(19)

25/Sep/2024

Page No.:
Date: / /

(30)

Day-11

→ HTTP | HTTP Methods | HTTP Headers

Difference between HTTP and HTTPS

HTTP (Hypertext Transfer Protocal) :- HTTP transfers data in plain text, meaning the information you send, like "abc" is transmitted as-is. There's no encryption, so data is vulnerable to interception and can be easily read by third parties.

HTTPS (Hypertext Transfer Protocol Secure) :- HTTPS adds a layer of security by encrypting the data before it's trasmitted. This ensures the data is unreadable while in transit and only becomes readable once it reaches the intended server or client. this encryption keeps the Communication secure from eavesdropping or tampering.

⇒ What are HTTP headers

Metadata → Key-value Sent along with request & response

→ Caching, authentication, mango State
X-Prefix → 2012 (X- deprecated)

- Request Headers → from Client
- Response Headers → from Server
- Respresentation Headers → encoding / compression
- Payload Headers → data

Send data like _ id :-
Email :-

Most common Headers

- Accept → application / json
- User → Agent
- Authorization →
- Content → Type
- Cookie
- Cache → Control

CORS

Access → Control - Allow - origin
Access → Control - Allow - Credentials
Access → Control - Allow - Method

Security

Cross → Origin - Embedder - Policy
Cross → origin - Opener - policy
Content → Security - Policy
X - Xss → protection

Page No.:
Date: / /

HTTP methods

Basic set of operations that can be used to interact with server.

- GET → retrieve a resource
- HEAD → No message body (response headers only)
- OPTIONS → What operations are available
- TRACE → loopback test (get some data)
- DELETE → remove a resource
- PUT → replace a resource
- POST → interact with resource (mostly add)
- PATCH → change part of a resource.

Most Commanly Method

Get, Post, Put, Delete and Sometime, Patch.

HTTP Status Code

- 1xx → Informational
- 2xx → Success
- 3xx → Redirection
- 4xx → Client error
- 5xx → Server error

100 → Continue
102 → Processing
200 → OK
201 → Created
202 → accepted
307 → temporary redirect
308 → Permanent redirect
400 → Bad request
401 → Unauthorized
402 → Payment required
404 → Not found
500 → Internal Server Error
504 → Gateway time-Out

⇒ URL (Uniform Resource Locator)
- Specifies the location and how to access a resource on the web
- Example:
  ↳ https://www.example.com/po

⇒ URI (Uniform Resource Identifier)
- A broader term that identifies a resource by location, name or both, it includes URL and URNs.
- Example: http://www.example.com/resource.

⇒ URN (Uniform Resource Name)
- A type URI that names a resource without indicating its location.
- Example: urn:isbn:0451450523 (for a Book)

Page No.:
Date: / /

Day - 12

→ Router and Controller with debugging

Overview of Routes and Controllers:

In modern web application, we use routes and controllers to organize our backend code. let's break down why we use these two concepts and how they interact in your/our project.

1. **Routes:**

Routes define "what URL pattern what functionality" in the application. Every route corresponds to a specific API endpoint, and when the user makes a request to that (like! /register), the corresponding function (or controller), is called to handle the request.

Why should we use routes ?

- Seperation of Concerns! Routes focus on handling URL paths, while Controllers focus on the actual business logic
- Scalability! As the app grows, its easier to manage different URL paths and functionalities

2.

#

(34) Routes Code

```
import {Router} from "express"
Import {registerUser} from
        "../controllers/user.controller.js"
Const router = Router()
router.route("/register").post(registerUser) //post
export default router
```

request for user registration

↓

Here, the route "/register" is defined. When a POST request is made to this route, it triggers the registerUser function in the controller.

2. Controller

Controllers handle the business logic of your application. They are responsible for processing request, interacting with database (if required), and sending responses back to the client

Why do we use controllers?

- Code Clarity and Organization: Controllers allow you to organize your business logic separately from routing. This keeps your/our Clean and modular.
- Error handing and async logic: Controllers also handle the asynchronous tasks and potential errors.

(35)

Controller code

```
import { asyncHandler } from '../utils/asyncHandler.js'
const registerUser = asyncHandler(async(req, res) => {
    res.status(200).json({
        message: "ok"
    })
})
export { registerUser }
```

1) Response for successfully user registration.
2)
3)

→ The registerUser Controller here handle the registration request and responds with a status of 200 and JSON message of "ok"

(8) Asynch Handler Code

```
const asyncHandler = (requestHandler) => {
    return (req, res, next) => {
        promise.resolve(requestHandler(req, res, next)).catch
        ((err) => next(err))
    }
}
export { asyncHandler }
```

→ This function wraps your asynchronous Controller (registerUser), catching any errors and fowarding them to express's error handling middleware.

) Connecting everything in app.js :

You declare your routes in app.js so the application knows where to direct incoming requests.

(36)

app.js code,

```
import userRouter from "./routes/user.routes.js"
    // Declare routes.
app.use("api/v1/users", userRouter)
```

→ Here, all routes related to users (like/ (like: /register) will be prefixed with /api/v1/users. So the full Path for registration is /api/v1/users/register.

Summary →

- Routes handle URLs and map them to controllers.
- Controllers process the request and send a response.
- asyncHandler helps manage async code and error handling smoothly
- Express routes are setup in app.js to organize the endpoint.

Page No.:
Date: / /

Day-13

→ Logic building / Register Controller.

Multer Middleware (upload.field())

↳ To handle file uploads (eg. user avatar, cover images).

```
upload.field([
  { name: "avatar",      ← Profile picture
    maxCount: 1
  },
  {
    name: "coverImage",  ← Cover Image photo
    maxCount: 1
  }
])
```

maxCount: Limits the number of files allowed per field (in this case, only one file per field)

Why it's needed → This middleware process multipart form data (like images) and them to req.files. this helps in managing file uploads before passing the control to the next step, the register function.

- It's necessary when you expect multiple file fields.

// File Handling in Controller (registerUser):

→ this function will handle the logic after files are uploaded

What it does:

This (registerUser) function is called after (upload.fields()) meaning files are now available in (req.files) inside this function.

- We can access uploaded files like req.files.avatar and req.files.coverImage.
- If you'd also handle user registration logic, such as saving user data to the database.

Why Multer with fields:

↳ we use upload.fields() instead of upload.single() or upload.array() because you have two distict field (avatar and coverImage) for different purposes

What it helps with:

↳ Ensure that each field get handled independently.
↳ prevents one field from accepting more than the allowed file count (maxCount: 1)

- .trim() :- Remove extra space from strings.
  Why: Enswres clean and consistent data input avoiding issues caused by extra spaces.
- $or (MongoDB) :- Allows checking multiple conditions
  Why: Ensure that at least one condition is true (e.g. either email or username exists)

- req.files : Access uploaded files when using multer.
  why :- Allows processing and accessing uploaded files in req.files.

- User.create() : Creates and saves a new user in the database.
  why :- To store user information, including file paths, in MongoDB

- findById() : finds a document in MongoDB by its _id.
  why :- used to retrieve specific user data based on their unique ID

- .select() : specifies which fields to include/exclude in a MongoDB query.
  Why :- optimizes performance by returning only necessary fields

- res.status() : sets the HTTP response status code.
  why :- provides the correct status for the client (e.g., 200 for success, 500 for errors.

Day-14
How to use Postman and Register Controller

Acce
- Purpo
- Lifes
- Usag
- Secu

Ref
- pu
- lo
- Li

40

29/Sep/2024

Page No.:
Date: / /

Day- 15

→ Access Token and Refresh Token.

Access Token :-

- Purpose: Used to access protected resources (e.g., APIs)
- Lifespan: Short - lived (usually minuts to a few hour.
- Usage: Sent with each request to the server to prove the user's identity.
- Security: Needs to be secure, as it provides access to resources.

Refresh token :-

- purpose: Used to get a new access token without requiring the user to log in again.
- Life Span: Long - lived (can last for days, weeks or longer).
- Usage: Sent when the access token expires, allowing the generation of a new access token.

Long-Lived vs Short-Lived Tokens:

- Long-Lived Tokens: Last for extended periods (e.g refresh tokens).
They minimize user re-authentication but can pose security risks if compromised.

- Short-Lived Tokens: Expire quickly (e.g, access token), enhancing security by reducing the time a stolen token can be used.

(42)
8/04/2024

Page No.:
Date: / /

Day-18

→ How to Design Custom Middleware and logout functionality.

- Middleware: JWT Verification

Middleware is created to verify the JSON web Token (JWT) for secured routes to ensure only authenticated users can access specific parts of the application.

Step-1 Extract the token from the Cookies or Authorization Header.

Step-2 If no token is found, an error with status 404 (Unauthorized) is thrown.

Step-3 Verify the extracted token using jwt.verify with key from the environment variable (ACCESS_TOKEN_SECRET).

Step-4 fetch the user based on the token's decoded ID, excluding sensitive fields like Password and refreshToken.

Step-5 If the user is not found, throw an error (401 ! Invalid Access Token)

Step-6 Attach the verified user to req.user to be used in the subsequent route handler.

Page No.:
Date: / /

Logout functionality.

The logout functionality ensure the user is logged out securely by removing tokens and clearing cookies.

Step-1 Update the user's refreshToken field to undefined in the database to invalidate any previous sessions.

Step-2 set httpOnly and secure options for cookies to ensure security during logout

Step-3 clear both accessToken and refreshToken cookies.

Step-4 Send a response confirming that the user has successfully logged out with status 200.

⇒ Routes

- Login Routes-

  router.route("/login").post(loginUser) is used for loggin in the user.

- Secured Logout Route:

  router.route("/logout").post(verifyJWT, logoutUser) ensures that before a user can logout, their JWT is verified using the verifyJWT middleware

Page No.:
Date: / /

⇒ Why Middleware and Logout Are Important:

- Middleware :- Ensure that only authenticated users with valid JWTs can access secured routes, protecting sensitive user data and resources.

- Logout : By clearing Cookies and setting refreshTok- to undefined, it prevents unauthorized access from expired or reused tokens, enhancing security.

Note ← → JWT (JSON Web Token): A compact, URL-safe means of representing claims between two parties.

APIError: Custom error class used to handle API errors systematically.

asyncHandler: Utility function to wrap async route handlers and avoid redundant try-catch blocks.

4 lectures

Day-20

1. changeCurrentPassword → Allow a user to change their password.

Step-1 Extract oldPassword and newPassword from the requested body.

Step-2 find the user by their ID (req.user?._id).

Step-3 check if the provided oldPassword is Correct using the method isPasswordCorrect.

Step-4 if incorrect, throw an error with the message "Invalid old Password".

Step-5 if Correct, update the user's password to newpassword and save the changes without validation (validateBeforeSave: false)

Step-6 Send a success response "Password Change Successfully"

why :- Ensures users can securely update their passwords by verifying their old password.

2. getCurrentUser :-

→ Fetches the details of the currently loged-in user.

Step

Step Simply returns the current user data (req.user) with a success message

```
Const getCurrentUser = asyncHandler(async(req, res) => {
    return res
    .status(200)
    .json(200, req.user, "current user fethched Successfully")
})
```

Why :- Useful for displaying or confirming the logged-in user's details.

3. updateAccountDetails :-

↳ Allows users to update their account details like fullName and email

Step-1 Extract fullName and email from the request Body.

Step-2 Check if both fields are present; if not, throw an error "All fields are required.

Step-3 find the user by ID and update the fields with the new values ($set : {fullName, email}).
↳ Operator

Step-4 Return the updated user data without the password (.select("-password"))

Why :- Provides a way for users to update personal information securely without affecting their password.

4. updateUserAvatar: → Allow users to update their profile picture (avatar).

Step-1 Get the file Path of the avatar from req.file?.path.
if no file is provided, throw an error "Avatar file is missing".
Upload the avatar to Cloudinary and get the URL
If upload fails, throw an error "error while uploading avatar.
Update the user's avatar field with the new URL
($set: { avatar: avatar.url })
Return the updated user data with a success message

Why :- Lets users change their avatar image by uploading it and storing the Cloudinary URL

5. updateUserCoverImage: → just like "updateUserAvatar"

Common Concepts

Error Handling: Uses custom ApiError for validated input (e.g, missing fields, wrong password)

Cloudinary Integration: uploads media (avatars and CoverImages) to Cloudinary and retrieves the URL's.

MongoDB updates: Uses findByIdAndUpdate with $set to update specific like email, fullName, avatar, and coverImage.

Async & Await & asyncHandler: Makes sure each function is asynchronous and errors are caught and handled properly.

Let s

Th

Now

48

6/oct/2024

Day - 21

Page No.:
Date: / /

→ Understanding the Subscription Schema

Subscription Schemes

- Subscriber
- Channel

→ there are only two values.

Let say → We have some Users like:- x, y, z,
and some Channels:- CAC, HCC, FCC
But in the end, User's and Channels
Both are just users.

The structure of Document

Subscriber
Channel

→ Now, every time the user Subscribes to a channel, a document is created.

Now let's see how the document will be created and how we will retrieve data from it

Now Suppose we have a user → x and they have Subscribed to a channel called CAC. How will the document for this be created?

Document

Ch → CAC Subs → x	Ch → CAC Subs → y	Ch → CAC Subs → z

And it will continue like this moving forward as well.

Ch → MCC Subs → z	Ch → PCC Subs → z

z → Subscribed multiple channels.

⇒ So Here the Same user can Subscribe to multiple channels, and multiple users can also Subscribe to CAC. A seperate document is being created for each of them, and as users keep subscribing, a huge document set will be prepared!

⇒ The main point here is that if we need to count the Subscribers for a Channel, we will Select all the documents where the Channel is CAC. instead of Counting Subscribes we will match the channels.

like :-

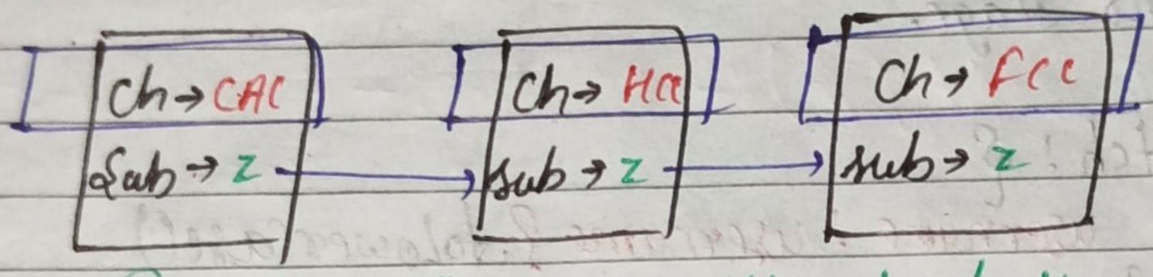
1	2	3
Ch → CAC Subs → x	Ch → CAC Subs → y	Ch → CAC Subs → z

→ Here are three (3) Subscriber

Simple we select channels not Subscribers.

Now, we need to determine which channels we have Subscribed to. let's say we have a user 'x'. How can we find out how many channels 'x' has Subscribed to?

So, we will select the Subscriber with the value 'x'



Now, from this, we will extract the list of channels.

Summary →

- We have documents being created under this everytimes
- As soon as we Subscribe to a channel, a document will be created.
- And as soon as someone Subscribes to the channel, a document will be created."

(51)

6/oct/2024

Note Params → URL

Page No.:
Date: / /

Day-22

→ Mongodb aggregation pipelines.

1. # Aggregation pipeline :- In MongoDB, an aggregation pipeline is used for data Processing and transformation. It Consists of a sequence of stages, where each stage takes input, processes it, and passes the output to the next stage.

⇒
```
{
  $match: {
    username: username?.toLowerCase()
  }
} //$match: Filters the documents based on the username. This act like a Where clause in SQL
```

⇒
```
{
  $lookup: {
    from: "Subscriptions"
    localField: "_id"
    foreignfield: "channel"
    as: "subscribers.
  }
}
```

$lookup :- Performs a left Join with another collection. In this Case, it's used twice.
- → first, to join with the Subscriptions collection to fetch all subscribers of the user.
- → Second, to fetch all channels to which the user is subscribed.

(52)

⇒ $ado

⇒
```
$addFields: {          //→ $addfields: Adds new fields to each
    subscribersCount: {          ↗field.   document based
        $size: "$subscribers"              on calculations
    },                                     or conditions.
    channelsSubscribedToCount: {
        $size: "$subscribedTo
    },
    isSubscribed: {
        $cond: {
            if: {$in: [req.user?._id, "$subscribers.
                                    subscriber"]},
            then: true,
            else: false
        }
    }
}
```

2. Operators →

- $size :— The $size operator counts the number of elements in an array, used here to determine the count of subscribers and subscribed channels.
- $in :- The $in operator checks if a specified value exist within an array, used here to see if the logged-in user is among the subscribers.
- $cond & $if :- The $cond operator acts like an if-else statement, using if to evaluate a condition and returning a specified value based on whether the condition is true or false

```
{
  $project: {
    fullName: 1,
    username: 1,
    subscribersCount: 1
    ....
    ....
  }
}
```

$project :- Specifies which fields should be included in the final output. this helps limit the data sent back in the response, improving performance and security.

2. # $lookup (Left Join) :- $lookup :- is used for joining two collections in MongoDB, Similar to SQL left join.

it's used to →

- Get Subscribers of a user's channel (joining on channel)
- Get channels to which a user is subscribed (joing on Subscriber)

4. Security & Performence :-

- Limiting Output fields with $project :- By selecting only necessary fields, you reduce data exposure, making the API more secure and efficient.
- Validation :- Ensuring username is valid and non empty prevents unnecessary database queries.

(54)

7 Oct 2024

Day - 23

→ Sub Pipelines and routes

Sub-Pipeline :- A Sub-pipeline in MongoDB is a series of aggregation stages used within a $lookup to process documentation documents from the joined collection. It works like a mini-aggregation pipeline, allowing you to filter, sort, project, or even perform additional lookups on the joined documents.

Why Use a Sub-pipeline ?

- **Advance filtering :-** Apply conditions directly to the joined documents.
- **Transformation :-** Modify or reshape joined documents before merging them.
- **Nested Lookup :-** Perform deeper joins, such as fetching related data from multiple collections with in a single aggregation operation.

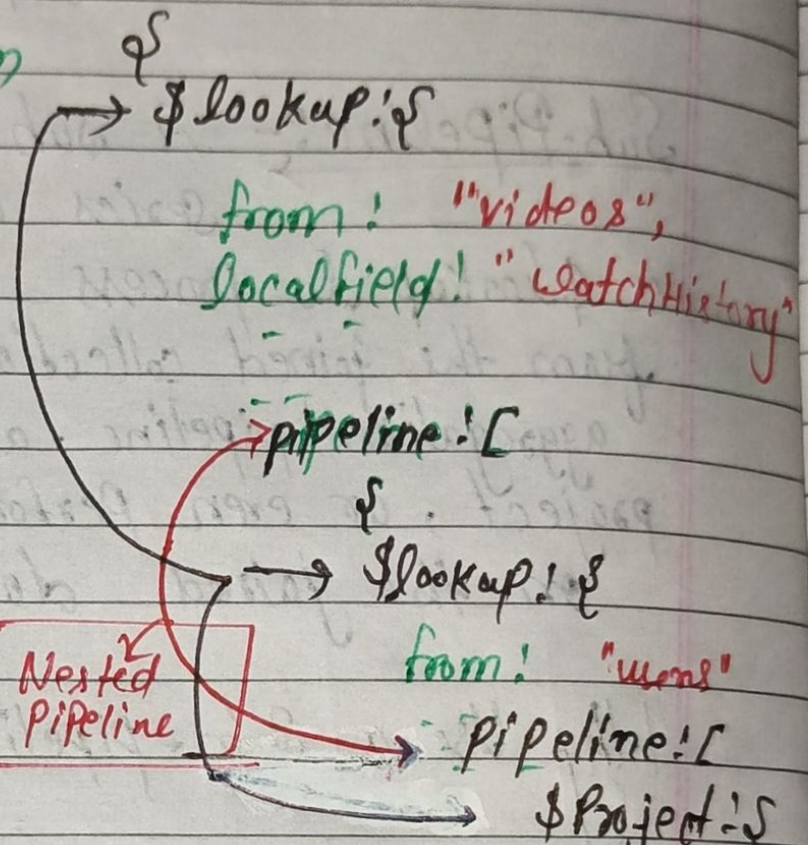
**$First :-** The first ($first) operator extract the first element from an array, useful here to Select the first owner (since there should be only one owner for each video).

55

⇒ Nested $lookup :-

- Here, $lookup is used within another $lookup, allowing the code to fetch data from the users collection while looking up videos.

- This enables multi-layer data fetching. In this case, after getting the watch history, it further retrieves details about the owner of each video.



⇒ Using $project in Nested $lookup :-

3/3/23

- In the nested $lookup for users, $project is used to select only specific fields (like, fullName, username, and avatar).

This reduce the amount of data fetched by including only the necessary fields, making the response more Efficient and secure.