UNIT-4: Express JS

Introduction

Express.js is a small framework that works on top of Node.js web server functionality to simplify its APIs and add helpful new features. It makes it easier to organize your application's functionality with middleware and routing. It adds helpful utilities to Node.js HTTP objects and facilitates the rendering of dynamic HTTP objects.

Features

- Develops Node.js web applications quickly and easily.
- It's simple to set up and personalize.
- It can be used to design single-page, multi-page and hybrid web applications.
- Allows you to define application routes using HTTP methods and URLs.
- Includes a number of middleware modules that can be used to execute additional requests and responses activities.
- Simple to interface with a variety of template engines, including Jade, Vash, and EJS.
- Allows you to specify a middleware for handling errors.

Using Express.js, one can create REST API 5 times faster & using less Code.

Environment Setup

Node Package Manager(npm)

npm is the package manager for node. The npm Registry is a public collection of packages of open-source code for Node.js, front-end web apps, mobile apps, robots, routers, and countless other needs of the JavaScript community. npm allows us to access all these packages and install them locally. You can browse through the list of packages available on npm at npmJS.

How to use npm?

There are two ways to install a package using npm: globally and locally.
Globally − This method is generally used to install development tools and CLI based packages. To install a package globally, use the following code.

npm install -g <package-name>

Locally − This method is generally used to install frameworks and libraries. A locally installed package can be used only within the directory it is installed. To install a package locally, use the same command as above without the **-g** flag.

npm install <package-name>

Whenever we create a project using npm, we need to provide a **package.json** file, which has all the details about our project. npm makes it easy for us to set up this file. Let us set up our development project.

Step 1 − Start your terminal/cmd, create a new folder named Express and cd (create directory) into it

- mkdir Express
- cd Express
- code Filename

Step 2 − Now to create the package.json file using npm, use the following code.

npm init

It will ask you for the following information.

```
{ About to write to D:\FSD-2\pkp\package.json:
{
  "name": "pkp",
  "version": "1.0.0",
  "description": "First express",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Khushbu Patel",
  "license": "ISC"
}
```

Note: By using **npm init -y** it will set automatic values in above fields.

*** proxy if required:** npm config set proxy http://192.168.10.252:808

Step 3 − Now we have our package.json file set up, we will further install Express. To install Express and add it to our package.json file, use the following command −

npm install express --save

The **--save** flag can be replaced by the **-S** flag. This flag ensures that Express is added as a dependency to our **package.json** file. This has an advantage, the next time we need to install all the dependencies of our project we can just run the command *npm install* and it will find the dependencies in this file and install them for us.

This will generate package-lock.json file.

This is all we need to start development using the Express framework.

Command to check version of express: **npm list express**

Routing

For routing with different pages, it is too easy than Node.js. There is no need to write any if-else ladder to compare requested page.

Step-1 Import Express

- const expr=require("express")
- const app=expr();

The first line imports Express in our file, we have access to it through the variable Express. We use it to create an application and assign it to var app.

Step-2

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.

The following examples illustrate defining simple routes.

Method -1 : app.get(route, callback)

This function tells what to do when a **get** request at the given route is called. The callback function has 2 parameters, ***request(req)*** and ***response(res)***.
The request **object(req)** represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc. Similarly, the response object represents the HTTP response that the Express app sends when it receives an HTTP request.

Respond with Hello World! on the homepage:

```
app.get('/', (req, res) => {
  res.send('Hello World!')
})
```

(**callback**- An asynchronous function that is called when the server starts listening for requests.).

Method 2: app.post(route, callback)

Respond to POST request on the root route (/), the application's home page:

```
app.post('/', (req, res) => {
  res.send('Got a POST request')
})
```

Note:
The **app.all()** function is used to route all types of HTTP requests. Like if we have POST, GET, PUT, DELETE, etc, requests made to any specific route, let's say */user*, so instead of defining different APIs like app.post('/user'), app.get('/user'), etc, we can define single API **app.all('/user')** which will accept all type of HTTP request.

Step-3 Response methods

The methods on the response object (res) in the following table can send a response to the client, and terminate the request-response cycle.

res.set("content-type", "text/html")- To set MIME as content-type of a browsing page.
res.write("string") function to write something on browser.
res.send()

- **res.set()**
  This function is used to set the response HTTP header field to value. To set multiple fields at once, pass an object as the parameter.

  Syntax: res.set(field , value)

  Parameters: The field parameter is the name of the field and the value parameter is the value assigned to the field parameter.
  Return Value: It returns an Object.

- **res.send()**
  - Since send( ) function can be called once and its like response is sent to client, now nothing can be happened. So after res.send() method, other send() or write() method can not be used. send() method is equivalent to end() method of Node.JS.
  - If you pass in a string, it sets the Content-Type header to text/html .
  - if you pass in an object or an array, it sets the application/json Content-Type header, and parses that parameter into JSON.
  - send() automatically sets the Content-Length HTTP response header.
  - send() also automatically closes the connection.

Response methods

The methods on the response object (res) in the following table can send a response to the client, and terminate the request-response cycle

Method	Description
res.end()	End the response process.
res.json()	Send a JSON response and close connection*.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.

Step-4

app.listen(port, [host], [backlog], [callback])

This function binds and listens for connections on the specified host and port. Port is the only required parameter here.

Sr No.	Argument & Description
1	**port** A port number on which the server should accept incoming requests.
2	**host** Name of the domain. You need to set it when you deploy your apps to the cloud.
3	**backlog** The maximum number of queued pending connections. The default is 511.
4	**callback** An asynchronous function that is called when the server starts listening for requests.

*** In Node JS we learnt http module to route on different pages. (Ref of Node*)**

```
var http=require("http");
var server=http.createServer( (req,res) =>{
   if(req.url=="/")
   {  res.writeHead(200,{"content-type":"application/json"});
      res.write("<h1>Thank you..!</h1>");
      res.end();    }
else if(req.url=="/student"){
   res.writeHead(200,{"content-type":"text/plain"}); //plain shows code as it is
   res.write("<i> Home page1 </i>");
   res.end();}});
server.listen(6001);
```

How the Express Works?

Example

p1.js

```
const expr = require("express");
const app = expr();
app.get ("/", (req,res)=>
{
   res.set ("content-type","text/plain");
   res.send ("<h1>Hello</h1>");
});
app.get ("/about", (req,res)=>
{
   res.set ("content-type","text/html");
   res.write ("Hello");
   res.send ();
   //res.write ("hello");
});
app.listen (5504,()=>
{    console.log ("server started");
})
```

Output

In terminal > node p1.js

Console >>

server started

In browser >>

- localhost:5504
  <h1>Hello</h1>
- localhost:5504/about
  Hello

Note: Like node JS we cannot set headers after they are sent to the client. Means if we send response by writing in "res.write()" then "res.send()" must be blank.

// Res.write() after the res.send() will generate an error in terminal "write after end". (Commented line)

Basic Note:

- **res.end:** comes from NodeJS core. In Express JS if you need to end request in a quick way and do not need to send any data then you can use this function.
- **res.send:** Sends data and end the request.
- **res.json:** Sends data in JSON format and ends the request.
- **res.json():** allows for extra formatting of the JSON data - if this is not required **res.send()** can also be used to return a response object using Express. Both of these methods also end the response correctly, and there's no further action required.

Route parameters

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the **req.params** object, with the name of the route parameter specified in the path as their respective keys.

To use the dynamic routes, we should provide different types of routes. Using dynamic routes allows us to pass parameters and process based on them.

Here is an example of a dynamic route:

```
var express = require('express');
var app = express();
app.get('/:id', (req, res)=>{
 res.send('The id you specified is ' + req.params.id);
});
app.listen(3000);
```

To test this go to http://localhost:3000/123. The following response will be displayed.
You can replace '123' in the URL with anything else and the change will reflect in the response.

Note : To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

```
var expr = require("express");
var app = expr();
app.get('/user/:userId/test/:test', (req, res) => {
    req.params;
    res.send(req.params);
  });
  app.listen(5000)
```

Output: { "userId": "342", "test": "889" }
Request URL: http://localhost:5000/user/342/test/889
Output due to req.params: { "userId": "342", "test": "889" }

Also apply route path in app.get with proper URL

Task-1
Route path: '/things/:name/:id'
Request URL : http://localhost:3000/things/practice/12345.
Output:{“name”: “practice” , “id”:12345}

Task-2
Route path: /flights/:from-:to
Request URL: http://localhost:3000/flights/ADI-NYC
Output: { "from": "ADI", "to": "NYC" }

Json Response Passing

We can define a JSON object and directly pass it inside res.send(). For this, JSON.stringify() method is not required. To send JSON object in res.write(), convert JSON object to string using JSON.stringify().

Example : Write Express JS script to request server to display json object on browser.

Send JSON object using res.write	**Send JSON object using res.send**

const expr=require("express")
const app=expr();
student={name:"LJU",age:28}
app.get("/",(req,res)=>{
res.set("content-type", "application/json")
res.write(JSON.stringify(student))
res.send()
})
app.listen(6007)
	const expr=require("express")
	const app=expr();
	student={name:"LJU",age:28}
	app.get("/",(req,res)=>{
	res.set("content-type","application/json")
	res.send(student)//res.send automatically converts data in string format
	})
	app.listen(6007)

Output: {"name":"LJU","age":28}	**Output:** {"name":"LJU","age":28}

You can send JSON to the client by using **res.json()** , a useful method.

It accepts an object or array, and converts it to JSON before sending it:

res.json(student) in above case also work. It will send data and close the connection.

Example : Write Express JS script to request server to display json object values on browser.

```
const expr=require("express")
const app=expr();
student={name:"ABC",age:28}
   app.get("/",(req,res)=>{
        res.write(student.name +"\n" + JSON.stringify(student.age))  //student age must be converted to string type before sending to server
                                                or
         //res.write(student.age + " ") //needs to convert age into string by Concatenation.
       res.send()
   })
app.listen(6007)
```

Output in browser: ABC
28

Example : Write Express JS script to request server to display json object (Array of Objects) in table form on browser.

```
const expr=require("express")
const app=expr();
student={
   u1:[{name:"LJU",id:2},
   {name:"LJU1",id:3},
   {name:"LJU2",id:4},
   {name:"LJU3",id:5},
   {name:"LJU4",id:6}]}
      app.get("/student",(req,res)=>{
        res.set("content-type","text/html")
         res.write("<center><table cellspacing='5px' cellpadding='8px'
border='1px solid'><tr><th>Name</th><th>ID</th></tr>")
         for(i of student.u1){
            res.write("<tr><td>" + i.name + "</td>")
            res.write("<td>" + JSON.stringify(i.id) + "</td></tr>")
         }
        res.write("</table></center>")
        res.send()
      })
app.listen(6007)
```

Output:

Name	ID
LJU	2
LJU1	3
LJU2	4
LJU3	5
LJU4	6

Example : Write an express js script to define one JSON array of 3 objects having properties name and age. Short these objects according to age. If user request sorted names in url then all names along with age should be printed according to descending order of age. Also, display these sorted values on “Sort page” and display JSON object on “Home page”.

```
const expr = require("express")
const app = expr();
 //home page
app.get ("/",(req,res)=>{
student = [{name:"abc",age:28},
        {name:"def",age:40},
        {name:"xyz",age:10}]
   res.set ("content-type","text/html")
   res.send (student)
})
//sort page
app.get('/sort', (req, res) => {
const sortedNames=student.sort((a, b) => a.age - b.age);
res.send(sortedNames);
});
```

// Alternate approach of “sort page” More Specific Based on Demand//

```
app.get('/sort', (req, res) => {
student.sort((a, b) => a.age - b.age).reverse();
const sortedage = student.map(user => user.age)
const sortedname = student.map(user=>user.name)
  res.set("content-type","text/html")
        for(k=0;k<student.length;k++){
res.write("<center><h2>"+JSON.stringify(sortedname[k])+"="+
              JSON.stringify(sortedage[k])+"</h2></center>");}
//res.write(JSON.stringify(sortedname)); - Array of name
//res.write(JSON.stringify(sortedage)) – Array of age

res.send()
});
```

Output:

On home page

[{name:"abc",age:28}, {name:"def",age:40}, {name:"xyz",age:10}]

On sort page (*Specific

Def = 40
abc = 28
xyz = 10

Example : Set 404 status code on page not found error. (Ref*)

```
const expr=require("express");
const app=expr();
app.get("/",(req,res)=>
{
res.write("Home Page"); res.send();
})
app.get("/student",(req,res)=>
{
res.write("Student Page"); res.send();
})
app.get("*",(req,res)=>
{
res.status(404).send("page not found");
})
app.listen(8081,()=>{console.log("server started");})
```

How to Link HTML, CSS & JS file in Express JS

To use HTML, CSS, JS files, we can create separate folders to store HTML & CSS files and separate folder to store JS files with other package.json files.

Methods:

express.static(root, [options…])

Express provides a built-in middleware **express.static(folder_name)** to serve static files, such as HTML,images, CSS, JavaScript, etc. Here, folder_name is a root directory to serve static assets.

app.use()

This function is used to mount the specified middleware function(s) at the path which is being specified. It is mostly used to set up middleware for your application.

Syntax: app.use(path, callback)

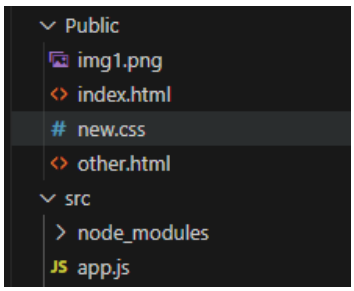
path: It is the path for which the middleware function is being called.
callback: It is a middleware function or a series/array of middleware functions.

You simply need to pass the name of the directory where you keep your static assets, to the **express.static** middleware to start serving the files directly.

For example, if you keep your images, CSS, and JavaScript files in a directory named public, you can do this −

For example: app.use(express.static('../public'));

We will keep a few images and HTML file in **public** sub-directory as follows –




```
var express = require('express');
var app = express();
app.use(express.static( '../public'));
 app.listen(8081, () =>{
   console.log("server start")})
```

Save the above code in a file named **app.js** and run it with the following command.

```
$ node app.js
```

Now open **localhost:8081/img1.png** in any browser and see observe your Image on web page.

Note:

***path.join(paths)[Concept of node]** : This method joins the specified path segments into one path.

**The path that you provide to the express.static function is relative to the directory from where you launch your node process. If you run the express app from another directory, it's safer to use the absolute path of the directory that you want to serve.

***We can also provide a path prefix for serving static files. For example, if you want to provide a path prefix like '/static'.

app.use('/static', express.static('public')).

******__dirname** is an environment variable that tells you the absolute path of the directory containing the currently executing file.

Folder structure

All Files are in One Folder	Industry Approach	All Logic files are in src folder	Non Logic files are in public folder
Express **|-----1.js** **|-----index.html** **|-----1.css** **|-----1.png**	**Express** **|-------src** **|---------|---1.js** **|-------public** **|---------|---index.html** **|---------|---1.css** **|---------|---1.png**	**Express** **|------src** **|--------|--1.js** **|---index.html** **|---1.css** **|---1.png**	**Express** **|---1.js** **|-----public** **|------|---index.html** **|------|---1.css** **|------|---1.png**
Code			
app.use (expr.static ('./')) Using "_ _ dirname" **app.use (expr.static (__dirname))**	**app.use (express.static('../public'))** **Here we can also join the path** **const sp = path.join (__dirname ,"../public");** **app.use (expr.static(sp));**	**app.use (express.static("../"))** **Here we can also join the path** **const sp = path.join (__dirname ,"../");** **app.use(expr.static(sp));**	**app.use (expr.static("public"))**

Example-1: Create one folder for front end and one folder for the backend programming.

/public/index.html

```
<html>
   <head>
      <link rel="stylesheet" href="express.css">
   </head>
   <body>
      <p class="lj_p"> File name must be index.html </p>
   </body></html>
```

/public/express.css

```
.lj_p {
   font-size:100;
   color:blueviolet;
}
```

/src/1.js

```
const expr = require ("express")
const app = expr();

// From here it will load HTML file from public  folder but name of file must be index.html

app.use(expr.static("../public"))
app.listen (5200)
```

Output in browser:

File name must be index.html

//Another way to load html file using sendFile() and path.join()

```
const expr = require ("express")
const path = require ("path");
const app = expr();
const staticpath = path.join (__dirname,"../public");
app.get('/',(req,res)=>{
res.sendFile(staticpath +"/index.html”);
})
app.listen (5200)
```

Output: File name must be index.html (without CSS)

Note: Here css file **will not be loaded** automatically. So, app.use() method is preferred where dependency stack is present in files.

Example 2 (Different File Name)

/public/2.html

```
<html> <head>
      <link
rel="stylesheet"
href="express.css">
   </head>
   <body>
      <p
class="lj_p">File
name can be
different.</p>
   </body>
</html>
```

/public/express.css

```
.lj_p {
   font-size:100;
   color:blueviolet;
}
```

/src/2.js

```
const expr = require ("express")
const path = require ("path");
const app = expr();
app.use
(expr.static("../public",{index:"other.html"}))
// If name is different then index.html
app.listen(5200)
```

Output in browser:

File name can be different.

Example to understand concept of all files in different folders.

Express
|-------src
|---------|---app.js
|-------html
|---------|---1.html
|-------css
|---------|---1.css
|-------image
|---------|--- 1.png

express/src/app.js

```
const express = require("express");
const app = express();
app.use(express.static("../css"))
app.use(express.static("../img"))
app.use(express.static("../html",{index:"1.html"}))
app.listen(5006)

//const port= 5006
//app.listen(port,()=>console.log(`http://localhost:${port}`))
```

express/html/1.html

```
<html>
   <head>
<link rel="stylesheet" href="1.css">
   </head>
   <body>
      <p class="p"> Hello </p>
      <img src="1.png">
   </body>
</html>
```

express/css/1.css

```
.p{
   font-size: 30px;
   color:tomato;
   text-align: center;
}
img{ width:400px; height:auto;}
```

HTTP Methods- GET, POST, Middleware

The HTTP method is supplied in the request and specifies the operation that the client has requested. GET and POST both are two common HTTP requests used for building REST API's. GET requests are used to send only limited amount of data because data is sent into header while POST requests are used to send large amount of data because data is sent in the body.

Express.js facilitates you to handle GET and POST requests using the instance of express.

GET method

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect. Get method facilitates you to send only limited amount of data because data is sent in the header. It is not secure because data is visible in URL bar. It facilitates you to send only limited amount of data because data is sent in the header. It is not secure because data is visible in URL bar.

Example: Write express script to get form data using get method and display data in JSON format in next page named “process_get”

index.html

```
<form method="get" action="/process-get">
<lable> User Name</lable>
<input type="text" name="firstname"/>
</br>
<lable> Password</lable>
<input type="password" name="password"/>
<input type="submit">
</form>
```

Trail.js (Both Trail.js and index.html is in one folder)

```
var express = require('express');
var app = express();
app.use(express.static(__dirname))
app.get('/process-get', (req, res)=> {
response = {
      Username:req.query.firstname,
      Password:req.query.password
   };
   console.log(response);
   res.send(response);
})
app.listen(8000)
```

Trail2.js (Trail2.js is in src folder and index1.html is in public folder)

```
const expr=require("express")
const app=expr();
   app.use(expr.static("../public",{index:"index1.html"})); // If name is different than
index.html
   app.get("/process-get",(req,res)=> {
      const response = {
         fname:req.query.firstname,
         pass:req.query.password,
      };
      console.log(response)
      res.send(response)
   })
   app.listen(5000)
```

Example: Write express js script to print message in next line splitting by "." And use get method to submit the data. HTML file contains form of text area for the message and submit button.

public/2.html

```
  <form method="get" action="/login">
      <textarea rows="10" cols="10" name="message">Hi.Hello.how are you
      </textarea>
      <button name="Submit" value="Submit">Submit</button>
   </form>
```

src/2.js

```
var expr = require("express");
var app = expr();
app.use(expr.static("../public",{index: '2.html'}));
app.get("/login",(req,res)=>{
   res.set("content-type","text/html");

   t1 = (req.query.message).split(".");
   for(i in t1){
      res.write(t1[i]+ "</br>");
   }
   res.send();
});
app.listen(5121,()=>{
   console.log("server start");
});
```

POST method

The POST method requests that the server accept the data enclosed in the request as a new object/entity of the resource identified by the URI. Post method facilitates you to send large amount of data because data is sent in the body. Post method is secure because data is not visible in URL bar.

Express.urlencoded()

The **urlencoded()** function is a built-in middleware function in Express.
It parses incoming requests with URL-encoded payloads and is based on a body parser.

- Express.urlencoded() expects request data to be sent encoded in the URL, usually in strings or arrays **{extended:Boolean Value}**

The extended option allows to choose between parsing the URL-encoded data with the querystring library (when false) or the qs library (when true).

Defaults to true, but using the default has been deprecated.

***Note:
• New version of express contains built-in body-parser module(As it is third party module). • For Express vesion of 4.0 above We don't need to install it and import it to our program. • We can access it's functionality by just adding below line to our program.

```
var expr = require("express");

app.use(expr.urlencoded());
```

Installation * (Only required in old version)**

npm install body-parser

API

```
var bodyParser = require('body-parser')

app.use(bodyParser.urlencoded({ extended: false }));
```

- Body-parser parses is an HTTP request body that usually helps when you need to know more than just the URL being hit.
- It provides four express middleware for parsing JSON, Text, URL-encoded, and raw data sets over an HTTP request body.
- Using body-parser allows you to access req.body from within routes and use that data.

Example: Write express script to get form data using post method and display data in JSON format in next page named “process”.

public/p.html

```
<html>
<body>
<form action="/process" method="post">
First Name: <input type="text" name="first_name">  <br>
Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

src/p.js

```
var express = require('express');
var app = express();

app.use(express.urlencoded({ extended: false }));

app.use(express.static(“../public”,{index: 'p.html'}));

app.post('/process', function (req, res) {
  res.send(req.body);
})

app.listen(7002)
```

Example: Write express js script to perform the tasks as asked below.

1) Create one HTML file named ljform.html and add one form which contains username, password and submit button. Data should be submitted by HTTP post method.

2) Submit button is of black color with white text. (External CSS)

3) On home page form should be displayed and while submitting the form, on next page named “/login” if username is admin then it will display “Welcome admin” else display “Please login with Admin name”.

src/3.js

```
var expr = require("express");
var app = expr();

app.use(expr.urlencoded());
app.use(expr.static(‘../public’,{index: ' ljform.html'}));
app.post("/login",(req,res)=>{
```

```
    if( req.body.uname == 'admin' ){
        res.write(`<h1 style="color:green">Hey ${req.body.uname},Welcome!</h1><br>`)
    }
    else{
        res.write(`<h1 style="color:red">Please login with Admin name</h1>`);
    }
    res.send();
})
app.listen(5222);
```

public/ ljform.html

```
<html>
    <head><title>Post Method</title>
        <link rel="stylesheet" href="1.css">
    </head>
    <body>
        <form method="post" action="/login">
        <div class="lj-div">
            <label>User name:</label>
            <input type = "text" name="uname" ></input>
        </div>
        <div class="lj-div">
            <label>Password:</label>
            <input type = "text" name="password" ></input>
        </div>
        <button name="Submit" value="Submit">Submit</button>
    </form>
    </body>
</html>
```

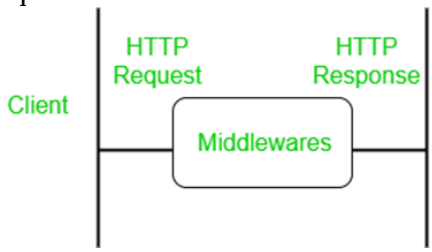
CSS file is also in public folder

```
button{
    padding: 10px;
    background-color: black;
    color: #fff;}
```

Middleware

What is Middleware in Express JS

Express.js is a routing and Middleware framework for handling the different routing of the webpage and it works between the request and response cycle. Middleware gets executed after the server receives the request and before the controller actions send the response.



Middleware is a request handler that allows you to intercept and manipulate requests and responses before they reach route handlers. They are the functions that are invoked by the Express.js routing layer.

It is a flexible tool that helps in adding functionalities like logging, authentication, error handling, and more to Express applications.

Middleware Syntax
The basic syntax for the middleware functions is:

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

- Middleware functions take 3 arguments: the request object, the response object, and the next function in the application's request-response cycle, i.e., two objects and one function.
- Middleware functions execute some code that can have side effects on the app and usually add information to the request or response objects. They are also capable of ending the cycle by sending a response when some condition is satisfied. If they don't send the response when they are done, they start the execution of the next function in the stack. This triggers calling the 3rd argument, next().
- The middle part (req,res,next)=>{} is the middleware function. Here we generally perform the actions required before the user is allowed to view the webpage or call the data and many other functions. So let us create our own middleware and see its uses.

Advantages of using middleware:

- Middleware can process request objects multiple times before the server works for that request.
- Middleware can be used to add logging and authentication functionality.
- Middleware improves client-side rendering performance.
- Middleware is used for setting some specific HTTP headers.
- Middleware helps for Optimization and better performance.

Types of Middleware

Express JS offers different types of middleware and you should choose the middleware on the basis of functionality required.

- **Application-level middleware:** Bound to the entire application using app.use() or app.METHOD() and executes for all routes.

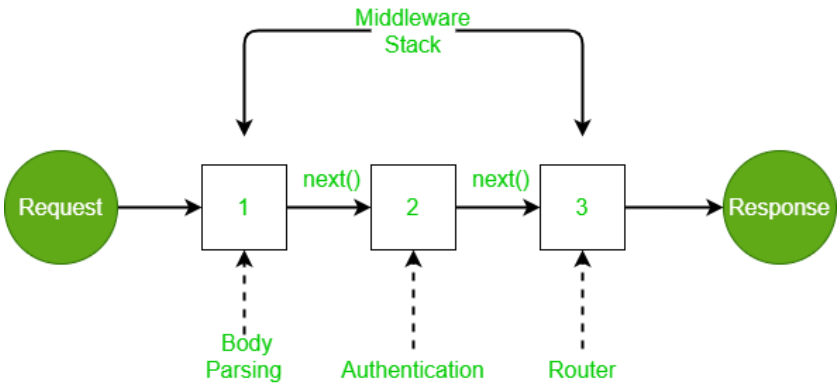
```
For Example: app.use((req, res, next) => {
                console.log('Time:', Date.now())
                 next()
                 })
```

- **Router-level middleware:** Associated with specific routes using router.use() or router.METHOD() and executes for routes defined within that router.
- **Error-handling middleware**: Handles errors during the request-response cycle. Defined with four parameters (err, req, res, next).
- **Built-in middleware**: Provided by Express (e.g., express.static, express.json, etc.).
- **Third-party middleware**: Developed by external packages (e.g., body-parser)

Middleware Chaining: Middleware can be chained from one to another, Hence creating a chain of functions that are executed in order. The last function sends the response back to the browser. So, before sending the response back to the browser the different middleware process the request.

The **next()** function in the express is responsible for calling the next middleware function if there is one.

Modified requests will be available to each middleware via the next function –



Middleware chaining example

In the above case, the incoming request is modified and various operations are performed using several middleware, and middleware is chained using the next function. The router sends the response back to the browser.

Example: Below is example to understand concept of middleware.

```
var expr = require("express");
var app = expr();

//Method 1
const cb=(req,res,next)=>
{
   console.log("Initalized");
   res.set("content-type","text/html")
   res.write("<strong>First</strong>");
   next();
}
const cb1=(req,res,next)=>
{
   res.write("<p>Addition = " + (5+5) + "</p>");
   next();
}
app.use("/ee",cb,cb1);
app.get("/ee",(req,res)=>
{
   res.write("<h1>Hello Welcome to LJU</h1>");
   res.send();
});

//Method 2
app.use(
   "/",
   (req, res, next) => {
     console.log("request received on"+new Date());
     next();
   },
   (req, res, next) => {
      res.set("content-type","text/html")
      res.write("Hello");
      next();
     },
   (req, res) => {
     res.write(`<div>
     <h2>Welcome to LJU</h2>
     <h5>Tutorial on Middleware</h5>
   </div>`);
   res.send();
   }
 );
app.listen(6001)
```

Output: Observe Sequence by running both the methods

Cookie

Cookies are small piece of information i.e. sent from a website and stored in user's web browser when user browses that website. Every time the user loads that website back, the browser sends that stored data back to website or server, to recognize user.

How cookies work

When a user visits a cookie-enabled website for the first time, the browser will prompt the user that the web page uses cookies and request the user to accept cookies to be saved on their computer. Typically, when a makes a user request, the server responds by sending back a cookie (among many other things).

This cookie is going to be stored in the user's browser. When a user visits the website or sends another request, that request will be sent back together with the cookies. The cookie will have certain information about the user that the server can use to make decisions on any other subsequent requests.

For example, **Facebook** from a browser.
When you want to access your Facebook account, you have to log in with the correct credentials. When you first make a login request and the server verifies your credentials, the server will send your Facebook account content. It will also send cookies to your browser. The cookies are then stored on your computer and submitted to the server with every request you make to that website. A cookie will be saved with an identifier that is unique to that user.

When you revisit Facebook, the request you make, the saved cookie, and the server will keep track of your login session and remember who you are and thus keep you logged in.

As explained, cookies are stored in the browser, and no sensitive information can be stored in them. They are typically used to save a user's preferences.

- **cookie-parser** - cookie-parser looks at the headers in between the client and the server transactions, reads these headers, parses out the cookies being sent, and saves them in a browser. In other words, cookie-parser will help us create and manage cookies depending on the request a user makes to the server.

Run the following command to install these NPM packages:

```
npm install cookie-parser
```

We will create a simple example to demonstrate how cookies work.

Step 1 - Import the installed packages

To set up a server and save cookies, import the cookie parser and express modules to your project. This will make the necessary functions and objects accessible.

```
const express = require('express')
const cookieParser = require('cookie-parser')
```

Step - 2 Get your application to use the packages

You need to use the above modules as middleware inside your application, as shown below.

```
//setup express app
const app = express()
// let's you use the cookieParser in your application
app.use(cookieParser());
```

This will make your application use the cookie parser and Express modules.

Step -3 Add/Read/Delete Operation

- **Add/Set Cookie: res.cookie()** method is used for setting the cookie name to value. The value parameter can be a string or an object converted to JSON.

  Syntax: res.cookies("name","value","expiry-date")

- **Display cookie:** To view cookies from your server req.cookies function is used. addthe following console to that route.

  console.log('cookies:' ,req.cookies)

- **Delete cookie** : The **res.clearCookie()** function is used to clear the cookie specified by name. This function is called for clearing the cookies which has already been set. For example, if a user cookie is set, then it can be cleared using this function.

  Syntax:res.clearCookie("cookie-name")

How Long Does a Cookie Store?

The time of expiry of a cookie can be set when the cookie is created. If time is not set by the user then by default, the cookie is destroyed when the current browser window is closed.

Adding Cookies with Expiration Time:-

You can add cookies that expire. Just pass an object with property 'expire' and set to the time when you want it to expire from your browser. Following is an example of this method.

// res.cookie(name, 'value', {expires:new Date(Date.now()+10000)}); Expires after 10000 ms from the time it is set.

The other way to use{ maxAge: oneDay } - this sets the cookie expiry time. The browser will delete the cookie after the set duration elapses. The cookie will not be attached to any of the requests in the future. In this case, we've set the maxAge to a single day as computed by the following arithmetic.

// res.cookie(name, 'value', {maxAge: 360000}); This cookie also expires after 360000 ms from the time it is set.

Example: Add/Read/Delete cookie

```
var expr=require("express")
var app=expr()
var cp=require('cookie-parser')
app.use(cp())
app.get("/",(req,res)=>
{
   res.cookie("fname","Khushbu") //Sets fname = Khushbu
   res.cookie("lname","Patel")
   res.cookie("contact","001100",{expires:new Date(Date.now()+10000)})
     //expires after 10 secs
   res.cookie('email', 'express@gmail.com',{maxAge:2000}); //expires after 2 secs
   res.cookie('city', 'ahmedabad');
res.send("Cookie set");
})
app.get("/cookie",(req,res,next)=> // To print on Console
{
  console.log(req.cookies)
  console.log(req.cookies.fname)
  res.write(JSON.stringify(req.cookies));
  res.write(req.cookies.fname);
  res.send()
})
app.get("/clear",(req,res)=>{          // To print coockie on another page →
localhost:5001/clear
// const cookies = req.cookies
//for (let prop in cookies) {
//     console.log(prop)
//   res.clearCookie(prop); //Or res.cookie(prop, '', {expires: new Date(0)});
// }
   res.clearCookie("fname")
   res.send(req.cookies)   // Use req.cookies method to check the saved cookies . If want to
write coockie using res.write use JSON.stringify
})
app.listen(5001)
```

To confirm that the cookie was saved, go to your browser's Inspect element > select the application tab > cookies > select your domain URL.

To check if your cookie is set or not, just go to your browser, fire up the console, and enter −

console.log(document.cookie);

Example: Write an express js script to set cookies of submitted values of form. Perform following tasks.

- Create a HTML file which contains a form with fields first name, last name, password and a submit button.
- Once form submitted, store all these entered values to the respective cookies on '/next' page.
- Then redirect user to "/admin" page and clear the cookie set for the last name. Display remaining set cookie values on this page. (Using get method)

In form.html file:

```
<form method="get" action="/next">
First Name : <input type = "text" name="uname" ></input>
Last Name : <input type = "text" name="lname" ></input>
Password : <input type = "password" name="password" ></input>
<button name="Submit" value="Submit">Submit</button>
</form>
</body></html>
```

In cookie.js file:

```
var expr=require("express")
var app=expr()
var cp=require('cookie-parser')
app.use(cp())
app.use(expr.static('../public',{index: 'trial.html'}));
app.get("/next",(req,res,next)=>
{
  // const response= {
  //     u:req.query.uname,
  //     p:req.query.password }
  // res.cookie("uname",response.u)
  res.cookie("uname",req.query.uname)
   res.cookie("password",req.query.password)
                // Only for ref*
                //res.cookie("password", req.query.password ,{ secure: true, httpOnly: true })
                //always use true value when you want cookies to be created on an HTTPS s
                //secure origin.
  res.cookie("lname",req.query.lname);
  res.redirect("/admin");  //  alternative of next()*
})
app.get("/admin",(req,res)=>{
  res.clearCookie('lname');
  res.write(" Welcome : " + req.cookies.uname);
  res.write(" Lname : " + req.cookies.lname);
  res.send()
});
app.listen(6959);
```

Session

The difference between session and cookie

Session	Cookies
A session is stored at server side	A cookie is stored at client side
It can store a data ranging between 5mb – 10mb	It can only store a data of 4kb
It is destroyed when user logout.	It is destroyed when user closes the page or it will remain until the defined time.
express-session middleware is required to create a session.	It doesn't require any middleware to create a cookie. Express provide built in support.
Session id is used as a identifier.	Key-value pair data is used as a identifier.
The performance of session is slower due to server interaction.	The performance of cookie is faster, as data is stored locally.
It is more secure as data are stored at server side.	It is less secure as data are stored at client side.
It is used for storing user-specific data.	It is used to store user preference data.

How sessions works

When the client makes a login request to the server, the server will create a session and store it on the server-side.

When the server responds to the client, it sends a cookie. This cookie will contain the session's unique id stored on the server, which will now be stored on the client.

Why Use Sessions?

Sessions are used to store information such as User ID over the server more securely, where it cannot be altered. This prevents the information from being tampered with.

In addition to this, sessions can transfer the information from one web page to another in the form of value.

When the client makes a login request to the server, the server will create a session and store it on the server-side. When the server responds to the client, it sends a cookie. This cookie will contain the session's unique id stored on the server, which will now be stored on the client. This cookie will be sent on every request to the server.

We use this session ID and look up the session saved in the database or the session store to maintain a one-to-one match between a session and a cookie. This will make HTTP protocol connections stateful.

HTTP is stateless; in order to associate a request to any other request, you need a way to store user data between HTTP requests. Cookies and URL parameters are both suitable ways to transport data between the client and the server. But they are both readable and on the client side. Sessions solve exactly this problem. You assign the client an ID and it makes all further requests using that ID. Information associated with the client is stored on the server linked to this ID. Install the using the command:

```
npm install express-session
```

Express-session options and how to use them

To set up the session, you need to set a couple of Express-session options, as shown below.

```
const oneDay = 1000 * 60 * 60 * 24; // creating 24 hours from milliseconds
app.use(sessions({
   secret: "thisismysecrctekey",
   saveUninitialized:true,
   cookie: { maxAge: oneDay },
   resave: false
}));
```

- **secret -** a random unique string key used to authenticate a session. It is stored in an environment variable and can't be exposed to the public. The key is usually long and randomly generated in a production environment.
- **resave -** Forces the session to be saved back to the session store, even if the session was never modified during the request.

  resave = true (Default) - It means when the modification is performed on the session it will re write the req.session.cookie object.

  resave = false -It will not rewrite the req.session.cookie object. the initial req.session.cookie remains as it is.

- **saveUninitialized -** Forces a session that is "uninitialized" to be saved to the store. A session is uninitialized when it is new but not modified. Choosing false is useful for implementing login sessions, reducing server storage usage, or complying with laws that require permission before setting a cookie.

Uninitialised = false - It means that Your session is only Stored into your storage, when any of the Property is modified in req.session

Uninitialised = true (Default) - It means that Your session will be stored into your storage Everytime for request. It will not depend on the modification of req.session.

- **cookie:** { maxAge: oneDay } - this sets the cookie expiry time. The browser will delete the cookie after the set duration elapses. The cookie will not be attached to any of the requests in the future.

Examples: write express script to print how much time user visit the page. For ex., if user visit first time "you have visited page First time" message will print. if user visit second time "you have visited page second time" message will print. And so on.

```
const expr=require("express");
const app=expr();
const sess=require("express-session");
app.use(sess(
   {
      resave:false,
      saveUninitialized:false,
      secret:"pkp123"
   }
));
app.get("/",(req,res)=>
{
   if(req.session.page_views)
   {
      req.session.page_views++;
      res.send(`<h1 style="color:blue;"> You have visited page ${req.session.page_views} times <h1>`);
   }
   else{
      req.session.page_views=1,
      res.send(`<h1 style="color:green;"> Welcome! Thank you for visiting our website!<h1>`);
   }

});
app.listen(8003,()=>
{
   console.log("server running at 8003");
});
```

Examples: write a script to meet following requirements:

1) Create index.html file page and open it on localhost.
2) After clicking submit button, it should jump on “savesession” page. Store username in session.
3) After saving session, redirect to “fetchsession” page and read value. Put a LOGOUT link button here.
4) Jump on delete session page after clicking LOGOUT button.
5) Destroy the session on this page and redirect to index.html page**.**

session.js

```
var expr = require("express");
var app = expr();
var es = require("express-session");
app.use(es({
   resave:false,
   saveUninitialized:false,
   secret:"mykey1"
}));
app.use(expr.static("../public",{index:"trial.html"}))
app.get("/savesession",(req,res)=>{
     var uname=req.query.uname;
      var password=req.query.password;

   req.session.uname = uname;
   req.session.password = password;
   res.redirect("fetchsession")
})
app.get("/fetchsession",(req,res)=>{
   res.write("<h1>Welcome " + req.session.uname +"</h1>")
   res.write("<a href='/deletesession'>Delete Session</a>")
   res.send();
});
app.get("/deletesession",(req,res)=>{
  req.session.destroy();
  res.redirect('/')

//Also try res.clearCookie('connect.sid')
//res.send("Session Cookie cleared")
});
app.listen(6177);
```

session.html

```
<form method="get" action="/savesession">
   <div><input type = "text" name="uname" placeholder="Username" ></input></div>
   <div><input type = "password" name="password" placeholder="Password"></input></div>
   <input class="lj_in" type="submit" name="submit" value="Submit">
</form>
```