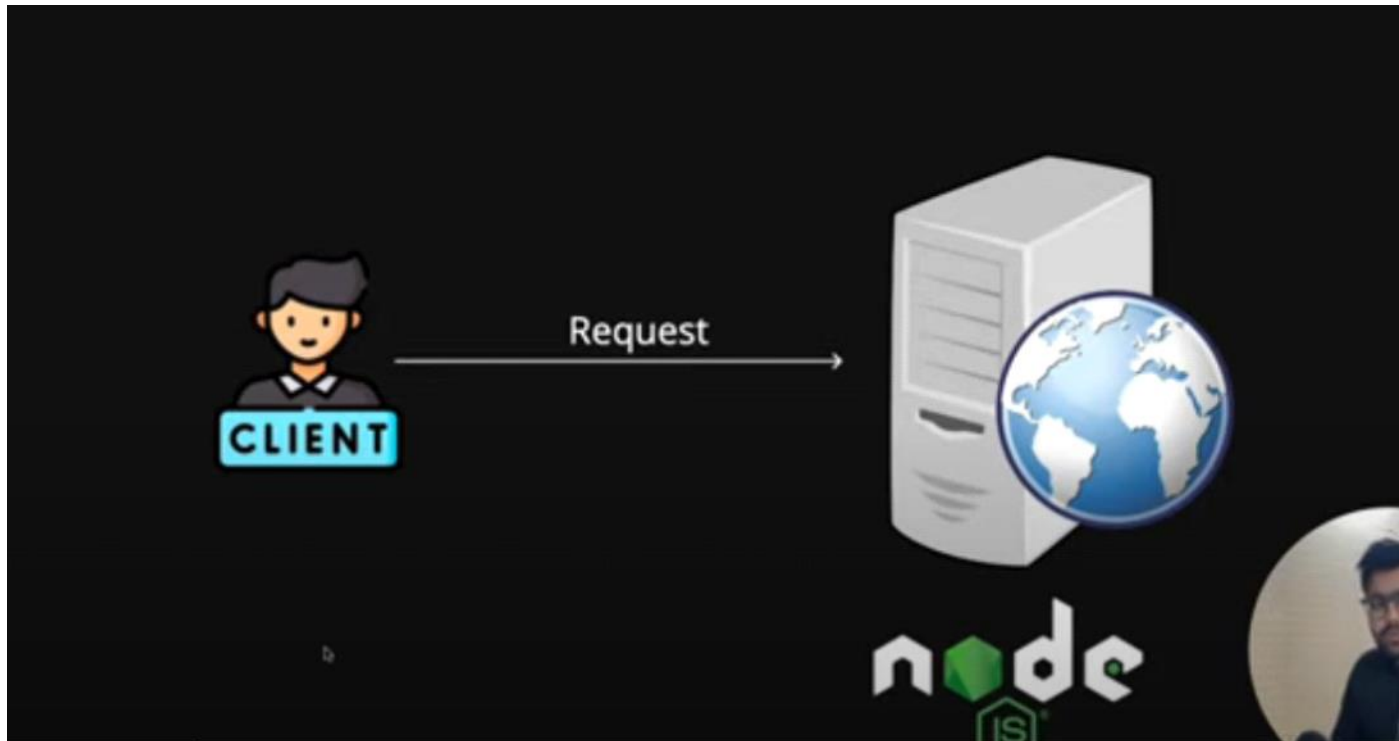
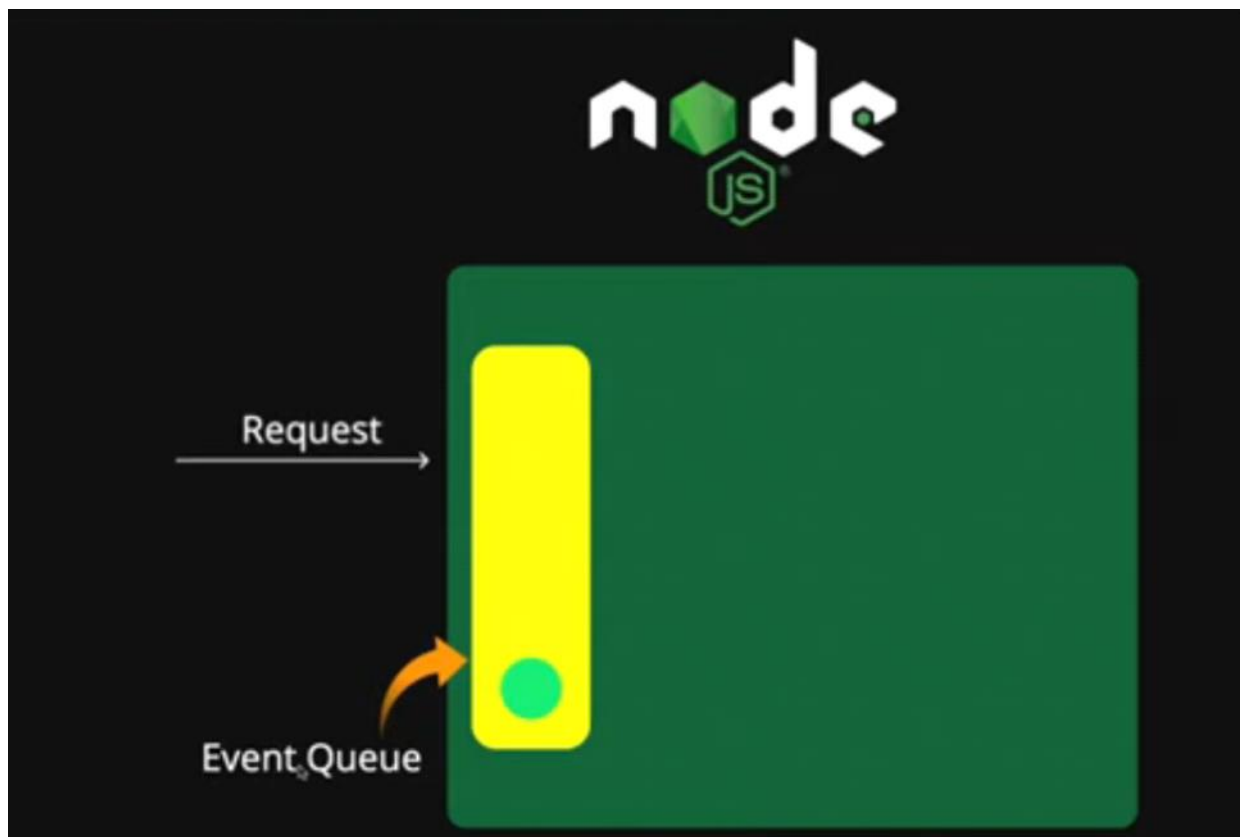


How NodeJS works:6

16 March 2024 11:47

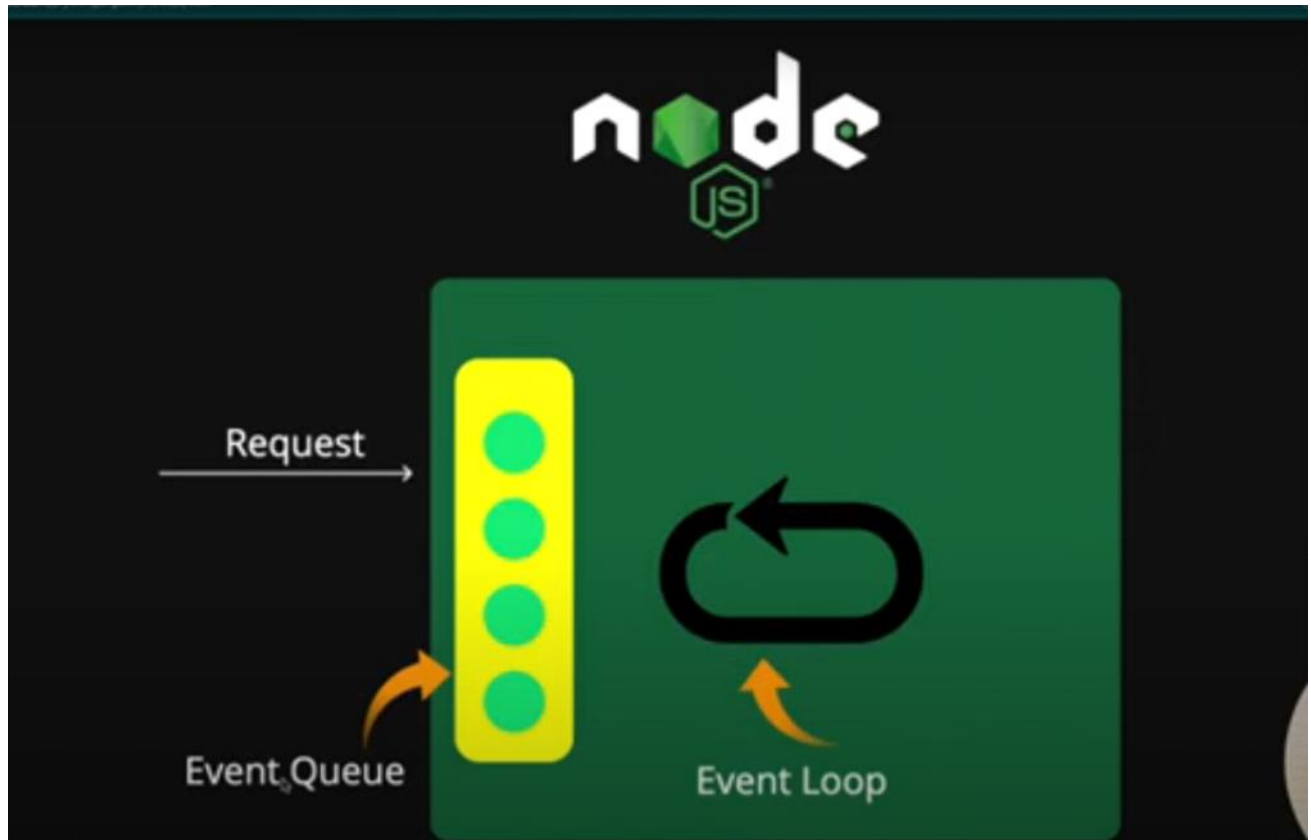


- In this scenario, the server is based on Node.js, although it could also be implemented in PHP or any other programming language.
- Now, observe this green box; it represents the Node.js server.
- The initial request is made by the client, after which all subsequent requests are queued in the event queue.

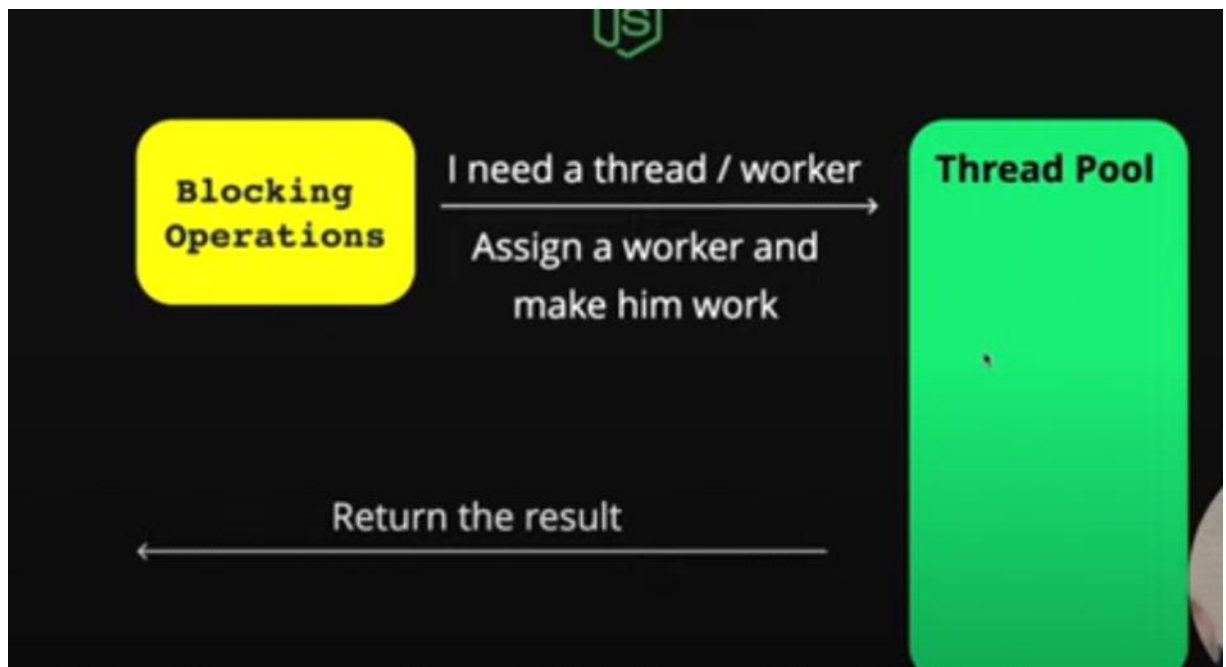


Event Queue

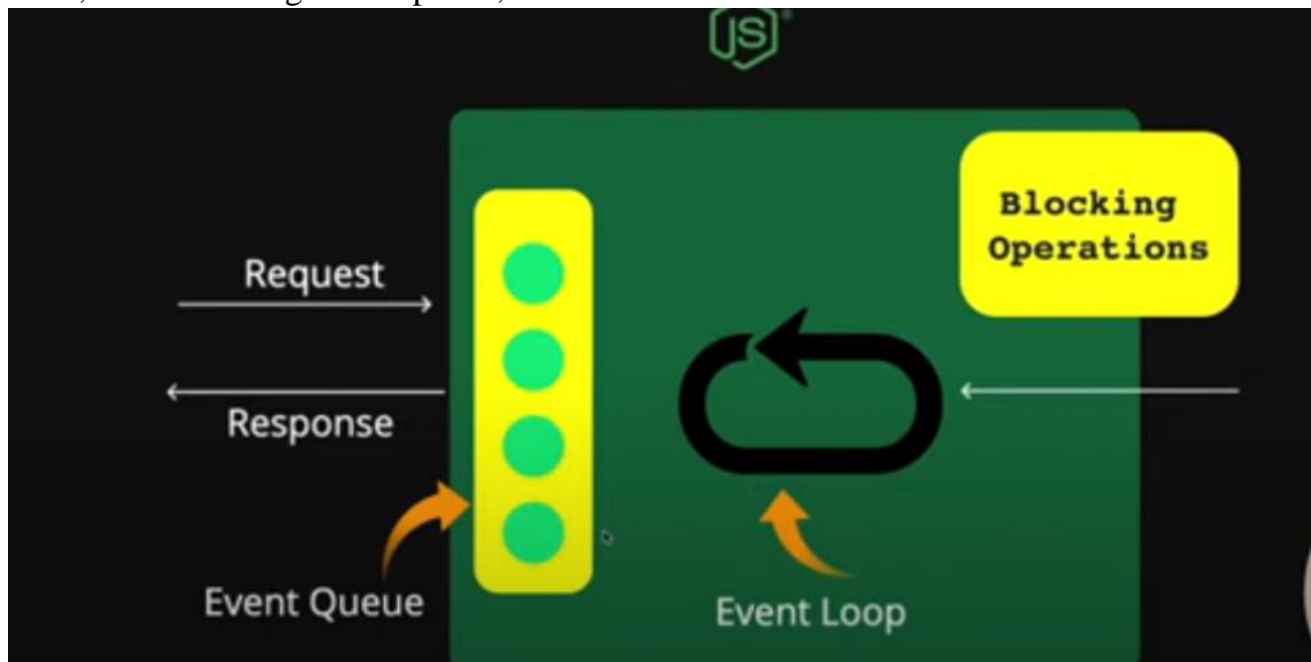
- The event loop monitors the event queue, processing requests in the order they were received using the FIFO (First In, First Out) principle.
- Then



- When we select requests from the event queue, they can be of two types: either blocking or non-blocking operations.
- Initially, the event loop checks whether it's a blocking or non-blocking operation. If it's non-blocking, the event loop processes it and sends the response back to the user accordingly.
- However, if it's a blocking operation, the thread is passed to the thread pool. In the thread pool, there are threads which you can consider as workers. Each blocking request is assigned to a thread, and it carries out the task before returning.



- Now, after receiving this response, we will send it back to the user.



- If it's non-blocking, it gets processed directly.(remember this)
- There are limited threads available. Let's assume there are 4 threads, and 4 blocking requests are being processed. Therefore, incoming requests will have to wait. When we execute synchronous code, it continues to the next line only after the current line has finished executing.
- For instance, in the case of reading contacts, the program won't proceed until the contacts are read.

```
2
3 console.log("1");
4 // Non - Blocking...
5 fs.readFile("contacts.txt", "utf-8", (err, result) => {
6   console.log(result);
7 });
8
9 console.log("2");
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ... zsh + v

```
● piyushgarg@Piyushs-MacBook-Pro hello-world % node file.js
1
2
Piyush Garg: +911111111111
Jhon Garg: +9122222222
```

- Can you increase the size of the threads, yes, but upto a limit, size depends on the cpus (cores) you have

```
// Default Thread Pool Size = 4
// Max? - 8core cpu | 8
```

- length of cpu

```
const os = require("os");

console.log(os.cpus().length);
```

HTTP Server

16 March 2024 11:48

npm init

- It will create a package.json file
- Create a index.js name file in the root location



- ```
const http = require("http");
```
- Http is inbuilt function of Nodejs
- Now we will create server using createServer , now this arrow function is responsible for processing our incoming requests, it has a request handler which has access to req, res

```
const myServer = http.createServer(() => {});
```

- ```
const http = require("http");  
  
const myServer = http.createServer((req, res) => {  
  console.log("New Req Rec.");  
  res.end("Hello From Server");  
});
```

- We will end our response with res.end
- Now we need a port to actually run our server
- Ports are sort of doors, on which door you want to run your server
myServer.listen(8000) :this is compulsory , but you can add a optional statement , that if everything goes right you can add a console to indicate that

```
const http = require("http");

const myServer = http.createServer((req, res) => {
  console.log("New Req Rec.");
  res.end("Hello From Server");
});

myServer.listen(8000, () => console.log("Server Started!"));
```

- Just change scripts to this , and then you can run your node file by just using npm start

```
"scripts": {
  "start": "node index"
},
```

OUTPUT DEBUG CONSOLE TERMINAL ...

```
rg@Piyushs-MacBook-Pro server % npm start
```

- You can actually see what information request can provide you if you do this, every information is available which is sent by the user
- You have to write this in the call back function & then you can see what's there the user have to offer
- If you want any particular thing, then do req. , otherwise just log the whole req , but it will be a very big object .

```
console.log(req. |
```

- By ctrl + C you can end the server
- Ok now we are also mixing file handling with this
- We are using non blocking req, because if we use synchronous ones and if all the threads are full then user have to wait, and we don't want that
- When we run the server, a file will be created named log.txt and what will be appended in that file 'log' which we have defined above, so when the file will be appended then only we will send response from res.end

```
1 const http = require("http")
2 const fs = require("fs")
3 const myServer = http.createServer((req, res) => {
4   ...const log = `${Date.now()};New Req Received\n`;
5
6   ...fs.appendFile("log.txt", log, (err, data) => {
7     ...res.end("Hello from server again ")
8     ...})
9
10  })
11 myServer.listen(8000, () => { console.log("server started at 8000") })
```

```
log.txt
1 1710669865584;New Req Received
2 1710669865709;New Req Received
3
```

- Two times , one because of the favicon
- So now we actually have log of the user, that which time we have requests, we can also print ip address of the user on each request
- You also have req.url , which will tell you on which route user is going
- So from the below code you have done two things, firstly you logged req received from which route in log.txt and second you are sending response for particular routes

```
Js index.js > [?] myServer > http.createServer() callback > fs.appendFile("log.txt") callback
1 const http = require("http")
2 const fs = require("fs")
3 const myServer = http.createServer((req, res) => {
4   const log = `${Date.now()};${req.url}New Req Received\n`;
5
6   fs.appendFile("log.txt", log, (err, data) => {
7     switch (req.url) {
8       case "/": res.end("Hello from the Server again");
9       break;
10      case "/about": res.end("You are in about route");
11      break;
12      default: res.end("404 not found");
13    }
14  })
15
16 })
17 myServer.listen(8000, () => { console.log("server started at 8000"); })
```

- Always use non blocking tasks!

Handling url's

17 March 2024 15:46

http: not secure

https: http + secure

- ## URL

Uniform Resource Locator

https://www.piyushgarg.dev/

Protocol: Hypertext Transfer Protocol Secure	Domain - User Friendly Name of IP Address of My Server
--	---

Path: / HomePage or Root Page

- ## Nested PATH

piyushgarg.dev/project/1

- ## Query Parameters

piyushgarg.dev/about?userId=1&a=2

- If you simply search tic tac toe in youtube, this is what actual url becomes

https://www.youtube.com/results?search_query=javascript+tic+tac+toe+

- The server we created in the last lect, in the log.txt favicon route was also coming so to remove that we did this, just we ended that response

```
dex.js > [?] myServer > [?] http.createServer() callback
const http = require("http")
const fs = require("fs")
const myServer = http.createServer((req, res) => {
  if (req.url === '/favicon.ico') return res.end();
  const log = `${Date.now()} ${req.url} New Req Received\n`;
  fs.appendFile("log.txt", log, (err, data) => {
    switch (req.url) {
      case "/": res.end("Hello from the Server again ");
      break;
      case "/about": res.end("You are in about route");
      break;
      default: res.end("404 not found");
    }
  })
})
```



```

    ...})
  })
  myServer.listen(8000, () => { console.log("server started at 8000") })

```

```

1673414623584: / New Req Received
1673414641429: /about New Req Received
• 1673414671286: /about?myname=piyush New Req Received

```

- Now when we entered our search parameters, then we got the info in the log.txt but in the page there were 404 not found, because our http module don't separate or understand that this is query paramter, for the http modules it's just another route
- So for this thing we have another package
- `npm i url`,
- url dependencies is created in the package.json after above command
- Now what you did, you just parsed your url in this function then you got this info about the url

```

const myServer = http.createServer((req, res) => {
  if (req.url === "/favicon.ico") return res.end();
  const log = `${Date.now()}: ${req.url} New Req Received\n`;
  const myUrl = url.parse(req.url);
  console.log(myUrl);
});

```

```

Server Started!
Url {
  protocol: null,
  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?myname=piyush',
  query: 'myname=piyush',
  pathname: '/about',
  path: '/about?myname=piyush',
  href: '/about?myname=piyush'
}

```

- Now you know that this separated query from the actual url, now our module can understand that what is query and what is actual pathname
- Earlier, what was happening is that if we went to 'about' and then added '?' followed by query parameters, it didn't separate the query and thought that it is an invalid path. But now, it knows that it's just a search parameter and the path is still 'about'.
- Now the switch case we are applying it to `myUrl.path` rather `req.url`

```

const myUrl = url.parse(req.url);
console.log(myUrl);
fs.appendFile("log.txt", log, (err, data) => {
  switch (myUrl.pathname) {
    case "/":
      res.end("HomePage");
      break;
    case "/about":
      const qp =
        res.end("I am Piyush Garg");
      break;
    default:
      res.end("404 Not Found");
  }
});

```

- And now it's easy to understand from the following code what you did, added `true` so that it know that we also have to parse query paramters

```

  slashes: null,
  auth: null,
  host: null,
  port: null,
  hostname: null,
  hash: null,
  search: '?myname=piyush&userId=1',
  query: { myname: 'piyush', userId: '1' },
  pathname: '/about',
  path: '/about?myname=piyush&userId=1',
  href: '/about?myname=piyush&userId=1'
}

```

```

const myServer = http.createServer((req, res) => {
  if (req.url === "/favicon.ico") return res.end();
  const log = `${Date.now()}: ${req.url} New Req Received\n`;
  const myUrl = url.parse(req.url, true);
  console.log(myUrl);
  fs.appendFile("log.txt", log, (err, data) => {

```

```

fs.appendFile("log.txt", log, (err, data) => {
  switch (myUrl.pathname) {
    case "/":
      res.end("HomePage");
      break;
    case "/about":
      const username = myUrl.query.myname;
      res.end(`Hi, ${username}`);
      break;
    case "/search":
      const search = myUrl.query.search_query;
      res.end("Here are your results for " + search);
    default:
      res.end("404 Not Found");
  }
});

```

One of the query name , it is like this

```

path: '/about?myname=piyush&userId=1&search=dog',
href: '/about?myname=piyush&userId=1&search=dog'

```

```

const http = require("http")
const fs = require("fs")
const url = require("url")
const myServer = http.createServer((req, res) => {
  if (req.url === '/favicon.ico') return res.end();
  const log = `${Date.now()} ${req.url} New Req Received\n`;
  const myUrl = url.parse(req.url, true)
  console.log(myUrl)
  fs.appendFile("log.txt", log, (err, data) => {
    switch (myUrl.pathname) {
      case "/": res.end("HomePage")
      break;
      case "/about":
        const username = myUrl.query.myname
        res.end(`Hi1 , ${username}`)
        break;
      default: res.end("404 not found")
    }
  })
})
myServer.listen(8000, () => { console.log("server started at 8000") })

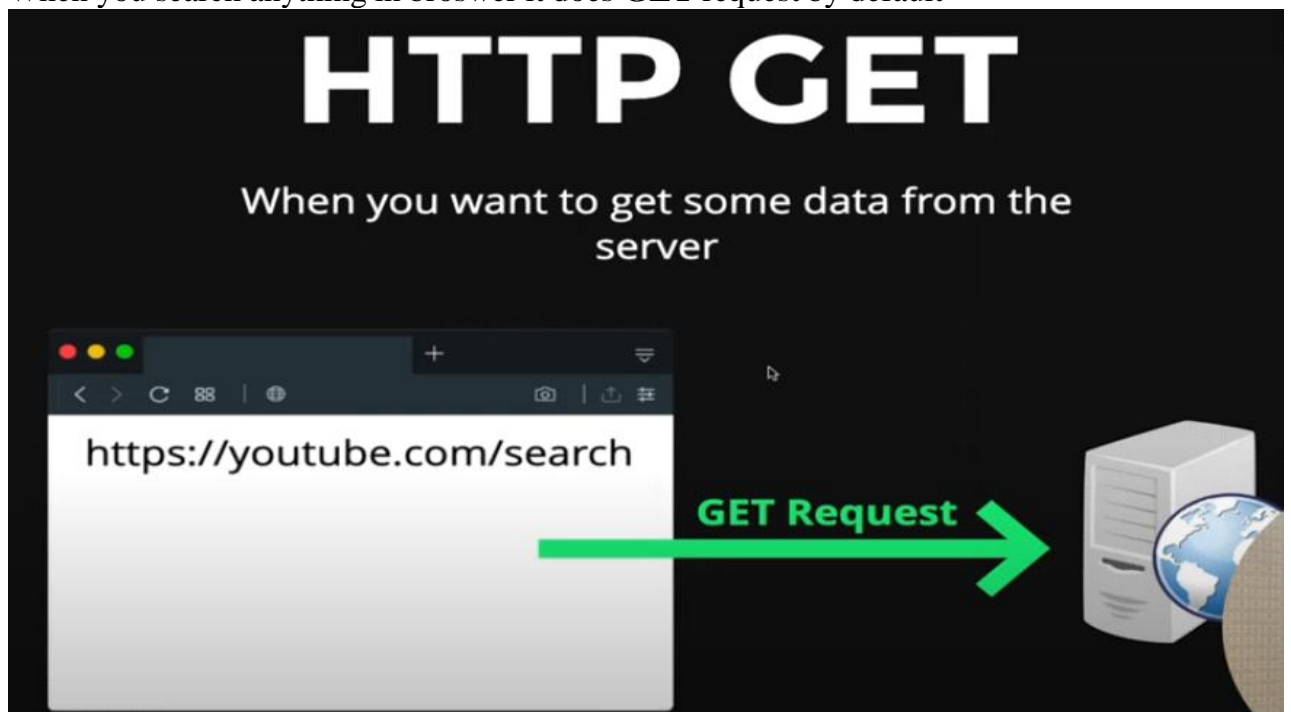
```

HTTP Methods:

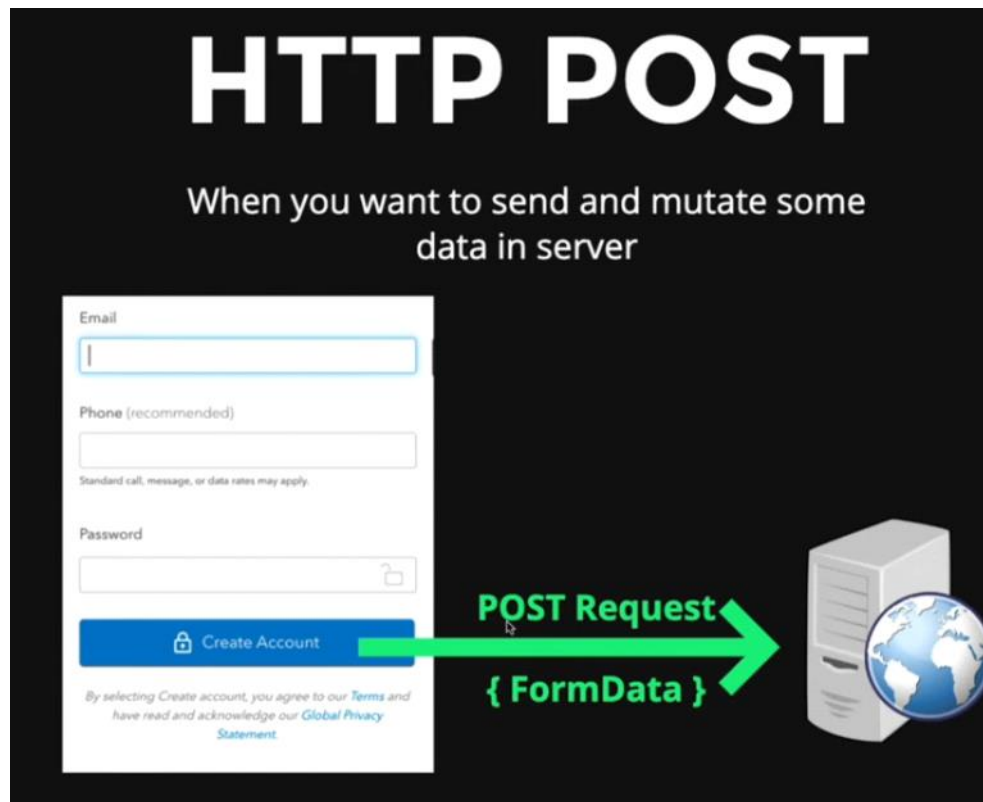
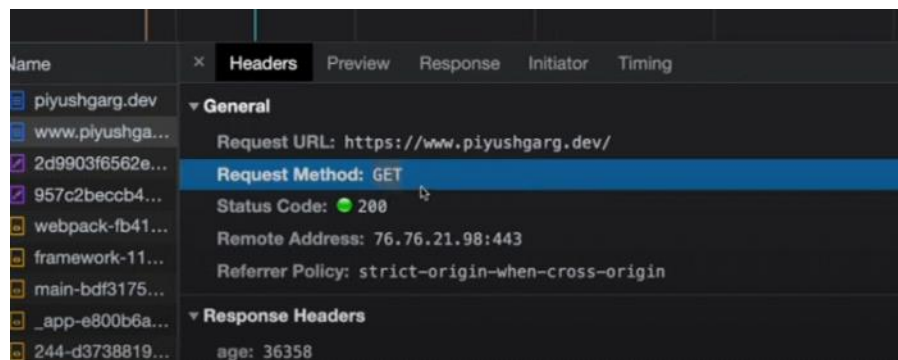
17 March 2024 21:58



- When you search anything in browser it does **GET** request by default



- Search any url on the browser, open network section in dev tools there you can see the
- request method as GET



- When we fill forms in websites then we are using post request at that point
- PUT:generally when you upload image on any website, you can understand it like this, when you put something on the website
- PATCH:when you want to update anything
- DELETE
- In the existing code, you can also find what type of request happened by req.method

```
const myServer = http.createServer((req, res) => {
  if (req.url === "/favicon.ico") return res.end();
  const log = `${Date.now()}: ${req.method} ${req.url} New Req Received\n`
  const myUrl = url.parse(req.url, true);
  console.log(myUrl);
});
```

```
1673500400725: GET / New Req Received
1673500481782: GET / New Req Received
1673500493351: GET /about New Req Received
```

- You can also create a case for signup, where if it is a post request then you can store the things in the database also
- All the ss of code are of the previous lectures
- So what will happen is that , when we will open /signup then default GET req will happen , and if there is a form and we will it and then send it then it will be a post request

```
case "/search":
  const search = myUrl.query.search_query;
```

there is a form and we will it and then send it then it will be a post request

```
case "/search":
  const search = myUrl.query.search_query;
  res.end("Here are your results for " + search);
case "/signup":
  if (req.method === "GET") res.end("This is a signup Form");
  else if (req.method === "POST") {
    // DB Query
    res.end("Success");
  }
}
```

- Now the code above you can see is not structured properly, there are lot of if else statements , so for this thing we have express, which we will study in the upcoming lectures

Express

17 March 2024 22:44

- Summary of the code:

We made the server from the help of http module, and inside that server we have a callback function which handles the whole server

```
const http = require("http");
const fs = require("fs");
const url = require("url");

const myServer = http.createServer((req, res) => {
  if (req.url === "/favicon.ico") return res.end();
  const log = `${Date.now()}: ${req.method} ${req.url} New Req Received\n`;
  const myUrl = url.parse(req.url, true);

  fs.appendFile("log.txt", log, (err, data) => {
    switch (myUrl.pathname) {
      case "/":
        if (req.method === "GET") res.end("HomePage");
        break;
      case "/about":
        const username = myUrl.query.myname;
        res.end(`Hi, ${username}`);
        break;
      case "/search":
        const search = myUrl.query.search_query;
        res.end("Here are your results for " + search);
      case "/signup":
        if (req.method === "GET") res.end("This is a signup Form");
        else if (req.method === "POST") {
          // DB Query
          res.end("Success");
        }
      default:
```

- In this code we have to make different cases for different routes as well as different cases for different methods & we have to use different different packages for url handling also, which is kind of a mess
- In below code we just made another function and just put all the logic of the server inside that function


```

const http = require("http")
const fs = require("fs")
const url = require("url")
function myHandler(req, res) {
  if (req.url === '/favicon.ico') return res.end();
  const log = `${Date.now()} ${req.url} New Req Received\n`;
  const myUrl = url.parse(req.url, true)
  console.log(myUrl)
  fs.appendFile("log.txt", log, (err, data) => {

    switch (myUrl.pathname) {
      case "/": res.end("HomePage")
      break;
      case "/about":
        const username = myUrl.query.myname
        res.end(`Hi!, ${username}`)
        break;
      default: res.end("404 not found")
    }
  })
}

const myServer = http.createServer(myHandler)
myServer.listen(8000, () => { console.log("server started at 8000") })

```

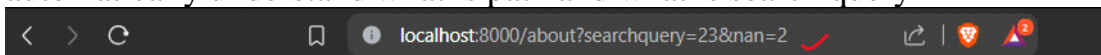
- Command: npm I express
- App basically here is a handler function
- See the code carefully (how clean the code became)

```

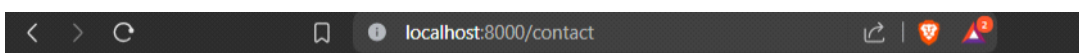
const http = require("http")
const express = require("express")
const app = express();
app.get('/', (req, res) => {
  return res.send("Hello from Home Page")
})
app.get('/about', (req, res) => {
  return res.send("Hello from About Page")
})
const myServer = http.createServer(app)
myServer.listen(8000, () => { console.log("server started at 8000") })

```

- Now if you go to /about or / you will see the responses, also if you have a search query with the route then you don't have to use any other package to handle that, it will automatically understand what is path and what is search query

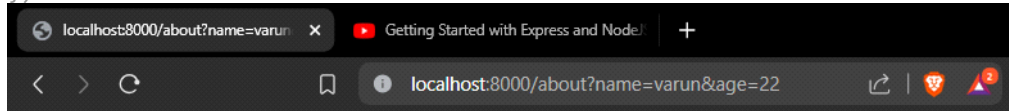


- Hello from About Page
- When you were using the previous code, you also have to handle the case of the route which is not listed, but it will show this by default



- Cannot GET /contact
- Now see, many things are already built in , in the express Just added this code

```
app.get('/about', (req, res) => {
  return res.send("Hello from About Page" + "" + req.query.name + " " + req.query.age)
})
```



- Now if we are using express, then we don't need http, so we can remove that remaining code of http

```
const express = require("express")
const app = express();
app.get('/', (req, res) => {
  return res.send("Hello from Home Page")
})
app.get('/about', (req, res) => {
  return res.send(`Hello ${req.query.name}`)
})
app.listen(8000, ()=>console.log("server started at 8000"))
```

- Express is just a framework, internally http is used inside express

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

Each route can have one or more handler functions, which are executed when the route is matched.

Route definition takes the following structure:

```
app.METHOD(PATH, HANDLER)
```

-

Where:

- app is an instance of express.
- METHOD is an [HTTP request method](#), in lowercase.
- PATH is a path on the server.
- HANDLER is the function executed when the route is matched.

- You can also uninstall the dependencies which you don't need
Command: `npm uninstall url`

Versioning NodeJS

18 March 2024 14:27

- Suppose there is any dependency installed in your app, and it has a version

```
"dependencies": {  
  "express": "^4.18.2"  
}
```

```
^4.18.2 | 4.18.2 -> < 5.0.0 |  
4.17.9  
4.18.1  
4.18.2  
4.18.3  
4.18.4
```

- This symbol indicate that , you can't change fourth, if you are updating the dependencies , you can change the second part and third part according to you, but you can't change fourth

- ^ - Install all Recommended and Minor Fixes Automatically

- You should not change the first part of the version, in the existing code, otherwise your code will break, because that is the major release
- You should be very cautious if you are updating the first part

```
// Version  
  
//  
4.18.3  
  
1st Part -> 4  
2nd Part -> 18  
3rd Part -> 2  
  
// 3rd Part (Last Part) - Minor Fixes (Optional)  
Latest -> 4.18.5
```

```
// 3rd Part (Last Part) - Minor Fixes (Optional)  
Latest -> 4.18.5  
  
// 2nd Part - Recommended Bug Fix (Security Fix)  
Latest -> 4.19.1  
  
// 3rd Major Release - Major / Breaking U  
5
```

Breaking update

- If you want to install any particular version: ***npm i express@4.17.2***

```
~4.18.1
~4.18.2
~4.18.3
~4.18.4
```

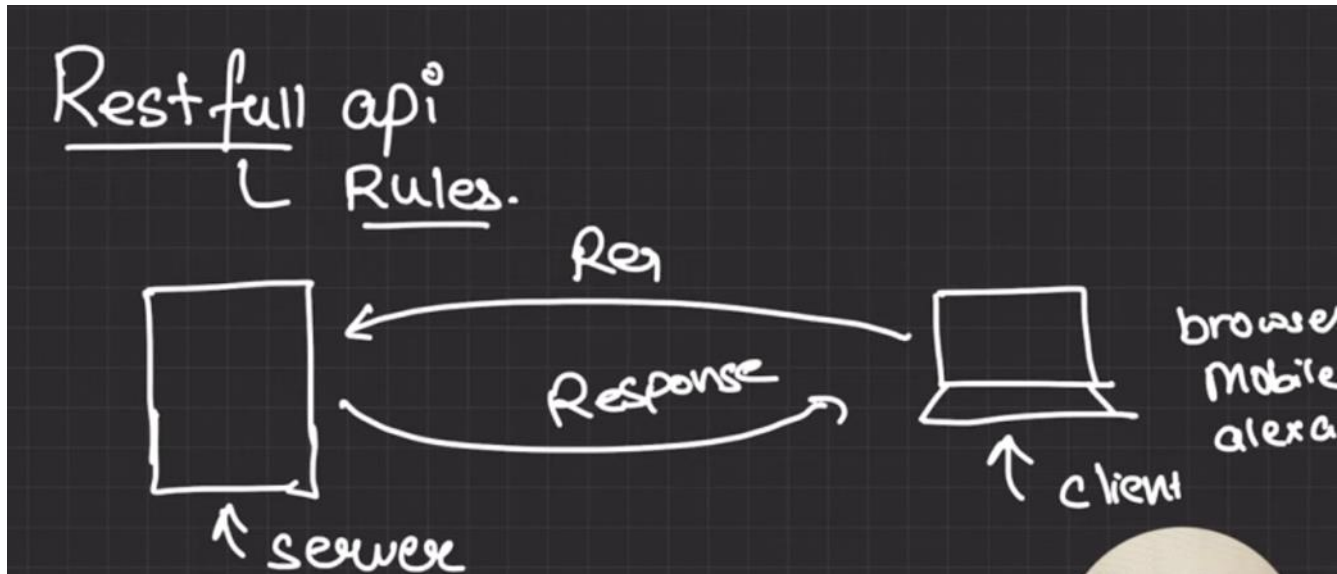
- This is for the middle version, this sign will let the last part change only
- Like this you can write the version you want to write

```
{
  "dependencies": {
    "foo": "1.0.0 - 2.9999.9999",
    "bar": ">=1.0.2 <2.1.2",
    "baz": ">1.0.2 <=2.3.4",
    "boo": "2.0.1",
    "qux": "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0",
    "asd": "http://asdf.com/asdf.tar.gz",
    "til": "~1.2",
    "elf": "~1.2.3",
    "two": "2.x",
    "thr": "3.3.x",
    "lat": "latest",
    "dyl": "file:../dyl"
  }
}
```

REST API

18 March 2024 15:01

- REST or RESTFUL API
- There are some rules which the server and client should practice so that we can say, our server is made up of restful Api's



1. Server client architecture:
 - Server is a different entity and client too, and they should not be dependent on each other
 - You sent a requ, suppose yuou want some blogs, and the response can come in diffrenet formats, etext, images, html document, json etc
 - If we did a get request, so server fetched the blogs from the database and we converted the data into html document, and it is sent as a response
 - If your client is a browser then no issue html will render easitly on the sfreen, but in case of other things we cant' render our html document
 - Now if response is in the form of json then the client doesn't have any problem reading it and it can be easily rendered even at tge mobile screen so we as a client can **independently** process the data
 - So if you know that the client is a browser then html is very fast to render, but if you knowt that you are working on a cross platform app, so json is the best alternative
2. Always respect all HTTP methods
 - jiska kaam ussi ko saaje, dooja kare toh.....

2. Always respect all http methods

GET POST PUT Patch DELETE

GET /user — users data read kro and return data

POST /user
└ handle new user creation

Patch /user — update the user

✓ POST / update User → user update

✓ POST / create user → create

• `res.send()` `res.json()`

- You can send your key value pair in `res.json` above
- SSR: one step less than csr, because as the data comes it is rendered
- CSR: first data is fetched and then rendered (done by react)

Building REST API'S

18 March 2024 15:40

- ```
scripts": {
 "start": "node index.js"
```

- Change script to this, so that you can run it by npm start
- So what we will be doing here is that , we are making a rest api in which we will deal with json data, and all the things are written below, like when we will do get request on /user route then it will list all user

- ```
REST API - JSON  
  
GET /users - List all users  
  
GET /users/1 - Get the user with ID 1  
GET /users/2 - Get the user with ID 2  
  
POST /users - Create new user  
  
PATCH /users/1 - Edit the user with ID 1  
  
DELETE /users/1 - Delete the User with ID 1
```

- So right now we don't have any db, so we will use the hardcoded data
- We used mokaroo.com for our fake data
- Code to simply list all the users

```
const express = require("express")  
const app = express();  
const user = require("./MOCK_DATA.json")  
const port = 8000;  
//Routes  
app.get("/users", (req, res) => {  
  return res.json(user)  
})  
app.listen(port, () => console.log(`server started at ${port}`))
```

- Now we are doing a change basically, so when it's /user then we can directly open html document there but otherwise suppose if we are using the api in any other device then we will return json only

- ```
REST API - JSON

GET /users - HTML Docuemnt Render
GET /api/users - List all users JSON - Done

GET /api/users/1 - Get the user with ID 1
GET /api/users/2 - Get the user with ID 2

POST /api/users - Create new user
```

- So in the route if we have api, then we will give json response otherwise html
- We are actually making a hybrid server which is a good practice
- Now for the /users route

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

- Now for the /users route

```
const express = require("express")
const app = express();
const users = require("./MOCK_DATA.json")
const port = 8000;
//Routes
app.get("/users", (req, res) => {
 const html = `

${users.map(user => `- ${user.first_name}`).join("")}`
 res.send(html)
})
app.get("/api/users", (req, res) => {
 return res.json(users)
})
app.listen(port, () => console.log(`server started at ${port}`))

```

- So now if you will go to /user , then it will be a html document
- We need /users/id dynamically right,

```
Dynamic Path Parameters
GET /api/users/:id
:id -> Variable | Dynamic
```

```
app.get("/api/users/:id", (req, res) => {
 //you extracted the id from the url
 // const id = req.params.id -> this will give us a string
 const id = Number(req.params.id)
 //now you have to search for that user whose id is in the params, and for each
 user we are searching for the id
 const user = users.find((user) => user.id === id);
 return res.json(user)
})
```

#### Example 1: Finding a Number Greater Than 10

```
javascript Copy code
const numbers = [5, 12, 8, 25, 3];
const found = numbers.find(element => element > 10);
console.log(found); // Output: 12
```

#### Example 2: Finding an Object by Property

```
javascript Copy code
const users = [
 { id: 1, name: 'John' },
 { id: 2, name: 'Alice' },
 { id: 3, name: 'Bob' }
];

const user = users.find(user => user.id === 2);
console.log(user); // Output: { id: 2, name: 'Alice' }
```

- You will get a particular user detail like this

```

1 // 20230117100136
2 // http://localhost:8000/api/users/4 ⓘ
3
4 {
5 "id": 4,
6 "first_name": "Pedro",
7 "last_name": "Downer",
8 "email": "pdowner3@fema.gov",
9 "gender": "Male",
10 "job_title": "Assistant Manager"
11 }

```

- If we have to use same route for different methods we can merge them in one also

```

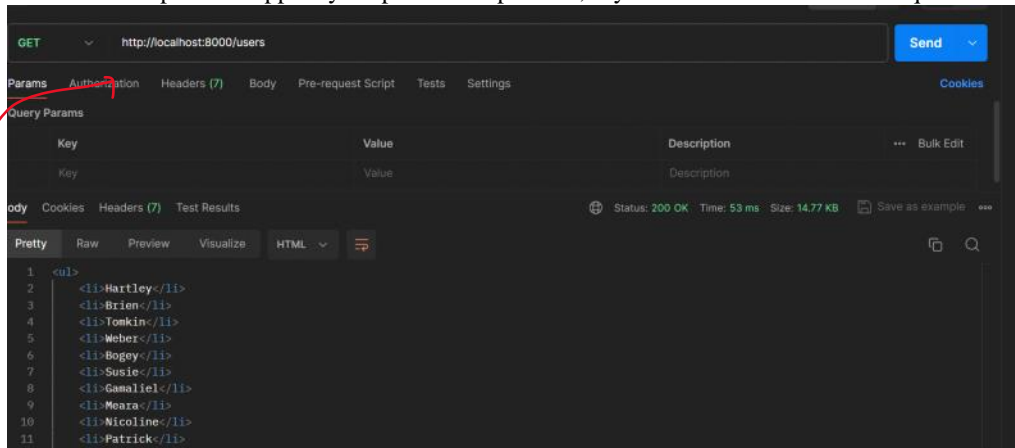
app
 .route("/api/users/:id")
 .get((req, res) => {
 const id = Number(req.params.id)
 const user = users.find((user) => user.id === id);
 return res.json(user)
 })
 .patch((req, res) => {
 //Edit user with id
 return res.json({ status: "Pending" })
 })
 .delete((req, res) => {
 //Delete user with id
 return res.json({ status: "Pending" })
 })
app.post("/api/users", (req, res) => {
 //TODO: create a new user
 return res.json({ status: "Pending" })
})

```

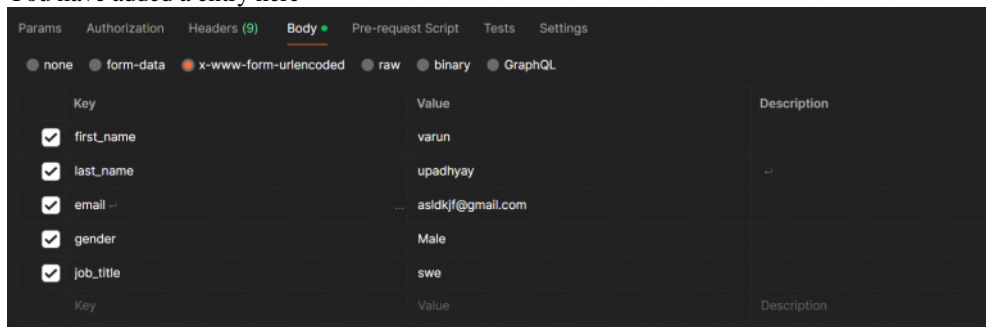
# Postman for REST

19 March 2024 10:17

- We are going to test our api routes in postman
- Just download postman app on your pc or dextop client , if you are unable to send the request



- Https doesn't work in localhost because we don't have ssl certificate
- Now as we know that we have made a post request and we want to send some data onto it , so we will go to body in postman and there we can have the data in form of key value pair, that is JSON data
- You have added a entry here



- We want to save this data to that file, so we have to append it, and we can do that by fs(file handling)
- when we send something from the frontend , express automatically takes it in the body

```
app.post("/api/users", (req, res) => {
 const body = req.body;
 console.log("Body", body);
 return res.json({ status: "pending" });
});
```

- So if you send the data on this route as a post request and write this code then you will get undefined in the console, because express doesn't know that which type of data it is ,so we have to use a middleware or plugin
- We will study middleware later , right now just write the plugin

```
app.post("/api/users", (req, res) => {
 const body = req.body;
 console.log("Body", body);
 //TODO: create a new user
 return res.json({ status: "Pending" })
})
```

- Everything is same, I have just added a middleware

```

Js index.js > ...
1 const express = require("express")
2 const app = express();
3 const users = require("../MOCK_DATA.json")
4 const port = 8000;
5
6 //Middleware - Plugin
7 app.use(express.urlencoded({extended: false}))
8 //Routes
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```

PS C:\Desktop\node-practice\rest_api> npm start

> rest_api@1.0.0 start
> node index.js

server started at 8000
Body [Object: null prototype] {
 first_name: 'varun',
 last_name: 'upadhyay',
 'email\n': 'asldkjf@gmail.com',
 gender: 'Male',
 job_title: 'swe'
}

```

- Now using our body we can add this data into our mockdata.json
- Updated code with patch and delete

```

const express = require("express")
const fs = require("fs")
const app = express();
const users = require("../MOCK_DATA.json");
const { stringify } = require("querystring");
const port = 8000;
//Middleware - Plugin
app.use(express.urlencoded({ extended: false }));
//Routes
app.get("/users", (req, res) => {
 const html = `${users.map(user => `${user.first_name}`).join("")}`;
 res.send(html)
})
app.get("/api/users", (req, res) => {
 return res.json(users)
})
app
 .route("/api/users/:id")
 .get((req, res) => {
 const id = Number(req.params.id)
 const user = users.find((user) => user.id === id);
 return res.json(user)
 })
 .patch((req, res) => {
 // getId stores the Id from the given Parameters in the URL.
 const getId = Number(req.params.id);
 // body stores the body in which we've to make changes.
 const body = req.body;
 // Finding the user Id from the user array.
 const userIndex = users.findIndex((user) => user.id === getId);
 // If we found a user with its Id then gotUser stores that object.
 const gotUser = users[userIndex];
 // Here gotUser has the user Object and body has the changes we have to made.
 const updatedUser = { ...gotUser, ...body };
 // After Merging them, Update the users Array.
 users[userIndex] = updatedUser;
 fs.writeFile("../MOCK_DATA.json", JSON.stringify(users), (err, data) => {
 return res.json({ status: "Success" }, updatedUser)
 })
 })
 .delete((req, res) => {
 //Delete user with id
 const id = Number(req.params.id)
 // Find the index of the user to delete
 const userIndex = users.findIndex((user) => user.id === id);
 if (userIndex !== -1) { // If the user with given ID exists
 const deletedUser = users.splice(userIndex, 1)
 //now deleted user is the updated list of the users after deleting our user
 // Write the updated users array back to the file
 fs.writeFile("../MOCK_DATA.json", JSON.stringify(users), (err, data) => {
 if (err) {
 console.error(err);
 return res.status(500).json({ status: "Error", message: "Failed to delete user" });
 }
 })
 }
 })

```

```

 if (err) {
 console.error(err);
 return res.status(500).json({ status: "Error", message: "Failed to delete user" });
 }
 console.log("User deleted successfully");
 return res.json({ status: `Deleted the user with ${id}`, deletedUser });
 })
} else {
 return res.status(404).json({ status: "Error", message: `User with ID ${id} not found` });
}
})
app.post("/api/users", (req, res) => {
 const body = req.body;
 // users.push(body) - in starting we were not giving id so we were writing like this, but now we are also giving id, so we will
 right
 //Like this below
 users.push({ ...body, id: (users.length + 1) })
 fs.writeFile("./MOCK_DATA.json", JSON.stringify(users), (err, data) => {
 return res.json({ status: "Success", id: users.length })
 //if the user is already in the data with the id, then you don't need to do + 1 again
 })
})
app.listen(port, () => console.log(`server started at ${port}`))

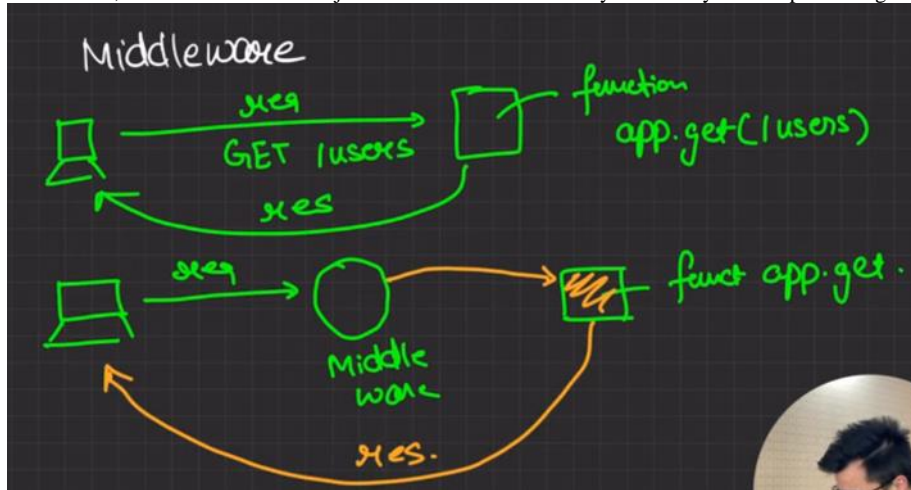
```



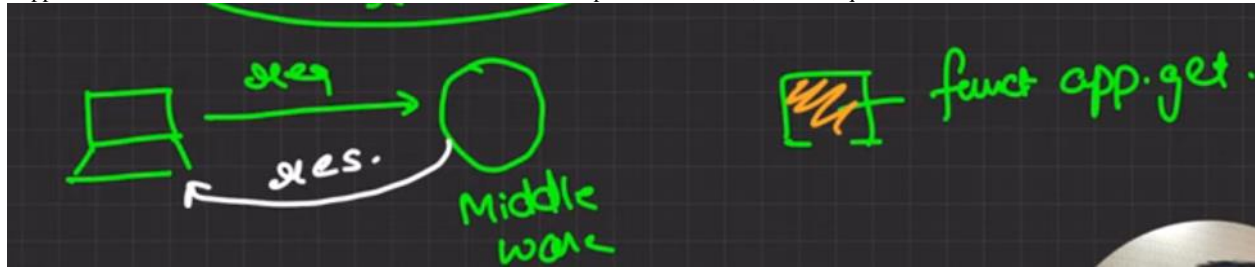
# Express Middleware

19 March 2024 17:05

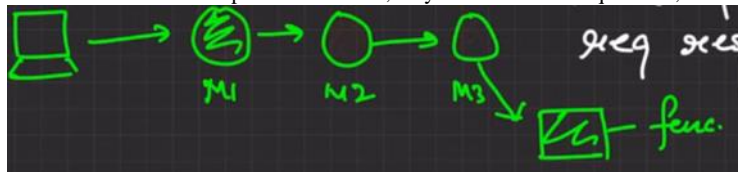
- In first part we have the req, which reach to client and then that client return a response directly,
- In the second part we have a middleware in between client and the user, so what that request doesn't reach to client , it will first reach middleware, and that middleware is just a function which basically can do any kind of processing on that request



- Like we can verify user, check that whether request is secure or not
- Suppose the middleware didn't validated , so it will return the response from there , means request will not reach to the client



- One code can have multiple middlewares , they have access to req and res, each middleware have their own functions



Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

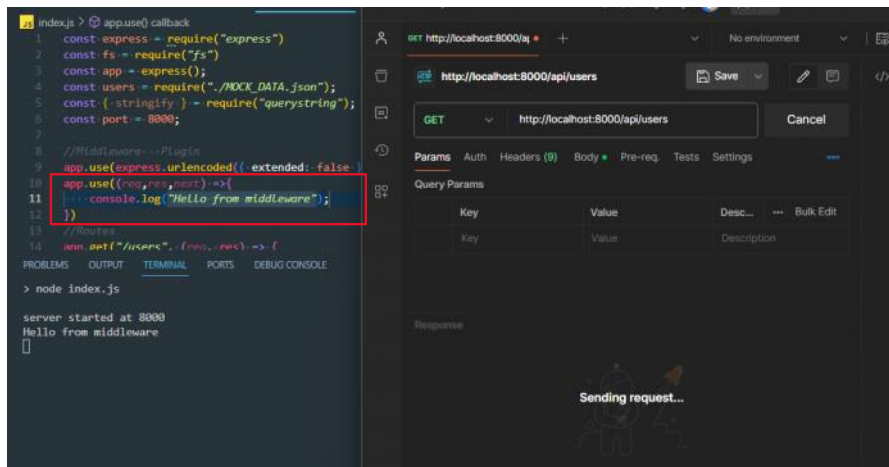
**Middleware** functions are functions that have access to the [request object](#) (`req`), the [response object](#) (`res`), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named `next`.

- Middleware functions can perform the following tasks:
  - Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware function in the stack.
- If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.
- Middleware runs orderwise, first above will run then second, and then the next

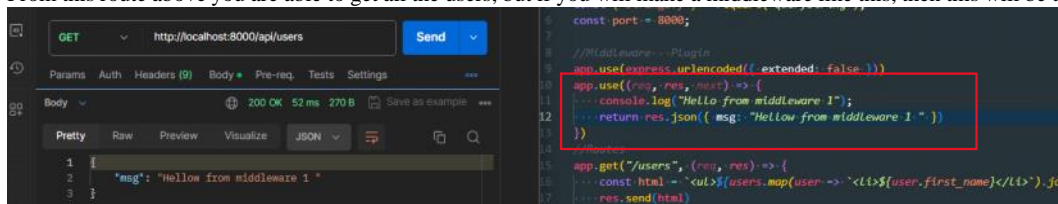
```
// Middleware - Plugin
app.use(express.urlencoded({ extended: false }));

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

- When you wrote this middleware, what happened is that you didn't send any response or you didn't use `next` , so you just left the Request hanging there



- So as a middleware you have that power, you can just send a response from there and the flow of code won't go down
- From this route above you are able to get all the users, but if you will make a middleware like this, then this will be the response



- Now what you did, you just added an next, it means you can go to next thing (middleware or route) after this middleware

```
app.use((req, res, next) => {
 console.log("Hello from middleware 1");
 next();
})
```

- Now we made another middleware, and in first middleware we added something in the req, and we accessed that thing in the Second middleware or anywhere else below that middleware, even in the routes too

```
app.use((req, res, next) => {
 req.myUserName = "varun"
 console.log("Hello from middleware 1");
 next();
})
app.use((req, res, next) => {
 console.log("This is middleware 2", req.myUserName)
 next()
})
```

```
server started at 8000
Hello from middleware 1
This is middleware 2 varun
```

Middleware functions can perform the following tasks:

- Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware function in the stack.
- Cdoe at this point of time

```
const express = require("express")
const fs = require("fs")
const app = express();
const users = require("./MOCK_DATA.json");
const { stringify } = require("querystring");
const port = 8000;
//Middleware - Plugin
app.use(express.urlencoded({ extended: false }))
app.use((req, res, next) => {
 req.myUserName = "varun"
 console.log("Hello from middleware 1");
 next();
})
app.use((req, res, next) => {
 console.log("This is middleware 2", req.myUserName)
 next()
})
//Routes
app.get("/users", (req, res) => {
 const html = `${users.map(user => `${user.first_name}`).join("")}`
 res.send(html)
})
app.get("/api/users", (req, res) => {
 return res.json(users)
})
```

```

app
 .route("/api/users/:id")
 .get((req, res) => {
 const id = Number(req.params.id)
 const user = users.find((user) => user.id === id);
 return res.json(user)
 })
 .patch((req, res) => {
 const getId = Number(req.params.id);
 const body = req.body;
 const userIndex = users.findIndex((user) => user.id === getId);
 const gotUser = users[userIndex];
 const updatedUser = { ...gotUser, ...body };
 users[userIndex] = updatedUser;
 fs.writeFile("./MOCK_DATA.json", JSON.stringify(users), (err, data) => {
 return res.json({ status: "Success" }, updatedUser)
 })
 })

 .delete((req, res) => {
 //Delete user with id
 const id = Number(req.params.id)
 // Find the index of the user to delete
 const userIndex = users.findIndex((user) => user.id === id);
 if (userIndex !== -1) { // If the user with given ID exists
 const deletedUser = users.splice(userIndex, 1)
 //now deleted user is the updated List of the users after deleting our user
 // Write the updated users array back to the file
 fs.writeFile("./MOCK_DATA.json", JSON.stringify(users), (err, data) => {
 if (err) {
 console.error(err);
 return res.status(500).json({ status: "Error", message: "Failed to delete user" });
 }
 console.log("User deleted successfully");
 return res.json({ status: `Deleted the user with ${id}`, deletedUser });
 })
 } else {
 return res.status(404).json({ status: "Error", message: `User with ID ${id} not found` });
 }
 })
})
app.post("/api/users", (req, res) => {
 const body = req.body;
 // users.push(body) - in starting we were not giving id so we were writing like this, but now we are also giving id, so we will
 //Like this below
 users.push({ ...body, id: (users.length + 1) })
 fs.writeFile("./MOCK_DATA.json", JSON.stringify(users), (err, data) => {
 return res.json({ status: "Success", id: users.length })
 })
 //if the user is already in the data with the id, then you don't need to do + 1 again
})

app.listen(port, () => console.log(`server started at ${port}`))

```

- Below code shows you the use of middleware, how you can make a middleware yourself and what info it can Get you
- Both are different usecase of next, see in the code

```

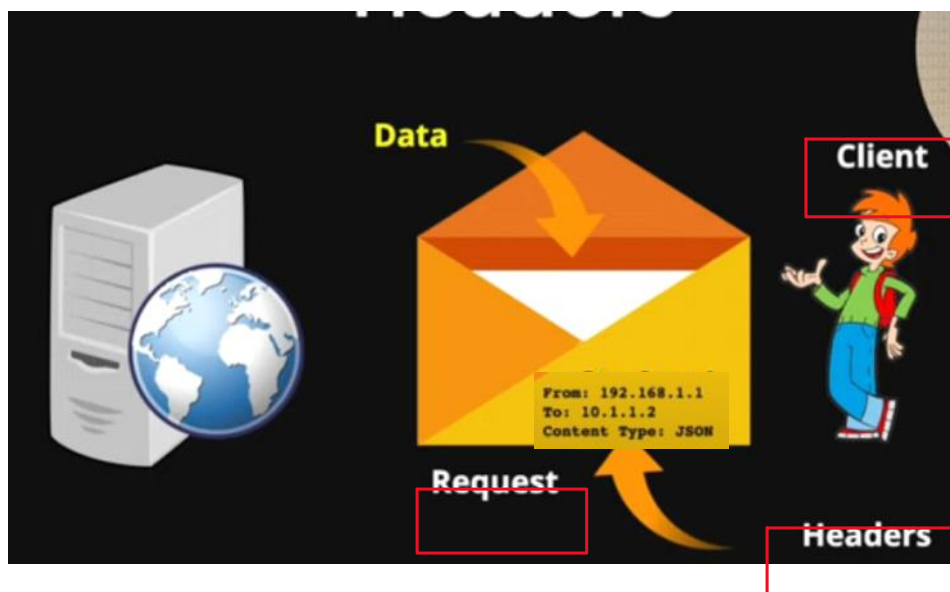
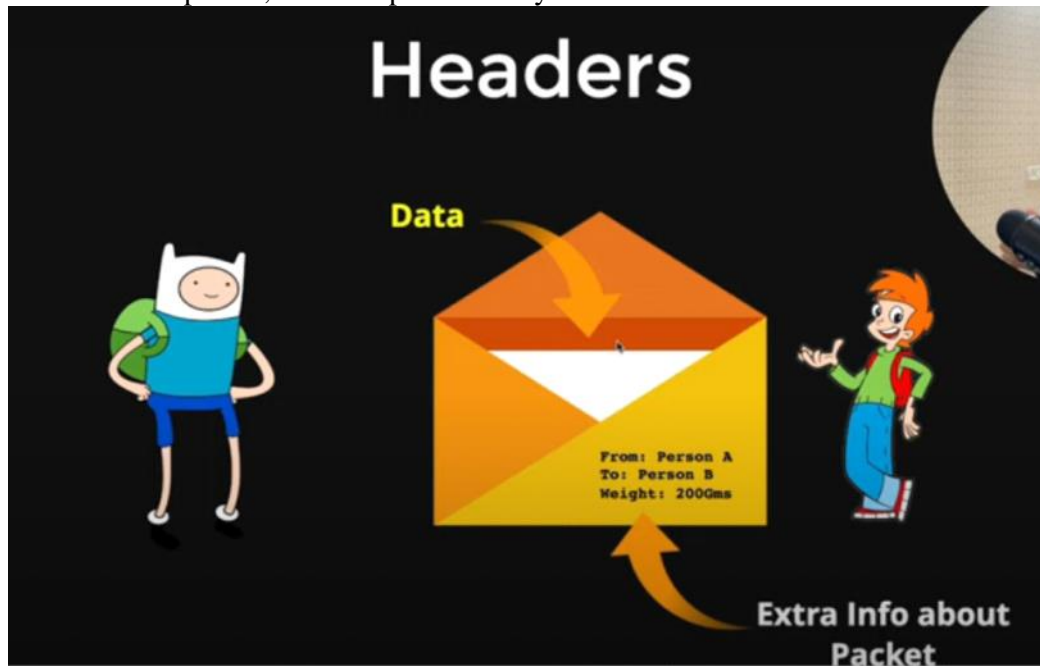
app.use((req, res, next) => {
 //below statement indicate that when this date thing is appended in Log.txt
 //then callback function will be called and which is just next()
 fs.appendFile("Log.txt", `${Date.now()} ${req.method} ${req.path} ${req.ip}\n`, (err, data) => {
 next()
 })
 // next();
})

```

# HTTP Headers

20 March 2024 00:30

- You sent a letter from one person to another person, and it has some data, and on the outside it has some extra info about the packet, now compare it with your actual client server architecture

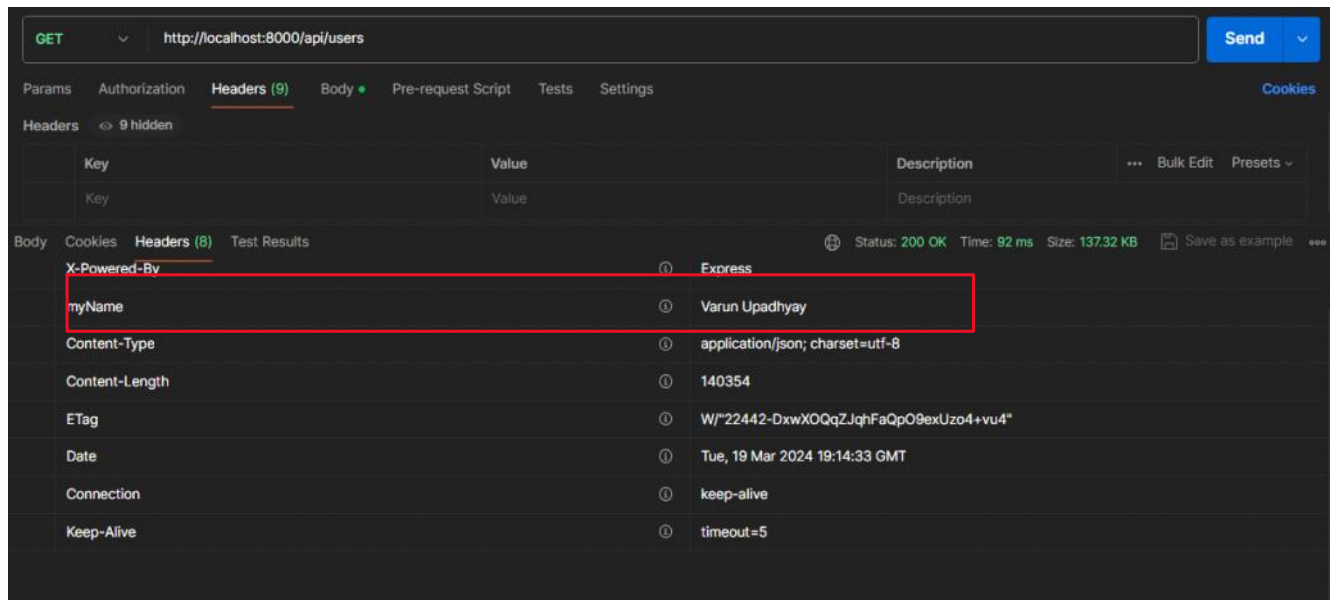


**HTTP Headers are an important part of the API request and response as they represent the meta-data associated with the API request and response.**

**Headers carry information for The request and Response Body.**

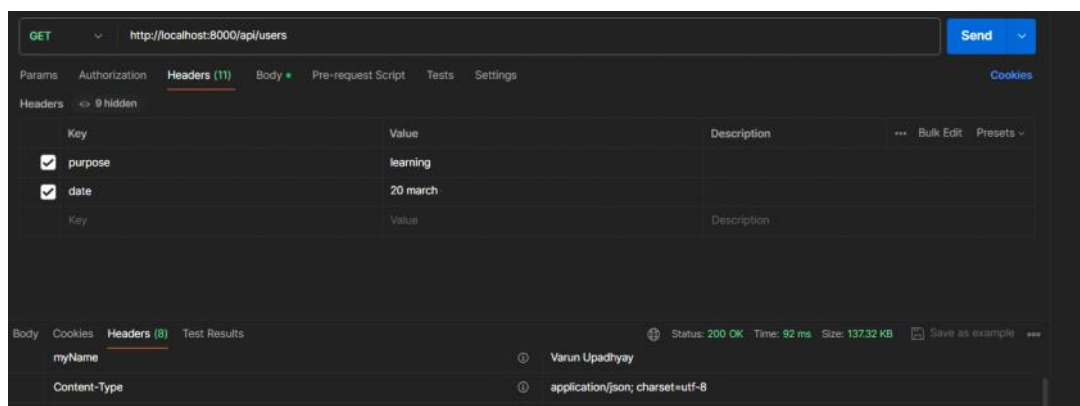
An HTTP header is a field of an HTTP request or response that passes additional context and metadata about the request or response. For example, a request message can use headers to indicate its preferred media formats, while a response can use header to indicate the media format of the returned body. 6 days ago

- You can see headers in the header section in the dev tools, you can see two type of header there , req header and res header
  - In postman too, there are response header and request header you can explore that yourself
  - So these headers are built in headers, but you can also make your own headers
- In one of the route I just added a header



```
app.get("/api/users", (req, res) => {
 res.setHeader("myName", "Varun Upadhyay")
 return res.json(users)
})
```

- Now this is the response header, you can also add something in your header in the req header
- Add like this



```
app.get("/api/users", (req, res) => {
 console.log(req.headers)
 return res.json(users)
})
```

- Console:

- ```
{
  purpose: 'learning',
  date: '20 march',
  'user-agent': 'PostmanRuntime/7.37.0',
  accept: '*/*',
  'cache-control': 'no-cache',
  'postman-token': '7ba7a7f2-c2df-4a09-a08d-47346a6fb13d',
  host: 'localhost:8000',
  'accept-encoding': 'gzip, deflate, br',
  connection: 'keep-alive',
  'content-type': 'application/x-www-form-urlencoded',
  'content-length': '93'
}
```
- Good practices:
Always add "X" to custom headers

```
app.get("/api/users", (req, res) => {
  res.setHeader("X-MyName", "Piyush Garg"); // C
  // Always add X to custom headers
  return res.json(users);
});
```


Status codes

20 March 2024 00:57

• <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

- Study each status codes
- It is generally good practice to send status codes associated with each tasks
- So 201 is sent when we send a post or put request or we create something
- Initially it was like this

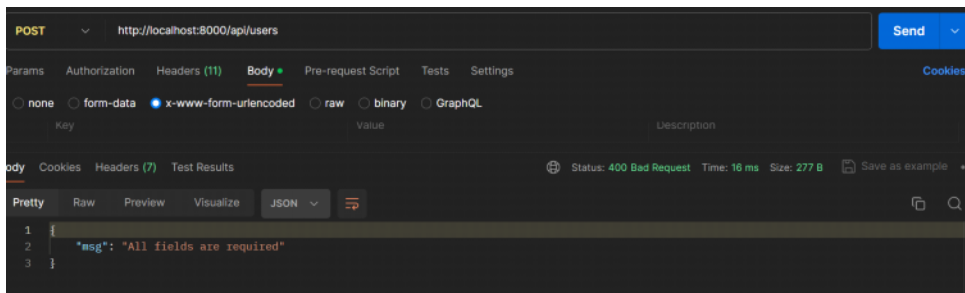
```
app.post("/api/users", (req, res) => {
  const body = req.body;
  users.push({ ...body, id: users.length + 1 });
  fs.writeFile("./MOCK_DATA.json", JSON.stringify(users), (err, data) => {
    return res.json({ status: "success", id: users.length });
  });
});
```

But we added an status to it

```
return res.status(201).json({ status: "success", id: users.length });
```

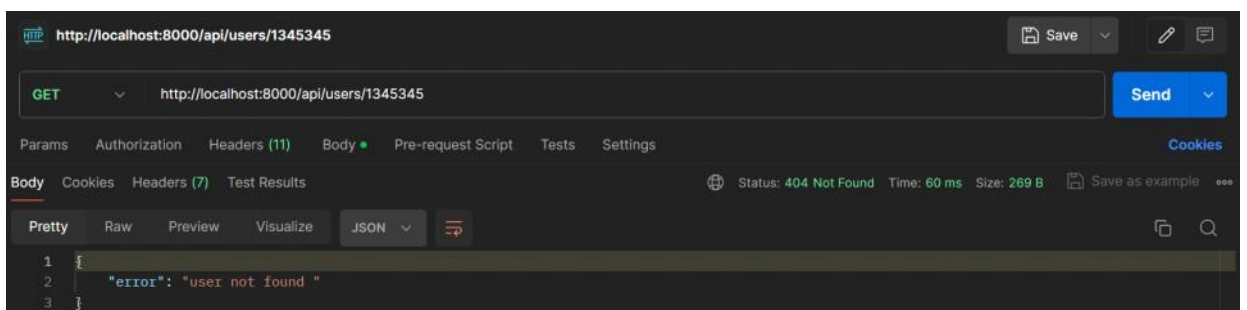
- We have to start our server each and every time , so we will use nodemon which will automatically start our server
- Mostly 200 and 400 is used
-

```
app.post("/api/users", (req, res) => {
  const body = req.body;
  //add more details
  if (!body || !body.first_name || !body.email)
    return res.status(400).json({ msg: "All fields are required" });
  users.push({ ...body, id: (users.length + 1) });
  fs.writeFile("./MOCK_DATA.json", JSON.stringify(users), (err, data) => {
    return res.status(201).json({ status: "Success", id: users.length });
  })
});
```



- Study about general status codes, then try to use it in your code

```
app
  .route("/api/users/:id")
  .get((req, res) => {
    const id = Number(req.params.id)
    const user = users.find((user) => user.id === id);
    if (!user) {
      return res.status(404).json({ error: "user not found " });
    }
    return res.json(user)
  })
```





- As in sql we have tables, in mongo db we have collections
- In cmd I wrote some commands and these are their results below

```
C:\Users\varun>mongosh
```

```
Current Mongosh Log ID: 65fb1a0dc343b7738e24bb25
```

```
Connecting to: mongodb://127.0.0.1:27017/?
```

```
directConnection=true&serverSelectionTimeoutMS=2000
```

```
&appName=mongosh+1.8.0
```

```
Using MongoDB: 6.0.6
```

```
Using Mongosh: 1.8.0
```

For mongosh info see: <https://docs.mongodb.com/mongodb-shell/>

The server generated these startup warnings when booting

2024-03-02T18:14:36.990+05:30: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted

Warning: Found ~/.mongorc.js, but not ~/.mongoshrc.js. ~/.mongorc.js will not be loaded.

You may want to copy or rename ~/.mongorc.js to ~/.mongoshrc.js.

```

test> show dbs
admin      40.00 KiB
cloud-note 180.00 KiB
config     72.00 KiB
local      92.00 KiB
test       188.00 KiB
varun      80.00 KiB
test> use test
already on db test
test> show collections
het
notes
users
test> db.users.find( { } )
[
  {
    _id: ObjectId("646c75d2e1eada829750f0ed"),
    name: 'varun',
    email: 'hey.u@gmail.com',
    password: '1234',
    timestamp: ISODate("2023-05-23T08:14:10.151Z"),
    __v: 0
  },
  {
    _id: ObjectId("6475a495fb6dbe52114a5dbc"),
    name: 'vun',
    email: 'hey@gmail.com',
    password: '1234',
    timestamp: ISODate("2023-05-30T07:24:05.281Z"),
    __v: 0
  },
  {
    _id: ObjectId("6475b433900429219bd59a5c"),
    name: 'adadasd',
    email: 'heyvarun@gmail.com',
    password: '1234sdf',
    timestamp: ISODate("2023-05-30T08:30:43.277Z"),
    __v: 0
  },
  {
    _id: ObjectId("6475b48666cb677b324357b3"),
    name: 'adadasd1',
    email: 'nahheyvarun@gmail.com',
    password: '1234sdf3',
    timestamp: ISODate("2023-05-30T08:32:06.953Z"),
  }
]

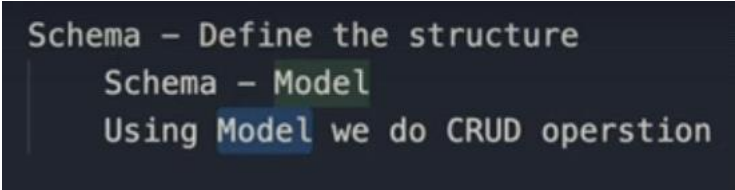
```

```
    __v: 0  
  }  
]
```

NodeJS & MongoDB

20 March 2024 22:50

- We use mongoose package to connect to our MongoDB from our server



```
Schema - Define the structure
Schema - Model
Using Model we do CRUD operation
```

- In mongoose we have a schema ,then we make a model from that schema and using that model we do CRUD operations
- Code upto this point

```
const express = require("express")
const fs = require("fs")
const app = express();
const users = require("./MOCK_DATA.json");
const { stringify } = require("querystring");
const port = 8000;
//Middleware - Plugin
app.use(express.urlencoded({ extended: false }))
app.use((req, res, next) => {
  //below statement indicate that when this date thing is appended in
  log.txt
  //then callback function will be called and which is just next()
  fs.appendFile("log.txt", `${Date.now()} ${req.method} ${req.path}
  ${req.ip}\n`, (err, data) => {
    next()
  })
  // next();
})
// app.use((req,res,next)=>{
//   console.log("This is middleware 2", req.myUserName)
//   next()
// })
//Routes
app.get("/users", (req, res) => {
  const html = `<ul>${users.map(user => `<li>${user.first_name}
</li>`).join("")}</ul>`
  res.send(html)
})
app.get("/api/users", (req, res) => {
  console.log(req.headers)
  return res.json(users)
})
app
  .route("/api/users/:id")
  .get((req, res) => {
    const id = Number(req.params.id)
    const user = users.find((user) => user.id === id);
    if (!user) {
      return res.status(404).json({ error: "user not found " });
    }
    return res.json(user)
  })
  .patch((req, res) => {
    // getId stores the Id from the given Parameters in the URL.
```

```

    const getId = Number(req.params.id);
    // body stores the body in which we've to make changes.
    const body = req.body;
    // Finding the user Id from the user array.
    const userIndex = users.findIndex((user) => user.id === getId);
    // If we found a user with its Id then gotUser stores that
object.
    const gotUser = users[userIndex];
    // Here gotUser has the user Object and body has the changes we
have to made.
    const updatedUser = { ...gotUser, ...body };
    // After Merging them, Update the users Array.
    users[userIndex] = updatedUser;
    fs.writeFile("./MOCK_DATA.json", JSON.stringify(users), (err,
data) => {
        return res.json({ status: "Success" }, updatedUser)
    })

    .delete((req, res) => {
        //Delete user with id
        const id = Number(req.params.id)
        // Find the index of the user to delete
        const userIndex = users.findIndex(user => user.id === id);
        if (userIndex !== -1) { // If the user with given ID exists
            const deletedUser = users.splice(userIndex, 1)
            //now deleted user is the updated list of the users after
deleting our user
            // Write the updated users array back to the file
            fs.writeFile("./MOCK_DATA.json", JSON.stringify(users), (err,
data) => {
                if (err) {
                    console.error(err);
                    return res.status(500).json({ status: "Error",
message: "Failed to delete user" });
                }
                console.log("User deleted successfully");
                return res.json({ status: `Deleted the user with ${id}`,
deletedUser });
            })
        } else {
            return res.status(404).json({ status: "Error", message: `User
with ID ${id} not found` });
        }
    })
app.post("/api/users", (req, res) => {
    const body = req.body;
    //add more details
    if (!body || !body.first_name || !body.email)
        return res.status(400).json({ msg: "All fields are required" })
    // users.push(body) - in starting we were not giving id so we were
writing like this, but now we are also giving id, so we will right
//like this below
    users.push({ ...body, id: (users.length + 1) })
    fs.writeFile("./MOCK_DATA.json", JSON.stringify(users), (err, data)
=> {
        return res.status(201).json({ status: "Success", id:
users.length })
        //if the user is already in the data with the id, then you don't
need to do + 1 again
    })
})

```

```

}))
app.listen(port, () => console.log(`server started at ${port}`))

```

- This means that first name is required and when it is entered then only then entry will be inserted in the db, but if you don't enter last name it's ok

```

//Schema
const userSchema = new mongoose.Schema({
  firstName:{
    type:String,
    required:true
  },
  lastName:{
    type:string
  }
})

```

- Now below code has all crud operations , all the route logic has changed, now we are easily interacting with db

```

const express = require("express")
const fs = require("fs")
const mongoose = require("mongoose");
const app = express();
const { stringify } = require("querystring");
const port = 8000;
//Connection
const nameOfApp = "yt-app"
mongoose.connect(`mongodb://127.0.0.1:27017/${nameOfApp}`)
  .then(() => console.log("MongoDB connected "))
  .catch((err) => { "Mongo Error", err })
//Schema
const userSchema = new mongoose.Schema({
  firstName: {
    type: String,
    required: true
  },
  lastName: {
    type: String
  },
  email: {
    type: String,
    required: true,
    unique: true
  },
  jobTitle: {
    type: String
  },
}, { timestamps: true })
//Model
const User = mongoose.model('user', userSchema)
//Middleware
app.use(express.urlencoded({ extended: false }))
app.use((req, res, next) => {
  fs.appendFile("log.txt", `${Date.now()} ${req.method} ${req.path} ${req.ip}\n`, (err, data) => {
    next()
  })
})
app.get("/users", async (req, res) => {

```



```

    //if in find you left empty that means all the users
    const allDBUsers = await User.find({})
    const html = `

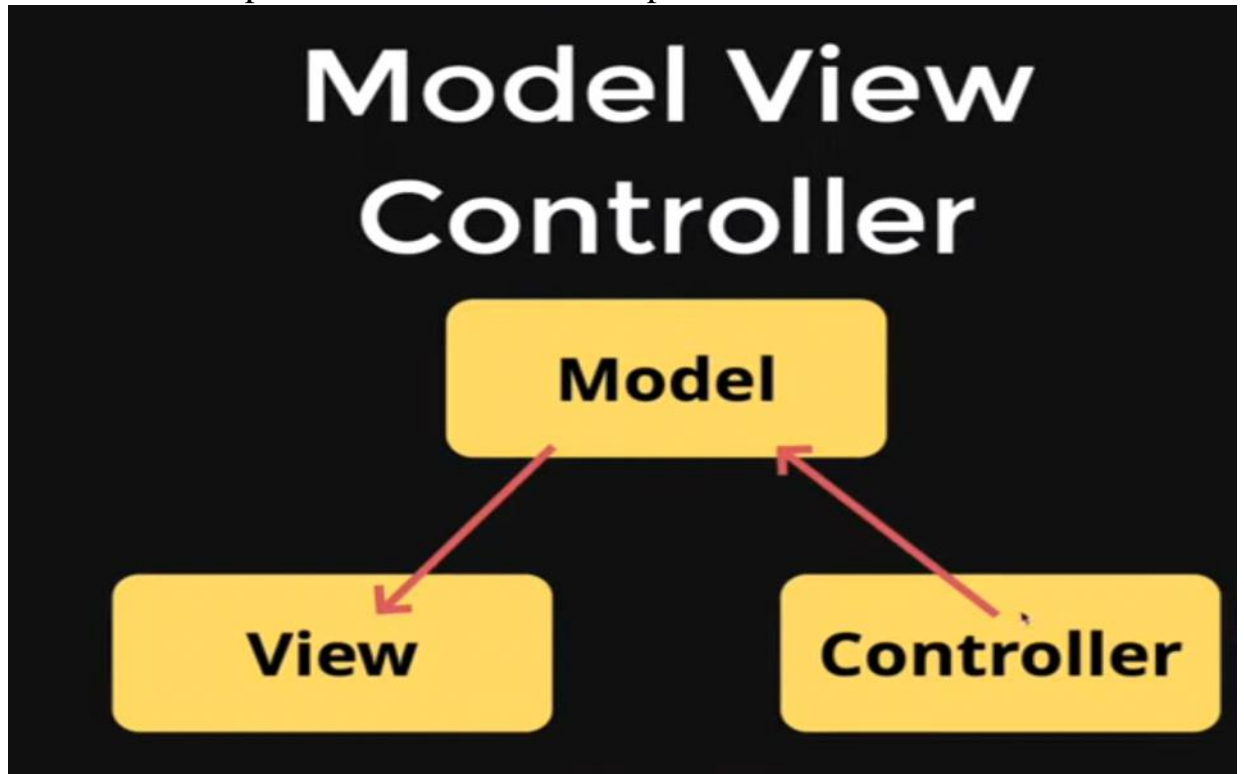
${allDBUsers.map(user => `- ${user.firstName} -
    ${user.email}</li>`).join("")}</ul>`
    res.send(html)
  })
  app.get("/api/users", async (req, res) => {
    const allDBUsers = await User.find({})
    return res.json(allDBUsers)
  })
  app
    .route("/api/users/:id")
    .get(async (req, res) => {
      const user = await User.findById(req.params.id)
      if (!user) {
        return res.status(404).json({ error: "user not found " });
      }
      return res.json(user)
    })
    .patch(async (req, res) => {
      //right now we have changed only last name, but we can get that
      info from the frontend
      await User.findByIdAndUpdate(req.params.id, { lastName:
      "Changed" })
      return res.json({ status: "Success" })
    })
    .delete(async (req, res) => {
      await User.findByIdAndDelete(req.params.id);
      return res.json({ sttus: "Success" })
    })
  app.post("/api/users", async (req, res) => {
    const body = req.body;
    if (!body || !body.first_name || !body.email)
      return res.status(400).json({ msg: "All fields are required" })
    const result = await User.create({
      firstName: body.first_name,
      lastName: body.last_name,
      email: body.email,
      jobTitle: body.job_title
    })
    console.log("result", result);
    return res.status(201).json({ msg: `Success, user created ` })
  })
  app.listen(port, () => console.log(`server started at ${port}`))

```

Model View controller

22 March 2024 11:00

- Controller manipulates model and model updates the view



- The code which we have written here in index.js, we are just segregating that code into a MVC pattern
- See in the github repo, made five commits , step by step so you can easily understand how we handled the code

- You can do authentication on the basis of these two patterns

Authentication Patterns

Statefull

Which maintains state or data or server side

Stateless

Which has no state

- you are parking a car in the parking lot, the parking boy there gives you a receipt or token based on your car and maintain a state like this, so when you came to get the car back it takes receipt back and delete the state
- So state is something which you can map it to somethinaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Statefull

State

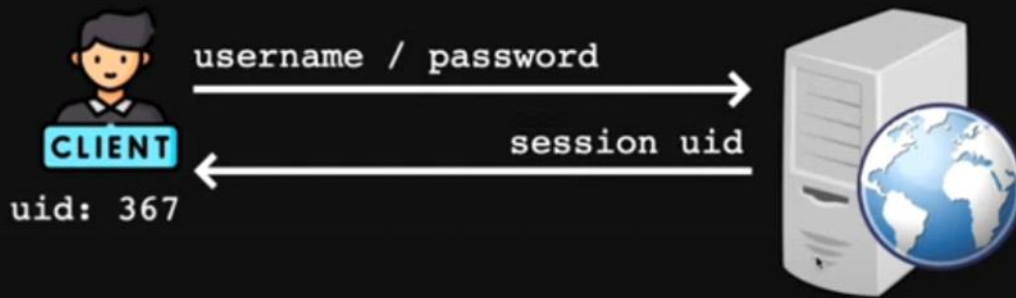
Number 24: DLXXXX123

Number 23: DLXXXX134

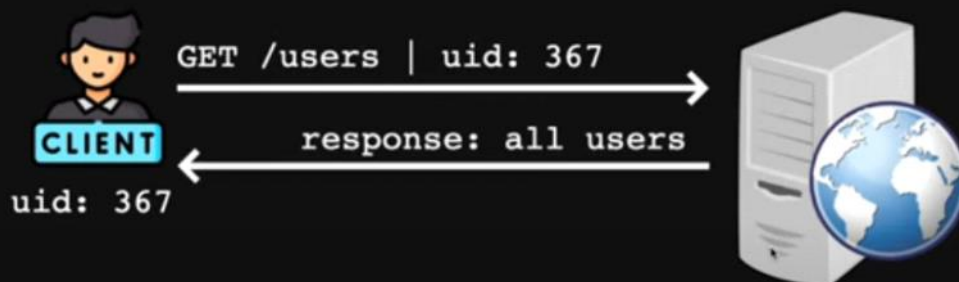
Number 26: DLXXXX167

Number 27: DLXXXX123

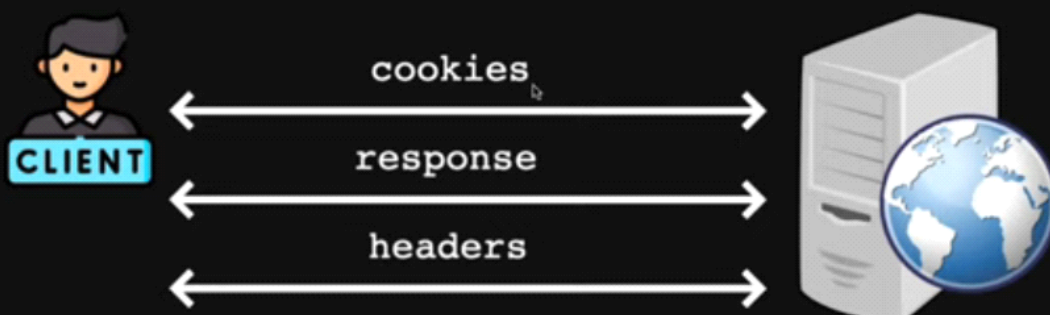
Statefull



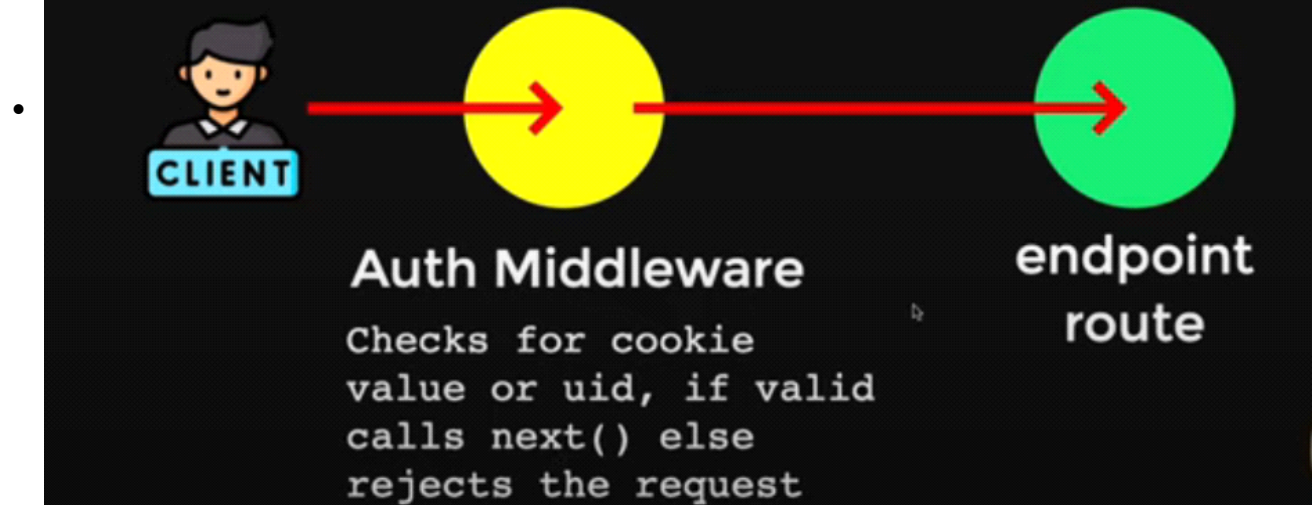
Statefull



How to transfer uid?



Express Flow



- Now I am providing the git repo

Jwt authentication

05 April 2024 12:27

- In stateful authentication for some reason if we lost our state, or we restart the server then all the users get logged out
- Stateful authentication is memory intensive, that means it takes the memory of the server, and we have limited memory there
- Now moving on to stateless
- So how do we store our data, if we don't have any state in the backend
- we store our data/payload inside the token/parking ticket and we put a stamp onto it so that no one else other than the owner can use it, so look closely you can read it but can't use it



- Session id is changed when you login everytime, but token remains the same, keep your token safe
- The tokens can only be modified by those who possess the secret key.
- Try experimenting:

When logged in, you receive a token. Attempting to modify its email address, name, or any payload details without the secret key on jwt.io will fail. However, with the secret details, you can successfully make these changes.

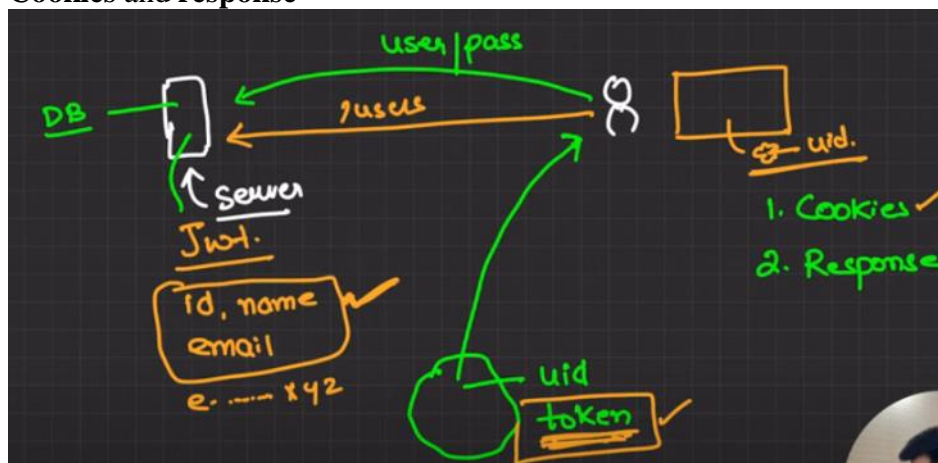
-

Cookies

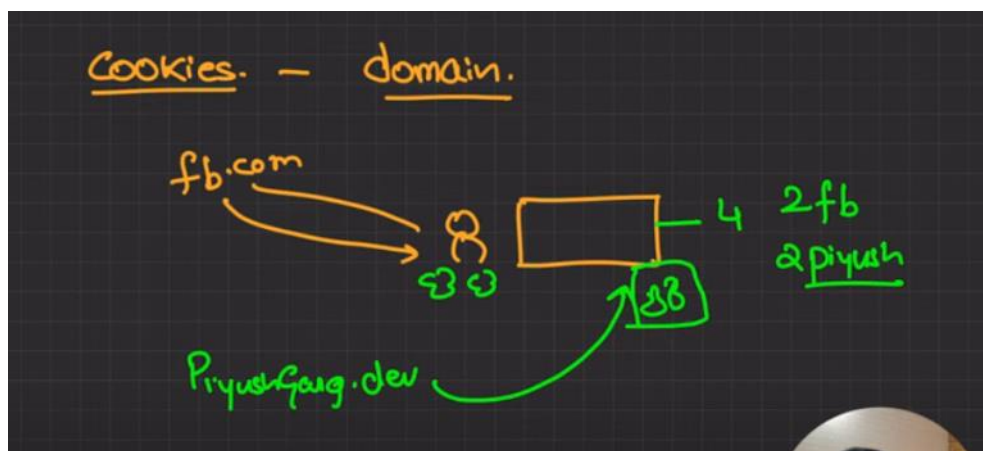
06 April 2024 14:34

- what if we store sessions in database, then there will be no problem in stateful authentication right?? what will be the problems
- Performance: Accessing a database for every request to retrieve or update session data can introduce latency and affect the overall performance of the application, especially under heavy load.
- Complexity: Implementing database-backed session management adds complexity to the application architecture. You need to handle database connections, manage transactions, and ensure data integrity, which can increase development and maintenance efforts.
- There are two ways in which we can give token to user in a secure way

Cookies and response



- Cookies are made by server always (res.cookie), we store token in cookies, it's the browser default behaviour to store the cookies
- So when we do any requests, get or post or anything, then the cookies go with it (sent by browser)
- Cookies are domain specific, suppose you are logging in fb.com and then two cookies were made similarly in piyushgargdev.com, so if you again request to piyushgargdev.com then only those two cookies will go that were made when you logged in to the piyushgargdev.com



- Earlier we were doing like this,
`res.cookie("uid", token)`
- But you can pass an extra argument here

res.cookie(name, value [, options])

Sets cookie name to value. The value parameter may be a string or object converted to JSON.

The options parameter is an object that can have the following properties.

Property	Type	Description
domain	String	Domain name for the cookie. Defaults to the domain name of the app.
encode	Function	A synchronous function used for cookie value encoding. Defaults to <code>encodeURIComponent</code> .
expires	Date	Expiry date of the cookie in GMT. If not specified or set to 0, creates a session cookie.
httpOnly	Boolean	Flags the cookie to be accessible only by the web server.
maxAge	Number	Convenient option for setting the expiry time relative to the current time in milliseconds.
path	String	Path for the cookie. Defaults to <code>/</code> .
partitioned	Boolean	Indicates that the cookie should be stored using partitioned storage. See Cookies Having Independent Partitioned State (CHIPS) for more details.
priority	String	Value of the "Priority" Set-Cookie attribute.
secure	Boolean	Marks the cookie to be used with HTTPS only.
signed	Boolean	Indicates if the cookie should be signed.
sameSite	Boolean or String	Value of the "SameSite" Set-Cookie attribute. More information at https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site-00#section-4.1.1 .

```
res.cookie("uid", token, {  
  domain: "www.google.com",  
});
```

- So now if you try to login , you will not be able to, because this cookie will only be accessible to google.com now
- \you can even set the expiry date to a token
- <https://expressjs.com/en/api.html>
- You can't see cookies in mobile, because cookies are the features of browser only
- Now there is another way that is **response**
- Server will make a token & we will send this token by `res.json({token})`, now it's the job of user to save token, user can either save it in device or file \
- But the point is how you will take this res thing, like cookie was sent automatically with every request
- With every route you send something like this

Token

`res.json({token})`

Route:

header: { Authorization: "Bearer <token>" }

Cookie — auto.
Only for browser

Header.

Json —> Header "Authorization"
Bearer " "

Authorization

07 April 2024 15:28

- **Authentication:** Authentication is the process of verifying the identity of a user or system. It answers the question "Who are you?". This typically involves users providing credentials, such as a username and password, which are then validated against a database or authentication service. Once the user's identity is confirmed, they are granted access to the system or resources.
- **Authorization:** Authorization, on the other hand, is the process of determining whether a user or system has permission to access a specific resource or perform a particular action. It answers the question "What are you allowed to do?". Once a user is authenticated, authorization mechanisms ensure that they only have access to the resources or actions that they are allowed to access.