database or something gets updated, we can **run some code** that does something, sends an e-mail or log something into another collection or whatever we need to do.
- So stitch is not just a toolset which helps us build beautiful UIs or client side code, we can still execute code in the cloud with so-called functions, so essentially just code snippets which we define where we don't have to write the boilerplate of parsing incoming requests or exposing routes in a restful API but where we just write a code we want to execute and then we can either call that code directly from inside the client or from inside another function or we set up a trigger with "react to events" to execute a certain code snippet when for example something gets inserted.
- **Mongodb stitch as a whole is kind of mongodb's answer to firebase.**
- For accessing the database, we got **Stitch queryAnywhere** which is their solution to allow us to run queries based on the rules we set up for the users of our app directly to our database.
- There also is **mongodb mobile**, which is basically a local mongodb database which you can install on mobile devices so that you can store data and sync it to the cloud even when you are offline. So store the data when you're offline, sync it to the cloud when you're back online.
- React to events, that is **stitch triggers**, that's simply the name of the service which allows us to configure triggers that will then call **stitch functions**. These functions will be on the MongoDB Atlas server so user can't see these.
- Additionally there's a feature called **stitch services** which allows us to integrate stitch with other services, like for example AWS S3. That would be useful for uploading files because stitch has no built-in file storage, so we got no solution for storing files, mongodb itself is a bad solution, a database should not be used as a file storage and there is no other service that would allow us to store files.
- Stitch really helps you build complete applications with everything you need, authentication, server side logic, even file storage if you use a service like S3 and of course, you have the entire mongo database at your disposal too.
- If you're building modern applications and you want to focus on your code, on your logic, then using a solution like stitch together with all the power and the cool features mongodb, the database gives you is really a great approach that allows you to build amazing applications quickly.

Idea of using Stitch



MongoDB Realm Cloud

- MongoDB Realm Cloud is a hosted application backend that enables real-time data sync between the Realm SDKs and MongoDB Atlas. It also includes a suite of tools for building cross-platform applications like user authentication providers, serverless functions & triggers, and dynamic data access rules.
- Docs: <https://docs.mongodb.com/realm/cloud>

Tips and Tricks

- As database administrator, you can use `db.runCommand` in the shell to run administrative command.
- Images should be stored on a file storage and not in the database. If you also have a web application with upload and so on, you would store the uploaded images in a file storage and then only store the path to the image in the database, not the image itself because this simply bloats your database, is pretty inefficient, a lot of data to transfer and just not what a database is built for.