EXPRESS

What is ExpressJS
↓
Using Middleware
↓
Working with Requests & Responses
↓
Routing
↓
Returning HTML Pages (Files)

⇒ Express: FRAMEWORK

Server logic is complex
→ data Buffer
→ Different Headers
→ Header & HTML

Express heavily depends upon Middlewares:
↓
Incoming Request is funneled through bunch of functions through expressJS.

Request
↓
Middleware
↓ next()
Middleware
↓ res.send()
Response

app.use( );
↳ Method for defining middlewares

Middlewares
→ app.use (Application level)
→ router.use (Router level)
→ express.static, express.urlencoded (Built in)
→ bodyParser, cookieParser (3rd Party Middleware)

* res.sendFile() takes absolute path.

Forever
Page No.
Date : / /201

⇒ Serving Static Files :

make an exception for a folder to serve static files (access file system)

```
app.use (express.static (path.join ("__dirname", "public")));
```

Managing data.

↓

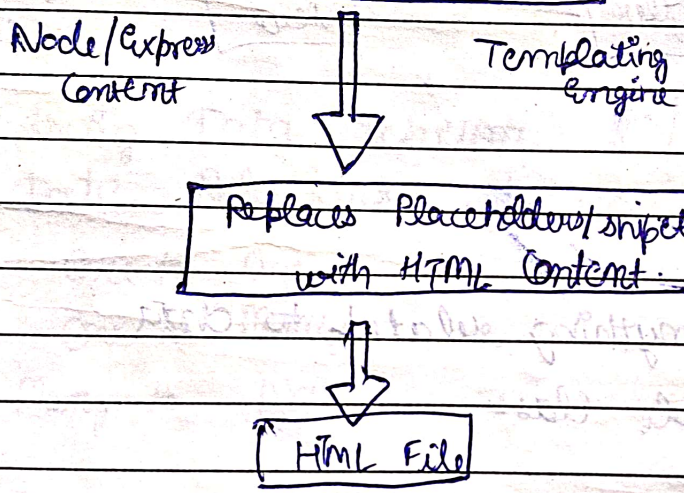
Render Dynamic Content in our Views

↓

~~Understanding Templating Engines~~

⇒ Templating Engines

↳ HTMLish Templates

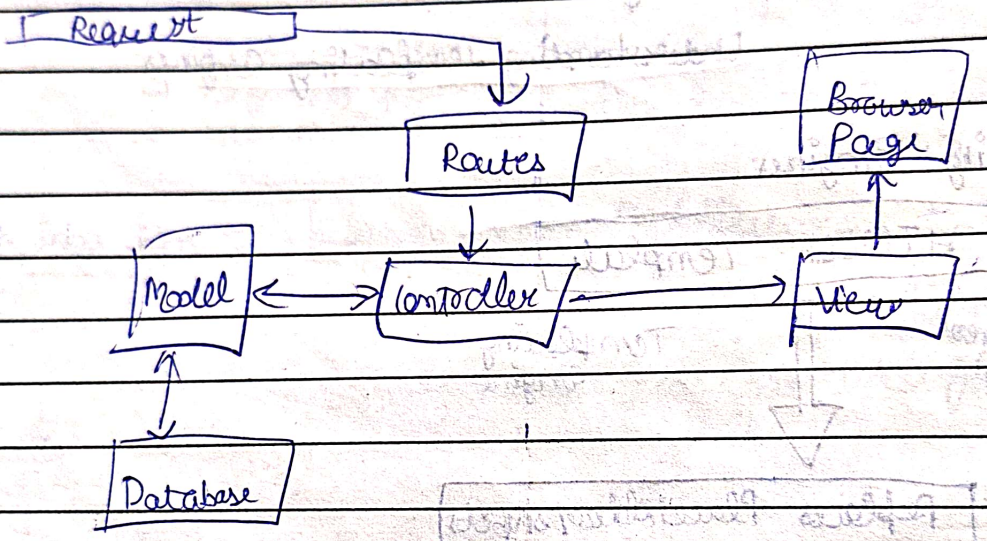


eg. EJS, Pug (Jade), Handlebars

```
app.set ('view engine', 'ejs' / 'pug');
app.set ('views', 'folder of views');
res.render (file, data);
```

MVC Architecture

Models	Views	Controllers
→ Represent your data in your code	→ What the user sees	→ Connecting your Models & your views
→ Works with your data	→ Decoupled from your application code	→ Contains the "in-between" logic



Model → It can be anything related to data
- File
- Simple class
- SQL
- Mongo.

⇒ Dynamic Routes

/:id → Anything, and now name is id.
req.params.id.

Databases

Forever
Page No.
Date: / /201

Goal: Store Data & make it easily accessible

↓

Use a Database — Quicker Access than with a file

↓ SQL — e.g mySQL

↓ NoSQL — e.g mongoDB

What's SQL ? (Structured Query Language)

⇒ Data is stored in tables
⇒ Each table has columns / attributes
⇒ Each data is represented as row / fields / records

Chars:

⇒ Strong Data Schema , / id name age.
⇒ Data Relations]- Connected Tables
  One to One
  One to Many
  Many to Many.

What's NoSQL?

⇒ Data is stored in collections
⇒ Each collections has document
  ↳ looks like javascript objects

Chars:

⇒ Weak Data Relation
⇒ No Strong Data Schema

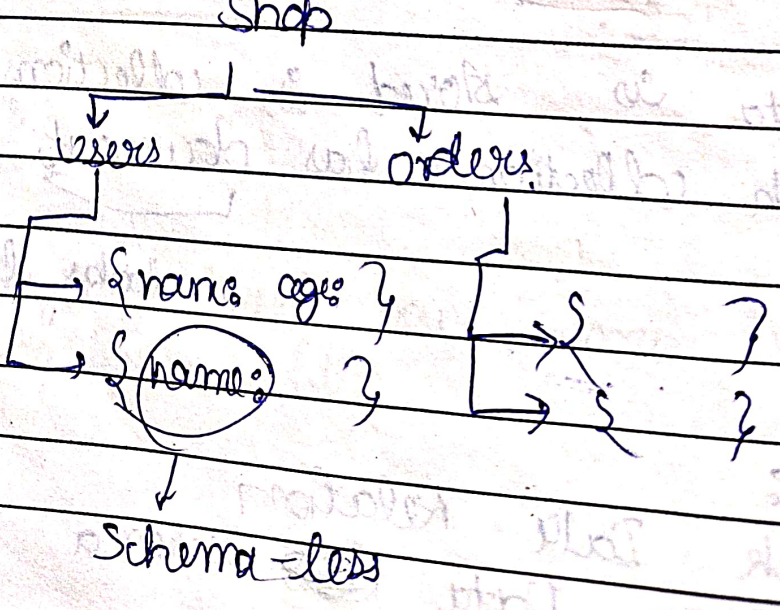
Horizontal scaling
↓
Add more servers
(and merge Data into one DB)

Vertical Scaling
↓
Improve Server Capacity/ Hardware.

SQL	NoSQL
→ Data uses Schemas Relations	→ Schema less → No (or few) relations
→ Data is distributed across multiple tables	→ Data is typically merged/ nested in a few collections
→ Horizontal scaling is difficult / Vertical scaling is possible.	→ Both horizontal & vertical scaling is possible
→ Limitations for lots of read & write queries per second	→ Great performance for read & write requests

Mongo DB

Database
↓
Collections
↓
Documents



Forever
Page No.
Date: / /201

JSON format looks like (BSON) format actually.
↓
Binary Json.

```
{
    key     value
    " "  :  {          object
    }
    " "  : [ { } , { } ]     Arrays of objects
}
```

→ JSON Format.

* MongoDB fetches other relations, not neither actually duplicating them nor do they relate them.
* They just embed it and whenever we need it it fetches to make it fast & efficient

Hence, Relations in MongoDB can be called as:

Nested / Embedded Documents

```
{
    name:
    address : {
        street:
        city :
    }
}
```

Reference

```
{
    username:
    favbook : ['id1', 'id2']
}

{ _id: 'id1',
  name:
}
```

⇒ Shell :

$ sudo service mongod start
$ mongo

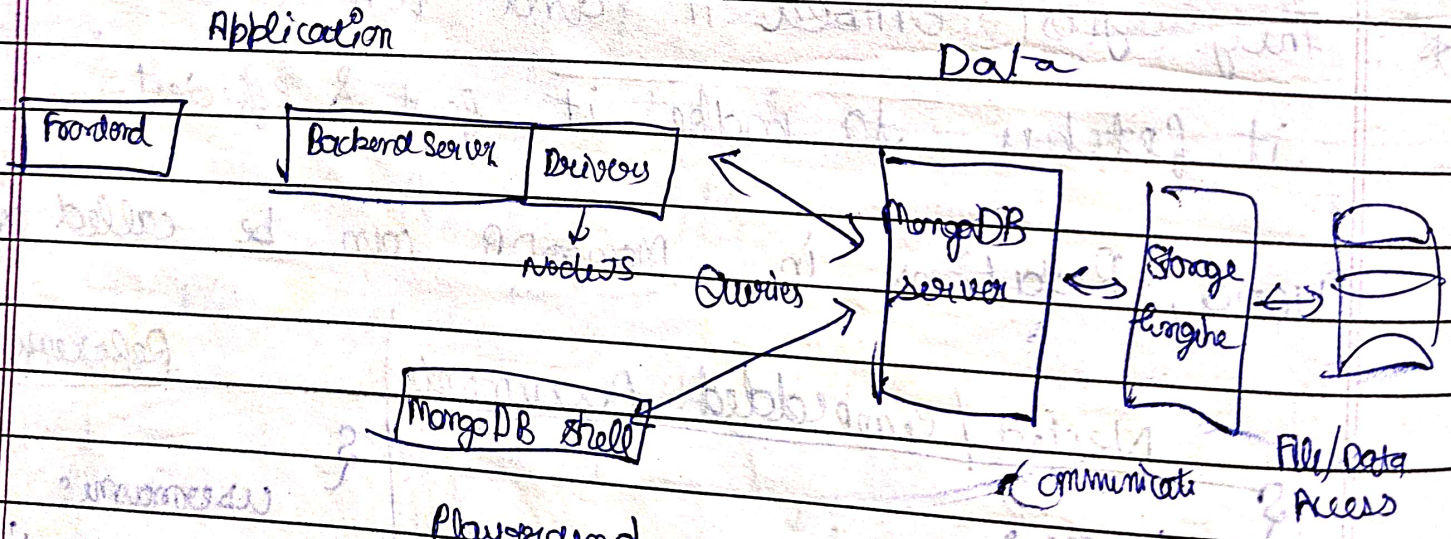
show dbs
use new_db

& many more commands.

⇒ Drivers:

Drivers are just the ecosystem allowing different languages to work with the mongoDB server.

Working with MongoDB



Forever
Page No.
Date : / /201

Create, Read, Update & delete

⇒ Basic CRUD:

I. Create
insertOne (data, options)
insertMany (data, options)

II. Read
find (filter, options)
findOne (filter, options)

III. Update
updateOne (filter, data, options)
updateMany (filter, data, options)
replaceOne (filter, data, options)

IV. Delete
deleteOne (filter, options)
deleteMany (filter, options)

⇒ show dbs
→ show collections → basic commands.
⇒ use <db-name>
⇒ db.dropDatabase()

db.<collection_name>.insertOne (Object)
insertMany ([Array of objects]) → Insertion

db.<collection_name>.find ({ filter: }).pretty()

filter can be anything

({"distance": 100}) , operators

{"distance": {$gt: 100}}
greater than 100

.findOne (filter).

Date : / /201

→ db.<collection-name>.updateOne(filter, {atomic operator});

$set : { option1 : "value1" }

.updateMany(filter, { });

.update(filter, object);

↓

Changes the whole object

.replaceOne(filter, object);

Same like update but more reasonable

→ delete is same as insert or read

✱ .find() gives a cursor

which can be used to apply other methods.

for ex – toArray()

forEach(( ) => { })

→ PROTECTION:

Projection

db.<collection-name>.find({}, {key: 1, key2: 0}).limit(5)

Not included

↑ cursor modifiers

Forever
Page No.
Date : / /201

⇒ EMBEDDED DOCUMENTS:

Data

```
{   Array
"hobbies" : [
        "sports",
        "cooking"
]
}
```

access :

findOne({ }).hobbies

find({ hobbies : "sports" })

↓

look up h array.

```
{   Array of Objects.
hobbies : [
    {   "sports" : [ "cricket", "baseball" ]
    }, {
        "description" : { "status" : "lazy" }
    } ]
}
```

Later

↓

```
{
"hobbies" : {
        "sports" : "cricket",
        "swim" : "champ"
}
}
```

find({ "hobbies.sports" : "cricket" })

⇒ DATA SCHEMAS AND DATA TYPES:

Isn't MongoDB schema-less?

↓

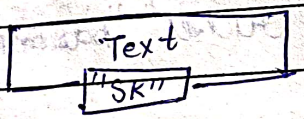
→ MongoDB enforces no schemas! Documents don't have to use the same schema inside of one collection.

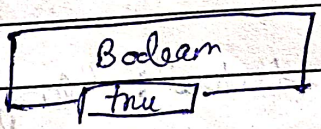
↓

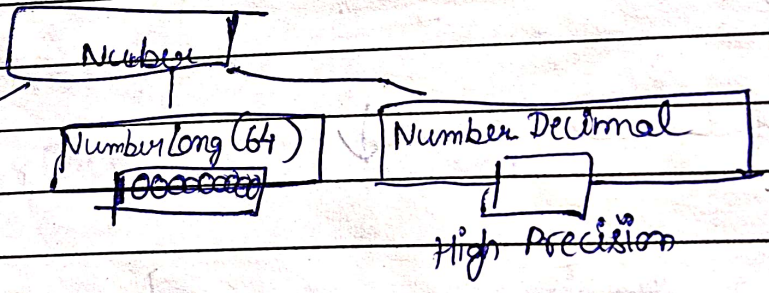
→ But that doesn't mean that you can't use some kind of schema.

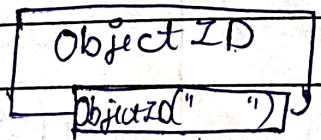
So, ultimately we kind of use both SQL & mongoDB world.

Data Types: (A document max. size is 16 MB)



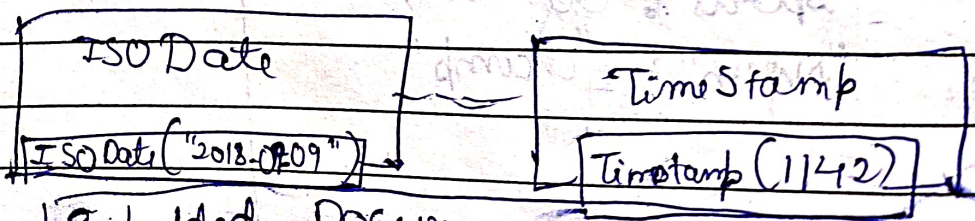






↓

Based on Time Sorting



Max level - 100 | Embedded Document + Array

Forever
Page No.
Date : / /201

⇒ Relations - Options

Is
the
Using Embedded documents (May have duplication)
Using References (Split data)

(a) One to One
Person → Car

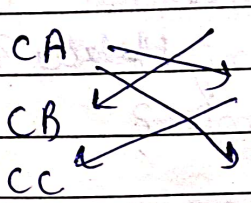
→ If we use references, it may be fast with small data but not optimal with large data,
So, we use embedded documents.

→ Again, when an application isn't interested in whole document but only parts, we will use reference

(b) One to Many
Thread → Q1
→ Q2

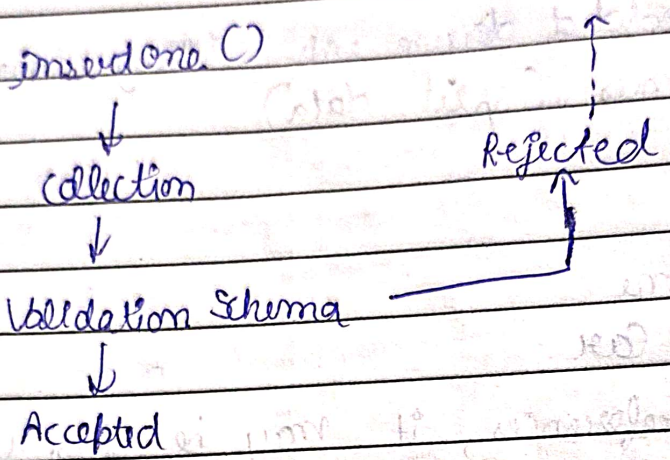
Embedded → Normally.
References → To not hit 16 MB limit

(c) Many to Many



Embedded, Lots & lots of data duplication
References → Maybe a good idea.

⇒ Schema - Validation:



Validation Level	Validation Action
Which document gets validated?	What happens if validation fails?
strict → All inserts & updates	error → Throw error & deny update/insert
moderate → All inserts & updates to correct docs.	warn → log warning but proceed

Summary:

I. In which format will you fetch data?
II. How often will you fetch & change your Data?
III. How much data will you save (how big it is?)
IV. How is your Data related?
V. Will Duplicates hurt you (=> many Updates?)
VI. Will you hit any storage limits?

Forever
Page No.
Date : / /201

⇒ Explore shell

1. Start MongoDB server:

(a) sudo service mongod start

Uses /var/lib/mongodb folder for dB

Uses /var/log/mongodb/mongod.log for logs.

(b) sudo mongod --dbpath <path>
--logpath <log-file-path>

starts mongod server in specified folder.

(c) sudo mongod --config <path-file_config>
save setting to path & start it.

⇒ Deeper Dive into Insert:

(i) insertOne] more descriptive & returns id
(ii) insertMany]
(iii) insert] confusing.

Bulk Write in insertMany

☆ If it fails, it continues upto failed

Ordered Insert:

Date : / /201

* ordered Insert:
Every element you insert is processed standalone but if one fails it cancels the operation but it doesn't roll back.

To change this default behaviour.

add this option

db.< >.insertMany([{},{}], {ordered: false}, option);

* write Concern:

It describes the level of acknowledgment from MongoDB for write operations to an instance or multiple, standalone.

writeConcern {
w: <value>, j: <boolean>, wtimeout: <number>
}

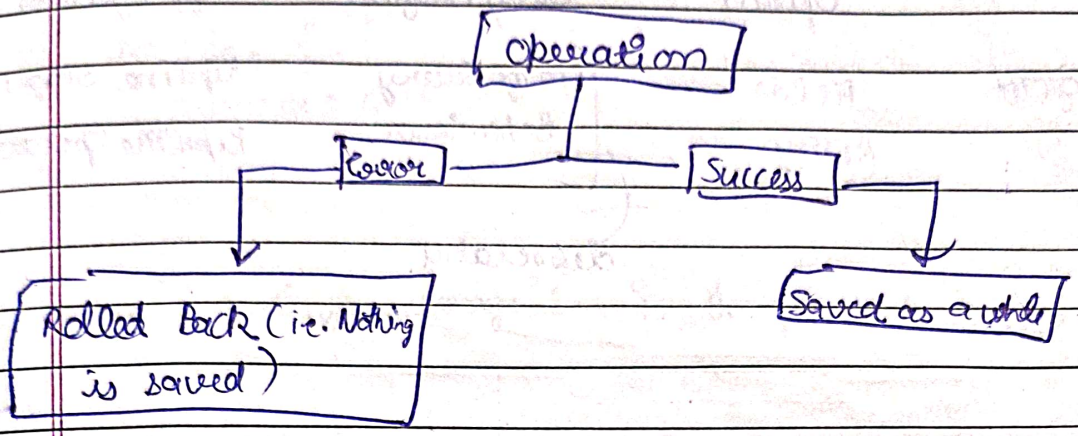
w: <value> → No. of instances.

j: <boolean> → true → Server + On-disk Journal Acknowledgment
false/undefined → Server Acknowledgment

wtimeout: <number> → Timeout: return error after specified time.

w: 0 → Acknowledged → false
doesn't guarantee if write operation is done or not

⇒ Atomicity :- on a document level, a guarantee is there if it is done or not.



⇒ Mongoimport

$ mongoimport ← json File path -d <db-name>

-c <collection name> --jsonArray --drop
↳ drop before inserting.

⇒ Deeper Dive Into Reads:

Access Current DB

```
db.myCollection.find({age: 32})
```

myCollection: Access this collection
find(): Method
{age: 32}: Filter. (Equality/single value)
age: field
32: value

Range Filter

```
{ age: { $gt: 30 } }
```

age: Field
$gt: operator
30: value

Date : / /201

Operators

Read
- Query & Projection

Update
- Fields
- Arrays

Query Modifiers
- Change Query Behavior
- Deprecated

Aggregation
- Pipeline Stages
- Pipeline Operators

For READ

Query selectors

- → Comparison
- → Logical
- → Element
- → ~~Geospatial~~
- → Evaluation
- → Array
- → Comments

Projection Operators

- $
- $elemMatch
- $meta
- $slice

a. Query Selectors:

i) Comparison

$eq	$lt	$lte	$in (SQL in)
$ne	$gt	$gte	$nin
			↓ not in

```
{$runtime : {$in: [x, y]}});
```

Forever
Page No.
Date : / /201

For embedded documents

I. {
ratings: {
average: 6
}
}

({"ratings.average": {$gt: 6}})

II. { genre: ["", "", ""] }

({genre: {$eq: "Drama"}})

({genre: "Drama"})

For array, it needs to find only one matching element

({genre: ["Drama"]})

It gives exact equality.

III) Logical:

$or

({$or: [{"rating.average": {$lt: 4}}, {"name": "Drama"}]});

$nor $and $nt.

iii) Element

→ $ exist (~~just~~ check for field
(~~only~~ fail for null values)

({ age : { $exist : true } })

→ $ type

({ phone : { $ type : "number" } })

"double"
"string"

iv) Evaluation

→ $ regex (regular expression)
Selects docs where values match a specified regular exp.

({ summary : { $ regex : /musical/ } });

Not quite efficient.

→ $ expr (later in aggregation module)

({ $expr : { $gt : ["$volume", "$target"] } });

Forever
Page No.
Date : / /201

v) Array

```
{
hobbies : [
        {
            frequency : 2
            title : sports
        },
        {
            frequency: 3
            title : Cooking.
        }
]
```

({"hobbies.title" : "sports"})

$ size

({"hobbies" : { $size: 3}});

$ all

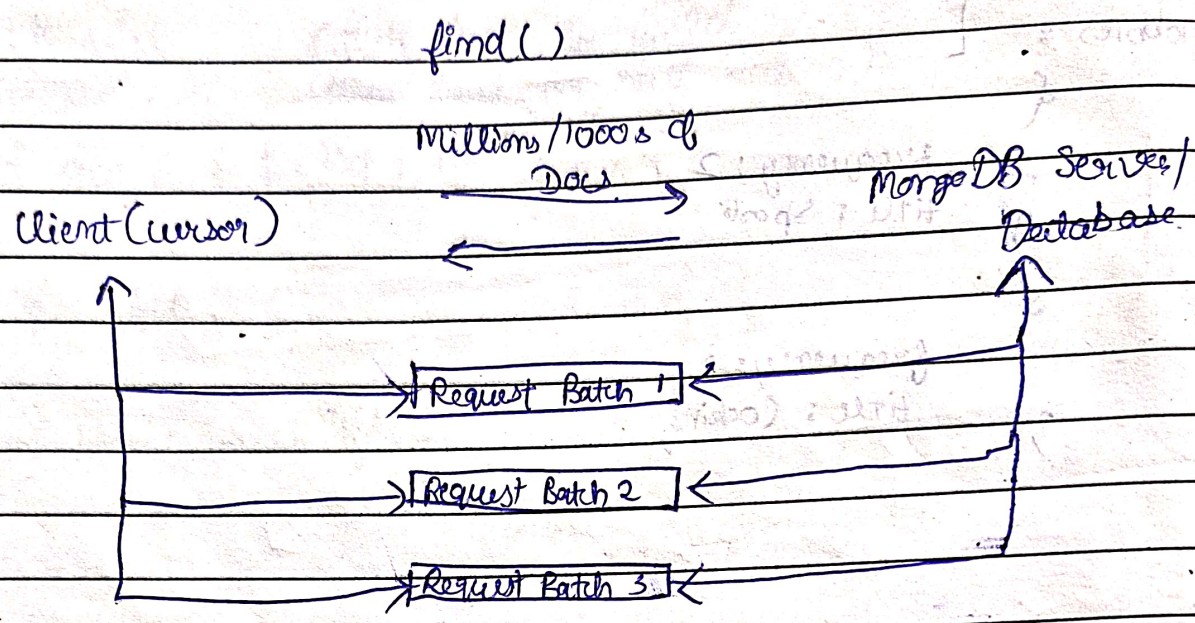
({"genre": { $all: ["action", "thriller"] }});

$ elemMatch

~~({"$and": [{"hobbies.title": "sport"}, {"hobbies.frequency~~

({"hobbies": { $elemMatch: { title: "sports", frequency: {$gte: 3}}});

⇒ CURSORS:



const dbCursor = db.<collection>.find()

dbCursor.count()
dbCursor.hasNext()
dbCursor.next() (Shows next doc in cursor)
dbCursor (Shows 20 a/c to shell)

dbCursor.forEach((doc) => printjson(doc));

print(tojson(doc));

dbCursor.sort({"rating.average": 1, runtime: -1});
Ascending
Descending

Already executed
~~dbCursor~~.skip(10)

.limit(5)

Forever
Page No.
Date : / /201

⇒ Projection

By default always including

db.<coll>.find({}, {_id: 0, name: 1}).

a) $

b) $elemMatch

Projects the first element in an array that matches the specified $elemMatch condition

c) $slice

Limits the no. of elements projected from array.

⇒ Diving Deeper into Update:

db.<coll>.updateOne({}, {$set: {}});

It can add if not present
It can override (only) an array.

$inc.

Add / subtract

{$inc: {age: 2}, $set: {age: 4}};

↓ increases if present appends if absent

Error! Two operations on same field.

Date : / /201

filter

$ min

{$ min : { age: 5 } }

if age is more than 5
it will be updated

$ max

{ $ max: { age: 3 } }

if age is less than 3
it will be updated.

{$ mul : { age: 5 } }

multiplied by 5

$ unset

Removes the attributes

{ $ unset: { age: "" } }

$ rename → Rename the attributes

{ $ rename : { age: "totalAge" } }

option

$ upsert

If set to true, creates new document when no document matches the query criteria.

Default - false.

Forever
Page No.
Date : / /201

{{}, {}, { upsert : true }}

Array operators

{ {}, { $set : { "hobbies.$" : { } } } }

changes the entire document with $ sign only that document is updated.

"hobbies.$.newAttribute" : true

"hobbies.$[]"

Changes all documents in the array.

{ {}, { $set : { "hobbies.$[el].goodfrequency" : true } }, identifier

{ arrayFilters : [{ "el.frequency" : { $gt : 2 } }] }

* Updates to all documents in array with certain filters

{ $push : { hobbies : { } } }

$each

$sort

$pull: Removes all array elements that match a specified query.

$pop: Removes first or last element in array.

{ $pop: { hobbies: 1 } }

1 → last

-1 → first.

$addToSet: Adds elements in array that are unique.

→ Diving Deeper into Delete operations:

db.<coll-name>.deleteOne({})

db.<coll-name>.deleteMany({}) → Delete all entries in collection.

db.<coll-name>.drop()

db.dropDatabase()

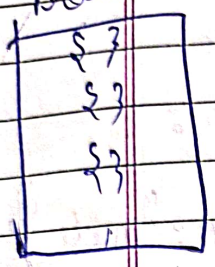
Retrieving Data Efficiently

INDEXES

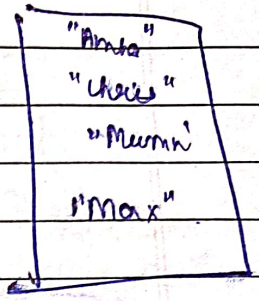
Forever
Page No.
Date : / /201

db. .find ({ name : "Max" })

No Index → Collection Scan ⇒



With Index → Ordered → Index Scan



db. <coll_name> . explain( ). find( )
update( )
delete( )

db. <coll_name>. explain("executionStats"). find( )

db. <coll_name> . createIndex ({ <field_name> : 1 })
Ascending

db. <coll_name> . dropIndex ({ <field_name> : 1 })

Compound Index

. createIndex ({ <field> : 1, <field> : 1 })

Default Index

db. <coll_name>. getIndexes ( )

↓

There is one by default for "_id",

Forever

Page No.

Date : / /201

Configuring Indexes

I. Add unique ({<field>:1}, {unique: true})

II. Partial Index

- createIndex({<field>:1}, {partialFilterExpression: {gender: "male"}})

↓

Smaller Index used when only a set of data is used

Time To Live Indexes

- createIndex({<field>:1}, {expireAfterSeconds: 10})

↓

→ Deletes document whenever it gets triggered.

→ Works on only Date Object

→ Only works on primary Index

⇒ Query Diagnosis & Query Planning

explain()

"queryPlanner"	"executionStats"	"allPlansExecution"
Shows summary for executed query + Winning Plan	Shows detailed summary for executed Query + Winning Plan + Possibly Rejected Plans	Shows detailed summary + Winning Plan + Plan Decision Process

Forever
Page No.
Date : / /201

→ Efficient Queries

Milliseconds Process time

IXSCAN ↑ COLLSCAN

Should be close or 0 [# of keys (in index) examined
# of document examined
# of docs returned] Should be close or 0

Covered Query

If an index have the ~~total~~ information about the document then there is no need to reach at the collection and fetch one.

ex. Let's say we have an index of name field

Now db.find({name: "Sanskar"}, {_id:0, name:1})

Here name is in the index.

So # of docs examined = 0

Hence, this is called covered query

⇒ REJECTING plans in MongoDB

I. Looks at Indexes which can help
II. Use all available approaches to search for some threshold documents.
III. Whoever wins is the winning plan.

Forever
Page No.
Date : / /201

IV. Now, mongoDB caches the winning plan to make future queries faster. (only for some future)

Cache is deleted when
- → Write Threshold (1000)
- → Index is rebuilt
- → Other Indexes are Added/Removed
- → Server Restarts

multi-key Index

on an array of collections/ docs

I. • createIndex ({ hobbies : 1 })
↓
array of "sports", "cook"

• find ({ "hobbies" : "sports" })
↓
Index scan is used

II. • createIndex ({ addresses : 1 })
↓
array of documents

• find ({ "addresses.street" : "main" })
↓
all scan is used

• find ({ "address" : { street : "main" } })
↓
Index Scan is used

Forever
Page No.
Date : / /201

For a multi-key index, mongoDB creates multi indexes

e.g- array has 4 docs, then we have 4 indexes

Text Index

- createIndex ({ description: "text" })

↓

Remove all the stop words and add all the keywords.

- find ({ $text: { $search: "awesome" } })

These are expensive indexes

Sorting in Text Index

- find ({ $text: { $search: "awesome t-shirt" }, score: { $meta: "textScore" } }).sort ({ score: { $meta: "textScore" } })

Configuration

- createIndex ({ name: "text", desc: "text" }, { default_language: "english", weights: { name: 1, desc: 10 } })

Forever

Page No.

Date : / /201

Note: Indexes lock the DB while getting created.
So, for production environment
set background: true

Working with Geospatial Data

GeoJSON object

({ name: { }, location : { type: "Point",
coordinates: [longitude, latitude]
}})

GeoJSON Query

(i) • find ({ location: { $ near : { $ geometry:
{ type: "Point",
coordinates: [longitude, latitude]
}, $ maxDistance :
$ minDistance :
}
});

Note: It requires · geospatial index

• createIndex ({ location: "2dsphere"})

(ii) Find locations inside a given polygon

• find ({ location: { $ geoWithin : { $ geometry:
{ type: "polygon",
coordinates: [[p1, p2, p3, p4, p1]]
}
}
});

(iii) Find places within a certain radius.

```
find ( { location : { $geoWithin : { $centerSphere : [ [long, lat],
                                                       1/6378.1
                                                       (Kms)
                                                     ]
                    } } } );
```

* "$Near" sort results also.

MongoDB Security

→ Authentication & Authorization
→ Transport encryption
→ Encryption at Rest

↑ Developer Concern

→ Auditing
→ Server & Network config & setup
→ Backup & Updates

↑ Server Admin Concern

1. Authentication & Authorization:

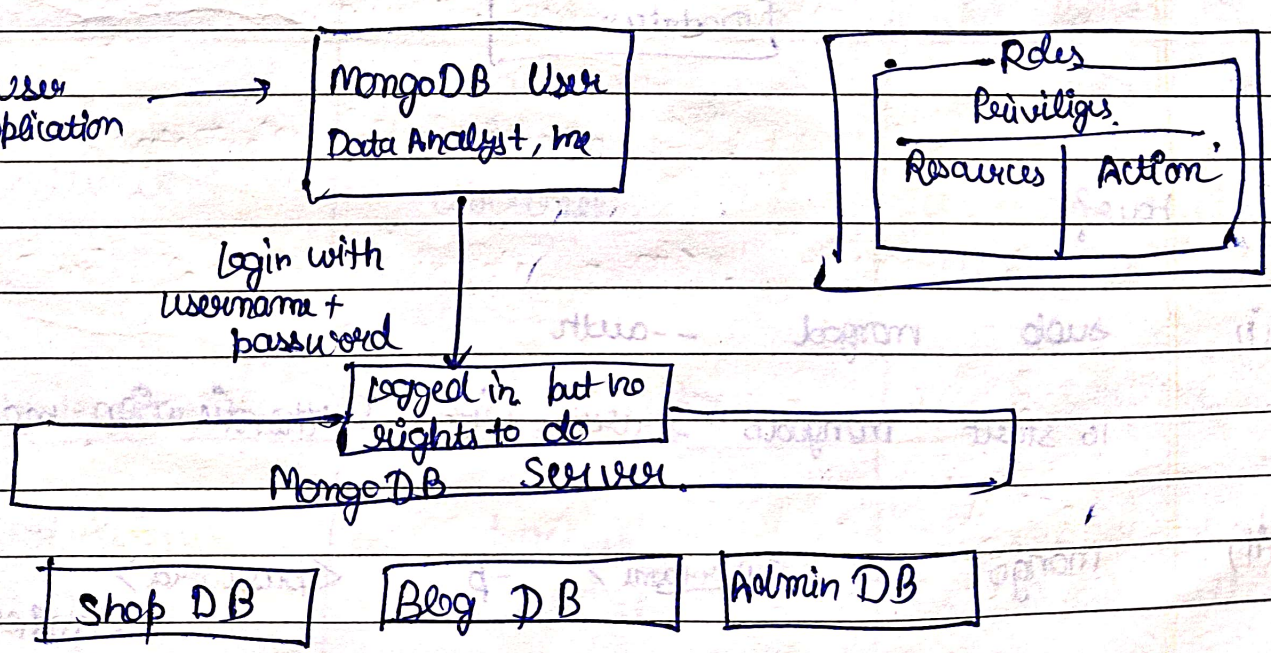
↓ Identifies valid users of the DB.

ex- You are employed & therefore may enter office

↓ Identifies what these users may do in the DB.

ex- You are employed as accountant and can do offi orders not software dev.

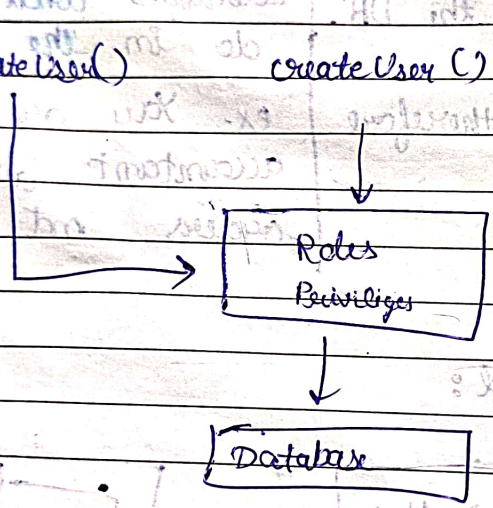
→ Role Based Access Control:



Why roles?

→ Different types of DB users

Administrator	Developer/your app	Data Analyst
Manage database config., create users	Needs to be able to do CRUD	Need to fetch data only
No need to be able to insert/fetch data	No managing users	No managing users, No CRUD



How?

(i) sudo mongod --auth

To start mongoDB server in authentication mode

(ii) mongo -u <username> -p <password> --authenticationDatabase admin

Before that, we need to have one user

Well, mongoDB has a local host connection which allows to add one user and this user is then allowed to create more users.

For this,

→ $ use admin

$ db.createUser({ user: "Max", pwd: "SK", roles: ["userAdminAnyDatabase"] });

$ db.auth ("Max", 'SK')

$ show dbs] - Now possible.

⇒ Built-in roles:

Database User	Database Admin	All Database Rles
read	dbAdmin	readAnyDatabase
readWrite	userAdmin	readWriteAnyDatabase
	dbOwner	userAdminAnyDatabase
		dbAdminAnyDatabase

ClusterAdmin	Backup/Restore	SuperUser
clusterManager	backup	dbOwner (admin)
hostManager	restore	userAdmin (admin)
clusterMonitor		root
clusterAdmin		userAdminAnyDatabase

ex.

for a shop database

```
$ use shop
$ db.createUser ({ user: "", pwd: "",
                   roles: ["readWrite"] })
```

method 1

```
$ use admin
$ db.logout()
$ use shop
$ db.auth ("", "")
$ db.products.find().pretty()
```

or

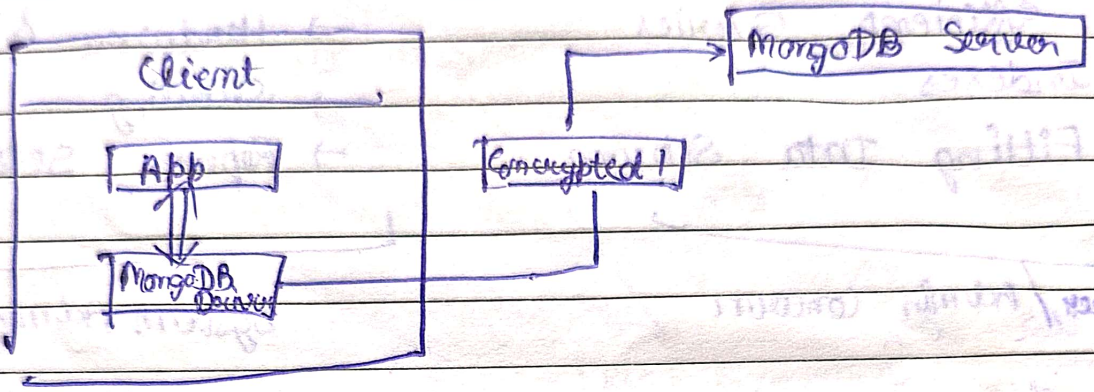
method 2

```
$ mongo -u ____ -p ____ --authenticationDatabase shop
$ use shop
$ db.products.find()
```

Update User

```
$ login as admin on admin DB
$ use shop (b/c user is in shop)
$ db.updateUser("vasu", { roles: [ "readWrite",
                                   (for current DB)
                                   { role: "readWrite", db: "test" } ] })
```

II. Transport x Encryption:



Performance Tolerance & Deployment

What influences Performance?

- → Efficient Queries
- → Indexes
- → Fitting Data Schema

Developer / Admin Concern

- → Hardware & Network
- → Sharding
- → Replica Sets

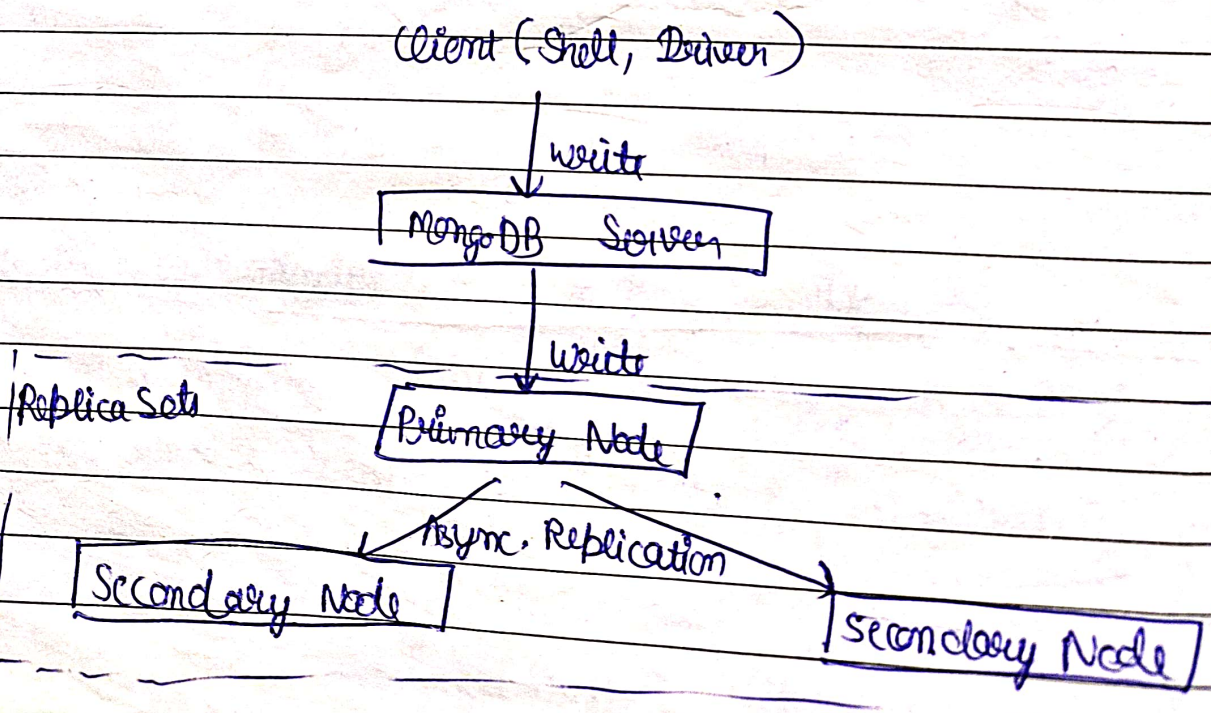
System Admin Concern

Capped collections

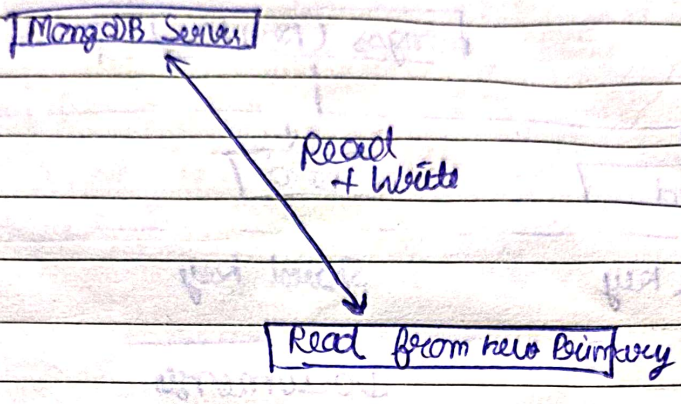
db.createCollection("<collection name>", {capped: true, size: 10000, max: 3})

- → Always sorted in these collections by id.
- → Remove first collection on size limit hit.

⇒ Replica Sets:



Why? Replica Sets Read.



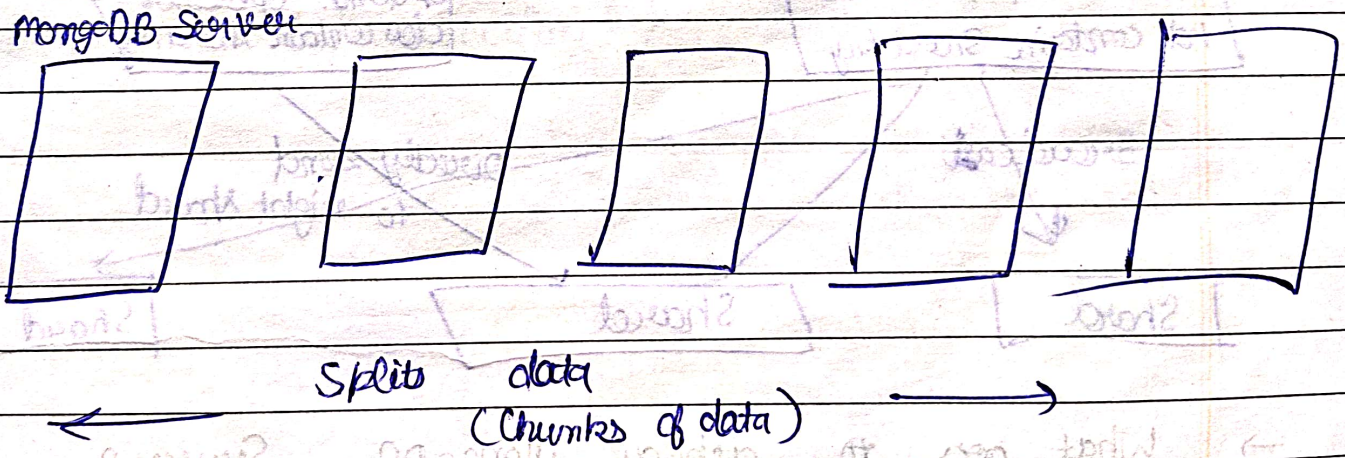
→ Backup / Fault Tolerancy

→ Improve Read Performance.

↓

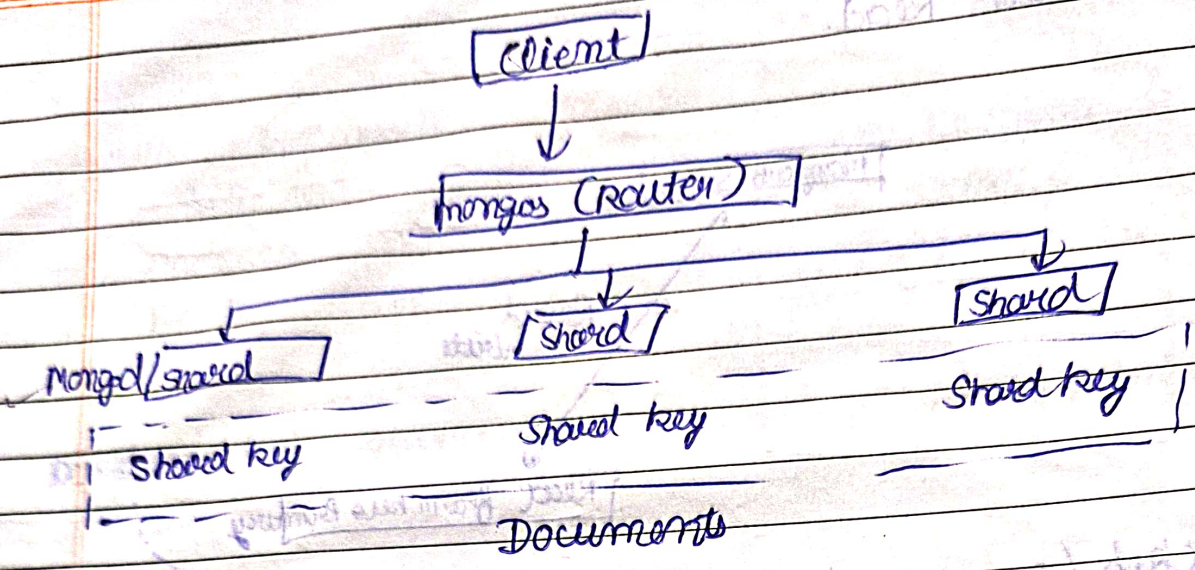
Read requests can go to all secondary nodes

⇒ Sharding (Horizontal Scaling)

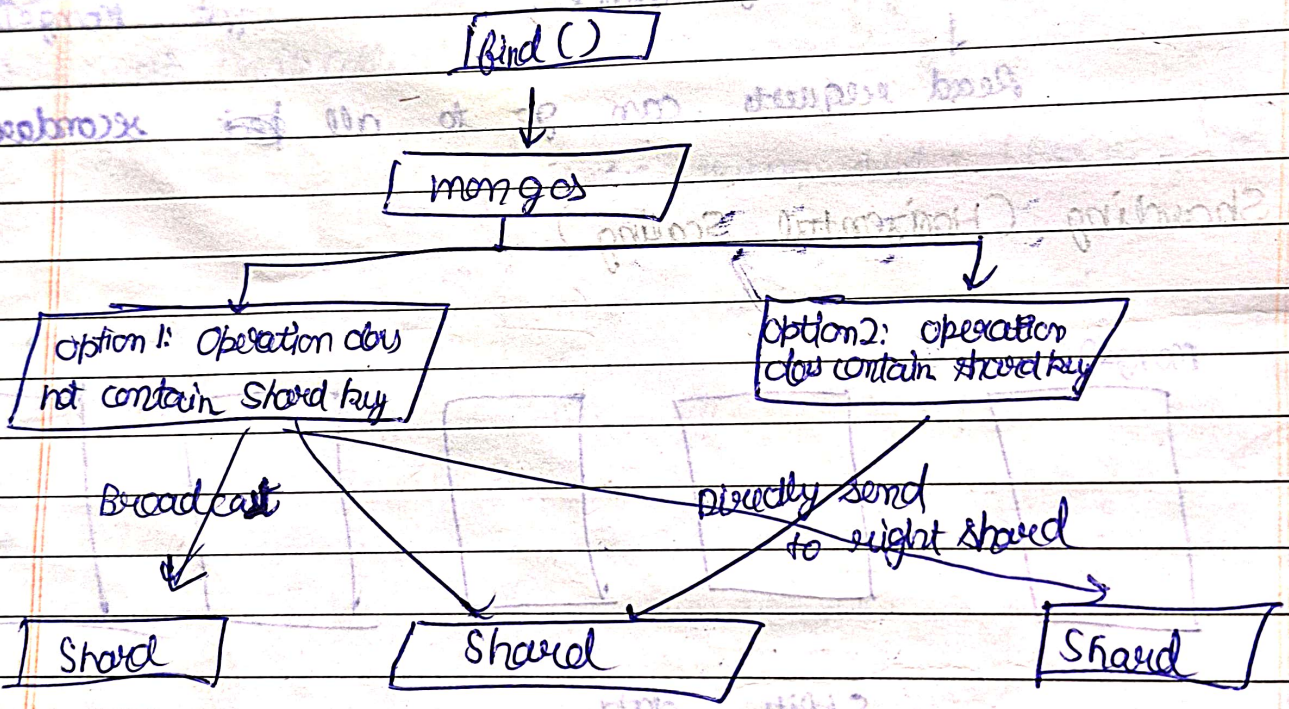


→ Data is distributed (not replicated) across shards

→ Queries are run across all shards



⇒ Queries & Sharding



⇒ What goes to deploy MongoDB Server?

→ Manage Shards → Secure User/Auth Setup → Protect Web Server/Network

→ Manage Replica Sets → Encryption (At Transport / at rest) → Regular Backups → Update Software

⇒ Transactions: (Uses sessions).

```
const session = db.getMongo().startSession();

session.startTransaction();

const V1 = session.getDatabase("db_name").col_name

const V2 = session.getDatabase("db_name").col_name
```

After this

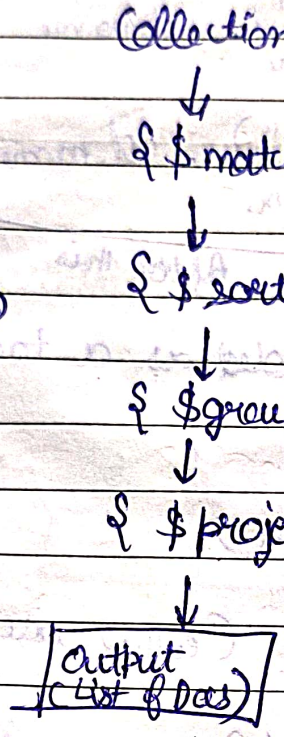
V1.deleteOne()
V2.deleteMany()
] Added as a todo for MongoDB

session.commitTransaction()

↑ changes done (either succeed or rollback)

session.abortTransaction() → cancels it!

⇒ AGGREGATION FRAMEWORK: (Alternative to find) Retriving Data efficiently & in a structured way



⇒ Every stages recieves the output of previous stage.

db.<coll.name>.aggregate([ ])

↑ Takes an array of operations

```
.aggregate ([
{ $match: { gender: "female" } },
{ $group: { _id: { state: "$location.state" },
totalPersons: { $sum: 1 } } },
])
```

Notes:

"$field" → refers to value.
"field" → refers to field.

```
{ $sort: { totalPersons: -1 } },

{ $project: { _id: 0, gender: 1,
    fullName: { $concat: [ "$name.first",
                           " World ",
                           "$name.last" ]
    }
```

$concat can recieve more complex statements

ex -

```
[ { $toUpper: "$name.first" } ]
```

length

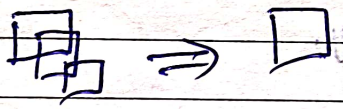
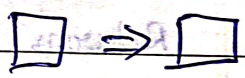
```
[ { $toUpper: { $substrCP: [ "$name.first", 0, 1 ]
  } },
  { $substrCP: [ "$name.first", 1,
    { $subtract:
      [ { $strLenCP:
          "$name.first"
        }, 1 ]
    }
  ]
}
```

ex- Project for a geoJSON object.

```
{ $ project: {
    _id: 0,
    name: 1,        (For next project)
                    (remember every pipe recieves op of previous stage
    email: 1,
    location: {
      type: "Point",
      coordinates: [
        "$ location. coordinates. longitude",
        "$ location. coordinates. latitude"
      ]
    }
  }
}
{ $ convert: { input: "$ location. coordinates. longitude",
               to: "double",
               onError: 0.0,
               onNull: 0.0
             }
}
```

ex- Project for a DOB

```
{ $project: {
    _id: 0,
    birthdate: {
      $convert: {
        input: "$dob.date",
        to: "date",
        onError: null
      }
    }
  }
}
```

$ group.	$ project
n: 1	1 : 1
	
sum, cnt, Average	Include / exclude fields, transform fields.

⇒ Array operators in Aggregation module:

I. ⇒ Push array hobbies into all Hobbies grouped by age.

```
db.<coll-name>.aggregate([
  {$group: { _id: { age: "$age" },
             allHobbies: { $push: "$hobbies" } }
  }
])
```

II. $unwind

Pass an array to $ unwind and flattens the array into multiple docs.

So, group hobbies by age is like.

db.<collname>.aggregate([

{$unwind: "$hobbies"},

{$group: {_id: {age: "$age"},

allHobbies: {$push: "$hobbies"}

}

}

])

We can use

This is now a single doc.

$addToSet to avoid duplicate values

III. $slice

Returns slice of an array.

e.g.

{$project: {_id: 0, examScore: {$slice:

["$examScores",]}

Any array

length to get

if it's (-ve) it takes from last

[array, start, length]

IV. $ size

Returns size of an array.

V. $ filter

filter out options from an array

eg.

```
{ $ project : { examScores : { $ filter : { input : "$examScores",
                                            as : "sc"
                                            cond : { $gt : [ "$$sc.score", 60 ] }
                             }
              }
}
```

(temp. name; syntax for temp. name)

Q. Find highest exam score of each person in array.

array look like this

```
[ {
    "_id" :
    "examScores" : [ {
                       "difficulty" : 3,
                       "score" : 75
                     },
                     { "difficulty" : 8,
                       "score" : 40
                     }
    ],
    another
```

⇒

```
[ {
    "_id" :
    "examScores" : 75
  },
  {
    "_id" :
    "examScores" :
  }
]
```

```
db.<coll>.aggregate [
  { $unwind : "$examScores" },
  { $project: { _id: 1, name: 1, score: "$examScores.score" } },
  { $sort: { score: -1 } },
  { $group: { _id: "$_id", name: { $first: "$name" },
      maxScore: { $max: "$score" }
    }
  }
])
```

* Some additional stages:

(i) Bucket
(ii) Bucket Auto.

```
db.<coll>.aggregate [
  { $bucketAuto: {
      groupBy: '$dob.age',
      buckets: 5
      output: {
        numPersons: { $sum: 1 },
        averageAge: { $avg: "$dob.age" }
      }
    }
  }
])
```

#Clips in visualisation of data.

(iii) limit stage :

{$ limit : 10 }

↑
get only first 10.

(iv) skip stage

{$ skip : 10 }

↑
skip first 10. (Depends on order)

(v) out stage

Takes out result and write into the new collection
already existing/new

{ $ out : "transformedData" }

Working with Numeric Data

⇒ Mongo shell is written in Javascript.

So, Int32 is stored as float64

If working from NodeJS driver same thing happens.

⇒ But, from python Driver, it differentiates Int32 and float64 because python differentiate them.

So, values written in MongoDB depends on client we are using.

Numbers Types in MongoDB

Integers (Int32)	Longs (Int64)	Doubles (64 bit)	High precision Doubles (128 bit)
→ only full nos,	only whole nos	Num. with Decimal	Num. with Decimal
→ -2^{31} to $2^{31}-1$	-2^{63} to $2^{63}-1$	Decimal values are approximated	Decimal values are stored with high precision (34 decimal digits)

⇒ Int32

db.persons.insertOne({ age: NumberInt(219.45) })

or "219.458" (Pass string then convert)

Stored as 219

greater $(2^{32}-1)$ ⇒ overflow

⇒ Int64

({ age: NumberLong(" ") })

Pass string

- During maths, be sure of typecasting.
- All Numbers are Float64 by default.

⇒ High precision Doubles

({ a : NumberDecimal ("0.345") })

From Shell to Driver

Shell

- → Configure DB
- → Create Collections
- → Create Indexes

Driver

- → CRUD operations
- → Aggregation pipeline

Connecting Driver of NodeJS

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect(URL, (err, client) => {
    if (err) {
        return console.log(err);
    }
    console.log('Connected to Server');
    const db = client.db('TodoAPP');

    client.close();
});
```

ObjectId ( )

Time Stamp + MachineId + Random Value. = _id

So _id.getTimestamp() → returns time.

db.collection('<coll-name>')
- findOne()
- find({})
- insertOne
- insertMany
- updateOne
- updateMany
- deleteOne
- deleteMany

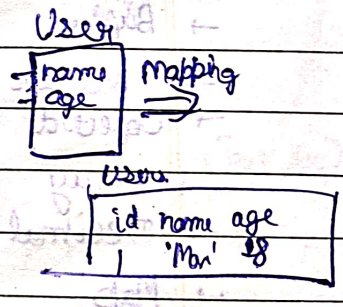
This gives a cursor

so
- .toArray()
- .forEach()

- findOneAndUpdate({}, {$set: {}}, options, ()=>{})
- findOneAndDelete({}, ()=>{}).

returns doc as well.

Drivers to ORM/ODM



Object-Relational Mapper → For relational DB
object-Document Mapper → For NoSQL DB.

→ Mongoose is an ODM that provides a straightforward & schema-based solution to model your application data on top of MongoDB's native Driver.

→ Built in type casting, validation, query building, hooks & more.

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var blogSchema = new Schema({
```

```
title : {
    type: String,
    required: true
},
age : Number
DOB : {
    type: Date,
    default: Date.now
}
};
module.exports = mongoose.model('Product', productSchema);
```

↓ gets converted to lower case & added plural form.

Schema Types

- String
- Number
- Date
- Buffer
- Boolean
- ObjectId
- Array
- Decimal128
- Map

Schema options.

- required
- default
- alias
- lowercase
- uppercase
- trim

Validators

- required
- minlength
- maxlength
- min
- max

For Date & Number

Note: Mongoose do type casting, it may be a trouble

⇒ Instance Methods:

```
BlogSchema.methods.<new_method_name> = function (cb) {

};
```

or

```
BlogSchema.method('new_method_name', function(){
```

⇒ Static Methods:
similar Way declaration.

⇒ CRUD

I. Create:

```
var Tank = compiled model
var document = new Tank (document);
document.save ((err) => {
});
```

or

```
Tank.create (doc, (err, doc) => {
});
```

or

```
Tank.insertMany ([ ], (err) => {
});
```

II. Read / Find

```
Tank.find ( ).where ('field').gt(5).exec((err,doc) => {
});
```

find By Id
find One

```
Tank.find ({ }, 'name age.', (err, docs) => {
});
```

(conditions, fields)

III. Update (runValidators : true).

```
Task.updateOne ({ }, { }, (err, res) => {

});
```

IV. Delete

```
Task.deleteOne ({ }, (err) => {

});
```

FRONTEND STORAGES

I. Cookie:

It is a small piece of data that a server sen to user's web browser.
The browser may store it & send it back with next request to same server.

So, we can say it remembers stateful information the stateless HTTP protocol.

They are used for:

→ Session Management
(See NodeJS session cookies)
→ Personalization
→ Tracking.

General Syntax
Set-Cookie : <cookie-name> = <cookie-value>

II. IndexedDB:

It is a low-level API for client-side storage of significant amounts of structured data, including file/ blobs.
It uses indexes to enable high-performance searches of data.

Features:

→ It is a transactional database system,
→ It has javascript-based object-oriented database.
→ Stores : JS objects, files, blobs etc.
→ Operations performed using IndexedDB are done async so as not to block apps.
→ Some storage limits depends on browser.

III. Web Storage API:

It provides mechanisms by which browsers can store key/value pairs, in a much more intuitive way.

There are two mechanisms within web Storage:

(a) session storage:

Maintains a separate storage area for each given origin that's available for the page open. (reloads, restores)

(b) local Storage:

Same thing, but persists even when browser is closed & reopened.

e.g.

localStorage.getItem(" ")

" .setItem(" "," ")

" .removeItem(" ")

" .key(n)

" .clear()

Note: window.n = x

IV. Cache API:

It provides a storage mechanism for Request/Response object pairs that are cached.

They are part of ServiceWorker life cycle. but it is exposed to windowed scopes as well

⇒ Service Worker (Optional)

A JS script that gets registered with the browser & stays registered even when offline.

Use Cases:

→ Caching assets & API calls
→ Push Notifications
→ Background data sync (x)
→ Used in progressive web apps

Security

The act/ practice of protecting websites from unauthorized access, use, modification, destruction, or disruption.

(😐 bakuuss)

I. XSS: (Cross side scripting)

Code injection attack that allows an attacker to execute malacious JS in another user's browser.

It can't be done directly. Instead, attacker exploits a vulnerability in website that the victim visits, which in turn delievers malacious JS to victim.

Q. How malacious JS is injected?

website (server) with comments

Attacker

<script> getcookies() </script> & send to my server.

User

Hey

Victim

I am innocent.

Result → Cookies, keylogging, phishing

⇒ Types of XSS:

i) Persistent
ii) Reflected
iii) Dom-based ? Invisible to server.

How to prevent it?

i) Encoding → user input TO data only not a code.
ii) Validation → filter user input

⇒ Star of Security:

I. Injections:

Code inside other code

e.g.1. SQL injection

' or 1=1 -- → Read all data
(comment)

'; DROP TABLE users; -- → drop table

e.g.2 In form fields

form → `<img src="/" onerror="alert('boom');" />`

Submit

if done via p.innerHTML

Better

var textNode = document.createTextNode(input);
p.appendChild(textNode);

How to fix them?

1. Sanitize input
2. Parametrize Queries
3. Knex.js or other ORMS
4. check express-validator.

II. 3rd party libraries:

Use,

nsp check (npm packag) # audit package.json

Snyk test # audit node_modules directory.

III. Logging:

winston or morgan

IV. XSS & CSRF:

Already discussed

Cross-site Request Forgery.

[redacted]

Attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.

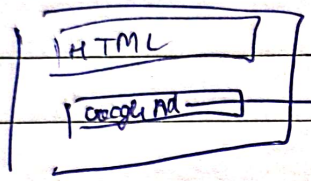
Eg. <img src = " https://www.example.com/index.php?action=delete&id=123" >

V. CORS:

It is a mechanism which aims to allow requests made on behalf of you & at the same time block some requests made by rogue JS & is triggered whenever a HTTP request is made to:

- a different domain
- a different sub-domain
- a different port
- a different protocol (HTTP, HTTPS)

e.g.



→ POST request to mybank.com

Headers:

→ Access-Control-allow-origin

→ " " " Credentials
Headers
Methods.

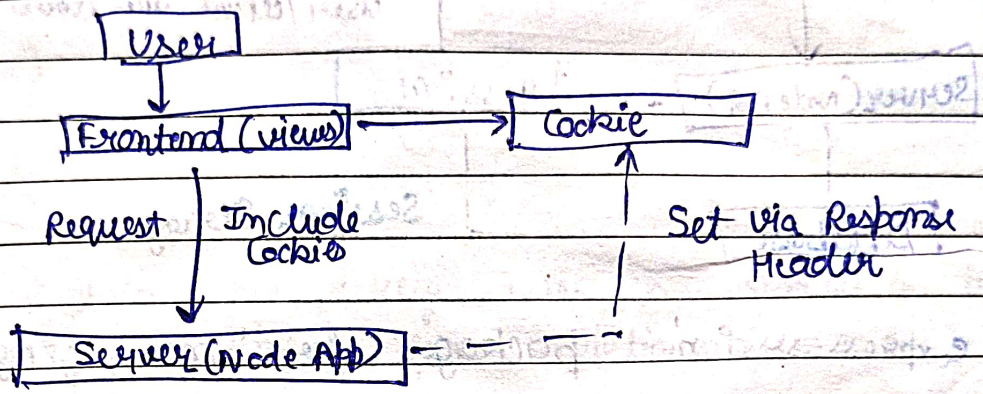
} some headers to do all this BS

⇒ Storing Information in Browser :

i) Cookies

ii) Session.

Cookies in NodeJS : (Stored on client side)



Q. Why cookies?

If we used some global variable to store some info it will be shared across every user for every request.

So, we use cookies.

res.setHeader('Set-Cookie', 'loggedIn=true')

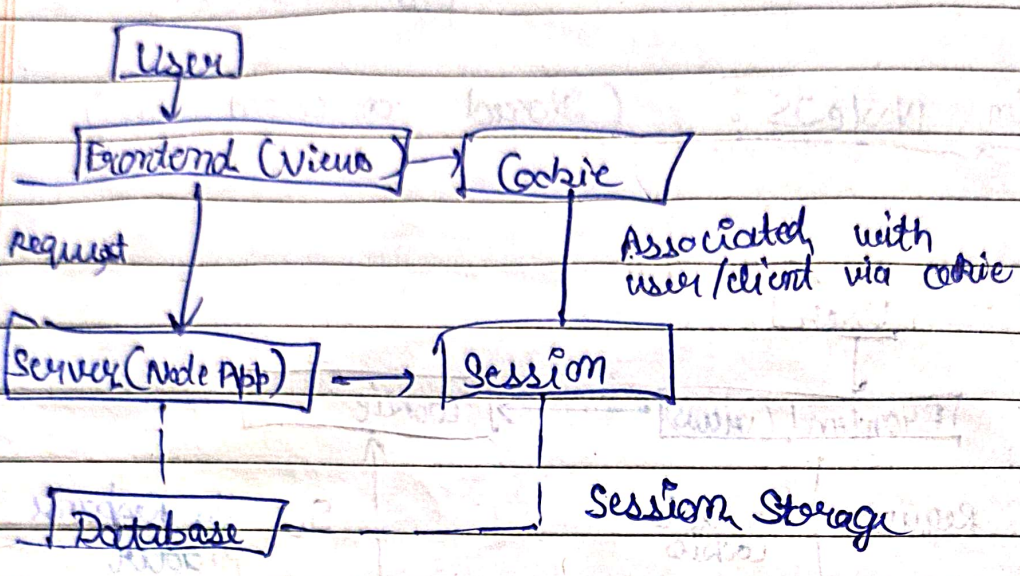
☆ Here, a cookie can be manipulated by client by setting to true/false.

Some options to Cookie

→ Secure
→ HttpOnly
→ Max-Age
→ Expires In

} Look documentation.

Session in NodeJS (stored on server side)



'express-session' package for managing session.

↳ options

i) cookie (only session ID is stored in cookie)
session data is stored on server side.

options: cookie, domain
expires,
maxAge
httpOnly
sameSite
secure
path

ii) resave : false

(if set to true) forces the session to be be saved back to session store even if session was never modified during the req

iii) saveUninitialized : false

iv) secret
used for signing the hash

⇒ app.use (
session ({
secret: " ",
resave: false,
saveUninit —
Cookie : {

}
})
);

Note: session data for every user is stored in memory which will be horrible for millions of request.

So, we use session stores

e.g.

connect-mongodb-session , connect-mongo, connect-redis

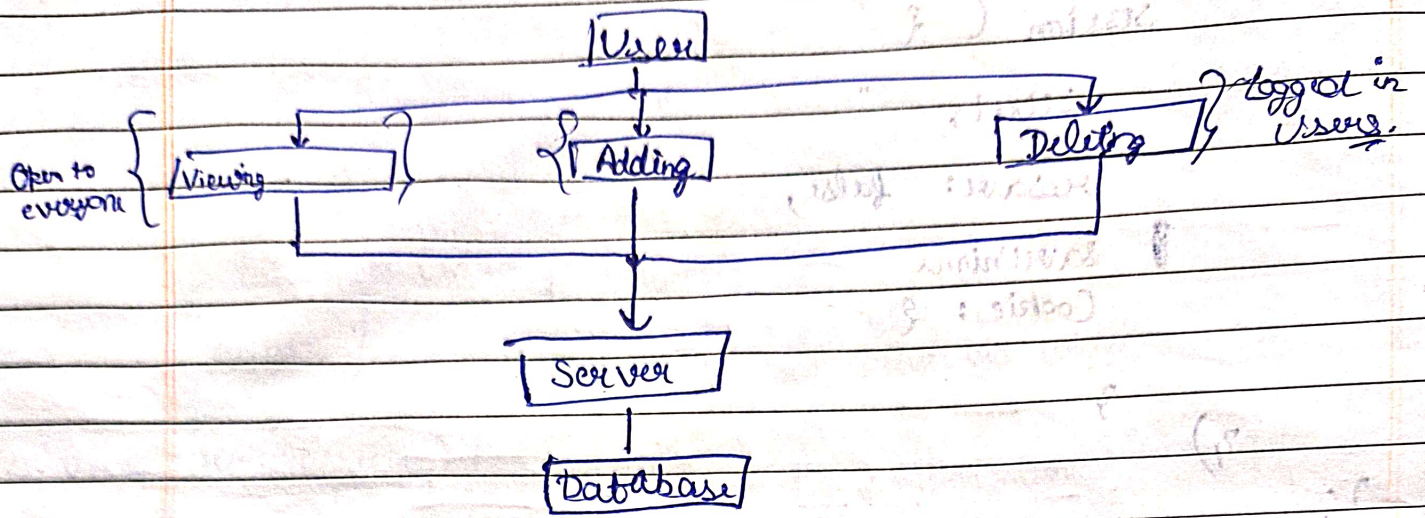
so,

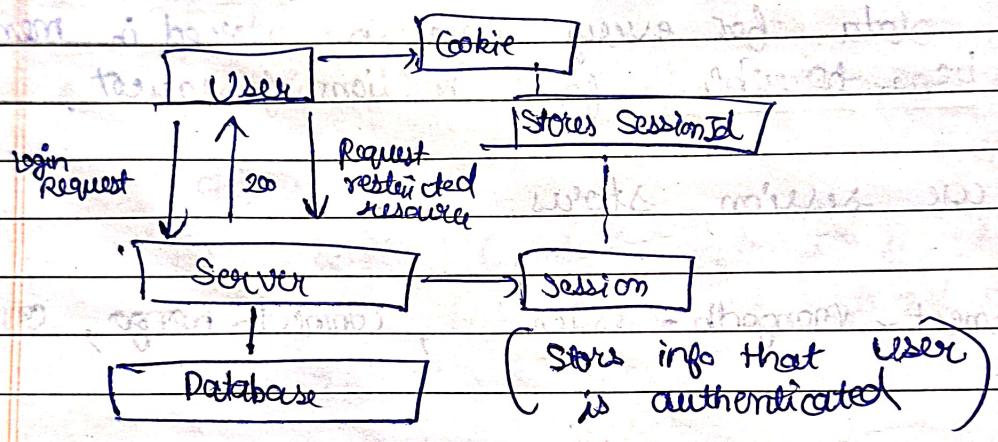
var MongoDBStore = require ('connect-mongodb-session')(session);

var store = new MongoDBStore ({
uri : DB_URL,
collection : <coll-name>
});

and add store option to app.use session.

User Authentication

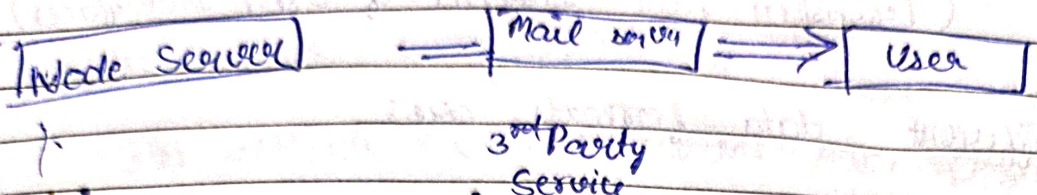




⇒ Storing password in the database :

```
bcrypt.genSalt(12, (err, salt) => {
    bcrypt.hash(password, salt, (err, hash) => {
        console.log(hash);
    });
});

bcrypt.compare(req.body.password, databasePassword, (err, result) => {
    if (err) return
    if (result === true) return true
});
```

⇒ Sending Mails:



mail server isn't an easy task.

⇒ nodemailer package is package handling mail servers Need a 3rd party service also.

⇒ Resetting User Password

Generate Random Token → send to email → when user request → Check for valid token → allow → Reset & saveHashed Password

(save it in user field find by email
Add expiry.

⇒ Validating x User-Input:

May use mongoose, but use server-side validation it's fast.

Check for 'express-validator'.

⇒ Error x Handling:

try-catch → Sync then, catch → Async

REST API:

Representational State Transfer.

(Transfer Data Instead of User Interfaces)

The different data formats are:

HTML	Plain Text	XML	JSON
Node.js	Node.js	<name> Node.js </name>	{ "title": "Node.js" }
Data + Structure	Data	Data	Data
Contains UI	No UI	No UI	No UI
Unnecessarily difficult to parse if you just need the data	Unnecessarily diff. to parse, no clear data structure	Machine-readable but relatively verbose, XML parser needed	Machine-readable and concise. Can easily be converted to JS

Rest Principles: (Useless Though)

→ Uniform Interface: Clearly defined API endpoints with clear req. & res. data structure

→ Stateless Interactions: Every req is handled sep.

→ Cacheable: Servers may set caching headers to allow the client to cache responses.

→ Client-Server: Server & client are seperated, client is not concerned with persistent data storage.

→ Layered System: Server may forward requests to other APIs.

→ Code on Demand: Executable code may be transferred from server to client.

⇒ Fixing CORS errors:

```
app.use ( (req, res, next) => {
    res.setHeader ('Access-Control-Allow-Origin', 'codepen.io');
    " " " ('" " " -methods', 'GET, POST, --');
    " " " - Headers',
        'Content-Type, Authorization');
});
app.use (cors());
```

Wild card (*)

Again wild card.

Also, we can use cors package.

⇒ Authentication in REST APIs:

Rule 1: Store Passwords in Hashed form:

Use bcryptjs. (Pure JS implementation of bcrypt C++ library)

⇒ For sync (Not recommended)

Hash password:

```
var bcrypt = require ('bcryptjs');
var salt = bcrypt.genSaltSync (10);   // or use a number
var hash = bcrypt.hashSync ("<pass>", salt);
```

Check password:

```
bcrypt.compareSync ("<pass>", hash);
```

Returns true or false.

⇒ For Async

```
bcrypt.genSalt(10, (err, salt) => {
    bcrypt.hash(Password, salt, (err, hash) => {

    })
})
```

Auto-gen a Salt & Hash

```
bcrypt.hash(Password, salt-number, (err, hash) => {
});
```

Rule 2: Generate JWT Tokens:

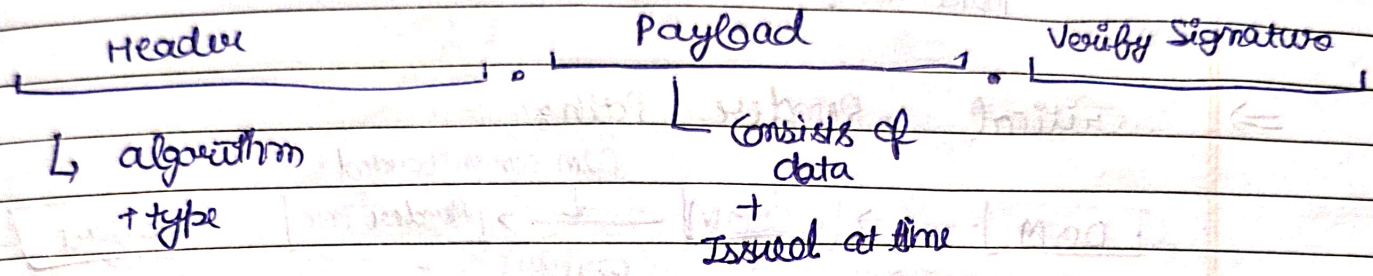
Rough Implem.

```
var data = {
    id: 4
}
var token = {
    data,
    hash: SHA256(JSON.stringify(data) + 'somesecret')
        .toString()
}

var resultHash = SHA256(JSON.stringify(token.data) +
    'somesecret').toString()

if resultHash === token.hash
```

jsonwebtoken is the npm package.

I. ~~jwt~~ sign (object-data , secret) → returns a token.

This token is a combination of:



II. jwt.verify (token , secret);

↓

returns data if succeeded

else returns error.

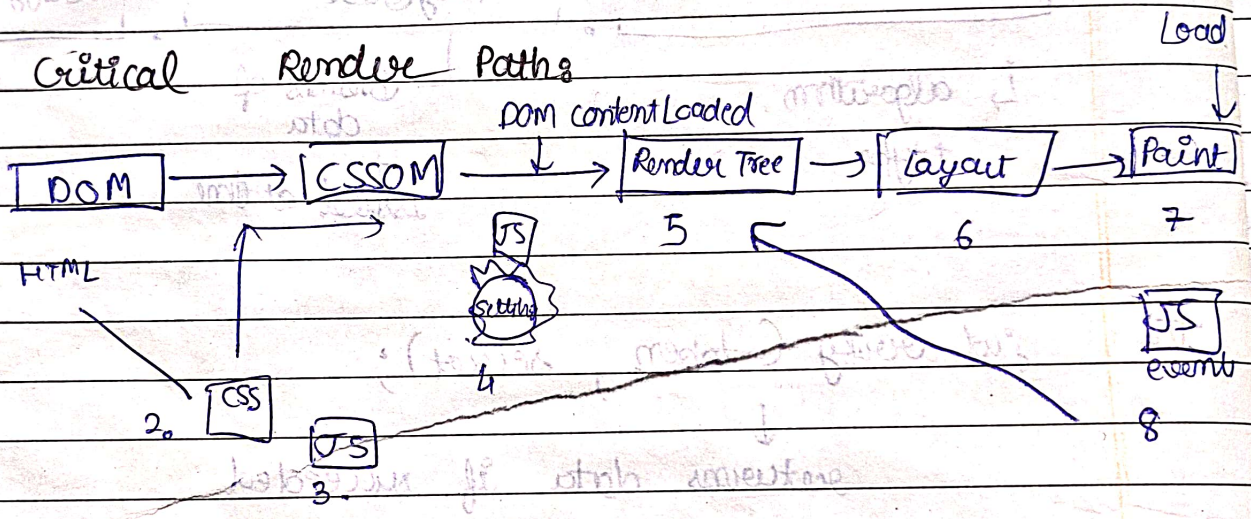
Performance

1. Network

Images
PNGs/JPGs → tinyPNG.com
JPGs → jpeg-optimizer.com
Remove Meta Data

HTML/CSS/JS
minify them.
Make them one.

⇒ Critical Render Paths



<Script>

For 3rd party scripts such as Iframes, Sharing Button (FB, twitter)

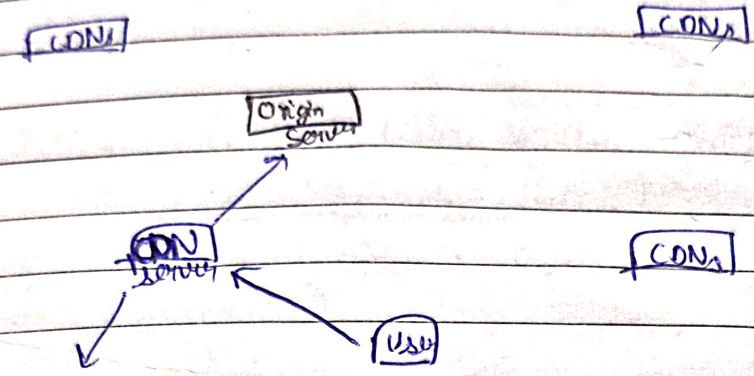
<Script async> Run with another thread

<Script defer> Runs after script downloaded in background

☆ Find Page speed at "page speed insights".
&
"webpage test Test"

2. Backend

⇒ CDNs: (Content Delievery Networks)



It do caching
→ Improves Page load
→ Higher Traffic load is handled
→ Blocks Bots/Spammers
→ Can prevent from DDOS.

⇒ GZIP: (google created Brotli) 20% faster than gzip.

In nodeJS, it is done by compression().

⇒ DB Scaling:

1.	Identify Inefficient Queries	Indexes
2.	Increase Memory	Hardware
3.	Vertical Scaling (Redis, Memcached)	Allows app-level caching
4.	Sharding	
5.	More Databases	Horizontal Scaling.
6.	Database Type	(It is bit more specific on application)

MongoDB, Key-Value (Redis), SQL, Graph Based

⇒ Load Balancing: (leave it on Nginx)