NAMASTE NODE.JS SEASON 2

Hand Written Notes

-By Shanmuga Priya

www.linkedin.com/in/shanmuga-priya-e-tech2

Season 2

Episode 1

Microservices vs Monolith - How to Build a project.

Software Development Life Cycle

→ In Industry, waterfall model approach is used.

1) what is waterfall model?

→ It is a methodology that is used in software & Product develop

→ It is a linear, sequential approach where each phase of the Project is completed before the next phase begins.

eg:

Requirements
↳ Design
↳ Development
↳ Testing
↳ Deployment
↳ Maintainance.

2) what are the things included in requirements?

→ what the project is about
→ what are the features we are going to build
→ How we are going to build
→ what are the different scenarios
→ who will be the audience
→ what will be the tech stack & so on.

→ Project Manager collects all these requirements

→ Product Manager + Designer → Mock Model.

2nd phase : Design Phase

→ Senior Engineer + Tech Lead defines the Design/architecture of the project (whether it is Monolith (or) Microservices)

→ Deciding the tech stack, how will they communicate

→ High Level Design (HLD) & Low level Design (LLD) are build in this phase.

3rd phase : Development

→ Developers are involved in developing this design & writing unit test cases

4th phase : Testing

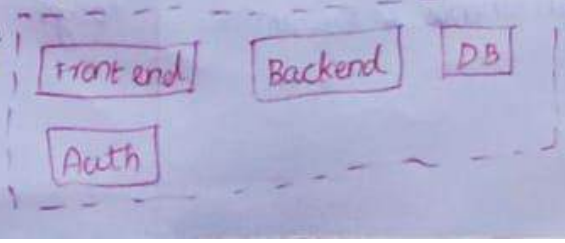
→ Testers test the project developed by the developers

5th phase : Deployment :

→ Devops manages the server & developers deploy the project after successfull testing phase.

→ It depends on company. if it is startup most of the works are done by developers only.

3) what is the difference between Monolithic & Microservices

Monolith	Microservices
→ The entire app is build as a single project. It has backend, Frontend, DB connection, Authentication, Emails, analytics, notification.	→ It consists of multiple small services, which are connected together where each services have their own responsibility such as one microservice for frontend, One for backend & so on.
eg: 	eg: 

Parameters	Monolith	Microservices
Development Speed	Slow	fast
code Repo	Single big repo	Multiple code repo. One repo for each Services
Scalability	As project grows. Scalability becomes tough	Each microservices are scaled independently
Deployment	Single deployment but whenever there is a change in code even its a single line we have to deploy the entire file once again	seperate deployment for each services.
Tech stack	Should follow the same tech stack throughout the entire project	Each services can have their own tech stack. Suppose one service can built using React, Other can use Angular.
Infra cost	low	high
Complexity	tough	easy when project is larger.
Fault Isolation	Even a single bug breaks the entire app	it affects only that particular service.
Testing	Easier testing	Testing is done independently for each services.
ownership	handled by single team	Each services are handled by diff team
Maintanence & Rewamps	tough	easy

Episode 2

Features, HLD, LLD & Planning

→ Planning is the most important & crucial part in project development.

→ Proper planning avoids refactoring the code.

→ 1st step in planning is gathering the requirements of the project

→ 2nd step is Planning the features of our app

→ 3rd step is Designing our app which include High Level Design (HLD), Low Level Design (LLD).

High Level Design:

→ HLD is designed by the senior engineer & Tech Leads.

→ They decide the project architecture like

*) what are the microservices the project should have?

*) what tech stack each microservices should use?

*) How these microservices communicate with each other?

*) what are the security practices should be taken?

*) optimization techniques & so on.

Low Level Design:

→ LLD is Designed by developers, SDE1, SDE2.

→ The decide on various factors like

*) what database should be used?

*) what are all the collections it should have?

*) How the document structure should be in each collection?

*) How the Schema & model should look like?

*) what are all the API it should have?

1) what is REST API?

→ A REST API (Representational State Transfer API) is a type of API that uses REST Architecture to connect applications.

→ It is nothing but a set of rules that allow programs to communicate with each other using HTTP requests.

→ REST API's are commonly used in client-server architecture and microservices, web API's.

2) what are the characteristics of REST API?

→ It should be lightweight, highly scalable, flexible, stateless, Uniform Interface.

Stateless:

→ Each request from client to server must contain all the information needed to process the request.

Uniform Interface:

→ REST API's uses standard HTTP methods

GET - to get data from the server

POST - to add a new data on the server

PUT - update the entire doc with the new one

PATCH - update only the part of the document

DELETE - to delete the document from a collection

Episode - 3

Creating our Express Server

1) What is express?

→ Express is a web application framework build on top of Node.js

→ It provide features for creating web server (or) API, Routing, Middleware & so on.

→ It is ideal for creating REST API's allowing developers to define routes that correspond to different HTTP ~~requests~~ methods.

Creating our Express server:

Step 1: Initialize the Project

```
npm init
```

→ It creates a "package.json" file

Step 2: Creating a src folder & new file in it

Step 3: Installing Express js

```
npm i express
```

→ It creates a "node_modules" folder & "package-lock.json" file.

Step 4: Creating Server

```
const express = require("express") → Importing express

const app = express() → new instance of express is created

// Defining routes
app.get('/test', (req, res) => {
    res.send("Hello from the server");
})
```

→ request handler function

Sending response back to client

```
// Creating/starting Server
app.listen(portNumber, () => console.log("started server"))
```

→ app.listen() starts the server at the specified port number and listens for incoming requests.

2) what is Node_modules folder?

→ This folder contains all the dependency that project requires along with the package we install and the nested dependencies.

3) what is Package.json & package-lock.json ?

→ Package.json :

→ this file list the direct dependencies (packages) that our Project need.

→ It include Package names & version ranges & it automatically update its version based on '^' (carat) (or) '~' (tilde) symbol before it.

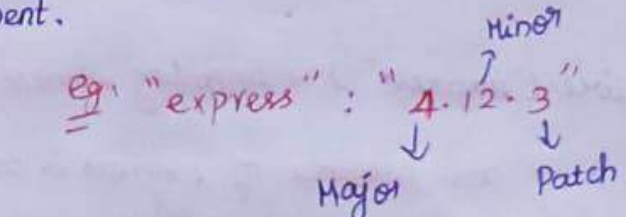
*) Carat - updates minor & patch version

*) tilde - updates only patch version.

*) if no '^' (or) '~' - exact installed version no update happens.

→ Package-lock.json:

→ this file locks the exact versions of the dependencies and their dependencies, ensuring consistent installation across different environment.



→ the version usually starts at version 1.0.0

4) what is Nodemon?

→ Nodemon is the utility tool for Node.js it automatically restart the server whenever there is a change.

npm i -g nodemon → global installation

→ without Nodemon, whenever we make change we need to stop the server by Ctrl+C & restart it with node app.js → entry point file.

→ with Nodemon, we can add this to package.json scripts for quick access.

```
{ scripts: {
    "start": "node app.js",
    "dev": "nodemon app.js"
  }
}
```

→ Start the app with npm run dev & it will restart the server every time whenever the file is saved.

Episode - 1

Routing and Request Handlers

1) will the order of middleware functions matters?

→ Yes, in express middlewares are processed in the order they are defined.

eg: → If you define a middleware with a path "/" it matches all the routes which starts with match & return the same response for all the routes even you hit the another path "/hello".

Soln:

→ To solve this place the generic "/path" at the end so that each path matches with the correct response.

> Always define specific routes before generic routes.
> order of the routes matters.

2) what is Postman?

→ It is an API platform for building, using & testing the API's.

3) what happens when you use +, *, ? in the path?

→ ? - indicates that letter is optional. we can also group things up with ()

eg: "/ab?c" → it works for both "/abc" & "/ac"
"/a(bc)?d" → it works for both "/ad" & "/abcd"

→ + - indicates that letter can be used multiple times but all other characters should be in the same order.

eg: "/ab+c" → it works for both "/abc" & "/abbbbc"

→ * - indicates that it can have anything at that position.

eg: "/ab*c" → it works for both "/abc" & "/abSHANC".

→ we can also use regex in the path.

eg:

```
app.get(/.*fly$/, (req, res) => {
    res.send("Hello")
})
```

4) How will you define a query Parameter in the route & how to access it?

→ we can add query Parameter using "?" and add more query Parameters to it using "&".

→ we can access the query parameter using req.query.

→ req.query returns an object with query parameter & its corresponding value.

eg: http://localhost:7777/user?userId=111&Password=abc

```
app.get("/user", (req, res) => {
    console.log(req.query) → { userId: "111", Password: "abc" }
    res.send("Hello")
})
```

5) How to make the Routes dynamic?

→ we can add dynamic value to the route using ":" and get the value using req.params

eg: http://localhost:7777/user/111

```
app.get("/user/:userId", (req, res) => {
    console.log(req.params) → { userId: "111" }
    res.send("Hello")
})
```

×

Episode - 5

Middlewares & Error Handlers

1) What is app.use() method?

→ app.use methods are used to add middlewares in request response cycle.

→ It accept all the HTTP methods by default. if we want any specific HTTP method for a particular use we can use methods like app.get(), app.post() instead of app.use.

→ The order of app.use() calls are matters becoz requests will go through the middleware in the order they are defined.

2) what happens when you don't send the response back inside a route handler?

→ when you don't send a response back the request will be left hanging & client making the request keep on waiting for a response from a server until the timeout occurs.

→ whenever we don't want to send any response back to the client we can call next() method to pass control to the next middleware function.

3) what happens when you call res.send()?

→ when we call res.send() method it sends response back to the client and ends the request-response cycle.

→ Not only that it also sets the necessary headers and status code automatically based on the type of data being sent.

4) what is middleware functions ?

→ function that execute for every request sent to the server.

→ They can modify the request & response objects, ends the req-res cycle (or) pass the control to the next middleware function using next() . It is used for authentication, logging, modifying req/res etc.

→ we can specify that middleware should run for all the routes (or) specific routes.

→ It consits 3 Parameters request, response & next fn.

eg: middleware for all request

```
app.use ((req, res, next) =>{
        next()
})
```

Middleware for specific routes

```
app.use ("/user", (req, res, next) =>{
        next()
})
```

5) How many middleware fn/ route handlers can one Route have?

→ Single route can have multiple middleware fn/route handlers but it should send only one response back. we can also pass them in array [ ].

→ These handlers are executed in the order they define & move to next handlers using next().

eg:

```
app.use ("/user", (req, res, next) =>{
        console.log("Handler 1")
        next()
},
(req, res, next) =>{
        console.log("Handler 2")
        res.send("Response")
})
```

6) what happens when you send a response in 1st route handler will it still execute the second handler function?

→ when we sent a res in 1st handler fn it sends a res to the client and ends the request so it will not execute the other handler fn defined below.

7) can we call next() function before (or) after sending a response?

→ Calling next() after response send:

→ we can call next() fn after the res is send back to the client to move to the next handler fn but it is unneccessary as the response is already sent we cannot modify it (or) send a new response in the 2nd handler.

→ if we try to send response in both handler function it will result in error: Cannot set headers after they are sent to the client:

→ calling next() before response send:

→ when next() is called before res.send() in 1st handler function it moves to the 2nd handler function and starts executing it once it finishes it return back & send the response back to client.

→ It is also bad practice if the 2nd handler attempts to modify the response.

Only call next() if we are not sending the response in the current handler. If you call res.send() there is no need for next()

8) what happens when you don't send response in any of the handler function just keep calling next()?

→ If we don't send a response and just keep calling next() in all the route handlers for a particular route, the request will continue and executes all the handlers associated with the route. However, if no response is ever sent it will result in a error 404 Not-found by express.

9) what is HTTP response status code & how to set it manually?

→HTTP response status code indicate whether a specific HTTP request has been successfully completed (or) not.

→ Commonly used Status Codes

200 → Success
201 → Created
400 → Bad request
401 → Unauthorized
403 → forbidden
404 → Not found
500 → Internal Server Error.

→ By default status code is 200.

→ we can set the status code manually in the response using status() function.

eg: res.status(401).send("Page Not Found").

10) what is the difference between app.use() & app.all()?

→ Both of these method accept all kind of HTTP Methods but the key difference is app.use() can matches all the routes if no path

is specified (or) to a specific route

→ app.all() always tied to a specific route.

11) How to handle error in node js?

There are 2 ways of handling errors

*) using try/catch block.

*) error handling middleware

error handling middleware:

→ It is similar to normal middleware fn but err should be the 1st parameter.

eg:

```
app.use("/", (err, req, res, next) => {
    if (err) {
        res.status(500).send("something went wrong")
    }
    next()
})
```

→ the above code catches the error wherever the error occurs in the entire app by placing this middleware fn at the end.

Always prefer using try/catch block

Episode - 6

Database, Schema & Models - Mongoose

1) How to connect MongoDB Database to your application?

Step1: creating a "config" folder & "database.js" file.

Step 2: Installation of Mongoose

-> Mongoose is a library that helps developers work with mongoDB by providing a Schema - based way to model data.

-> In Mongoose everything is schema based. for every MongoDB collection we create seperate schema.

-> The schema defines the structure of the document along with data type for each collection.

npm i mongoose

↳ It install the mongoose & include it in our app dependency.

Step 3: Connecting to DB cluster

```
const mongoose = require("mongoose")

const connectDB = async () => {
    await mongoose.connect( your MongoDB connection String + DB name )
}

connectDB().then(() => { console.log(" DB connected")
    })
    .catch((err) => { console.log(" Failed to connect")
})
```

-> Mongoose have connect fn which accept a MongoDB connection

string to connect to the cluster and return a promise.

Step 4: Placing / Importing the DB file to app.js

```
//app.js
const DB = require("./config/database")
```

→ whenever we require the DB file all the code from DB file is read inside app.js file.

correct way:

→ 1st our app should connect to DB & then only it should start the server. So instead of calling the connectDB fn in the Database.js we just export that fn and import in app.js & connect to the server only when DB connection is successful.

eg: //app.js

```
const { connectDB } = require("./config/database")

connectDB().then(() => {
    console.log("DB connected successfully")
    //Start the server
    app.listen(7777, () => {
        console.log("Server started")
    })
}).catch((err) => {
    console.log("error")
})
```

→ Best practice is to place connection string in .env file and access it using "dotenv" package.

27 what is Schema and how to create it?

Schema:

→ Schema is like a blueprint for the document in each collection

→ It defines how the document should have what are the fields each document should contain, what datatype it should be, whether the field is required (or) not, the length of each field & so on.

Creation of Schema

→ we create schema using mongoose.Schema() fn which accept an obj where we define each field of the document along with its datatype.

→ we create Model based on Schema. Model is what actually interacts with DB. It represents a MongoDB collection along with all the operations that can be performed on that collection.

→ we create model using mongoose.model (ModelName, SchemaName)
ModelName should always starts with Capital letter.

eg:

```
const mongoose = require("mongoose")

const userSchema = new mongoose.Schema({
    name: {
        type: String, required: true
    },
    email: { type: String, required: true
    }
})

const User = mongoose.model("User", userSchema)

module.exports = User.
```

3) How to add data manually in the database through API call?

Step:1 Creation of schema & model

Step:2 Using it to create a new document

→ we create a new document by creating a new instance of the model we defined & Pass the fields & data to it by new ModelName

→ we use Save() method to save the new instance to the DB.

→ Save() methods returns a promise so we need to await the response & send the response to the client.

eg:

```
app.post("/signup", async(req, res) => {
    const user = new User({
        name: "Shan",
        email: "test@gmail.com"
    })
    await user.save() → saving it to DB
    res.send("user created successfully")
})
```

→ New instance of User Model

→ we use Post http method since we are adding the data to DB.

⇒ when we hit the route with post method it will add this new document to user collection.

4) What are "_id" & "__v" field in the document?

→ Both fields are automatically created by MongoDB.

→ _id : id fields are used to uniquely identify the document.

→ __v : this field is used to track the version of the document whenever the doc is updated the version is incremented by default it is 0.

X

Episode -7.

Diving into the APIs

1) How to add a dynamic data in the database through API?

→ usually the data comes from frontend through API, we validate it and store it in DB & send res back to the client.

→ we can also send the data from postman for testing purposes.

2) How to pass data from postman for testing API?

→ Create a POST request with the API endpoint URL

→ Select the Body tab

→ Select the appropriate format for sending data. we choose raw format as we need to send json data.

→ select the format type to JSON

→ After entering the data in the body click on send

→ Postman will send the POST request with the specified body data.

3) How to receive the data passed in the body and store it in DB?

→ the data we sent is present in request body and can access it using req.body

→ As we are sending JSON data first we need to convert it to JS object before storing it to DB otherwise the req.body will be undefined.

→ To convert the JSON data to JS object we use a built in express middleware express.json()

→ This middleware has to be on top after imports as all the data

we recieve should convert first before storing it in DB.

→ store the data recieved in the place of hardcoded data.

eg:
```
app.use(express.json())

app.post("/signup", async (req, res) => {
    const user = new User(req.body)
    try{
        await user.save()
        res.send("user added successfully")
    }catch(err){
        res.status(400).send("Error saving the user")
    }
})
```

4) what is the difference between Js object and JSON?

JavaScript object	JSON
It is a data structure in JS used to store collection of data as key-value pairs	It is data format used for transferring data between servers and web applications.
Keys don't have to be in quotes	Keys should always be in quotes
It can hold any data types like strings, numbers, array, objects, functions, undefined and null	Can hold only certain data types cannot hold function or undefined.
eg: const user = { name: "Shan", age: 50, }	eg: const user = { "name": "Shan", "age": 50 }

5) How to get a data from a db?

→ 1st step: create an get API

→ 2nd step: query the model which contains that data either by using find(), findById(), findOne() depending upon the use case.

eg: finding one document from a collection.

```
app.get("/user", async (req, res) => {
    const userEmail = req.body.email

    try {
        const user = await User.find({email : userEmail})
                                  ↓
                              Model.find
                              Name
        res.send(user)
    } catch(err) {
        res.status(400).send("Something went wrong")
    }
})
```

→ 3rd step: send the email Id from postman in body section.

Note: If there are multiple user with the same email Id find method will returns all that users in an array.

→ If the user is not found it gives an empty array so we need to check the length of the result before sending response.

eg:

```
if (user.length === 0) {
    res.status(401).send("user Not found")
} else {
    res.send(user)
}
```

6) How to get all the document from a particular collection?

→ using Model.find({})

eg:

```
app.get("/allusers", async (req, res) => {
    try {
        const users = await User.find({})
        res.send(users)
    } catch (err) {
        res.status(500).send("internal server error")
    }
})
```

Note:

→ Since we are retrieving all the documents we no need to pass anything in the body.

→ It returns all the document in an array.

7) what is difference between find({query}) & findOne()?

→ find method with query:

returns all the document that matches the query in array.

→ findOne():

returns the first document which matches the query.

↳ It returns "null" if no document matches the query.

8) How to delete a document from collection?

→ There are various ways of deleting a document from the collection such as findByIdAndDelete(), findOneAndDelete()

eg:

```
app.delete("/user", asyn (req, res) => {
    const userId = req.body.userId.
    try {
        await User.findByIdAndDelete(userId)
        res.send("user deleted successfully")
    }
```

```
catch (err) {
    res.status (100).send ("Something went wrong")
}
});
```

9) How to update a document in the collection?

→ There are 2 ways of doing it either by using findByIdAND Update() (or) findOneAndUpdate()

→ findById And Update() method accept 3 values one is Id, 2nd is the updated data, 3rd is options (optional).

eg:

```
app.Patch ("/user", async (req, res) => {
    const userId = req.body.userId
    const data = req.body.
    try{
    const user = await User.findByIdand Update (
                    {_id : userId}, data )
    res.send ("user updated successfully")
    }
    catch (err) {
    res.status (400).send ("Something went wrong")
    }
});
```

×

Episode-8

Data Sanitization and Schema Validations

Data Validation :

→ Before adding the incoming data into the DB we need to validate the data. This can be done is many ways. one such way is Schema Validation.

→ In Schema for each field we can add several validators along with the data type like required.

→ required validator → accept a boolean value it tells whether that field is required (or) not.

eg:

```
const userSchema = new mongoose.Schema({
    name: { type: String,
            required: true
    }
});
```

→ we made the name field a mandatory one. If there is no name in the incoming data mongoose will not allow the data to insert into DB rather it throws an error.

→ Similarly, there are many validators like unique, minLength, maxLength & so on. that can be used in our Schema to validate the incoming data.

Kindly refer the mongoose documentation → Schema Types for more validators

1) How to create a custom validation function ?

→ we can also create our own custom validation by creating a validator function and attach it to a validate property.

→ The validator function returns true or false. we can also provide an error message if the validation fails using message Property.

→ By default, the validator fn runs only when the new document is created and not on update:

→ If we want to run the validation fn on update, we must set the option { runValidators : true } on update API.

eg: User.UpdateOne (id, data, { runValidators : true })

```
//custom validation fn in user Schema.
const userSchema = new Mongoose.Schema({
    age: { type: Number,
           Validate: {
               validator: function(value) {
                   return value >= 18 && value <= 65
               },
               message: "age must be above 18 & below 65"
           }
    }
})
```

2) How to add timeStamps in our document?

→ when we want to add timeStamps in our documents we can simply add { timeStamps : true } in our schema as one our field.

→ It will automatically add createdAt and updatedAt fields along with the time.

```
eg: const userschema = new mongoose.Schema({
        name: { type: String },
        age: { type: Number },
    }, { timeStamps : true })
```

Note:

→ The timestamp option has to be second argument to Schema.

API - Level validation:

What is API Level Validation?

→ API level validation is a process where data is validated when it is send through an API before being stored to DB.

→ It ensures only correct, complete and secured data is send to DB.

→ It allows only the required fields & filter out unnecessary fields that are injected by any attacks.

→ After the data passes API level validation, it is stored in DB.

→ It can be done along with the Schema - validation for better security.

eg: we are allowing only certain fields to be updated.

```
app.Patch("/user", async (req, res) =>{
    const userId = req.body.userId
    const data = req.body

    try {
      //API Level Validation
      const Allowed_fields = ["about", "PhotoURL", "Skill"]

      const isUpdateAllowed = Object.keys(data).every((k)=>
          Allowed_fields.includes(k))

      if(!isUpdateAllowed){
          throw new Error("update not allowed");
      }
      const user = await User.findByIdAndUpdate(userId,
                                    data)
      res.send("user updated successfully")
}
```

```
    catch(err){
        res.status(400).("update failed" + err.message)
    }
});
```

Code Explanation:

→ we are creating an "allowed_fields" array which include the fields which can be updated.

→ we are checking the incoming data with the allowed_fields

→ if there is any mismatch in the incoming data like addition of extra field it will throw an Error which is catched by catch block.

→ if there is no Error then data is stored in DB & the document updated successfully.

Data Sanitization:

1) what is data Sanitization?

→ Data sanitization is a process of cleaning (or) altering data to ensure it is free from errors, unwanted content (or) malicious input.

→ It prevents from SQL injection & cross site scripting (XSS) attacks

→ There are libraries like validator.js (or) express-validator makes the validation very easier.

eg: using validator.js

Step1: Installation of validator

npm i validator

Step 2: including it in code for validation

```
//email validation in schema
const validator = require("validator")

const userSchema = new mongoose.Schema({
    email: { type: String,
             required: true,
             validate: {
                 validator: (value) => {
                     if(!validator.isEmail(value)){
                         return false;
                     },
                     message: "Invalid Email"
                 }
             }
    })
```

Kindly refer the validator library to know what are the validators available.

Episode 9

Encrypting Password

1) What are all the steps has to be done before sending data to DB when signing up ?

There are 2 steps involved before sending the signup data to DB

*) validating the input Data

*) Encrypting the Password.

2) How to Encrypt the Password?

→ we can encrypt the password using "bcrypt" library.

→ using bcrypt we can hash the password and also can validate the password.

"npm i bcrypt"

→ we use "hash" function to hash the password. This has to returns a promise.

syntax/eg:

bcrypt.hash(password, salt)

→ Salt is the random piece of data that gets added to the password to make it unbreakable and unique.

→ Higher the No. of. Salt longer the time it takes, if it is less it is easy to break. So, the standard salt value is 10.

→ Once the Password is encrypted we cannot decrypt it.

→ we can store the hashed password in the DB.

eg: const bcrypt = require ("bcrypt")

app.post ("/signup" async(req, res) => {

```
try {
    // validating data
    validateSignup(req)

    // Encrypting password
    const { firstName, lastName, email, Password } = req.body

    const PasswordHash = await bcrypt.hash(Password, 10)

    // Storing the hashed password to DB.
    const user = new User({
        firstName, lastName, email,
        Password: PasswordHash
    })

    await User.save()
    res.send("user added succesfully")
} catch(err) {
    res.status(400).send("Error" + err.message)
}
});
```

3) How to compare the hashed password with the incoming Password while logging in ?

→ there is a fn called "compare" which is used to compare the incoming password with the hashed Password. It returns a promise.

syntax:

```
bcrypt.compare(password, hashedPassword)
```

eg:

```
app.post("/login", async (req, res) => {
    try {
        // Validate incoming data
        validateLogin(req)

        // get the document based on email Id from DB for password comparison
```

```
const {email, password} = req.body

const user = await User.findOne({email: email})
  if(!user){
      throw new Error("Invalid credentials")
  }
 //comparing the password

  const isPasswordValid = await bcrypt.compare(password, user.password)
  if(isPasswordValid){
      res.send("user logged in successfully")
  }else{
     throw new Error("Password not correct")
  }
}catch(err){
    res.status(400).send("Error" + err.message)
  }
})
```

Episode - 10

Authentication, JWT & Cookies

1) Why we need Authentication?

→ whenever a client req some data from the server TCP/IP Protocol is made between client & server and server checks / validate the user. If the user is a authorized (logged in) the server sends back the data to client and connection is closed

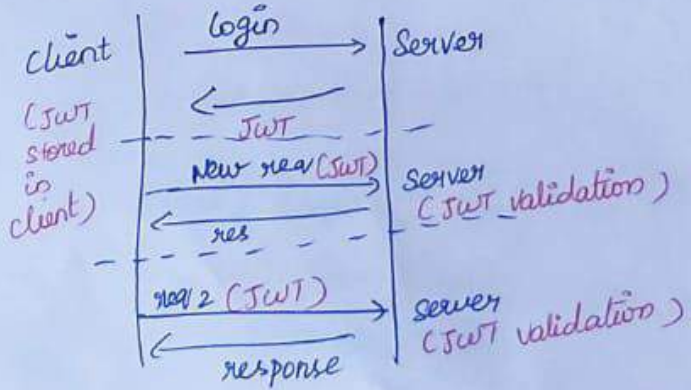
→ This Process of making an connection, validating the user, sending res back & closing the connection happens for all the requests made by a user.

2) How server validate the User who is making a request?

→ when the user logs in initially the server creates a JWT token and sends back to client along with response

→ The JWT token is stored on client side & everytime when the user makes an API call it sends back the token from client to server.

→ The server then validates the incoming Token whether it is malformed (or) not.



3) Where the JWT tokens get stored?

→ The JWT token gets stored in cookies in browser.

4) when the cookies will not work?

→ whenever the cookies (or) JWT token gets expired.

5) How to create a cookies & how to access it?

Creating cookie:

→ we can create a new cookie by res.cookie() which is given to us by Express.

→ res.cookie(Name, value)

→ Name of our cookie, value is the JWT token created.

Getting access to cookie:

→ we can get the cookie using req.cookies but we need to parse the cookie first inorder to read its value.

→ For that we use the middleware called Cookie-Parser

npm i cookie-parser.

app.use(cookieParser())

→ once we got the cookie we extract the JWT token from it & use for validation & send res back.

6) what is JWT?

→ JWT stands for JSON webToken is a way of securely sharing information between client and server.

→ It is commonly used for authentication and authorization in microservices.

7) what is the structure of JWT?

→ The JWT token consits of three parts seperated by dots (.)

*) Header → contain metadata like Token type (JWT) and hashing algorithm used (HS256)

*) Payload → contains the data that we want to send like user ID, role.

*) signature → It is used to validate the token whether some has changed (or) not. It is a combination of header, Payload and secret key we set.

8) How to create a JWT token?

→ To create a JWT token we first need to install the JWT library.

npm i jsonwebtoken

→ There is a fn called "sign()" which accepts the payload, secretkey which is used to create a JWT token.

eg:

```
app.post("/login", async (req, res) => {
  try {
    const {email, password} = req.body
    //checking user in DB
    const user = await User.findOne({email : email})

    if(!user){
      throw new Error("user not found")
    }
    // comparing Password
    const isPasswordValid = await bcrypt.compare(Password,
                                user.password)

    if(isPassword valid){
      //creating token
      const token = await Jwt.sign({_id: user._id},
```

```
        secret key )
    // Adding token to cookie & send back to client along with response
    res.cookie ("token", token)
    res.send ("login successfull")
  } else {
      throw new Error ("Invalid credentials")
  }
} catch (err) {
    res.status (400). send ("Error " + err.message)
}
})
```

7) How to verify the JWT token on subsequent request?

-> To validate the incoming Jwt token on further request there is a function called "verify" which accepts the incoming token & secret key which is used when creating a token.

eg:

```
app.get ("/profile", async (req, res) => {
    try {
        const cookies = req.cookies
        const {token} = cookies
        if (!token) {
            throw new Error ("Invalid token")
        }
        // decoding/validating the token to get the payload back
        const decodedToken = await jwt.verify (token, secretkey)
        const {_id} = decodedToken
        // finding user profile based on id stored in token
        const user = await User.findById (_id)
```

-> getting the cookie & token from that cookie

```
if (!user){
    throw new Error("user does not exist")
}
res.send(user)
} catch(err){
    res.status(400).send("Error" + err.message)
}
})
```

10) How to create a Auth middleware?

-> we need to provide access to all the route only after logged in except signup & login route.

-> Inorder to validate JWT token for all the route we are creating a middleware which will validate the token & sends back the user corresponding to it

-> Then we pass the req handlers after that to handle the api request.

Creating middleware

eg.

```
const userAuth = async (req, res, next) => {
    try {
        //getting token & validating it
        const {token} = req.cookies
        if (!token){
            throw new Error("Invalid token")
        }
        const decodedToken = await jwt.verify(token, secretKey)
        const {_id} = decodedToken
        //getting user
        const user = await User.findById(_id)
        if (!user){
            throw new Error("user not found")
        }
        //sending user
        req.user = user
```

```
        next()
    } catch (err) {
        res.status(400).send("ERROR " + err.message)
    }
})
```

using Middleware:

```
app.get("/profile", userAuth, async (req, res) => { })
```

11) How to set a expiry time for the JWT token?

→ we can set the expiry time for the token when creating it by passing a "expiresIn" prop as a 3rd argument to "sign" fn.

eg:
```
const token = await jwt.sign(data, secretKey, {
                                    expiresIn: '1h' })
```

12) How to set a expiry time for the cookies?

→ Similarly we can set the expiry time for cookie when creating it by passing "expires" prop as 3rd argument to "res.cookie( )"

eg:
```
res.cookie("token", token, { expires: new Date(Date.now
                                    +8 * 3600000) } )
```

13) what are Schema methods?

→ These are methods that are directly attached to Schema and are available for all the document that are created based on that Schema.

→ Arrow fn are not allowed for Schema methods as "this" keyword is undefined.

14) How to create Schema methods?

→ to create a Schema methods we can use "Schema.methods" fn to attach a fn to the schema.

eg: // userSchema.js

```
userSchema.methods.validatePassword = async function (pwd) {
    const isPasswordValid = await bcrypt.compare (pwd, this.password)
    return isPasswordValid
}
```

pwd → incoming pwd

this → this points to current user document (hashed pwd)

Usage of Schema Method:

```
const isPasswordValid = await user.validatePassword (password)
```

→ This Schema methods enable us to have a cleaner, modular and maintainable code.

Episode - 11

Diving into the API's and express Router

1) what is Express Router?

→ the Router is a way of organizing and managing routes of our application.

→ It is like a seperate mini express app that handles set of routes which we can connect to our main application.

2) why to use Express Router?

→ when the application grows manage all the routes in a single file becomes hard so router group related routes together in a seperate module (file).

→ It makes our code clean and maintainable.

3) How to create a express Router?

→ we can create a router using "express.Router()" fn. and attach the routes to it.

eg: //user.js

```
const express = require("express")
const router = express.Router()

// define routes for user
router.get("/user", (req, res) => {
    res.send("user list")
})

module.exports = router
```

4) How to connect the router file with the main file?

→ We connect the router just like we connect with middleware using "app.use()"

eg: //app.js

```
const express = require("express")

const app = express()

const userRouter = require('./users')

//connecting the user Router
app.use("/users", usersRouter)
```

→ whenever the request starts with "/users" it goes to the userRouter

×

Episode - 12

Logical DB Query & compound Indexes

1) How you reference a objectId in Schema?

-> the type of object Id is "mongoose.Schema.Types.ObjectId"

eg:
```
const userSchema = new mongoose.Schema({
    userId: {
        type: mongoose.Schema.Types.ObjectId
    }
})
```

2) how to compare objectId with the string version of id?

-> there is function called "equals()" which takes string version id & compare it with object Id. whether both are equal (or) not

eg:
```
ObjectId.equals(string id)
```

3) what are the types of middleware in Mongoose?

Mongoose Middleware:

→ Mongoose middleware (also known as "hooks") allow us to run some logic before (or) after certain actions such as save, update, delete in documents

→ we can also validate the documents using this middlewares.

Types of Mongoose Middleware:

*) Pre Middleware ⇒ Runs before certain action occurs

*) Post Middleware ⇒ Runs after certain action occurs.

Use cases of these Middleware:

*) Pre-Save Middleware: used for hashing password before saving to DB

*) post-Save Middleware: Logging changes after saving a document

*) Pre-remove Middleware: clean-up data before removing a doc

*) Post-remove Middleware: notify that doc is deleted.

4) How to create a Mongoose Middleware?

→ we attach the pre (or) post middleware function in the Schema

→ this middleware fn accept 2 arg: 1st → action 2nd → fn

→ Arrow fn is not allowed as it does not have "this" keyword.

eg:
```
Schema.type("action", fn(){ })
userSchema.pre("find", function(next){
    this.where({isActive: true}) → find active users
    next()
})
```

→ Don't forget to call next as it is a middleware.

4) what is Indexing and why we need it?

→ Indexing improves the efficiency of querying operation.

→ when a collection grows querying for single doc by scanning the larger data makes the querying process slow & inefficient.

thus Index allow us to quickly locate & retreive that single doc from the largest collection.

Index:

→ Indexes store a small portion of the collection's data in a way that makes search faster.

→ It is used for query optimization, sorting, uniqueness.

5) what are the type of Indexes in MongoDB?

*) Single field Index: created on single field.
*) Compound Index: created on multiple field
*) Multikey Index: Supports indexing array fields
*) Text Index: used for full-text search
*) Hashed Index: used for sharding in MongoDB
*) Unique Index: ensure that indexed fields have unique values.

6) How to create a Index?

→ we define index in the schema either by using "index()" method (or) as options in schema field definitions

eg: using index() method

```
const userSchema = new mongoose.Schema({
    name : String,
    age : Number
});
```

// compound index on name & age field

```
userSchema.index({ name: 1, age: -1 })

const user = mongoose.model("User", userSchema)
```

→ 1 -> ascending order
→ -1 -> descending order.

eg: Mongoose option

```
const userSchema = new.mongoose.Schema({
    name: { type: String,
            index: true   → single field index
          },
    email: { type: String,
             unique: true  → unique index.
           }
})
```

Episode - 13

Ref, Populate & Thought Process of writing APIs

Thought Process of writing API's

POST:

→ we should think what are all the possible ways that attackers can attack a POST API.

→ eg: Sending some random data to API by attackers.

→ Validate at Schema level, API level validation.

GET:

→ we should make sure the data we are sending back to client

→ check for authorization of the user and he has permission to access the protected routes

1) How to make connection between two collections ?

→ In MongoDB, we make connection between 2 collections using "reference". instead of embedding the entire document we store a reference (usually ID) of that doc2 in different collection.

→ This is a common way to represent one to many & many to many relationships.

2) what is ref ?

→ It is used to define a relationship between 2 collections

→ It specifies which model/collection a field is referencing to

eg: Lets create 2 collection

```
// user Schema
const userSchema = new mongoose.Schema({
```

name : String,
email : String

})

```
// Post Schema
const postSchema = new mongoose.Schema({
    title: String,
    content: String,
    author: {
        type: mongoose.Schema.Types.ObjectId,
        ref: "User"
    }
})
```

→ it made connection to user schema using Id as reference.

3) what is populate ?

→ It is used to retrieve the related document based on reference ID from the related collection.

eg:

→ "populate()" method accept 2 args (1st - field which has reference that we want to populate, 2nd - what are the data we want to retrieve from that reference document (optional)

→ we can give the 2nd args as [ ] sperated by , (or) as a single string seperated by space.

eg:

```
const post = await Post.findOne({ title: "first post" })
    .populate('author', ["name"])
console.log(post)
```

↓

it gives the post content along with author field & set with "_id" & "name" fields.

Episode - 14

Building feed API & Pagination :

1) what is Select method?

→ Select method is used to select fields that are need to be returned in the query result.

→ It is used to include only certain fields in the document & exclude the rest of the field.

2) How to do a Pagination in Mongodb?

→ There are two important fn in mongodb for pagination

* skip() → it is to skip the no. of. document
* limit() → it is to limit the no. of. document retrieved.

→ Skip can be calculated always using (Page - 1) * limit

→ both Page & limit values always come as "query parameters" in the request api

→ we can chain skip & limit methods on a query to get the desired documents.

eg:
```
app.get ("/feed", async (req, res) => {
  try {
    const feed = await
```

eg:
```
app.get ("/feed", async (req, res) => {
  try {
    const Page = ParseInt (req.query.Page) || 1
    let
    ~~const~~ limit = ParseInt (req.query.limit) || 10
    limit = limit > 50 ? 50 : limit
    const skip = (Page - 1) * limit.
```

if limit is greater than 50 we are limiting it to 50

```
const feed = await User.find({}).skip(skip).limit(limit)

    res.send(feed)
} catch(err){
    res.status(400).send("ERROR" + err.message)
}
})
```

X