Akash Pundir

System Programming I

School of Computer Science and Engineering

Whaaaaaaat is Expresss?

Express.js is a powerful and flexible Node.js ***framework*** *for building web applications and APIs.*

Framework?

A framework is **a software platform that provides a structured and standardized way to build web applications**.

It offers a set of pre-defined elements, tools, and approaches that you can leverage to construct something more complex and specific.

Think of it like **a pre-built toolkit that saves you from starting from scratch every time you need to build something**.

Install Express.js

npm install express

Creating a Simple Express Application

```
const express = require('express');
const app = express();
const port = 3000;

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

app.listen(port, () => {
  console.log(`Server is listening at http://localhost:${port}`);
});
```

Request Methods

- HTTP (Hypertext Transfer Protocol) defines several request methods to indicate the desired action to be performed for a given resource. These HTTP methods define the actions that clients (such as web browsers) can perform on resources hosted on servers.

- **GET**: The GET method is **used to request data from a specified resource**. It sends data in the URL query parameters. This method should only be used for retrieving data and should not have any side effects on the server.

- **POST**: The POST method is **used to submit data to be processed to a specified resource**. It sends data in the request body, which allows for sending larger amounts of data than the URL query parameters. This method is often used for creating new resources on the server or submitting form data.
- **PUT**: The PUT method is **used to update an existing resource or create a new resource if it does not exist at the specified URL**. It sends data in the request body and replaces the entire resource at the specified URL with the new data provided.

- **PATCH**: The PATCH method is **used to apply partial modifications to a resource.** It sends data in the request body to specify the changes that should be applied to the resource.
- **DELETE**: The DELETE method is **used to request the removal of the specified resource**. It sends no data in the request body, and the server is expected to delete the resource identified by the URL.

Write a code demonstrating how Express.js sets up a server to handle **GET** requests at **'/readfile’,** read the content of 'example.txt' and **send it as a response to browser**.

```
const express = require('express');
const fs = require('fs');
const app = express();

// Define a route handler for GET requests to '/readfile'
app.get('/readfile', (req, res) => {
    // Specify the file path
    const filePath = 'example.txt';
    // Read the file asynchronously
    fs.readFile(filePath, 'utf8', (err, data) => {
        if (err) {
            return res.status(500).send('Error reading the file');
        }
        // If successful, send the file content as the response
        res.send(data);
    });
});
// Start the server
app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Test Your Knowledge

Build a simple Express.js server that accepts GET requests containing query parameters representing user information. Your server should extract these query parameters, save them to a file, and respond with a success message.

Let’s perform a write operation in a file with inputs from the form.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Form Submission</title>
</head>
<body>
    <h1>Form Submission</h1>
    <form action="/submit" method="GET">
        <label for="name">Name:</label><br>
        <input type="text" id="name" name="name"><br>
        <label for="email">Email:</label><br>
        <input type="email" id="email" name="email"><br><br>
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

```
const express = require('express');
const fs = require('fs');
const app = express();

// Serve HTML form
app.get('/form', (req, res) => {
    // Read the HTML form file asynchronously
    fs.readFile('public/form.html', 'utf8', (err, data) => {
        if (err) {
            console.log(err);
            // If there's an error reading the file, send an error response
            return res.status(500).send('Error reading the file');
        }

        // If successful, send the HTML form content as the response
        res.send(data);
    });
});
```

```
// Handle form submission
app.get('/submit', (req, res) => {
    // Access data sent through query parameters
    const name = req.query.name;
    const email = req.query.email;
    // Do something with the data (e.g., save it to a file)
    fs.writeFile('data.txt', `Name: ${name}, Email: ${email}\n`, err => {
        if (err) {
            console.error('Error saving data to file:', err);
            return res.status(500).send('Error saving data');
        }
        // Respond with a success message
        res.send('Data saved successfully');
    });
});
// Start the server
app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

ExpressJS - Middleware

Middleware functions are functions that **have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle**. These functions are used to modify req and res objects.

Middleware serves as a bridge between a client request and server response in web applications.

- **Request Processing:** Middleware functions can preprocess requests before they reach the main application logic. This includes tasks like parsing request bodies, extracting cookies or session data, and performing initial authentication checks.

- **Authentication and Authorization:** Middleware is commonly used for authentication, ensuring that only authenticated users can access certain routes or resources. It can also handle authorization, determining whether authenticated users have the necessary permissions to perform specific actions.

- **Logging and Debugging:** Middleware can log request details, such as timestamps, request methods, URLs, and user agents, aiding in debugging and monitoring of applications. Logging middleware can also help track errors or unexpected behavior.

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

// Middleware function
app.use((req, res, next) => {
  console.log('This is a middleware function!');
  next(); // Call the next middleware function in the stack
});

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

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Multiple Middlewares

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

// Middleware function for logging incoming requests
app.use((req, res, next) => {
  console.log(`${new Date().toISOString()}: ${req.method} ${req.url}`);
  next(); // Pass control to the next middleware function or route handler
});

// Middleware function to add a custom header to outgoing responses
app.use((req, res, next) => {
  res.setHeader('X-Custom-Header', 'Hello from Custom Middleware!');
  next(); // Pass control to the next middleware function or route handler
});

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

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Test your Knowledge

- Write a middleware function in Express.js that logs the timestamp, HTTP method, and URL of every incoming request to a file named 'request.log'. Ensure that the log file is created if it doesn't exist and that new log entries are appended to it.

```
const express = require('express');
const fs = require('fs');

const app = express();

// Middleware function to log incoming requests to a file
app.use((req, res, next) => {
  const logFilePath = __dirname + '/request.log';
  const logEntry = `${new Date().toISOString()}: ${req.method} ${req.url}\n`;

  fs.appendFile(logFilePath, logEntry, (err) => {
    if (err) {
      console.error('Error writing to log file:', err);
    }
  });

  next(); // Pass control to the next middleware function or route handler
});
```

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

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

- Express provides built-in middleware for **parsing URL-encoded bodies**, which is commonly used for processing form data submitted in HTTP requests. This middleware is called express.urlencoded().

- When a client submits form data using the application/x-www-form-urlencoded content type, Express parses this data into a JavaScript object and makes it available in the req.body property of the request object.

Coming Up…..

- You are building an Express application that needs to handle form submissions containing URL-encoded data. Using the built-in URL-encoded body parser middleware, write a route handler to handle POST requests to the /submit endpoint. Parse the incoming URL-encoded data and log it to the console.

Hint : `//Middleware for Parsing URL Encoded bodies`

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

```
const express=require('express');
const fs=require('fs');
const app=express();

//Middleware for Parsing URL Encoded bodies
app.use(express.urlencoded({ extended: true }));

//Route Handler for Serving Form
app.get('/',(req,res)=>{
    const readStream=fs.createReadStream('public/index.html');
    readStream.pipe(res);
});

//Route Handler for POST request
app.post('/submit',(req,res)=>{
    console.log(req.body);
    let data=JSON.stringify(req.body);
    const writeStream=fs.createWriteStream('body.json');
    writeStream.write(data);
    writeStream.end();
    res.end("Done");

});

app.listen(3000);
```

Test your Knowledge

- Implement a basic login page using HTML for a web application.
- Serve this login page using Express.js on the /login route.
- Create a middleware function for authentication in Express.js, which validates user credentials against a hardcoded database.
- Define a protected route ‘/profile’ that requires authentication to access. If authenticated, it should display a personalized welcome message.
- Additionally, set up a public route ‘/’ that doesn't require authentication, displaying a generic welcome message.
- Ensure that the authentication middleware restricts access to the /profile route based on the provided credentials.

Let’s first make login page login.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    <form action="/profile" method="post">
        <input type="text" name="username" placeholder="Username">
        <input type="password" name="password" placeholder="Password">
        <button type="submit">Login</button>
    </form>
</body>
</html>
```

Now, Import Necessary Modules and create instance of express.

```
const express = require('express');
const fs = require('fs');

const app = express();
```

Make middleware to parse request bodies and simulate a user database

```
// Middleware to parse URL-encoded bodies
app.use(express.urlencoded({ extended: true }));

// Simulated user database
const users = [
    { id: 1, username: 'John', password: 'Wick' },
    { id: 2, username: 'Babu', password: 'Rao' }
];
```

Middleware for Authentication

```
// Middleware function for authentication
app.use('/profile',(req, res, next) => {
    // Extract username and password from request body
    const { username, password } = req.body;

    // Find user in the database
    const user = users.find(u => u.username === username && u.password === password);

    // If user is found, set it on the request object and proceed to next middleware
    if (user) {
        req.user = user;
        next();
    } else {
        // If user is not found, send 401 Unauthorized response
        res.status(401).send('Unauthorized');
    }
});
```

Now, let’s define public and login route handlers

```
// Public route
app.get('/', (req, res) => {
    res.send('Welcome to the public route!');
});

// Route to serve login form
app.get('/login', (req, res) => {
    fs.readFile(__dirname+'/login.html','utf-8',(err,data)=>{
        if(err){
            console.error(err);
            res.send("Error Displaying Form");
        }else{
            res.end(data);
        }
    });
});
```

Our Protected route

```
// Protected route
app.post('/profile', (req, res) => {
    res.send(`Welcome ${req.user.username}!`);
});

// Start the server
const port = 3000;
app.listen(port, () => {
    console.log(`Server is running on port ${port}`);
});
```

Requirements:

- Create an Express.js application with routes for user profile (/profile) and checkout (/checkout).
- Implement a middleware function for authentication that checks if a user is logged in before granting access to the /profile and /checkout routes. Assume that user authentication details are stored in a simulated database.
- Implement a logging middleware function to log details about incoming requests (such as URL, HTTP method, timestamp) and responses (status code, response time) for all routes.
- Ensure that the middleware functions are applied to the appropriate routes to enforce authentication and logging.
- Test the application by sending sample requests to the protected routes (/profile and /checkout) with and without authentication credentials, and verify that the middleware functions behave as expected.