Backend Programming with Express: Class 2

September 12, 2023

Basic Module System, Routes, Middleware

1 Module System

As we learned in the previous lab, Node.js employs a module system that organizes code into reusable components. Modules help organize and structure the Node.js applications by allowing you to break the code into smaller, reusable pieces.
There are mainly 2 types of module systems in Node.js:

1. **Core Modules**: Built-in modules provided by Node.js. You can use them in your applications without having to install or require additional packages. Examples include the `os` module for interacting with the operating system and `fs` module for file system operations.
2. **User-Defined Modules**: These are modules created by developers and can be used in multiple parts of an application. To create a user-defined module, you typically create a separate JavaScript file and then export functions, objects, or variables from that file using `module.exports`.

Create an Express project

You can either create an express app (shown in the last lab) or continue with the previous one.
**** To execute the file, open the terminal and go to the file directory (cd <filepath>), type node `filename.js`, and press enter.**

1.1 OS Module

The Node.js OS (Operating System) module is a built-in module in Node.js that provides a set of utility methods to interact with the operating system. It allows developers to access various operating system-related information and perform tasks like getting information about the operating system, user information, file paths, memory, and more.

Create a new JavaScript file (e.g., `os-module.js`) in your project directory. At the top of the file, import the OS module.

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

1.1.1 Perform Basic Operations

1. **os.userInfo()**: Returns information about the currently effective user

   ```
   console.log(os.userInfo());
   ```

2. **os.platform()**: Returns the operating system platform (e.g., 'darwin' for macOS, 'win32' for Windows).

3. We can also store this information in a variable and return them all together.

```
const currentOS = {
    name: os.type(),
    release: os.release(),
    totalMem: os.totalmem(),
    freeMem: os.freemem()
}
console.log(currentOS);
```

1.2 File Module

To handle file operations like creating, reading, deleting, etc., Node.js provides an inbuilt module called FS (File System).

Create a new JavaScript file (e.g., `file-module.js`) in your project directory. At the top of the file, import the File module.

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

1.2.1 Perform Basic Operations

There are two ways to do that. **Synchronous** and **Asynchrnous**.

- **Synchronous approach**: It executes the commands line by line. It first starts executing a command and waits for it to complete to get the result, and only after that, it executes the next command.
- **Asynchronous approach**: it does not wait for each operation to complete, rather it executes all operations in the first go itself. It allows Node.js to handle multiple operations concurrently without blocking the execution of other tasks. This asynchronous programming model is crucial for building scalable and high-performance applications, especially for I/O-bound tasks, such as reading files, making network requests, or handling database queries.

Imagine you have a sequence of five commands to execute: 1, 2, 3, 4, and 5. When using a synchronous process, these commands are executed sequentially. First, command 1 is executed, then command 2, and so on, with each command waiting for the previous one to complete. However, in an asynchronous process, things work differently. Suppose command 3 is an asynchronous operation, and the others are not dependent on it. In this case, when you start the process, commands 1 and 2 are initiated one after the other and executed sequentially. After that, command 3 begins its execution asynchronously in the background, not blocking the main execution flow. Now, here's where the asynchronous advantage comes in: while command 3 is still running, command 4 is also initiated and starts executing concurrently with it. This means that command 4 doesn't have to wait for command 3 to finish. Then, command 5 will execute. This asynchronous approach allows for more efficient resource utilization and responsiveness, especially when dealing with tasks that might take some time to complete, while other tasks can continue in parallel.

When to use which approach?
If your operations don't involve intensive tasks, such as fetching large amounts of data from a database, you can opt for a synchronous approach. However, when dealing with heavier tasks, the asynchronous method is recommended.

There are some main functionalities we can perform using the File system module. We will learn about – **write, read, append, rename and delete** a file in Node.js

1. **Write operation**: Here, we can write into a file. The system first checks if there's already an existing file or not and then creates a new file if the specified file does not exist. There are 2 ways to perform it. Synchronous and Asynchronous write. You can identify the synchronous versions of file-writing functions by the ***"Sync"*** suffix in their names, like `fs.writeFileSync`.

- **Synchronous write:**

```
fs.writeFileSync( file, data, options )
```

file: denotes the path of the file where it has to be written. It can be a string, Buffer, URL etc.
data: actual data that is to be written.
options: It is a string or object that can be used to specify optional parameters that will affect the output. **Example**:

```
fs.writeFileSync("./contents/demoFile.txt", "hello");
```

- **Asynchronous write**:
Syntax:

```
fs.writeFile( file, data, options, callback )
```

Example:

```
fs.writeFile(
    "./demoFile.txt",
    " Life is like a box of chocolates; you never know what you're gonna get \n",
    (error) => {
      if (error) {
        console.log(error);
      } else {
        console.log("Successful!!");
      }
});
```

One thing to note here, Asynchronous function needs one extra parameter callback. It gets executed when the file operation is complete. If there is an error while writing to the file, the error is shown through the callback function. Remember, all the asynchronous functions have a callback function.

For the rest of the example, we will use asynchronous process.

2. **Append operation**: Now, if we want to write more to the file, we use **appendFile**.

```
fs.appendFile(
    "./demoFile.txt",
    "- Forrest Gump ",
    (err) => {
      if (err) {
        console.log(err);
      }
      else {
       console.log("New text is appended!!");
       }
});
```

N.B. if we use `fs.writeFile` instead, the previous texts will be replaced by the new ones. In the case of, `fs.appendFile()` the new texts will be added at the end of the file.

3. **Rename Operation**: Syntax:

```
fs.rename( oldPath, newPath, callback )
```

Example:

```
fs.rename("./demoFile.txt", "./RenamedFile.txt", (err) => {
    if (err) {
        console.log(err);
    } else {
        console.log("Successful");
    }
});
```

4. **Read Operation:**
   Syntax:

   ```
   fs.readFile( filename, encoding, callback)
   ```

   Example:

   ```
   fs.readFile("./RenamedFile.txt", "utf-8", (err, data) => {
       if (err) {
           console.log(err);
       } else {
           console.log(data);
       }
   });
   ```

5. **Delete Operation:** To delete a file in Node.js, we can use the `unlink()`.

   ```
   fs.unlink("./demoFile.txt", (err) => {
     if (err) {
       console.log(err);
     } else {
       console.log("Deleted Successfully.");
     }
   });
   ```

TASK:

```
console.log("Before");
```

First read the file and show the data. Then write something more in that file and finally show the data again.

```
console.log("After");
```

Explain the behavior.

2 Routes

Routing in Express.js refers to the process of defining how your application responds to client requests for different URLs. It helps determine which code should run based on the URL or route requested by the client. **Route definition takes the following structure:**

```
app.METHOD(PATH, HANDLER)
or
app.METHOD(path, callback [, callback ...])
```

METHOD is an HTTP request method. Example: app.get(), app.post(), app.put(), and so on.
PATH is a path on the server. Example: "./about-us".
HANDLER is the callback function executed when the route and method are matched. In fact, the routing methods can have more than one callback function as arguments. With multiple callback functions, we need to provide next() as an argument to the callback function and then call it within the body of the function to hand off control to the next callback.

Example:
Create a file name `routes.js` and import Express.

```
const express = require("express");
const router = express.Router();
```

Router: It is used to create a new router object. This function is used when you want to create a new router object in your program to handle requests.

2.1 GET Request:

The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.
Let's send a **GET** request. In the `route.js` file type the following:

```
router.get("/getRequest", (req,res)=>{
    res.send("<h1>This is a GET request</h1>")
});
```

router.get("/getRequest ", (req, res) => ...): This code defines a route handler for HTTP GET requests to the "**/getRequest** " path. When a client makes a GET request to "**/getRequest** ", this route handler is executed.
res.send"<h1> This is a GET request < /h1>") sends an HTML response containing an **<h1>** element.
After that, we will export the router object.

```
module.exports = router;
```

it makes the routes.js file available for other parts of your Node.js application to use. We will then import and mount this router in the app.js file to define and handle routes.
In the `app.js` file, we will import the `routes.js`

```
const routes = require("./routes")
app.use(routes)
```

app.use(routes), is used to incorporate the routes defined in the routes router into the Express application. This modular approach to routing allows you to organize your code effectively and manage different parts of your application's functionality in separate files or modules.

Some other HTTP requests:

2.2 POST request:

A POST request is used to send data to the server, for example, customer information, file upload, etc.

```
router.post('/', function (req, res) {
    res.send('Got a POST request')
})
```

2.3 PUT request:

Replaces all current representations of the target resource with the uploaded content.

```
router.put('/user', function (req, res) {
    res.send('Got a PUT request at /user')
})
```

2.4 DELETE request:

Removes all current representations of the target resource given by a URI.

```
router.delete('/user', function (req, res) {
    res.send('Got a DELETE request at /user')
})
```

2.5 Passing a parameter:

We can create routes with parameters, including using an "id" as a parameter. Suppose, there are 10 items in a list. And we want to update a particular item, we can use it to retrieve the item from the list.

```
 router.get('/users/:id', (req, res) => {
      const userId = req.params.id;
      // Now you can use the userId in your code
      res.send(`User ID: ${userId}`);
});
```

The **":id"** part of the path is a placeholder for a dynamic parameter. It signifies that this route can handle requests with different user IDs provided in the URL.
`req.params.id` is used to retrieve the value provided in the URL. For example, if the client sends a GET request to **"/users/123"**, `req.params.id` will be "123".

3 Middleware

It is used to add functionality and processing to HTTP requests and responses in a web application. Middleware comes in the middle of the request and response cycle of the node.js execution. Middleware functions are executed sequentially in the order they are defined and can perform various tasks, such as modifying request or response objects, authentication, logging, error handling, and more.

Create another file `middleware.js` and write a function.

```
function logRequest(req, res, next) {
  res.send(‘Request received for: ${req.method} ${req.url}‘);
  next();
}
```

logRequest is a middleware function that takes three parameters: `req, res, and next`.
next(): This is a callback function that is used to pass control to the next middleware function in the request-response cycle. It allows the middleware to indicate when it has completed its task, and Express should continue processing the request with the next middleware or route handler.

```
module.exports = {
  logRequest
};
```

the middleware function is exported as an object with the key `logRequest`. This allows other parts of your application (e.g., your main Express application file) to import and use this middleware function.

Now in the app.js, import middleware.js .

```
const {logRequest}=require("./middleware")
```

one thing to notice,`logRequest` is in braces. It is used when we want to import a specific property or variable from a module. In this case, it imports the `logRequest` variable from the **'./middleware'** module. If we had used-

```
const logRequest=require("./middleware")
```

It would import the entire module as a single object, and we can access the exported variables or functions as properties of that object.

After that we define a route that uses a middleware

```
app.get("/middleware", logRequest, (req, res) => {
    console.log("We have implemented our first middleware!!");
});
```

Here, we have implemented `logRequest` as a middle function in the route definition. This means that before the route handler is executed, the `logRequest` middleware will run, logging information about the incoming request, including the HTTP method and URL.

This demonstrates how you can use middleware to add functionality to specific routes in your Express application. The `logRequest` middleware is used to log request details, but you can create more complex middleware functions to perform various tasks like authentication, authorization, and more.

Task

For this task, you will need to install **Postman** in your Visual Studio Code.
Postman: Postman is a popular software tool used for API development and testing. It provides a user-friendly interface for making HTTP requests to APIs, inspecting responses, and automating various aspects of API testing and development. You can use it while working with RESTful APIs, GraphQL, SOAP, or other types of APIs. Here's a simple tutorial you can follow Install Postman extension.

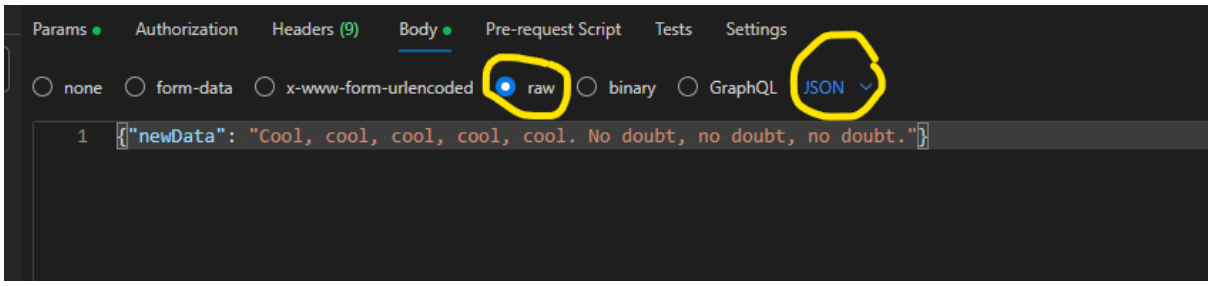
Task 1

You will create a basic authentication system, where users can sign up/sign in using email, user name, and password.

- **User Signup:** Users can sign up with their email, username, and password. Store user data, including OS-related information, in a json file.
- **User Login:** Users can log in using their email and password.
  - First verify the provided email and password against stored data. (Create a middleware: **isAuthenticated**)
  - If authenticated, allow access to a welcome page that shows the username.
  - Handle errors, such as incorrect email or password, and respond accordingly.
- **Password Reset:** Authenticated users can update their password.
  - Verify their previous password to confirm it matches.
  - Allow password reset if the previous password is correct.
  - Handle errors, such as incorrect previous password, and respond accordingly.
- **Error Handling:**
  - Implement error handling for various scenarios.
  - Show appropriate status codes and error messages.

Hints:

1. Figure 1 to post JSON data through Postman:



2. In the **route.js** file, include the following lines-

```
const bodyParser = require('body-parser');

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

3. To traverse through the json objects, first store the data in using-

```
let users= []
```

Then use the following line to traverse all the emails and check whether it matches with the require one or not.

```
const user = users.find((u) => u.email === email);
```

JSON data: JSON stands for "JavaScript Object Notation." It is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. JSON data consists of key-value pairs, similar to JavaScript objects. Here's an example of JSON data:

```
{
  "name": "John Doe",
  "age": 30,
  "city": "New York"
}
```

JSON data is widely used in web applications and APIs for exchanging structured data between the client and server. It's a common format for configuration files, data storage, and data transmission over HTTP in RESTful APIs.