----------------------------------------Node.js---------------------------------------------

Introduction:-

- Node.js is open source, cross-platform, JavaScript runtime environment.

- Node applications can be developed using either JavaScript or typescript.

- Node.js was released by Ryan Dahl on 27th May 2009, at Netscape.

- The applications (servers) developed by Node.js are called 'Single Threaded Event Loop'

--------------------------------------------------------------------------------------------

- Node.js contain infinite event loop.

- This loop can handle two types of requests

- Non - blocking I/O requests.

- Blocking I/O requests.

- Multiple clients requesting to server.

- these requests kept in Event queue.

- these requests sent to event loop.

- if the requests are non-blocking I/O requests, response will be sent immediately to clients.

- if the requests are blocking I/O requests, those are sent to thread pool.

- thread pool will pass requests to databases or file systems (third parties).

- the response given by third parties will be in thread pool.

- the response from thread pool will be in event loop.

- event loop will give this response back to clients.

--------------------------------------------------------------------------------------------

-- HTTP

- This is the native module.

- This module is available along with 'Node Engine'.

--------------------------------------------------------------------------------------------

-- URL

- This is native module.

- This module is used to read get parameters in http servers.

--------------------------------------------------------------------------------------------

-- query-string

- This is native module.

- This module is used to read the post parameters in http servers.

--------------------------------------------------------------------------------------------

-- fs

- This is native module.

- fs stands for 'File System'.

- This module is used to interact with flat files.

- Eg :- txt, xml, json, etc

--------------------------------------------------------------------------------------------

-- Express

- This is third party module.

- This module is used to develop 'ReST APIs' (web services)

- ReST API :- Representational State Transfer Application Programming Interface.

--------------------------------------------------------------------------------------------

-- MySQL

- This is third party module.

- This module is used to interact with MySQL database.

--------------------------------------------------------------------------------------------

-- mongodb

- This is third party module.

- This module is used to interact with mongodb without schema.

- Note :- rules and regulations of db are called as schema.

--------------------------------------------------------------------------------------------

-- Mongoose

- This is third party module.

- This module is used to interact with mongodb with schema.

--------------------------------------------------------------------------------------------

-- mssql

- This is third party module.

- This module is used to interact SQL server.

--------------------------------------------------------------------------------------------

-- multer

- This is third party module.

- This module is used to upload images to server.

--------------------------------------------------------------------------------------------

-- Socket.io

- This is third party module.

- This module is used to develop chat applications.

--------------------------------------------------------------------------------------------

-- jwt-simple

- This is third party module.

- This module is used to generate tokens for authentication purpose.

- This system is technically called as token based authentication system.

--------------------------------------------------------------------------------------------

-- body-parser

- This is third party module.

- This module is used to set MIME type.

--------------------------------------------------------------------------------------------

-- cluster

- This is third party module.

- This module is used to implement child process in http server.

- Implementing child process is called as load balancing.

--------------------------------------------------------------------------------------------

-- express-cluster

- This third party module.

- This module is used to implement load balancing in ReST APIs.

--------------------------------------------------------------------------------------------

-- Cookie-parser

- This is third party module.

- This module is used to work with cookies.

--------------------------------------------------------------------------------------------

- We can download all third party modules by using either 'npm' or 'yarn' tool.

- 'npm' stands for Node packaging manager.

- 'yarn' is latest tool used to download node modules in faster manner as compared to 'npm'.

- 'yarn' tool given by facebook

- All node modules are downloaded to 'node\_modules' directory in current path.

--------------------------------------------------------------------------------------------

https://shorturl.at/bw12r 12 Dec 2024

G21 BEE

KickStart with Node.js https://shorturl.at/6DjNO

Creating HTTP server

Passing Custom HTML response

=======================================================

Implementing HTTP Server

=======================================================

- ‘http’ is the predefined module used to create http servers.

- http is the native module, so no need to download it.

- ‘require()’ is used to import.

- Eg let http = require(‘http’)

- ‘createServer()’ is the predefined function in http module.

- This function is used to create the server.

- The argument to createServer() is arrow function.

- To this arrow function there are two arguments, ‘req’ and ‘res’.

- request and response objects provided by node engine respectively.

- req object is used to store client data.

- res object is used to send response to client.

- 'writeHead(-,-)' is the predefined function in res object.

- This function is used to set the MIME type.

- First argument to this function is the status code (200 - ok).

- Second argument is the JSON object.

- JSON key is ‘content-type’ and the value is ‘text/html’.

- ‘write(-)’ is the predefined function in res object.

- This function is used to append response to res object.

- ‘end()’ is the predefined function in res object.

- This function is used to lock the response.

Task:- Getting Custom HTML response from http server

let resText = `Welcome to HTTP server`

let myRes = `

<!DOCTYPE html>

<Html>

<head>

<style>

body

{

background-color: black;

}

h1{

color: black;

text-shadow: 0px 0px 2px red;

}

</style>

</head>

<body>

<h1>${resText}</h1>

</body>

</Html>

`

//import http module

let http = require('http')

let server = http.createServer((req, res) => {

//set MIME type

res.writeHead(200, { 'content-type': 'text/html' })

res.write(myRes)

res.end()

})

//assign port no

server.listen(8080)

console.log('Server listening port no 8080')

-------------------------------------------------------------------------------------------------------------

https://shorturl.at/bw12r 13 Dec 2024

G21 BEE

KickStart with Node.js https://shorturl.at/6DjNO

=======================================================

HTTP get parameters

=======================================================

- 'url' is the predefined module in node.

- url module is used to read get parameters in http server.

\*\*\*server.js\*\*\*

//import http module

let http = require('http')

//import url module

let url = require('url')

let server = http.createServer((req, res) => {

let obj = url.parse(req.url, true).query

let uname = obj.uname

let upwd = obj.upwd

//set MIME type

res.writeHead(200, { 'content-type': 'text/html' })

if (uname === 'admin' && upwd === 'admin')

res.write("<h1> Login Success </h1>")

else

res.write("<h1> Login Failed </h1>")

res.end()

})

//assign port no

server.listen(8080)

console.log('Server listening port no 8080')

//url :- http://localhost:8080/?uname=admin&upwd=admin

\*\*\*index.html\*\*\*

<!DOCTYPE html>

<html>

<head>

<link rel="stylesheet" href="style.css">

</head>

<body>

<form action="http://localhost:8080" method="get" class="box">

<h1>Login</h1>

<input type="text" placeholder="Username" name="uname">

<input type="password" placeholder="Password" name="upwd">

<input type="submit" value="Login">

</form>

</body>

</html>

\*\*\*style.css\*\*\*

h1 {

color: white;

text-transform: uppercase;

font-weight: normal;

}

body {

background: radial-gradient(white, black);

font-family: sans-serif;

}

.box {

background-color: black;

width: 300px;

margin: 50px auto;

padding: 40px;

border-radius: 20px;

text-align: center;

}

input {

margin: 20px auto;

text-align: center;

padding: 14px 10px;

width: 200px;

border-radius: 24px;

background: none;

}

input[type="text"],

input[type="password"] {

border: 2px solid skyblue;

color: lightyellow;

}

input[type="submit"]

{

border: 2px solid burlywood;

color: white;

cursor: pointer;

}

=================================================

HTTP post parameters

=================================================

- 'querystring' is the predefined module in nodejs.

- querystring module is used to read post parameters in http server

\*\*\*server.js\*\*\*

//import http module

let http = require('http')

//import querystring module

let qs = require('querystring')

let server = http.createServer((req, res) => {

//set MIME type

res.writeHead(200, { 'content-type': 'text/html' })

let body = ''

//listen to post parameters

req.on('data', (result) => {

body = body + result

})

//end callback to node engine

req.on('end', () => {

let obj = qs.parse(body)

let uname = obj.uname

let upwd = obj.upwd

if (uname === 'admin' && upwd === 'admin')

res.write("<h1 style = 'color:green'> Login Success </h1>")

else

res.write("<h1 style = 'color:red'> Login Failed </h1>")

res.end()

})

})

//assign port no

server.listen(8080)

console.log('Server listening port no 8080')

//url :- http://localhost:8080/?uname=admin&upwd=admin

\*\*\*index.html\*\*\*

<!DOCTYPE html>

<html>

<head>

<link rel="stylesheet" href="style.css">

</head>

<body>

<form action="http://localhost:8080" method="post" class="box">

<h1>Login</h1>

<input type="text" placeholder="Username" name="uname">

<input type="password" placeholder="Password" name="upwd">

<input type="submit" value="Login">

</form>

</body>

</html>

\*\*\*style.css\*\*\*

h1 {

color: white;

text-transform: uppercase;

font-weight: normal;

}

body {

background: radial-gradient(white, black);

font-family: sans-serif;

}

.box {

background-color: black;

width: 300px;

margin: 50px auto;

padding: 40px;

border-radius: 20px;

text-align: center;

}

input {

margin: 20px auto;

text-align: center;

padding: 14px 10px;

width: 200px;

border-radius: 24px;

background: none;

}

input[type="text"],

input[type="password"] {

border: 2px solid skyblue;

color: lightyellow;

}

input[type="submit"]

{

border: 2px solid burlywood;

color: white;

cursor: pointer;

}

**Similar Technologies Like Node.js**

•  Django (Python)

•  Ruby on Rails (Ruby)

•  Spring Boot (Java)

•  Flask (Python)

•  ASP.NET Core (C#)

**Django (Python)**

•  High-level, batteries-included web framework.

•  Built-in tools for security, database management, and authentication.

•  Scalable and secure, with automatic protections against common vulnerabilities.

•  Ideal for rapid development of complex applications.

•  Great for web apps like CMS, social networks, and e-commerce.

•  Users :- Instagram, Pinterest, Mozilla, Disqus, Washington Post

**Ruby on Rails (Ruby)**

•  Convention-over-configuration framework for fast development.

•  Follows the MVC architecture (Model-View-Controller).

•  Built-in ORM (ActiveRecord) for easy database management.

•  Ideal for prototyping and startups.

•  Strong ecosystem with many reusable libraries (gems).

•  Users :- Basecamp, GitHub, Shopify, Airbnb, Twitch

**Spring Boot (Java)**

•  Simplifies Java web development with minimal configuration.

•  Ideal for microservices and RESTful APIs.

•  Includes embedded servers (e.g., Tomcat, Jetty).

•  Part of the larger Spring ecosystem (security, cloud, etc.).

•  High performance and scalability for enterprise applications.

•  Users :- Netflix, Uber, Alibaba, eBay, Accenture

**Flask (Python)**

•  Lightweight and minimalistic web framework.

•  Flexible and extensible, ideal for small to medium-sized apps.

•  Commonly used for microservices and RESTful APIs.

•  Jinja2 templating for dynamic content rendering.

•  Easy to integrate with third-party extensions.

•  Users :- Airbnb, Netflix, Uber, Reddit, LinkedIn

**ASP.NET (C#)**

•  Cross-platform, high-performance web framework.

•  Suitable for building scalable, cloud-based applications.

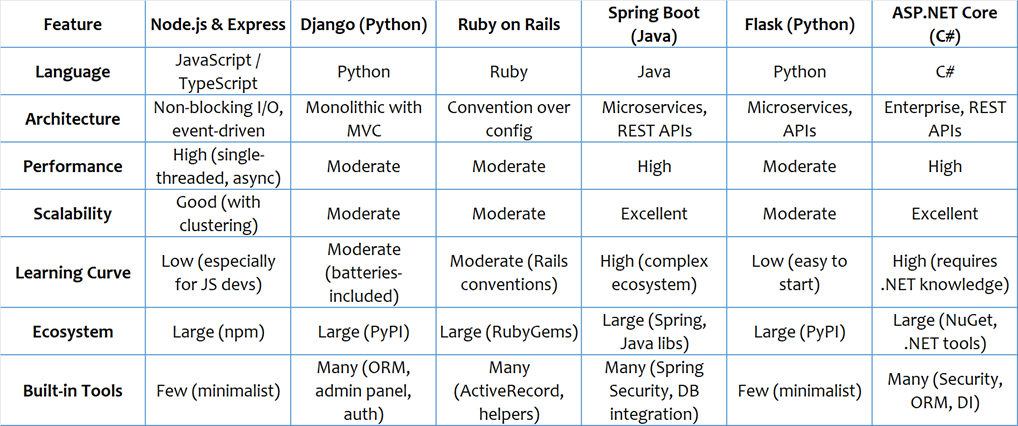
•  Modular architecture with built-in security features.

•  Great for microservices, especially with Azure integration.

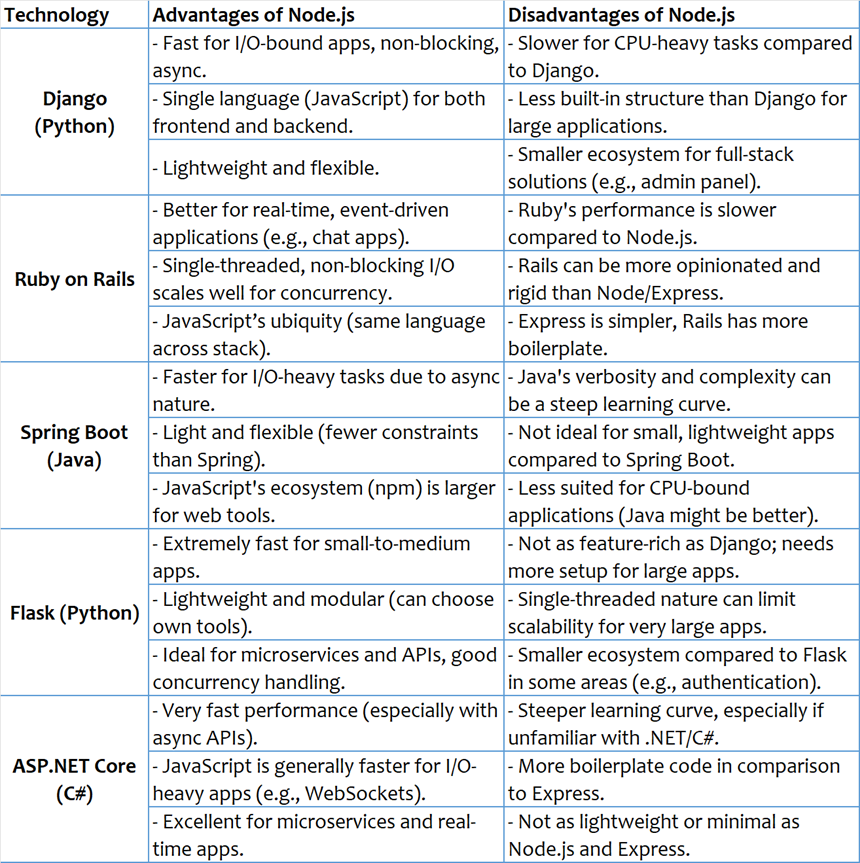
•  High speed and low overhead for large apps.

•  Users :- Stack Overflow, Microsoft, Trello, Siemens, IntellIJ IDEA

**Quick Overview**



**advantages and disadvantages of Node.js with above technologies**

****

**Application of Node.JS**

•  Real-time chats

•  Complex single-page applications

•  Real-time collaboration tools

•  Streaming apps

•  JSON APIs

handling exceptions(errors)

 - There are various ways in node.js to handle the exceptions

- Synchronous errors: Use try-catch blocks to handle exceptions.

- Asynchronous errors: Handle errors in callbacks by checking the first argument (err).

- Global errors: Use process.on('uncaughtException') to handle uncaught exceptions globally.

- Custom errors: You can create custom error objects using Error and manipulate their properties.

Creating endpoints

 - method is the predefined key from req object.

 - this key gives type of method in request.

 - url is the predefined key from req object.

 - this key gives url of the request.

 - setHeader is the predefined function from res object

 - this function is used to set headers.

 - statusCode is the predefined key from res object.

 - we can set status code using this key.

 - handleError is the user defined function.

 - we are handling exceptions(errors) from this function.

 - possible errors are

- 'Route not found'

- 'Invalid method'

- etc

 - if any error occurred statusCode will be 4xx.

//Import the 'http' module

const http = require('http')

//handle the errors

function handleError(res, errorCode, message) {

    res.statusCode = errorCode

    res.write(message)

    res.end()

}

// Create HTTP server

const server = http.createServer((req, res) => {

    // Set the response header indicating the content type is JSON

    res.setHeader('Content-Type', 'application/json')

    res.statusCode = 200

    console.log('url',req.url)

    console.log('Method',req.method)

    //GET method

    if (req.method == 'GET') {

        if (req.url == '/')

            res.write('Default get request')

        else if (req.url == '/login')

            res.write('Login get request')

        else

            handleError(res, 404, 'Route not found')

    }

    //POST method

    else if (req.method == 'POST') {

        if (req.url == '/')

            res.write('Default post request')

        else if (req.url == '/login')

            res.write('Login post request')

        else

            handleError(res, 404, 'Route not found')

    }

    else

        handleError(res, 404, 'Invalid Method')

    res.end()

})

// Start the server on port 8080

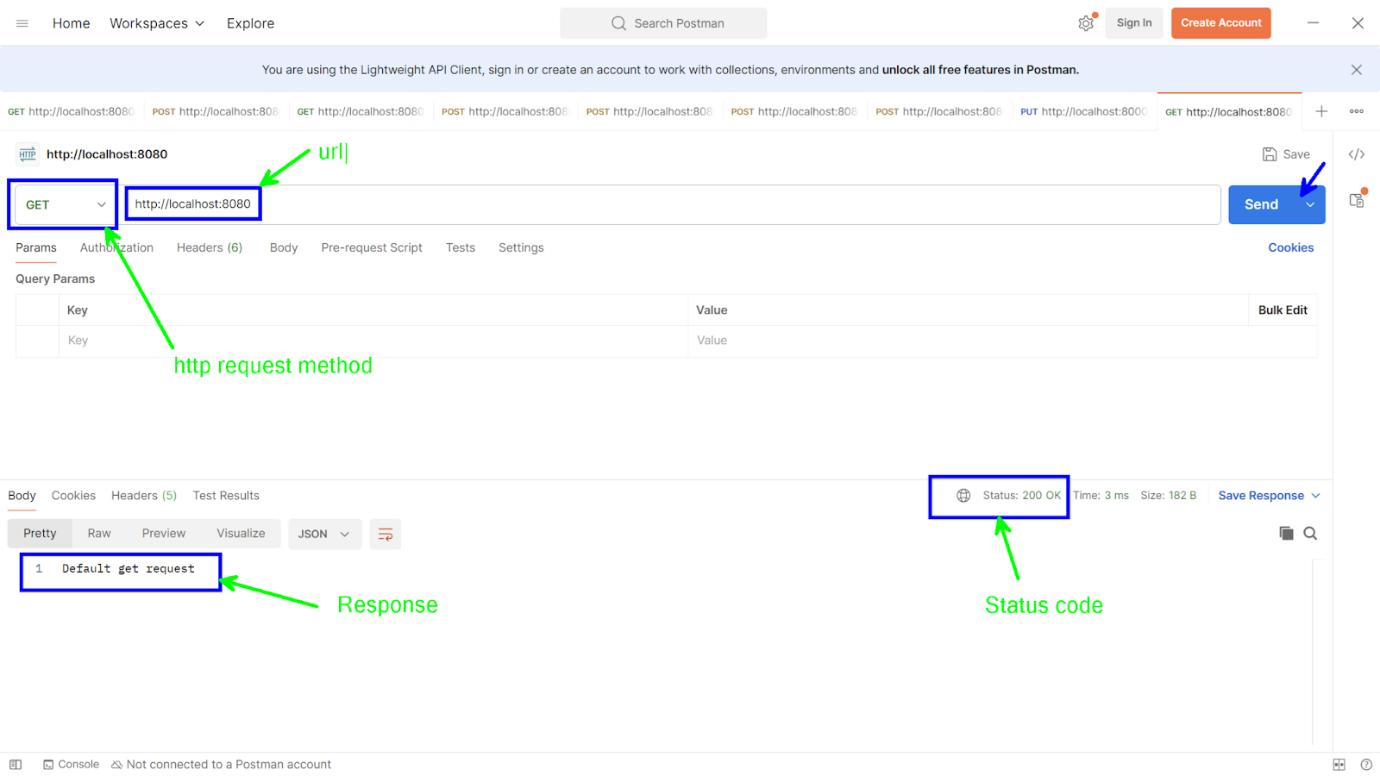
const PORT = 8080

server.listen(PORT, () => {

    console.log(`Server is running at http://localhost:${PORT}`)

})

//Test in Postman



Task:-

Create nodeserver

Display home page

on home page provide two links/buttons

login and dashboard

dashboard -> error route not found

on login navigate to login endpoint -> login get request

on login page accept username and password

as per authentication display respective messages

apply proper styling

//Import the 'http' and 'querystring' modules

const http = require('http')

const qs = require('querystring')

// handle the errors

function handleError(res, errorCode, message) {

    res.statusCode = errorCode

    res.write(message)

    res.end()

}

// Create HTTP server

const server = http.createServer((req, res) => {

    // Set the response header

    res.setHeader('Content-Type', 'text/html')

    res.statusCode = 200

    // Log the incoming request to see headers

    console.log(`Request method: ${req.method} | URL: ${req.url}`)

    //GET method

    if (req.method == 'GET') {

        if (req.url == '/') {

            res.write(homePage)

        }

        else if (req.url == '/login') {

            res.write(loginPage)

        }

        else {

            handleError(res, 404, 'Route not found')

        }

    }

    // POST method

    else if (req.method == 'POST') {

        if (req.url == '/') {

            res.write('Default post request')

        }

        else if (req.url == '/login') {

            let body = ''

            // Listen to post parameters

            req.on('data', (data) => {

                body += data

            })

            // end callback to node engine

            req.on('end', () => {

                // Log the body to see what data is being received

                console.log('Received Body:', body)

                // Parse the form data

                let obj = qs.parse(body)

                console.log('Parsed Object:', obj) // Check if we are parsing correctly

                // Extract username and password

                let uname = obj.uname

                let upwd = obj.upwd

                // Basic validation

                if (uname === 'admin' && upwd === 'admin') {

                    res.write("<h1 style='color:green'> Login Success </h1>")

                } else {

                    res.write("<h1 style='color:red'> Login Failed </h1>")

                }

                res.end()

            })

        } else {

            handleError(res, 404, 'Route not found')

        }

    }

    else {

        handleError(res, 404, 'Invalid Method')

    }

})

// Start the server on port 8080

const PORT = 8081

server.listen(PORT, () => {

    console.log(`Server is running at http://localhost:${PORT}`)

})

// Home Page HTML

let homePage = `<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" rel="stylesheet">

    <title>Home Page</title>

</head>

<body class="bg-light">

    <nav class="navbar navbar-expand-lg navbar-light bg-primary">

        <div class="container-fluid">

            <a class="navbar-brand text-white" href="#">Home</a>

            <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

                <span class="navbar-toggler-icon"></span>

            </button>

            <div class="collapse navbar-collapse" id="navbarNav">

                <ul class="navbar-nav ms-auto">

                    <li class="nav-item">

                        <a class="nav-link text-white" href="/login">Login</a>

                    </li>

                    <li class="nav-item">

                        <a class="nav-link text-white" href="/dashboard">Dashboard</a>

                    </li>

                </ul>

            </div>

        </div>

    </nav>

    <div class="container text-center my-5">

        <h1 class="display-4 text-primary mb-4">Welcome to Home</h1>

        <p class="lead">Please choose an action below:</p>

        <div>

            <a href="/login" class="btn btn-primary btn-lg mx-3">Login</a>

            <a href="/dashboard" class="btn btn-secondary btn-lg mx-3">Dashboard</a>

        </div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"></script>

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/js/bootstrap.min.js"></script>

</body>

</html>`

// Login Page HTML (same as before)

let loginPage = `

<!DOCTYPE html>

<html>

<head>

    <style>

        h1 {

            color: white;

            text-transform: uppercase;

            font-weight: normal;

        }

        body {

            background: radial-gradient(white, black);

            font-family: sans-serif;

        }

        .box {

            background-color: black;

            width: 300px;

            margin: 50px auto;

            padding: 40px;

            border-radius: 20px;

            text-align: center;

        }

        input {

            margin: 20px auto;

            text-align: center;

            padding: 14px 10px;

            width: 200px;

            border-radius: 24px;

            background: none;

        }

        input[type="text"],

        input[type="password"] {

            border: 2px solid skyblue;

            color: lightyellow;

        }

        input[type="submit"] {

            border: 2px solid burlywood;

            color: white;

            cursor: pointer;

        }

    </style>

</head>

<body>

    <form action="/login" method="post" class="box">

        <h1>Login Page</h1>

        <input type="text" placeholder="Username" name="uname" required>

        <input type="password" placeholder="Password" name="upwd" required>

        <input type="submit" value="Login">

    </form>

</body>

</html>`

**G21 BEE**

**Task :- Custom Navigation  Using static files**

Handling Static files

 - path is the predefined module in node.js.

 - this module is used for working with file and directory paths.

 - this module is having predefined functions like,

join()

resolve()

basename()

are used to handle and manipulate file paths.

 - \_\_dirname gives current path of server.js

<>

public

- index.html

server.js

\*\*\*index.html\*\*\*

<!DOCTYPE html>

<html>

    <head></head>

    <body>

        <h1 style="color: blue;">Welcome to static files</h1>

    </body>

</html>

\*\*\*server.js\*\*\*

// Import required modules

const http = require('http')

const fs = require('fs')

const path = require('path')

// Create an HTTP server

const server = http.createServer((req, res) => {

    const filePath = path.join(\_\_dirname, 'public','index.html')

    console.log('Path:- ', \_\_dirname)                    //?

    console.log('Base name', path.basename(\_\_dirname))   //?

    console.log('Resolve:- ', path.resolve(filePath))    //?

    fs.readFile(filePath, (err, data) => {

        if (err)

            console.log(err)

        else

            res.write(data)

        res.end()

    })

})

// Specify the port and start the server

const PORT = 8080;

server.listen(PORT, () => {

    console.log(`Server is running on http://localhost:${PORT}`)

})

Complete task given in last session using static files.

**G21 BEE**

**Routing in express**

**Routing methods**

**Route paths**

**Route parameters**

**Route Handlers**

=================================================

Routing in express

=================================================

 - module is the predefined object in node.

 - exports is the predefined key in module object.

 - exports key is used to export (JSON object or function)

**1. Routing Methods:**

* **Routing methods** define the HTTP method (GET, POST, PUT, DELETE) for a route in Express.

**2. Route Paths:**

* **Route paths** are the URL patterns used to define a route in Express. They can be static (e.g., /login) or dynamic, using parameters (e.g., /user/:id).

**3. Route Parameters:**

* **Route parameters** are dynamic placeholders in a route path, prefixed by a colon (:), which capture values from the URL. They are accessed via req.params (e.g., req.params.id).

**4. Route Handlers:**

* **Route handlers** are callback functions that are executed when a request matches a route. They handle the request and send a response using res.send() or res.json().

<>

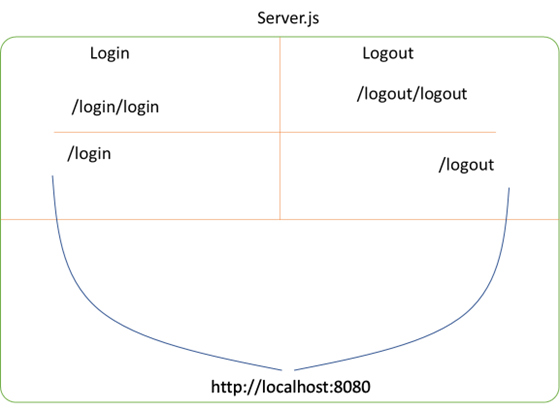
           login

                        - login.js

           logout

                        - logout.js

           - server.js



\*\*\*login.js\*\*\*

//import express module

let express = require('express')

//create router instance

let router = express.Router()

//create get request

router.get("/",(req, res)=>{

    res.send({ 'message': 'Welcome to Login module' })

})

//create one more get request

router.get("/login/:uname/:upwd", (req, res) => {

    //here we are reading url parameters using params

    let uname = req.params.uname

    let upwd = req.params.upwd

    if (uname == 'admin' && upwd == 'admin')

        res.json({ 'login': 'success' })

    else

        res.json({ 'login': 'failed' })

})

//export router instance

module.exports = router

\*\*\*logout.js\*\*\*

//import express module

let express = require('express')

//create router instance

let router = express.Router()

//create get request

router.get("/", (req, res) => {

    res.json({ 'message': 'Welcome to logout module' })

})

//create one more get request

//URL :- http://localhost:8080/logout/logout/?uname=admin&upwd=admin

router.get("/logout", (req, res) => {

    //here we are reading get parameters using query

    let uname = req.query.uname

    let upwd = req.query.upwd

    if (uname == 'admin' && upwd == 'admin')

        res.send({ 'logout': 'Success' })

    else

        res.send({ 'logout': 'Failed' })

})

//export router

module.exports = router

\*\*\*server.js\*\*\*

//import modules

let express = require('express')

let login = require('./login/login')

let logout = require('./logout/logout')

//create rest object

let app = express()

//use modules

app.use("/login",login)

app.use("/logout",logout)

//create port

let port = 8080

//assign port no

app.listen(port, () => {

    console.log(`Server running on port no :- ${port}`)

})

/\*

http://localhost:8080/login

http://localhost:8080/login/login/admin/admin

http://localhost:8080/logout

http://localhost:8080/logout/logout?uname=admin&upwd=admin

\*/

https://shorturl.at/bw12r   17 Jan 2025

**G21 BEE**

=================================================

**Middleware**

=================================================

**What is Middleware in Express?**

Middleware in Express is a function that processes a request before it reaches the route handler, or processes the response before it is sent back to the client.

These functions have access to three core objects:

1. req (Request Object): Contains data about the incoming request (e.g., URL, query parameters, headers, body).
2. res (Response Object): Allows the middleware to modify the outgoing response (e.g., setting status codes, sending a response body).
3. next (Function): A callback that passes control to the next middleware function in the stack. If you don’t call next(), the request-response cycle will be halted.

=================================================

Middleware lifecycle

=================================================

-          The Express middleware lifecycle is a sequence of steps through which HTTP requests pass as they are processed by the server.

-          Each middleware function is executed in the order it is defined, with each function having access to the request (req), response (res), and a next() function to pass control to the next middleware in the chain.

1.   Request Reception

·        When a request is made to the Express application, it first enters the server and is sent to the first middleware function.

2.       Middleware Execution

·        Middleware functions are executed in the order they are defined in the application. Each middleware function has access to the req (request object), res (response object), and the next() function, which is used to pass control to the next middleware in the stack.

3.      Routing Middleware

·        After general middleware, Express checks if there’s a matching route (like GET /users) based on the incoming HTTP method and URL. If a match is found, the corresponding route handler is executed.

·        If no route matches, a 404 error middleware can be triggered to indicate that the requested resource doesn’t exist.

4.      Response Sent

·        Once the middleware chain completes, the server sends the response to the client. This can either be a standard HTTP response or an error response.

5.       Error Handling

·        If any middleware throws an error or if there’s an issue during the request-response cycle, the error-handling middleware (which must have four parameters: err, req, res, next) is triggered.

·        This allows the server to handle errors centrally, providing a unified response or logging mechanism.

=================================================

Types of middlewares

=================================================

Application-level Middleware: Applies globally or to specific routes (via app.use()).  
  
  
Router-level Middleware: Applied to specific router instances or sub-routes (via router.use()).

// Import the Express library

const express = require('express')

// Create an instance of an Express application

const app = express()

// Create a router instance

const router = express.Router()

// Router-level middleware

// This middleware will be executed for every route defined on the router

router.use((req, res, next) => {

    console.log('Router-level middleware') // Logs a message every time a request hits the router

    next() // Passes control to the next middleware or route handler

})

// Define a GET route for "/users" on the router

router.get('/users', (req, res) => {

    res.send('User list') // Responds with a simple message "User list" when the "/users" route is accessed

})

// Use the router with the main Express app

// Any route starting with "/api" will now be handled by the router defined above

app.use('/api', router) // All routes that match "/api/\*" will pass through the router

// Start the Express server and make it listen on port 8080

app.listen(8080, () => {

    console.log('Server is running on port 8080') // Logs a message indicating the server has started and is listening on port 8080

})

//test url http://localhost:8080/api/users

Error-handling Middleware: Catches errors that occur in the application and handles them with a custom response.

// Import the Express library

const express = require('express')

// Create an instance of an Express application

const app = express()

// Example route that throws an error

// This route simulates an error by throwing it when accessed

app.get('/', (req, res, next) => {

    const error = new Error('Something went wrong') // Create a new error

    next(error) // Pass the error to the error-handling middleware

})

app.get('/hello', (req, res) => {

    res.send('Hello, World!') // Non-error route that responds normally

})

// Error-handling middleware

// This middleware catches any errors passed to the `next` function

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

    console.error(err.stack) // Log the full error stack for debugging

    res.status(500).send('Something went wrong!') // Send a 500 error response to the client

})

// Start the server and listen on port 8080

app.listen(8080, () => {

    console.log('Server is running on port 8080') // Log a message indicating the server is running

})

/\*

Test urls

http://localhost:8080/hello     -> without error

http://localhost:8080/          -> with error

\*/

Third-party Middleware: Middleware created by others that provides extra features or functionality (e.g., logging, CORS handling, etc.).

// Import the Express library to create the application

const express = require('express')

// Import the morgan library for HTTP request logging (third-party middleware)

const morgan = require('morgan')

// Create an instance of an Express application

const app = express()

// Use morgan middleware for logging HTTP requests

// The 'dev' format is a preset that provides colored status codes and concise logs

app.use(morgan('dev'))

// Example route that responds to a GET request at the root ('/')

app.get('/', (req, res) => {

    res.send('Hello, World!') // Sends 'Hello, World!' as the response to the client

})

// Start the server and listen on port 8080

app.listen(8080, () => {

    console.log('Server is running on port 8080') // Logs a message when the server starts

})

/\*

Test urls :-

http://localhost:8080/

http://localhost:8080/hello

\*/

Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| **Type of Middleware** | **Description** | **Scope** | **Usage** |
| **Application-level Middleware** | Middleware applied to the entire application. | Global (applies to all routes) | Used for tasks like logging, authentication, etc. |
| **Router-level Middleware** | Middleware applied to specific routers or route handlers. | Specific to a particular router/route | Useful for applying logic only to certain route groups. |
| **Error-handling Middleware** | Special middleware to handle errors that occur during request processing. | Global (can be applied at the end of all routes) | Used for catching and responding to errors like 404, 500, etc. |
| **Third-party Middleware** | Middleware provided by external libraries or packages. | Can be applied at the application or router level | Provides functionality like body parsers, session handling, security, etc. |

1. **Application-level Middleware**

//initialyse project

//>npm init -y

//download express module

//>yarn add express --save

// Define port

const port = 8080

// Import the express module

const express = require('express') // Express library for creating the server

//Create rest object

const app = express()

app.listen(port, () => {

  console.log(`Server is running on port ${port}`)

})

**Eg 1** Logging Incoming Requests

**Problem Definition:** Demonstrate to log HTTP method and URL for each incoming request using application-level middleware.

// Application-level middleware to log the HTTP method and URL of each incoming request

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

  console.log(`${req.method} ${req.url}`) // Log the request method and URL

  next() // Pass control to the next middleware or route handler

})

app.get('/', (req, res) => {

  res.send('Hello, World!') // Respond with 'Hello, World!' when the root URL is accessed

})

/\*

    Steps to Execute:

    >node server

    Open a browser or Postman and navigate to `http://localhost:8080/`.

    Check your terminal to see the HTTP method and URL being logged for the request.

\*/

**Eg 2** Setting a Custom Header for All Responses

**Problem Definition:** To set a custom header (X-Powered-By) for every HTTP response using application-level middleware.

// Application-level middleware to set a custom header for all responses

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

  res.setHeader('X-Powered-By', 'Express') // Set custom header 'X-Powered-By'

  next() // Pass control to the next middleware or route handler

})

app.get('/', (req, res) => {

  res.send('Hello, World!') // Respond with 'Hello, World!'

})

/\*

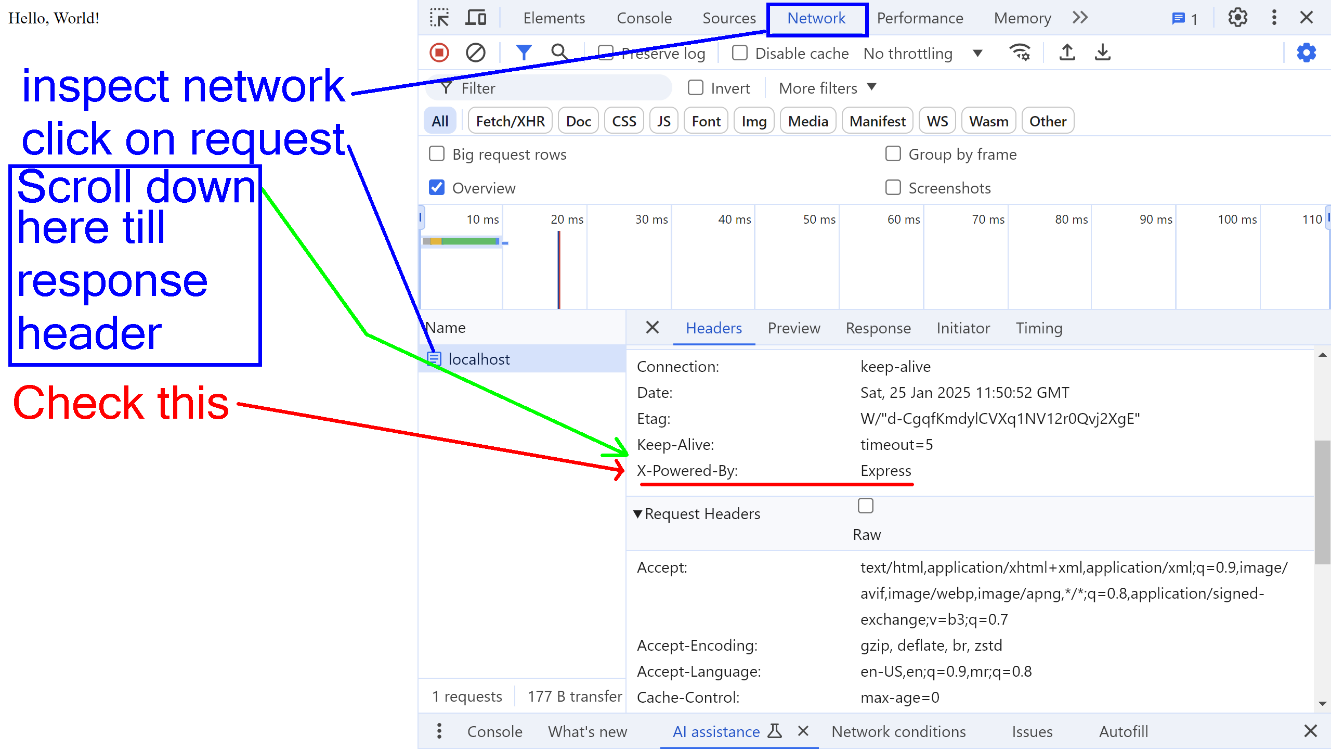
Steps to Execute:

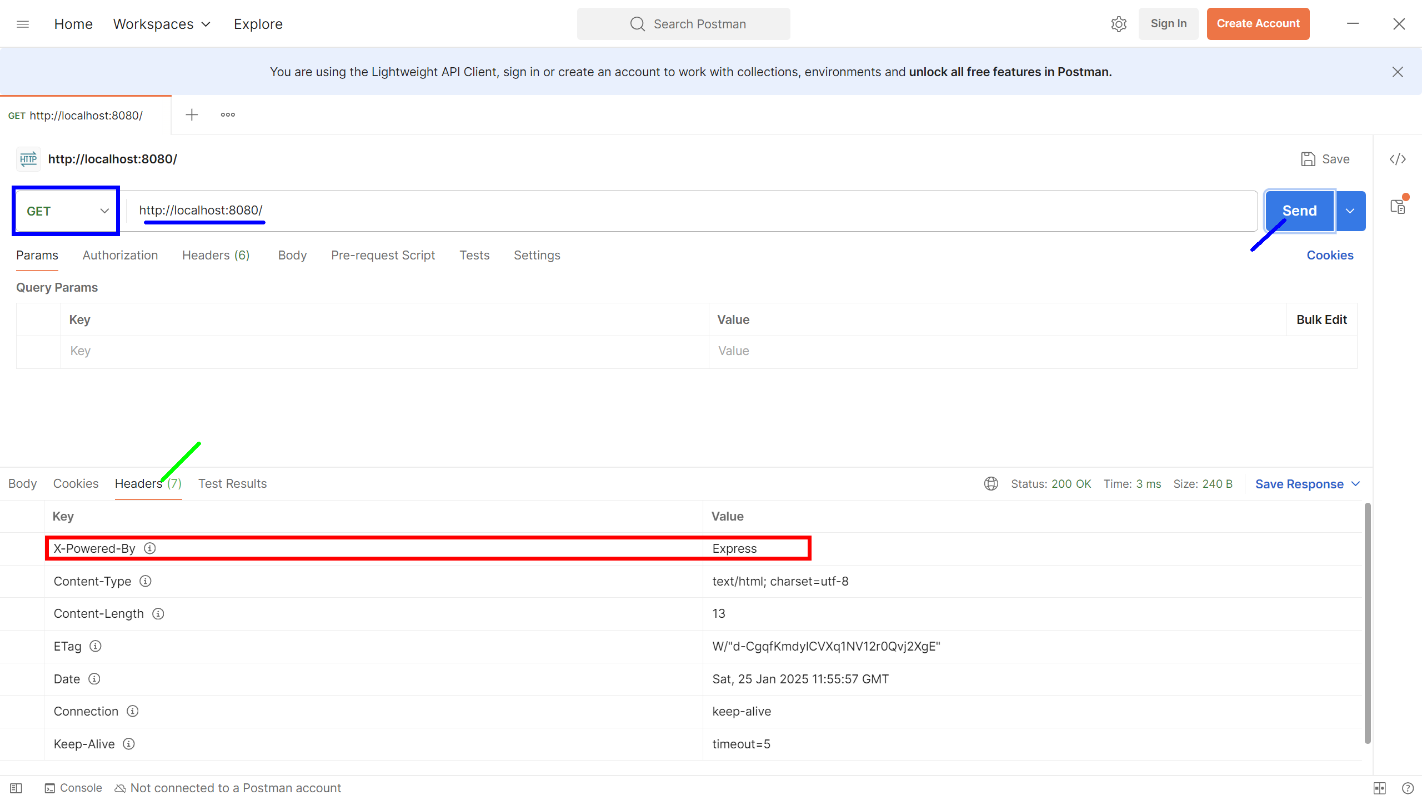
    >node server

    Open a browser or Postman and navigate to `http://localhost:8080/`.

    Inspect the response headers in the browser's developer tools or in Postman to see the `X-Powered-By: Express` header.

\*/





**Eg 3** Authentication for All Routes

**Problem Definition:** Add a basic authentication middleware for all routes, blocking access if no Authorization header is provided.

// Application-level middleware to require authentication for all routes

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

    if (!req.headers.authorization) { // Check if authorization header exists

        return res.status(403).send('Forbidden') // Respond with Forbidden if no authorization

    }

    next() // Pass control to the next middleware or route handler

})

app.get('/', (req, res) => {

    res.send('Hello, World!')

})

/\*

Steps to Execute:

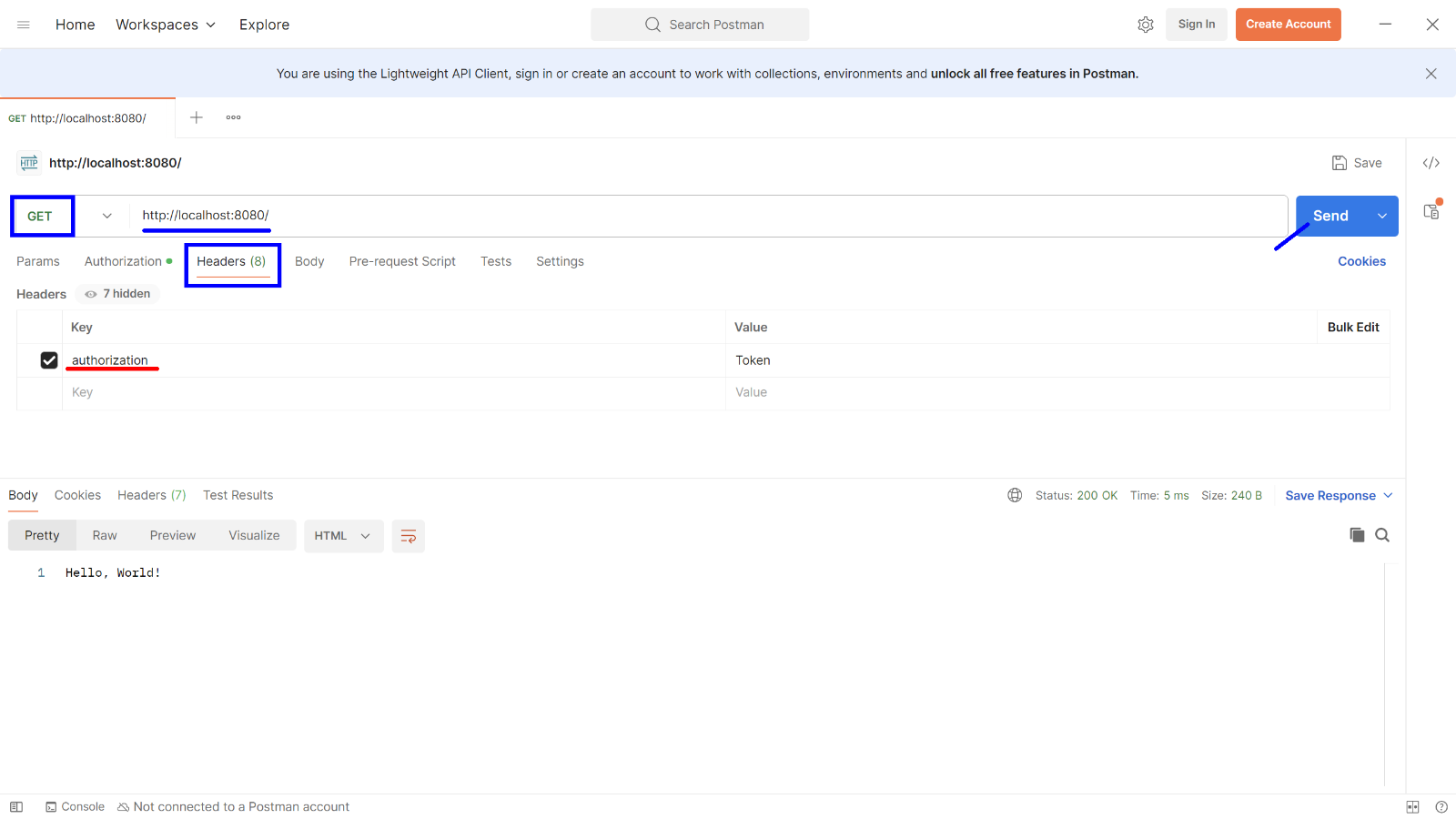
    >node server

    Open Postman and make a GET request to `http://localhost:8080/`.

    - Without any `Authorization` header, the response should be `403 Forbidden`.

    - Add an `Authorization` header with any value to see the successful response.

\*/



**Eg 4** Request Body Parsing

**Problem Definition:** To use application-level middleware to parse incoming JSON request bodies.

// Application-level middleware to parse incoming JSON request bodies

app.use(express.json()) // Parse JSON body

app.post('/', (req, res) => {

  res.send(`Received data: ${JSON.stringify(req.body)}`) // Respond with received data

})

/\*

Steps to Execute:

    >node server

    Open Postman, set the request method to `POST`, and use the URL `http://localhost:8080/`.

    In the "Body" section, select `raw`, choose `JSON`, and add data like:

    {

        "name": "Chitkara University",

        "department": "CSE"

    }

\*/

https://shorturl.at/bw12r   30 Jan 2025

 31 Jan 2025

6 Feb 2025

**G21 BEE**

* **Middlewares (Deep Dive)**

=================================================

**Middleware Deep Dive (For Practice, Applications out of scope currently)**

=================================================

1. **Application-level Middleware**

//initialyse project

//>npm init -y

//download express module

//>yarn add express --save

// Define port

const port = 8080

// Import the express module

const express = require('express') // Express library for creating the server

//Create rest object

const app = express()

app.listen(port, () => {

  console.log(`Server is running on port ${port}`)

})

**Eg 1** Logging Incoming Requests

**Problem Definition:** Demonstrate to log HTTP method and URL for each incoming request using application-level middleware.

// Application-level middleware to log the HTTP method and URL of each incoming request

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

  console.log(`${req.method} ${req.url}`) // Log the request method and URL

  next() // Pass control to the next middleware or route handler

})

app.get('/', (req, res) => {

  res.send('Hello, World!') // Respond with 'Hello, World!' when the root URL is accessed

})

/\*

    Steps to Execute:

    >node server

    Open a browser or Postman and navigate to `http://localhost:8080/`.

    Check your terminal to see the HTTP method and URL being logged for the request.

\*/

**Eg 2** Setting a Custom Header for All Responses

**Problem Definition:** To set a custom header (X-Powered-By) for every HTTP response using application-level middleware.

// Application-level middleware to set a custom header for all responses

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

  res.setHeader('X-Powered-By', 'Express') // Set custom header 'X-Powered-By'

  next() // Pass control to the next middleware or route handler

})

app.get('/', (req, res) => {

  res.send('Hello, World!') // Respond with 'Hello, World!'

})

/\*

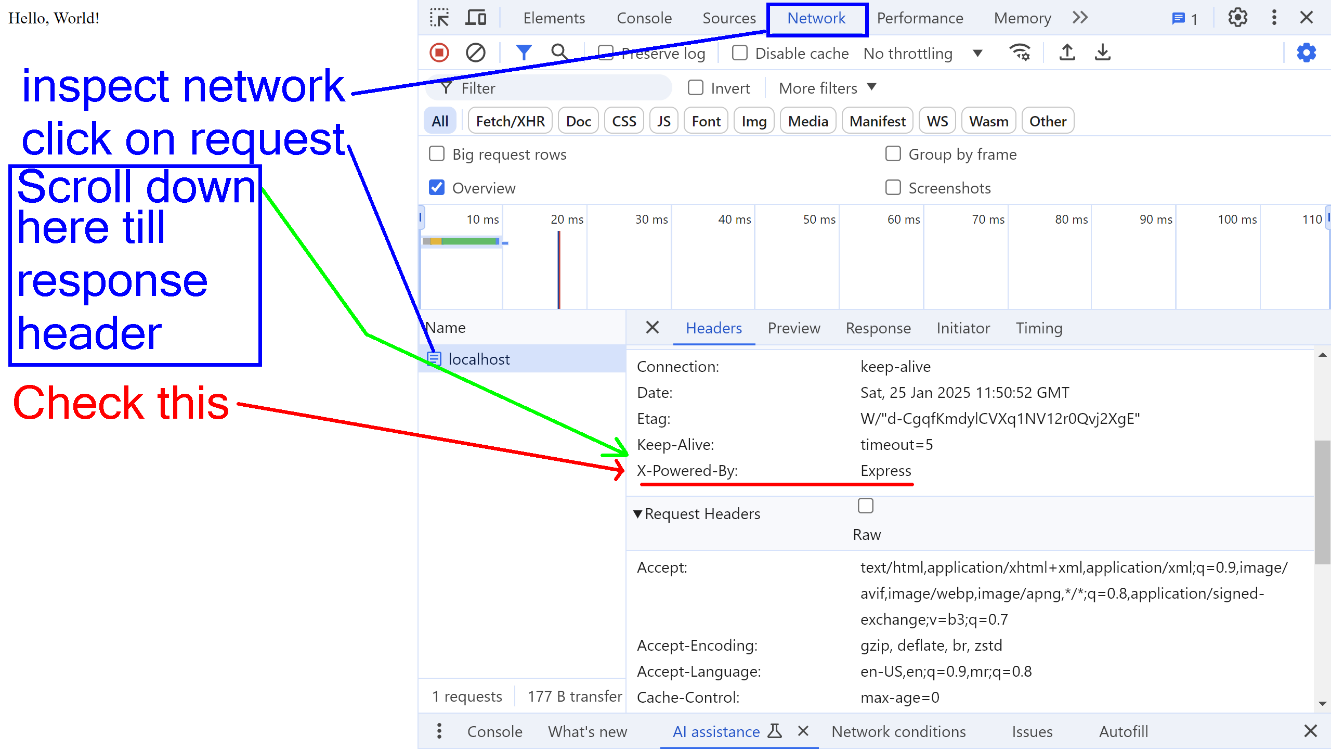
Steps to Execute:

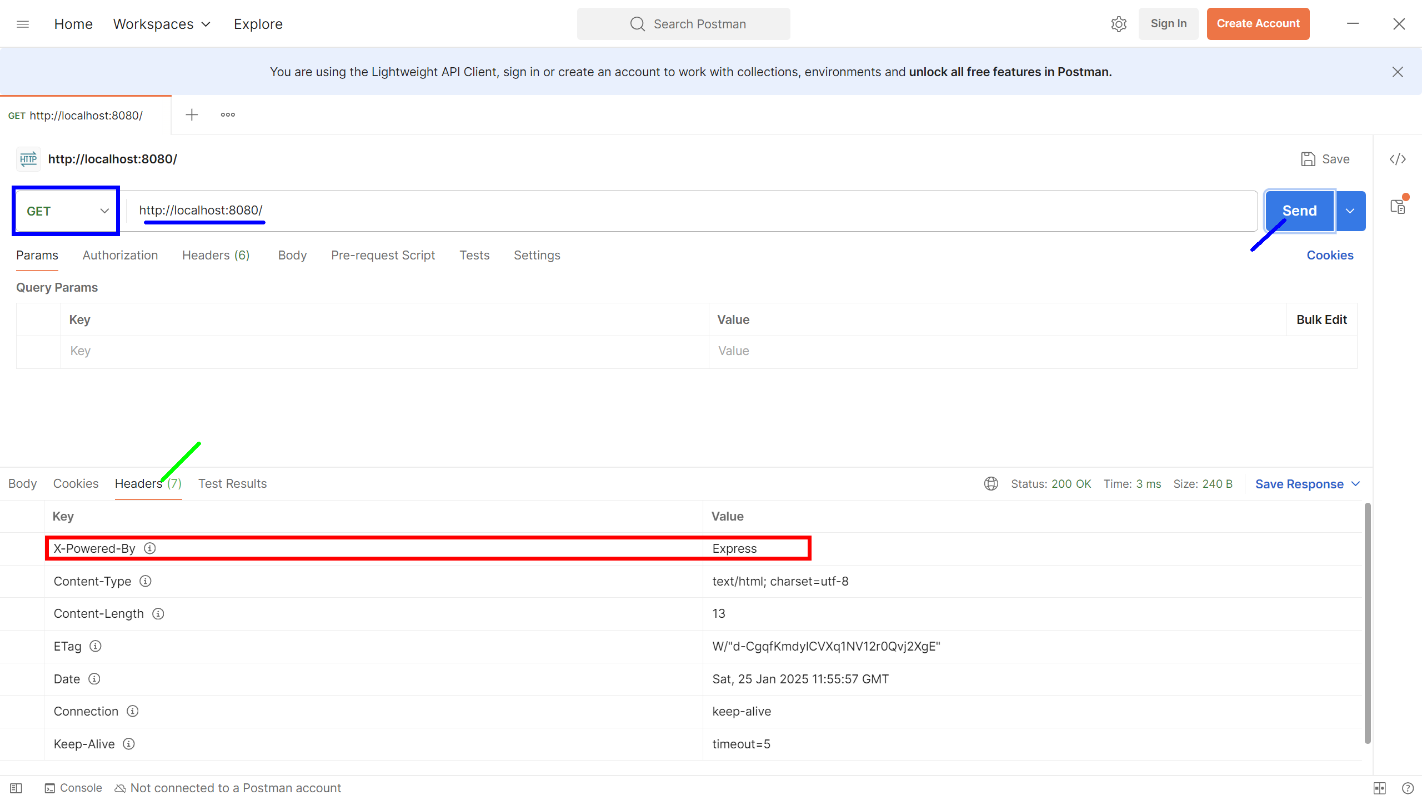
    >node server

    Open a browser or Postman and navigate to `http://localhost:8080/`.

    Inspect the response headers in the browser's developer tools or in Postman to see the `X-Powered-By: Express` header.

\*/





**Eg 3** Authentication for All Routes

**Problem Definition:** Add a basic authentication middleware for all routes, blocking access if no Authorization header is provided.

// Application-level middleware to require authentication for all routes

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

    if (!req.headers.authorization) { // Check if authorization header exists

        return res.status(403).send('Forbidden') // Respond with Forbidden if no authorization

    }

    next() // Pass control to the next middleware or route handler

})

app.get('/', (req, res) => {

    res.send('Hello, World!')

})

/\*

Steps to Execute:

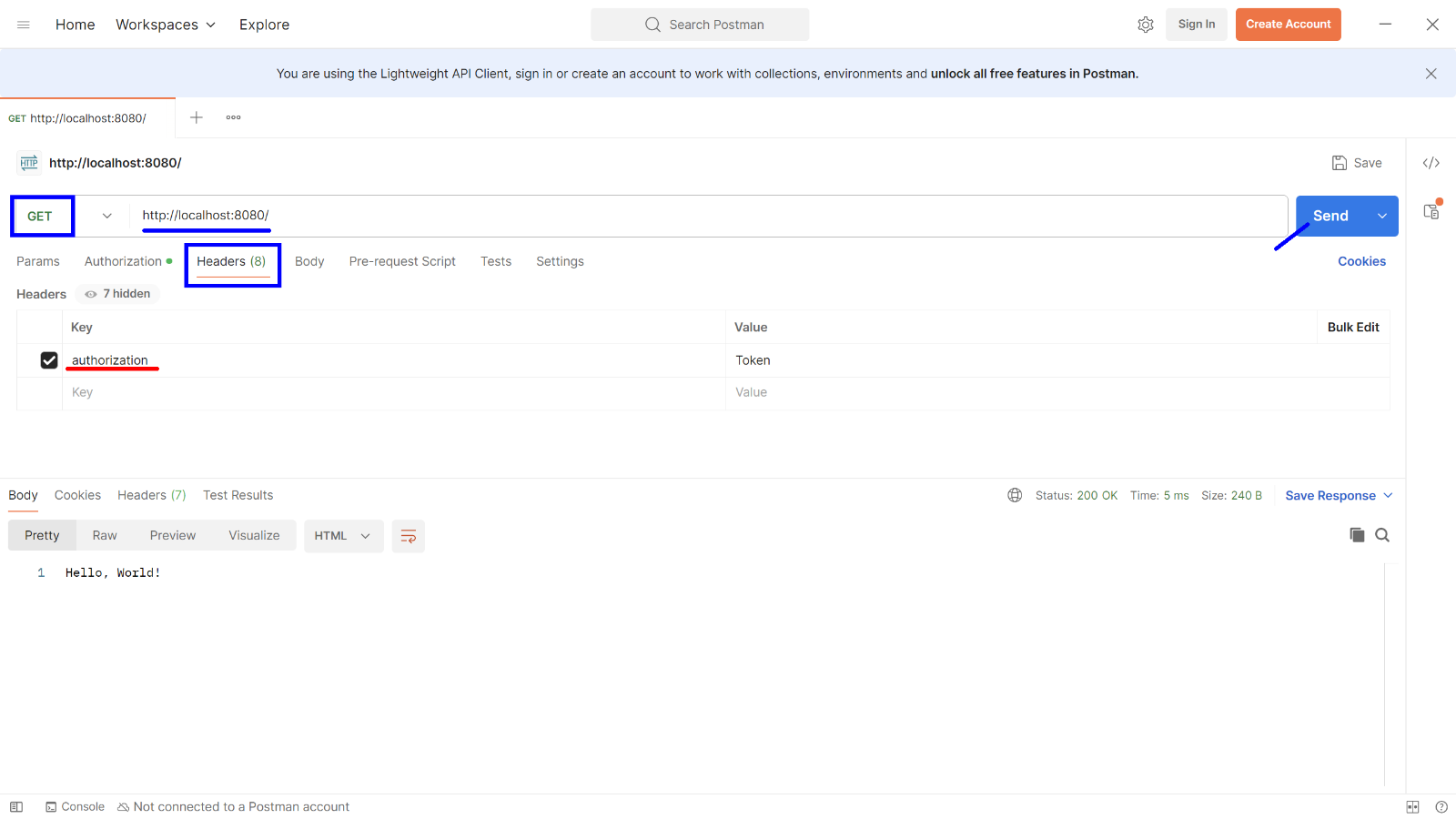
    >node server

    Open Postman and make a GET request to `http://localhost:8080/`.

    - Without any `Authorization` header, the response should be `403 Forbidden`.

    - Add an `Authorization` header with any value to see the successful response.

\*/



**Eg 4** Request Body Parsing

**Problem Definition:** To use application-level middleware to parse incoming JSON request bodies.

// Application-level middleware to parse incoming JSON request bodies

app.use(express.json()) // Parse JSON body

app.post('/', (req, res) => {

  res.send(`Received data: ${JSON.stringify(req.body)}`) // Respond with received data

})

/\*

Steps to Execute:

    >node server

    Open Postman, set the request method to `POST`, and use the URL `http://localhost:8080/`.

    In the "Body" section, select `raw`, choose `JSON`, and add data like:

    {

        "name": "Chitkara University",

        "department": "CSE"

    }

\*/

1. **Router-level Middleware**

//initialyse project

//>npm init -y

//download express module

//>yarn add express --save

//Define port

const port = 8080

// Import the express module

const express = require('express') // Express library for creating the server

//Create rest object

const app = express()

// Create a router instance

const router = express.Router()

app.listen(port, () => {

  console.log(`Server is running on port ${port}`)

})

**Eg 1** Logging Requests to Specific Routes

**Problem Definition:** To apply router-level middleware to log requests to specific routes (e.g., `/api/users`).

// Router-level middleware to log requests to "/api" routes

router.use((req, res, next) => {

  console.log('Request to /api route:', req.method, req.url)

  next() // Pass control to the next middleware or route handler

})

router.get('/users', (req, res) => {

  res.send('User list') // Respond with a list of users

})

app.use('/api', router) // Apply the router to all routes starting with "/api"

/\*

    Steps to Execute:

    >node server

    Open Postman or your browser and navigate to `http://localhost:8080/api/users`.

    Check your terminal for the log output of the HTTP request.

\*/

**Eg 2** Authorization for User Routes

**Problem Definition:** To implement route-specific authentication for routes related to users.

// Router-level middleware to check authentication for "/users" routes

router.use('/users', (req, res, next) => {

  if (!req.headers.authorization) { // Check if user is authenticated

    return res.status(401).send('Unauthorized') // Respond with Unauthorized if not authenticated

  }

  next() // Pass control to the next middleware or route handler

})

router.get('/users', (req, res) => {

  res.send('User list') // Respond with a list of users

})

app.use('/api', router) // Apply the router to all routes starting with "/api"

/\*

    Steps to Execute:

    >node server

    Open Postman and navigate to `http://localhost:8080/api/users`.

    Without an authentication middleware, you’ll get a `401 Unauthorized` response unless authentication is set up.

\*/

**Eg 3** Caching Data for Routes

**Problem Definition:** To implement a simple caching mechanism at the router level for a /products route.

// Simple in-memory cache object

const cache = {}

// Router-level middleware to cache data for "/products" routes

router.use('/products', (req, res, next) => {

  if (cache[req.url]) { // Check if the request URL is cached

    return res.json(cache[req.url]) // Serve cached data

  }

  next() // Continue to the next handler if no cache exists

})

router.get('/products', (req, res) => {

  const products = [{ id: 1, name: 'Product 1' }]

  cache[req.url] = products // Cache the response data

  res.json(products) // Send the response

})

app.use('/api', router) // Apply the router to routes starting with "/api"

/\*

    Steps to Execute:

    >node server

    Open Postman and send a GET request to `http://localhost:8080/api/products`.

    The first request will fetch and cache the data. Subsequent requests (if any) will serve the cached data.

\*/

**Eg 4** Limiting Number of Requests

**Problem Definition:** implement rate limiting for a /data route, allowing a limited number of requests.

let requestCount = 0

const RATE\_LIMIT = 5

// Router-level middleware to limit requests for "/api" routes

router.use((req, res, next) => {

  requestCount++ // Increment request counter

  if (requestCount > RATE\_LIMIT) { // Check if the rate limit is exceeded

    return res.status(429).send('Too many requests') // Respond with Too many requests

  }

  next() // Continue to the next handler

})

router.get('/api/', (req, res) => {

  res.send('Data accessed')

})

app.use(router) // Apply the router

/\*

    Steps to Execute:

    >node server

    Open Postman and send multiple GET requests to `http://localhost:8080/api/`.

    After reaching the rate limit (5 requests), you will receive a `429 Too many requests` response.

\*/

1. **Error-handling Middleware**

//initialyse project

//>npm init -y

//download express module

//>yarn add express --save

//Define port

const port = 8080

// Import the express module

const express = require('express') // Express library for creating the server

//Create rest object

const app = express()

app.listen(port, () => {

    console.log(`Server is running on port ${port}`)

})

**Eg 1** Basic Error Handling Middleware

**Problem Definition:** The server might encounter unexpected errors, so we need to handle them gracefully.

// Define a route that triggers an error

// Simulate an error by passing it to the next middleware

app.get('/', (req, res, next) => {

    const error = new Error('Something went wrong') // Simulate an error

    next(error) // Pass the error to the next middleware

})

// Error-handling middleware function

// Logs the error stack and sends a 500 error response

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

    console.error(err.stack) // Log the error stack trace for debugging

    res.status(500).send('Something went wrong!') // Respond with a 500 status and error message

})

/\*

    Steps to Execute:

    >node server

    Visit http://localhost:8080/ to simulate the error

\*/

**Eg 2** 404 Error Handling Middleware

**Problem Definition:** If a route is not found, the server should return a 404 error.

// Define a valid route for "/"

app.get('/', (req, res) => {

    res.send('Home Route') // Respond with a message for valid requests

})

// 404 Error-handling middleware (handles non-existing routes)

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

    res.status(404).send('Page not found') // Respond with 404 for invalid URLs

})

/\*

    Steps to Execute:

    >node server

    in postman

        Visit http://localhost:8080/ for the valid route

        Visit http://localhost:8080/invalid-route for a 404 error

\*/

**Eg 3** Async Error Handling Middleware

**Problem Definition:** Handling asynchronous errors, such as those in promises or async functions.

// Async route that might throw an error

// Simulating an asynchronous operation that throws an error

app.get('/', async (req, res, next) => {

    try {

        // Simulating an asynchronous operation that throws an error

        const result = await Promise.reject('Something went wrong')

        res.send(result)

    } catch (error) {

        next(error) // Pass the error to the error-handling middleware

    }

})

// Error-handling middleware for catching async errors

// Logs the error and sends a 500 status response

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

    console.error(err) // Log the error for debugging

    res.status(500).send('Internal Server Error') // Respond with a 500 status code

})

/\*

    Steps to Execute:

    >node server

    Visit http://localhost:8080/ to trigger the async error

\*/

1. **Third Party Middleware**

//initialyse project

//>npm init -y

//download express module

//>yarn add express --save

//Define port

const port = 8080

// Import the express and morgan module

const express = require('express') // Express library for creating the server

//Create rest object

const app = express()

app.listen(port, () => {

    console.log(`Server is running on port ${port}`)

})

**Eg 1** Morgan (HTTP Request Logging Middleware)

**Problem Definition:** We need to log incoming HTTP requests to monitor traffic and debug issues.

// Use morgan for HTTP request logging in 'dev' format

// This will log concise info about each request

app.use(morgan('dev')) // Logs requests with concise info

// Define a simple route for testing

app.get('/', (req, res) => {

    res.send('Hello World') // Respond to a GET request with a message

})

/\*

    Steps to Execute:

    >node server

    Visit   http://localhost:8080/

            AND

            http://localhost:8080/hello to see the logs in the console

\*/

**Eg 2** CORS Middleware (Cross-Origin Resource Sharing)

**Problem Definition:** Allowing cross-origin requests from a different domain or port.

const cors = require('cors')

//Create rest object

const app = express()

// Use cors middleware to enable cross-origin requests

// This will allow requests from any domain

app.use(cors()) // This enables CORS for all incoming requests

// Define a simple route for testing

app.get('/', (req, res) => {

  res.send('CORS is enabled') // Respond with a message

})

/\*

    Steps to Execute:

    >node server

    Visit http://localhost:8080/ to confirm CORS is working

\*/

**Eg 3** Helmet Middleware (Security Headers)

**Problem Definition:** Adding security headers to protect against various attacks (e.g., XSS, clickjacking).

const helmet = require('helmet')

//Create rest object

const app = express()

// Use helmet middleware to automatically set secure HTTP headers for the app

app.use(helmet())

// Define a route handler for the root URL ('/') which sends 'Hello World!' as the response

app.get('/', (req, res) => {

    // Send a 'Security applied' message when the root URL is accessed

    res.send('Security applied')

})

/\*

    Steps to Execute:

    >node server

    Visit http://localhost:8080/ to confirm Security is working

    What is observation in Postman / Browser?

Check number response headers with and without helmet

\*/

**Eg 4** cookie-parser Middleware

**Problem Definition:** Parse cookies from the incoming requests and make them accessible.

// Use cookie-parser middleware

app.use(cookieParser())

// Route to read cookies

app.get('/cookies', (req, res) => {

    const cookies = req.cookies

    res.json({ cookies })

})

// Route to set a cookie

app.get('/set-cookie', (req, res) => {

    res.cookie('user', 'Username')

    res.send('Cookie has been set')

})

// Route to clear a cookie

app.get('/clear-cookie', (req, res) => {

    res.clearCookie('user') // Clear the 'user' cookie

    res.send('Cookie has been cleared')

})

/\*

    Steps to Execute:

    >node server

    Visit http://localhost:8080/set-cookie to set a cookie

    Visit http://localhost:8080/cookies to view the cookies

    Visit http://localhost:8080/clear-cookie to clear the 'user' cookie

\*/