Contents

Backend Developer Roadmap in Node JS

To become a solid backend developer using **Node.js** and **Express.js**, you need to master several core topics. I'll list them from **basic to advanced**, covering both fundamental and practical skills.

🟢 1. Core Node.js Concepts (Foundation)

These are the building blocks of Node.js, essential before jumping into Express.js.

- **JavaScript Basics** – (ES6+, Async/Await, Promises, Callbacks)
- **Node.js Architecture** – (Single-threaded, Event Loop, Non-blocking I/O)
- **Global Objects & Modules** – (__dirname, __filename, require(), import/export)
- **File System Module (fs)** – (Reading/Writing files, Streams, Buffers)
- **Events & Event Emitter** – (Handling and triggering custom events)
- **HTTP Module** – (Creating a basic server using http.createServer)
- **Process & Child Processes** – (Environment variables, process.env, child_process)

🚀 2. Express.js Basics (Building APIs)

This is where you start building web applications.

- **Setting up an Express server** – (express() and app.listen())
- **Middleware in Express** – (Built-in, custom, and third-party middleware like morgan)
- **Routing in Express** – (GET, POST, PUT, DELETE, app.use(), app.get())
- **Request & Response Handling** – (req.body, req.params, req.query, res.json())
- **Serving Static Files** – (express.static())

🛠 3. Express.js Advanced Features

Now you enhance API functionality.

- **Error Handling Middleware** – (Centralized error handling, next())
- **Asynchronous Code & Error Handling** – (async/await, try-catch)
- **Express Router** – (Modularizing routes into separate files)
- **CORS & Security** – (cors package, Helmet, Rate limiting)
- **Logging & Debugging** – (winston, debug module)

🔗 4. Database Connectivity

Backend APIs often connect to a database.

- **MongoDB with Mongoose** – (Schema design, CRUD operations, Models)
- **SQL (PostgreSQL/MySQL) with Sequelize/Prisma** – (Relations, Transactions)
- **Database Indexing & Performance Optimization**

🔐 5. Authentication & Security

Securing your backend is a must!

- **JWT Authentication** – (Creating & verifying JWT tokens)
- **OAuth & Social Logins** – (Google, Facebook auth with Passport.js)
- **Hashing Passwords** – (bcryptjs for password hashing)
- **Session-based Authentication** – (express-session, Cookies)
- **Role-based Access Control (RBAC)** – (Admin/User permissions)

📦 6. APIs & Microservices

Develop scalable architectures.

- **Building RESTful APIs** – (REST principles, CRUD operations)
- **GraphQL Basics** – (Schema, Queries, Mutations, Apollo Server)
- **WebSockets & Real-time Apps** – (socket.io for chat apps, notifications)
- **Rate Limiting & API Security** – (express-rate-limit)
- **Caching** – (Using Redis for performance optimization)

⚙ 7. Deployment & DevOps

Get your app live and optimized.

- **Environment Variables** – (dotenv, .env files)
- **Deploying on Cloud** – (Vercel, Render, DigitalOcean, AWS, Heroku)
- **CI/CD Pipelines** – (GitHub Actions, Docker, Kubernetes)
- **Load Balancing & Scaling** – (Cluster Module, PM2)
- **Logging & Monitoring** – (Winston, Morgan, Sentry)

⚡ 8. Best Practices & Code Quality

Write maintainable and scalable backend code.

- **Project Structure & Clean Code** – (MVC Pattern, Modularization)
- **Testing** – (Jest, Mocha, Chai, Supertest for API testing)
- **Performance Optimization** – (Compression, Lazy Loading, Caching)

🏆 Bonus Topics (For Mastery)

If you want to go beyond and become an expert:

- 🔹 **Event-Driven Architecture** – (Node Event Emitters, Kafka, RabbitMQ)
- 🔹 **Serverless Functions** – (AWS Lambda, Firebase Functions)
- 🔹 **Blockchain & Smart Contracts** – (Interacting with Ethereum)

Introduction to Node.js

What is Node.js?

Node.js is an **open-source, cross-platform JavaScript runtime environment** that allows developers to execute JavaScript code outside of a browser. It is widely used for building scalable and high-performance applications.

Key Features of Node.js:

- **V8 Engine**: Node.js runs on the V8 JavaScript engine (also used in Google Chrome), which makes it fast and efficient.
- **Asynchronous and Event-Driven**: Node.js handles I/O operations (network requests, database access, file system operations) asynchronously, meaning it doesn't block the execution of other code.
- **Single-Threaded, Non-Blocking I/O**: Unlike traditional multi-threaded models, Node.js operates on a single thread but can handle multiple requests concurrently.
- **Rich Package Ecosystem**: Node.js has a vast collection of open-source modules available via npm (Node Package Manager), making development faster and easier.

V8 JavaScript Engine

V8 is the JavaScript engine that makes Chrome and Node.js run JavaScript fast. Let's break it down:

1. **What is V8?**
    - It is the engine that runs JavaScript code inside Google Chrome.
    - Node.js also uses it to run JavaScript outside the browser.
2. **What does it do?**

- It **parses** (reads) and **executes** JavaScript code.
- The browser provides extra features like the **DOM (Document Object Model)** and other Web APIs.

3. **Other JavaScript Engines:**
   - **SpiderMonkey** → Used in Mozilla Firefox
   - **JavaScriptCore (Nitro)** → Used in Safari
   - **V8** → Used in Chrome & Edge (Edge now uses Chromium)

How JavaScript Runs in V8 (Compilation & Performance)

1. **JavaScript was originally "interpreted"** (executed line by line).
2. **Now, V8 compiles JavaScript** before running it (this makes it faster).
3. **JIT (Just-In-Time) Compilation**
   - Instead of running directly, JavaScript is first **compiled into machine code** (fast format for computers).
   - This speeds up execution a lot, which is crucial for big applications like Google Maps.

---

npm (Node Package Manager) – Your JavaScript Toolbox

What is npm?

- **npm = Node.js package manager** (used to install and manage code libraries).
- It has **millions of reusable packages** for JavaScript developers.

Installing All Dependencies

- If a project has a package.json file, run:
- npm install
  - This installs all required libraries in the **node_modules** folder.

Updating Packages

- npm update
- Update a specific package:
- npm update <package-name>

Versioning with npm

- You can install a specific version of a package:
- npm install <package-name>@<version>

Understanding ECMAScript 2015 (ES6) and Beyond in Node.js

1. What is ECMAScript (ES6)?

- ES6 (ECMAScript 2015) introduced many modern JavaScript features.
- Node.js follows the latest updates of this standard by using the **V8 engine**.

👉 **To check available in-progress features:**
Run this command in your terminal:

```
node --v8-options | grep "in progress"
```

What is a Module in Node.js?

A **module** in Node.js is a reusable block of code that can be imported into other files. It helps in organizing the code and improving maintainability.

Types of Modules in Node.js:

1. **Core Modules**: Built-in modules provided by Node.js, such as http, fs, path, os.
2. **Local Modules**: Custom modules created by developers within a project.
3. **Third-Party Modules**: Modules installed via npm, like express, mongoose, dotenv.

The HTTP Module in Node.js

The **http module** in Node.js allows the creation of web servers and handling HTTP requests and responses.

Creating a Basic Web Server:

```
const { createServer } = require('node:http'); // Importing the HTTP module from Node.js
const hostname = '127.0.0.1'; // Defining the hostname (localhost IP)
const port = 3000; // Defining the port number where the server will listen
// Creating an HTTP server
const server = createServer((req, res) => {
  res.statusCode = 200; // Setting the HTTP status code to 200 (OK)
  res.setHeader('Content-Type', 'text/plain'); // Setting response header to indicate plain text content
  res.end('Hello World'); // Sending "Hello World" as the response body and ending the response
});
// Server starts listening on the specified hostname and port
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`); // Logging the server URL to the console
});
```

Running the Server:

1. Save the file as server.js.
2. Run the command:
3. node server.js
4. Open a browser and visit http://127.0.0.1:3000/ to see the output.

JavaScript Basics

1. **Asynchronous programming and callbacks**
2. **Timers**
3. **Promises**
4. **Async and Await**
5. **Closures**
6. **The Event Loop**

Asynchronous Programming and Callbacks JavaScript is single-threaded, meaning it executes one operation at a time. However, asynchronous programming allows tasks like I/O operations, network requests, and timers to run in the background without blocking the main thread. This is crucial in Node.js since it handles multiple requests efficiently.

Callbacks**:** A callback is a function passed as an argument to another function and executed later. Callbacks are commonly used in asynchronous operations. Callback functions are a way to ensure certain code runs only after another code has already finished execution.

Example:

```
// Function that takes a name and a callback function as arguments
function greet(name, callback) {
  console.log('Hello ' + name); // Print a greeting message
  callback(); // Call the callback function
}
// Callback function that logs when the function execution ends
function logEnd() {
  console.log('Function execution ended.');
}
// Calling the greet function with "Alice" and passing 'logEnd' as the callback
greet('Alice', logEnd);
// Output:
// Hello Alice
// Function execution ended.
```

- **processUserData:** This function takes two parameters: userId and callback.

- userId: The ID of the user whose data is to be fetched.
- callback: A function that will be executed after the data fetching operation completes.

- **setTimeout:** Simulates a delay of 1.5 seconds to mimic an asynchronous operation (like fetching data from a server).
  - After 1.5 seconds, it checks if userId is provided.
  - If userId is provided, it calls the callback function with null as the first argument (indicating no error) and a user object { id: userId, name: "John Doe" } as the second argument (indicating success).
  - If userId is not provided, it calls the callback function with an error message "User ID not provided" as the first argument and null as the second argument.

2. Function Call

javascript

```
// Function that simulates fetching user data from a database or API
function processUserData(userId, callback) {
  // Simulate a delay of 1.5 seconds before returning data
  setTimeout(() => {
    if (userId) {
      callback(null, { id: userId, name: "John Doe" }); // If userId exists, return user data (success case)
    } else {
      callback("User ID not provided", null); // If no userId, return an error message
    }
  }, 1500);
};
```

- **processUserData(1, callback):** Calls the processUserData function with 1 as the userId and an anonymous function as the callback.
  - Inside the callback function:
    - It checks if there is an error.
      - If there is an error, it logs the error message to the console.
      - If there is no error, it logs the user data to the console.

Timers Timers allow executing code after a specified delay or at intervals. JavaScript provides three main timer functions:

- setTimeout(fn, delay): Executes fn after delay milliseconds.
- setInterval(fn, delay): Repeats execution of fn every delay milliseconds.

- clearTimeout and clearInterval: Stop the execution of timers.

Example:

```
setTimeout(() => console.log("Executed after 2 seconds"), 2000);

let interval = setInterval(() => console.log("Repeating every 1 second"), 1000);

setTimeout(() => clearInterval(interval), 5000); // Stops after 5 seconds
```

A Promise in JavaScript is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. It provides a cleaner, more robust way to handle asynchronous operations compared to traditional callback functions, which can lead to callback hell.

Key Concepts

- **Pending:** The initial state; neither fulfilled nor rejected.
- **Fulfilled:** The operation completed successfully.
- **Rejected:** The operation failed.

Creating a Promise

```
// Creating a new Promise object
const myPromise = new Promise((resolve, reject) => {
  const success = true; // Change this to `false` to simulate failure

  if (success) {
    resolve("Operation succeeded!"); // If success is true, resolve the promise
  } else {
    reject("Operation failed."); // If success is false, reject the promise
  }
});
```

Handling Promises

You can handle promises using then and catch methods:

The then and catch methods are used to handle the outcome of a Promise in JavaScript.

then Method

The then method is used to handle the resolved value of a Promise. It takes up to two arguments:

1. A callback function for the onFulfilled case (when the Promise is resolved successfully).
2. An optional callback function for the onRejected case (when the Promise is rejected).

catch Method

The catch method is used to handle the rejected value of a Promise. It takes one argument, a callback function for the onRejected case.

```
myPromise
  .then((message) => {
    console.log("Success:", message); // If the promise is resolved, this runs
  })
  .catch((error) => {
    console.log("Error:", error); // If the promise is rejected, this runs
  });
```

Here's your code with comments on every line for better understanding:

```
const fetchData = (url, callback) => {
    // Create a new Promise that handles the fetch request
    const myPromise = new Promise((resolve, reject) => {
        fetch(url) // Fetch data from the given URL
            .then((response) => {
                if (response.ok) { // Check if the response is successful (status 200-299)
                    return response.json(); // Convert response to JSON
                } else {
                    reject("Failed to fetch data: " + response.statusText); // Reject if response is not OK
                }
            })
            .then((data) => {
                resolve(data); // Resolve the promise with fetched data
            })
            .catch((error) => {
                reject("Network error: " + error.message); // Reject if there is a network error
            });
    });
    // Use the promise to trigger the callback
    myPromise
        .then((data) => {
            callback(null, data); // No error, pass the fetched data to the callback
        })
        .catch((error) => {
```

```
            callback(error, null); // Pass the error to the callback function
        });
};
// Example usage
const url = "https://jsonplaceholder.typicode.com/posts";
fetchData(url, (error, data) => {
    if (error) {
        console.error("Error:", error); // Log error if fetching fails
    } else {
        console.log("Fetched data:", data); // Log the fetched data if successful
    }
});
```

Why is using a `Promise` better than not using it?

If we didn't use Promises and only relied on callback-based approaches, we would face multiple issues:

1. Avoiding Callback Hell (Better Readability & Maintainability)
   - If we had multiple nested callbacks, the code would become difficult to read and maintain.
   - With Promises, we can chain `.then()` and `.catch()`, making it cleaner.
2. Error Handling is Easier
   - Without Promises, error handling inside multiple nested callbacks would be messy.
   - With Promises, we can use `.catch()` to handle errors in one place.
3. Better Control Over Asynchronous Operations
   - Promises allow us to `resolve()` or `reject()` based on conditions.
   - With callbacks, we must rely on manually managing success/failure states.
4. Avoiding "Callback Inversion"
   - When using callbacks, functions pass execution to another function, making it harder to track flow.
   - Promises provide a more structured and predictable execution flow.
5. Improved Debugging
   - Promises work well with modern debugging tools and async/await, allowing better stack traces.
   - Callbacks often make debugging harder, especially when deeply nested.

Alternative: Using `async/await` (Even Cleaner)

If you want an even cleaner approach, you can use `async/await`:

```
const fetchDataAsync = async (url) => {
```

```
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error("Failed to fetch data: " + response.statusText);
        }
        const data = await response.json();
        console.log("Fetched data:", data);
    } catch (error) {
        console.error("Error:", error.message);
    }
};
// Example usage
fetchDataAsync(url);
```

👉 This removes explicit Promises and callbacks, making it even more readable!

Using Async/Await

Async/await syntax provides an even cleaner way to work with promises:

Asynchronous in JavaScript
In JavaScript, asynchronous operations don't block the execution of other code. This is achieved using constructs like callbacks, Promises, and the async/await syntax.
async Keyword
The async keyword is used to declare an asynchronous function. This function returns a Promise implicitly, and you can use the await keyword within it.
await Keyword
The await keyword can only be used inside an async function. It pauses the execution of the async function until the Promise is resolved or rejected.
Handling Errors
When using async and await, it's important to handle errors using try and catch blocks. If the awaited Promise is rejected, the code in the catch block will execute.

```
async function handleOperations() {
  try {
    const result1 = await asyncOperation1;
    console.log(result1);
    const result2 = await asyncOperation2;
    console.log(result2);
```

```
  } catch (error) {
    console.error(error);
  }
}
handleOperations();
```

In this example, async keyword is used to define an asynchronous function, and await pauses the function execution until the promise settles.

Summary

- **Promises** simplify asynchronous code, making it more readable and manageable.
- **Then** and **catch** methods handle success and error cases.
- **Chaining** allows for sequential asynchronous operations.
- **Async/await** provides a syntactically cleaner approach for dealing with promises.

Async and Await async and await make asynchronous code look synchronous and are used with Promises.

Example:

```
// Function that returns a Promise simulating data fetching
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Data fetched successfully!"); // After 2 seconds, resolve the promise
    }, 2000);
  });
}
// Async function to handle the asynchronous fetchData function
async function getData() {
  console.log("Fetching data..."); // Log before fetching starts

  try {
    const data = await fetchData(); // Wait for fetchData() to complete
    console.log(data); // Print the resolved data
  } catch (error) {
    console.error("Error:", error); // Catch and log any errors
```

```
  }


  console.log("Done fetching data."); // Log after fetching is completed
}
// Call the async function
getData();
```

Breakdown of the Example

1. **fetchData Function:**
    - Returns a Promise that resolves with the message "Data fetched successfully!" after a 2-second delay.
2. **getData Function:**
    - Declared as an async function.
    - Logs "Fetching data..." to the console.
    - Uses await to wait for fetchData to resolve and assigns the resolved value to the data variable.
    - Logs the data to the console.
    - Logs "Done fetching data." to the console after the asynchronous operation is complete.

Closures

A closure is a function that remembers the environment in which it was created. It allows a function to access variables from its outer (enclosing) scope even after that outer function has finished executing.

Here's an example to illustrate the concept:

```
function outerFunction() {
  const outerVariable = 'I am from the outer scope';
  function innerFunction() {
    console.log(outerVariable); // Accesses outerVariable even after outerFunction is done
  }
  return innerFunction;
}
const myClosure = outerFunction();
myClosure(); // Logs: 'I am from the outer scope'
```

Explanation:

- outerFunction creates a local variable outerVariable and defines innerFunction.
- innerFunction has access to outerVariable even after outerFunction has returned.

- myClosure holds the innerFunction, which still has access to outerVariable due to closure.

The Event Loop

The Event Loop handles JavaScript's asynchronous operations, ensuring that non-blocking code is executed efficiently. It processes the **Call Stack**, **Web APIs**, **Callback Queue**, and **Microtask Queue (Promises)** in a cyclic manner.

Example:

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

setTimeout(() => console.log("Timeout callback"), 0);

Promise.resolve().then(() => console.log("Promise resolved"));

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

Output:

Start

End

Promise resolved

Timeout callback

Promises are executed before setTimeout because they are in the **Microtask Queue**, which has higher priority than the **Callback Queue**.

🔲What is Undici?

- **Undici** is an HTTP client library for **Node.js** that powers the fetch API.
- It is built **from scratch** and does **not** use Node.js's built-in HTTP module.
- It is **high-performance** and good for handling many requests efficiently.

2️⃣Basic Usage of Fetch API with Undici

📌 GET Request (Fetching Data)

Here's how you can make a GET request to fetch data in Node.js using the node-fetch library.

```
// Import the node-fetch library

const fetch = require('node-fetch');

// URL of the API you want to fetch data from

const apiURL = 'https://jsonplaceholder.typicode.com/posts';

// Function to fetch data

async function fetchData() {

    try {
```

```
        const response = await fetch(apiURL);
        if (!response.ok) {
            throw new Error('Network response was not ok ' + response.statusText);
        }
        const data = await response.json();
        displayData(data);
    } catch (error) {
        console.error('There was a problem with the fetch operation:', error);
    }
}
// Function to display data
function displayData(data) {
    data.forEach(item => {
        console.log(`ID: ${item.id}, Title: ${item.title}`);
    });
}
// Call fetchData
fetchData();
```

Explanation

- **API URL**: Replace 'https://jsonplaceholder.typicode.com/posts' with the URL of the API you want to fetch data from.
- **Import node-fetch**: This line imports the node-fetch library, which allows you to use the fetch function in Node.js.
- **fetchData Function**: Uses fetch to make a GET request to the API.
    - await fetch(apiURL): Waits for the response from the API.
    - Checks if the response is OK. If not, throws an error.
    - await response.json(): Parses the response as JSON.
- **displayData Function**: Logs the fetched data to the console.
    - Iterates over the data and logs each item's ID and title.

📌 POST Request (Sending Data)

```
// Import the node-fetch library
const fetch = require('node-fetch');
// URL of the API you want to post data to
```

```
const apiURL = 'https://jsonplaceholder.typicode.com/posts';
// Data to be sent in the POST request
const data = {
    title: 'foo',
    body: 'bar',
    userId: 1
};
// Function to make a POST request
async function postData() {
    try {
        const response = await fetch(apiURL, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });
        if (!response.ok) {
            throw new Error('Network response was not ok ' + response.statusText);
        }
        const responseData = await response.json();
        console.log('Response:', responseData);
    } catch (error) {
        console.error('There was a problem with the fetch operation:', error);
    }
}
// Call postData
postData();
```

Explanation

- **API URL**: Replace 'https://jsonplaceholder.typicode.com/posts' with the URL of the API you want to post data to.
- **Data**: This is the data you want to send in the POST request. It is converted to a JSON string using JSON.stringify(data).

- **postData Function**: Uses fetch to make a POST request to the API.
    - method: 'POST': Specifies that this is a POST request.
    - headers: Sets the Content-Type to application/json to indicate that the request body contains JSON data.
    - body: Contains the data to be sent, converted to a JSON string.
    - await fetch(apiURL, { method, headers, body }): Sends the request and waits for the response.
    - Checks if the response is OK. If not, throws an error.
    - await response.json(): Parses the response as JSON and logs it to the console.

Node.js Module System

Node.js follows a **module system** to organize code into separate files. This helps make the code reusable, maintainable, and easy to manage.

1 Global Object

- In Node.js, the **global object** is an object that is accessible from anywhere in your application.
- It provides built-in functions and variables.
- Unlike in browsers (where the global object is window), in Node.js, the global object is **global**.

- **Example:**

```
console.log(global); // Prints all global properties

console.log(global.setTimeout); // Shows that setTimeout is a global function
```

---

2 Modules

- A **module** is just a JavaScript file that contains some code.
- Node.js organizes code into different files (modules) to keep things clean.
- There are **three types** of modules:
    1. **Built-in modules** (provided by Node.js, like fs, http, path)
    2. **User-defined modules** (files you create)
    3. **Third-party modules** (installed via npm, like express)

---

3 Creating a Module

- To create a module, you write some code inside a separate file and **export** it using module.exports.

math.js

```
function add(a, b) {
    return a + b;
}
module.exports = add; // Exporting function
```

4️⃣ Loading a Module

- To use a module in another file, we **import** it using require().

app.js

```
const add = require('./math'); // Importing math.js
console.log(add(5, 3)); // Output: 8
```

5️⃣ Module Wrapper Function

- Every module in Node.js is **wrapped** inside a function before execution.
- This function provides the following arguments:
    - exports → Shortcut for exporting
    - require → Function to import modules
    - module → Object representing the current module
    - __filename → Path of the current file
    - __dirname → Directory of the current file

- **Example:**

```
console.log(__filename); // Prints the full path of this file
console.log(__dirname);  // Prints the directory where this file is located
```

6️⃣ Path Module

- **Joining Paths:** Combines multiple path segments into one.
- **Resolving Paths:** Converts a sequence of paths into an absolute path.
- **Extracting Parts:** Retrieves information like directory name, base name, extension, etc.
- **Normalization:** Normalizes a path, resolving .. and . segments.

```
// Import the path module from Node.js
const path = require('path');
// Join multiple path segments into a single path
```

```
const fullPath = path.join('/users', 'md', 'projects');
// Log the full joined path
console.log('Full Path:', fullPath);
// Resolve a sequence of paths into an absolute path
const absolutePath = path.resolve('users', 'md', 'projects');
// Log the absolute path
console.log('Absolute Path:', absolutePath);
// Get the last portion (file name) of a path
const fileName = path.basename('/users/md/projects/index.html');
// Log the file name
console.log('File Name:', fileName);
// Get the directory name of a path
const dirName = path.dirname('/users/md/projects/index.html');
// Log the directory name
console.log('Directory Name:', dirName);
// Get the extension of the path
const extName = path.extname('/users/md/projects/index.html');
// Log the extension name
console.log('Extension Name:', extName);
```

OS Module

- The os module is a **built-in module** that provides system-related information.

- **Example:**

```
const os = require('os');
console.log(os.type());        // OS type (Windows/Linux/Mac)
console.log(os.freemem());     // Free memory in bytes
console.log(os.totalmem());    // Total memory in bytes
console.log(os.platform());    // OS platform (win32, linux, darwin)
```

Node.js Module System

Node.js provides **built-in modules** that help developers perform different tasks easily, such as working with files, handling events, and creating servers.

1️⃣ File System (FS) Module

- The **fs (File System) module** is used to interact with files and directories.
- It allows you to:
    - Read files
    - Write files
    - Delete files
    - Rename files
    - Create directories, etc.

- **Example: Reading a File**

app.js

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

// Read file asynchronously

fs.readFile('example.txt', 'utf8', (err, data) => {

   if (err) {

      console.error(err);

      return;

   }

   console.log(data); // Prints the content of example.txt

});
```

- **Example: Writing to a File**

```
fs.writeFile('output.txt', 'Hello, Node.js!', (err) => {

   if (err) throw err;

   console.log('File written successfully!');

});
```

2️⃣ Events Module

In Node.js, the events module is at the heart of handling and working with events. It allows you to create, fire, and listen to events in an asynchronous and event-driven manner.

What is an Event in Node.js??

An **event** is essentially a signal that something has happened in the application. For example:

- A user clicks a button (in a UI application).
- A file finishes downloading.

- A server receives a request.

In Node.js, events help to implement the non-blocking, asynchronous nature of its runtime. When an event occurs, associated listeners (or callbacks) respond to them.

The Event Module

The events module in Node.js provides the EventEmitter class, which allows you to create and manage custom events. The most common pattern is:

1. **Emit an Event**: You "trigger" an event to signal that something happened.
2. **Listen to an Event**: You "register" a listener that runs when the event is triggered.

How Events and Listening Work

Here's an example to illustrate:

javascript

```
// Import the events module
const EventEmitter = require('events');
// Create an instance of EventEmitter
const myEmitter = new EventEmitter();
// Create a listener for the 'greet' event
myEmitter.on('greet', (name) => {
  console.log(`Hello, ${name}!`);
});
// Emit the 'greet' event
myEmitter.emit('greet', 'MD');
```

What's Happening Here:

1. **Event Emitter Creation**: The myEmitter instance is created from the EventEmitter class.
2. **Registering a Listener**: The .on() method listens for a specific event ('greet' in this case). When the event is triggered, the callback function runs.
3. **Emitting the Event**: The .emit() method triggers the event, optionally passing along arguments ('MD' here, passed into the listener's function).

Key Methods and Properties

- on(event, listener): Registers a listener for the specified event.
- emit(event, ...args): Triggers an event, optionally passing arguments to the listener.
- once(event, listener): Registers a one-time listener that is removed after the event is triggered.
- removeListener(event, listener): Removes a specific listener for an event.
- removeAllListeners(event): Removes all listeners for a specific event.

This pattern makes Node.js powerful in building scalable and event-driven applications, such as servers, real-time systems, or asynchronous workflows.

3️⃣ Event Arguments

- We can **pass data (arguments)** while emitting an event.
- This helps send information along with an event.

- Example: Passing Arguments in Events

```
emitter.on('userLoggedIn', (username) => {
    console.log(`User ${username} has logged in.`);
});
emitter.emit('userLoggedIn', 'Luffy');
// Output: User Luffy has logged in.
```

4️⃣ Extending EventEmitter

- Instead of using EventEmitter directly, we can create our own **custom class** that extends it.
- This is useful when we need custom event-handling logic inside a class.

- Example: Creating a Custom EventEmitter Class

```
const EventEmitter = require('events');
class Logger extends EventEmitter {
    log(message) {
        console.log(message);
        this.emit('messageLogged', { id: 1, text: message });
    }
}
const logger = new Logger();
logger.on('messageLogged', (eventData) => {
    console.log('Listener received:', eventData);
});
logger.log('Hello, World!');
```

Explanation:

1. We create a class Logger that **extends** EventEmitter.
2. The log() method prints a message and emits an event.

3. When the event is emitted, a listener handles it.

5️⃣ HTTP Module

- The http module helps create web servers and handle requests.
- It is used to build backend applications in Node.js.

- Example: Creating a Simple Web Server

```
const http = require('http');
const server = http.createServer((req, res) => {
   res.writeHead(200, { 'Content-Type': 'text/plain' });
   res.end('Hello, this is a Node.js server!');
});
server.listen(3000, () => {
   console.log('Server is running on port 3000...');
});
```

Explanation:

1. We import the http module.
2. We create a server that sends "Hello, this is a Node.js server!" as a response.
3. The server listens on **port 3000**.

Code With Mosh:

Explanation

Here's a fully commented version of your code along with a detailed explanation of every function, object, and event mechanism used.

Explanation of Key Concepts

1. EventEmitter

- **What it does:** EventEmitter is a built-in module in Node.js that allows objects to communicate asynchronously through event-driven programming.
- **How it works:**
    - It emits (triggers) named events.
    - It listens (reacts) to those events when they occur.

2. emit Method

- **What it does:** Triggers an event.
- **Example from your code:**

js

CopyEdit

emitter.emit('messageLogged', { id: 1, url: "http://" });

  - Here, emit sends a signal that the messageLogged event has occurred with an object { id: 1, url: "http://" } as data.

3. on Method (Listener)

- **What it does:** Listens for an event and executes a function when the event is emitted.
- **Example from your code:**

js

CopyEdit

logger.on('messageLogged', (arg) => {

  console.log('Listener called', arg);

});

  - This listens for the messageLogged event and executes a function when the event occurs.

Fully Commented Code

app.js (Main file)

javascript

CopyEdit

// Import the built-in EventEmitter module from Node.js

const EventEmitter = require('events');

// Import the Logger class from logger.js

const Logger = require('./logger');

// Create an instance of the Logger class

const logger = new Logger();

// Register a listener for the 'messageLogged' event

// The callback function gets executed when the event is emitted

logger.on('messageLogged', (arg) => {

  console.log('Listener called', arg); // Logs the event data when triggered

});

// Call the log function, which will internally emit 'messageLogged' event

```
logger.log('message');
```

---

logger.js (Module File)

```
javascript
CopyEdit
// Import the built-in EventEmitter module to handle events
const EventEmitter = require('events');
// Define a URL (not used, but can be useful for logging requests)
var url = 'http://mylogger.io/log';
// Define a Logger class that extends EventEmitter
class Logger extends EventEmitter {
  // Define a method 'log' that takes a message as an argument
  log(message) {
    // Print the message to the console
    console.log(message);

    // Emit an event named 'messageLogged' with an object as data
    // The event listener in app.js will receive this data
    this.emit('messageLogged', { id: 1, url: "http://" });
  }
}
// Export the Logger class so it can be used in other files
module.exports = Logger; // The correct export (previously, it was 'log' which was incorrect)
```

Detailed Explanation of Objects, Functions, and Event Mechanism

Term	Explanation
EventEmitter	A built-in class in Node.js that enables event-driven programming. Objects can emit and listen for events.
emit(eventName, data)	Triggers an event with an optional data object. In this case, it emits 'messageLogged' with { id: 1, url: "http://" }.
on(eventName, callback)	Listens for a specific event and executes a callback when the event occurs. The callback receives event data.
Logger (class)	A class that extends EventEmitter to allow logging messages and emitting events.

log(message)	A method inside Logger that logs a message and emits the 'messageLogged' event.
{ id: 1, url: "http://" }	The data object passed when the event is emitted. It can be any structured data.

Flow of Execution

1. logger.js defines a class Logger that extends EventEmitter.
2. The log(message) function prints a message and emits the messageLogged event.
3. app.js creates an instance of Logger.
4. It registers a listener for messageLogged, which logs the received event data.
5. Calling logger.log('message') prints 'message' and triggers the event.
6. The event listener logs 'Listener called' along with the event data.

PDF, DOCX, PPT, TEXT file read using library in node

PDF File Parsing

Package Installation Command:

bash

```
npm install pdf-parse
```

Example Code:

javascript

```
// Import the built-in 'fs' (file system) module
const fs = require('fs');
// Import the 'pdf-parse' library for parsing PDF files
const pdf = require('pdf-parse');
// Specify the path to the PDF file
const filePath = '../Introduction to Node Note-01.pdf';
// Read the PDF file as a buffer
fs.readFile(filePath, (err, data) => {
    // Handle any errors that occur while reading the file
    if (err) throw err;
    // Parse the PDF data
    pdf(data).then(pdfData => {
        // Log the text content of the PDF to the console
        console.log(pdfData.text);
```

```
    }).catch(error => {
        // Handle any errors that occur while parsing the PDF
        console.error("Error reading file", error);
    });
});
```

Parsing Different Types of Files in Node.js

1. PDF Files

- **Library:** pdf-parse
- **Usage:** Extracts text content from PDF files.
- **Example:** The code snippet above demonstrates how to read and parse a PDF file.

2. DOCX Files

- **Package Installation Command:**

bash

```
npm install mammoth
```

- **Example Code:**

javascript

```
// Import the built-in 'fs' (file system) module
const fs = require('fs');
// Import the 'mammoth' library for parsing DOCX files
const mammoth = require('mammoth');
// Specify the path to the DOCX file
const filePath = 'example.docx';
// Read the DOCX file as a buffer
fs.readFile(filePath, (err, data) => {
    // Handle any errors that occur while reading the file
    if (err) throw err;
    // Parse the DOCX data to extract raw text
    mammoth.extractRawText({ buffer: data })
        .then(result => {
            // Log the text content of the DOCX to the console
            console.log(result.value);
        })
```

```
    .catch(error => {
        // Handle any errors that occur while parsing the DOCX
        console.error("Error reading file", error);
    });
});
```

3. PPT Files

- **Package Installation Command:**

bash

```
npm install pptx2json
```

- **Example Code:**

javascript

```
// Import the built-in 'fs' (file system) module
const fs = require('fs');
// Import the 'pptx2json' library for parsing PPTX files
const pptx2json = require('pptx2json');
// Specify the path to the PPTX file
const filePath = 'example.pptx';
// Parse the PPTX data to extract content
pptx2json(filePath).then(result => {
    // Log the parsed data from the PPTX file to the console
    console.log(result);
}).catch(error => {
    // Handle any errors that occur while parsing the PPTX
    console.error("Error reading file", error);
});
```

4. Text Files

- **Library:** Built-in fs module (no need to install)
- **Usage:** Reads plain text files.
- **Example Code:**

javascript

```
// Import the built-in 'fs' (file system) module
const fs = require('fs');
```

```
// Specify the path to the text file
const filePath = 'example.txt';
// Read the text file with UTF-8 encoding
fs.readFile(filePath, 'utf8', (err, data) => {
    // Handle any errors that occur while reading the file
    if (err) throw err;
```

getRequest.js:

```
// Import the 'express' module.
const express = require('express');
// Create an instance of an Express application.
const app = express();

// Define an array of course objects.
const courses = [
  {id: 1, name: 'course1'},
  {id: 2, name: 'course2'},
  {id: 3, name: 'course3'}
];

// Define a route for the root URL ('/') that sends 'Hello World' as the response.
app.get('/', (req, res) => {
  res.send('Hello World');
});

// Define a route for '/api/courses' that sends the list of courses as the response.
app.get('/api/courses', (req, res) => {
  res.send(courses);
});

// Define a route for '/api/courses/:id' that sends the course with the given ID as the response.
app.get('/api/courses/:id', (req, res) => {
  // Find the course with the given ID.
  const course = courses.find(c => c.id === parseInt(req.params.id));
  // If the course is not found, send a 404 status and an error message.
  if (!course) res.status(404).send('The course with the given ID was not found.');
  // Send the course as the response.
  res.send(course);
});

// Define a route for '/api/posts/:year/:month' that sends the request parameters as the response.
app.get('/api/posts/:year/:month', (req, res) => {
  res.send(req.params);
});

// Define the port to listen on, using the environment variable PORT or defaulting to 3000.
const port = process.env.PORT || 3000;
// Start the server and listen on the defined port, logging a message to the console.
app.listen(port, () => console.log(`Listening on port ${port}...`));

/*
This code sets up a basic Express server with the following routes:
- GET '/' which responds with 'Hello World'.
- GET '/api/courses' which responds with a list of courses.
```

```
- GET '/api/courses/:id' which responds with a specific course based on the provided ID.
- GET '/api/posts/:year/:month' which responds with the request parameters (year and month).
The server listens on a port defined by the environment variable PORT or defaults to port 3000.
*/
```

postRequest.js:

```
//postRequest.js
// Import the Joi module for data validation.
const Joi = require('joi');
// Import the 'express' module.
const express = require('express');
// Create an instance of an Express application.
const app = express();

// Use the express.json() middleware to parse JSON bodies.
app.use(express.json());

// Define an array of course objects.
const courses = [
  {id: 1, name: 'course1'},
  {id: 2, name: 'course2'},
  {id: 3, name: 'course3'}
];

// Define a route for '/api/courses' that sends the list of courses as the response.
app.get('/api/courses', (req, res) => {
  res.send(courses);
});

// Define a route for POST requests to '/api/courses' to add a new course.
app.post('/api/courses', (req, res) => {
  // Define a schema for validating the request body.
  const schema = {
    name: Joi.string().min(3).required()
  };

  // Validate the request body against the schema.
  const result = Joi.validate(req.body, schema);
  // If validation fails, send a 400 status and the error message.
  if (result.error) {
    res.status(400).send(result.error.details[0].message);
    return;
  }

  // Check if the name is not provided or is less than 3 characters long.
  if (!req.body.name || req.body.name.length < 3) {
    return res.status(400).send('Name is required and should be at least 3 characters long.');
  }

  // Create a new course object with an id and name from the request body.
  const course = {
    id: courses.length + 1,
    name: req.body.name,
  };
  // Add the new course to the courses array.
  courses.push(course);
  // Send the new course as the response.
  res.send(course);
});
```

```
// Define the port to listen on, using the environment variable PORT or defaulting to 3000.
const port = process.env.PORT || 3000;
// Start the server and listen on the defined port, logging a message to the console.
app.listen(port, () => console.log(`Listening on port ${port}...`));

/*
This code sets up a basic Express server with the following routes:
- GET '/api/courses' which responds with a list of courses.
- POST '/api/courses' which allows adding a new course to the list. It validates the request body to
ensure the 'name' field is provided and is at least 3 characters long.
The server listens on a port defined by the environment variable PORT or defaults to port 3000.
*/
```

putRequest.js:

```
// Import the Joi module for data validation.
const Joi = require('joi');
// Import the 'express' module.
const express = require('express');
// Create an instance of an Express application.
const app = express();
// Use the express.json() middleware to parse JSON bodies.
app.use(express.json());
// Define an array of course objects.
const courses = [
  {id: 1, name: 'course1'},
  {id: 2, name: 'course2'},
  {id: 3, name: 'course3'}
];
// Define a route for '/api/courses' that sends the list of courses as the response.
app.get('/api/courses', (req, res) => {
  res.send(courses);
});
// Define a route for POST requests to '/api/courses' to add a new course.
app.post('/api/courses', (req, res) => {

  const result = validateCourse(req.body);
  const { error } = validateCourse(req.body);
  // If validation fails, send a 400 status and the error message.
  if (error) {
    res.status(400).send(error.details[0].message);
    return;
  }
  if (!req.body.name || req.body.name.length < 3) {
    return res.status(400).send('Name is required and should be at least 3 characters long.');
  }
  const course = {
    id: courses.length + 1,
    name: req.body.name,
  };
  courses.push(course);
  res.send(course);
});
app.put('/api/courses/:id', (req, res) => {
  // Find the course with the given ID.
  const course = courses.find(c => c.id === parseInt(req.params.id));
  // If the course is not found, send a 404 status and an error message.
  if (!course) {
    res.status(404).send('The course with the given ID was not found.');
    return;
  }
```

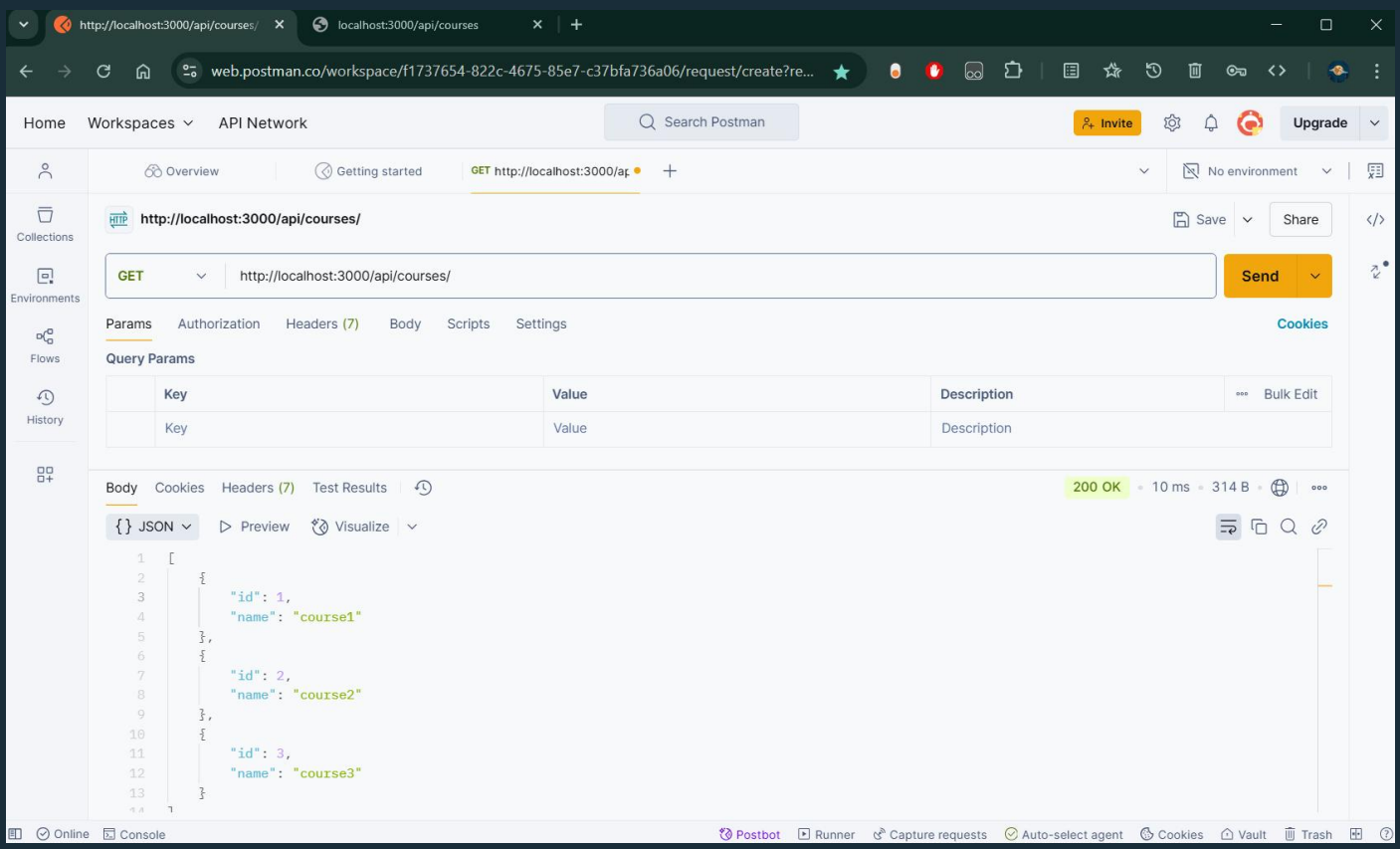
```
  const result = validateCourse(req.body);
  const { error } = validateCourse(req.body);
  // If validation fails, send a 400 status and the error message.
  if (error) {
    res.status(400).send(error.details[0].message);
    return;
  }
  // Update the course with the new name.
  course.name = req.body.name;
  // Send the updated course as the response.
  res.send(course);
});
function validateCourse(course) {
  const schema = {
    name: Joi.string().min(3).required()
  };
  return Joi.validate(course, schema);
}
const port = process.env.PORT || 3000;
// Start the server and listen on the defined port, logging a message to the console.
app.listen(port, () => console.log(`Listening on port ${port}...`));
```

deleteRequest.js:

```
// Import the Joi module for data validation.
const Joi = require('joi');
// Import the 'express' module.
const express = require('express');
// Create an instance of an Express application.
const app = express();
// Use the express.json() middleware to parse JSON bodies.
app.use(express.json());
// Define an array of course objects.
const courses = [
  {id: 1, name: 'course1'},
  {id: 2, name: 'course2'},
  {id: 3, name: 'course3'}
];
// Define a route for '/api/courses' that sends the list of courses as the response.
app.get('/api/courses', (req, res) => {
  res.send(courses);
});
// Define a route for POST requests to '/api/courses' to add a new course.
app.post('/api/courses', (req, res) => {

  const result = validateCourse(req.body);
  const { error } = validateCourse(req.body);
  // If validation fails, send a 400 status and the error message.
  if (error) {
    res.status(400).send(error.details[0].message);
    return;
  }
  if (!req.body.name || req.body.name.length < 3) {
    return res.status(400).send('Name is required and should be at least 3 characters long.');
  }
  const course = {
    id: courses.length + 1,
    name: req.body.name,
  };
```

```
  courses.push(course);
  res.send(course);
});
app.delete('/api/course/:id', (req, res)=>{
  const result = validateCourse(req.body);
  const { error } = validateCourse(req.body);
  // If validation fails, send a 400 status and the error message.
  if (error) {
    res.status(400).send(error.details[0].message);
    return;
  }
  const course = courses.find(c => c.id === parseInt(req.params.id))[0];
  if(!course) return res.status(404). send("The course with the given ID not found");
  const index = courses.indexOf(course);
  courses.splice(index, 1);
  res.send(course);
})
function validateCourse(course) {
  const schema = {
    name: Joi.string().min(3).required()
  };
  return Joi.validate(course, schema);
}
const port = process.env.PORT || 3000;
// Start the server and listen on the defined port, logging a message to the console.
app.listen(port, () => console.log(`Listening on port ${port}...`));
```



Middleware in Expres.js

Middleware in Express.js is a function that is executed between the request and the response.

It has access to the request object (req), the response object (res), and the next middleware function

in the application's request-response cycle. They sit between the incoming request and the final response and allow you to perform operations like request processing, authentication, data validation, or even error handling.

Middleware in Express.js refers to functions that execute during the lifecycle of an HTTP request. They sit between the incoming request and the final response and allow you to perform operations like request processing, authentication, data validation, or even error handling.

Types of Middleware in Express.js

```
1. **Application-Level Middleware**: Defined at the app level and commonly used for logging,
authentication, or parsing request bodies.
   const express = require('express');
   const app = express();
   // Application-level middleware
   app.use((req, res, next) => {
       console.log(`${req.method} request made to ${req.url}`);
       next(); // Pass control to the next middleware
   });
   app.get('/', (req, res) => {
       res.send('Hello, World!');
   });
   app.listen(3000, () => console.log('Server running on port 3000'));
   ```
2. **Router-Level Middleware**: Works on specific routes. These are defined using `Router()`.
   const router = express.Router();
   // Router-level middleware
   router.use((req, res, next) => {
       console.log('Middleware for this router');
       next();
   });
   router.get('/about', (req, res) => {
       res.send('About Page');
   });
   app.use(router); // Attach router to the app
   ```
3. **Built-in Middleware**: Predefined middleware functions provided by Express.js.
   - `express.json()`: Parses incoming JSON requests.
   - `express.urlencoded()`: Parses URL-encoded payloads.
   app.use(express.json());
   app.post('/data', (req, res) => {
       res.send(req.body); // Access parsed JSON data
   });
```

Middleware for Authentication

Let's create middleware to verify if a user is authenticated before accessing certain routes.

javascript

```
const express = require('express');
const app = express();
// Sample middleware for authentication
function authenticateUser(req, res, next) {
    const token = req.headers['authorization']; // Check for a token in the request headers
/*
The token in your code represents a piece of information, usually a string, that is used to verify the
identity or authenticity of a user. Tokens are a common method for securing communication between a
client (like a web browser or mobile app) and a server.
```

```
In this specific example:
The token is expected to be included in the request headers, under the authorization field.
It is compared against a predefined value (my-secret-token in this case) to determine if the user is
allowed to access the requested resource or endpoint.

*/
    if (token === 'my-secret-token') {
        console.log('User authenticated successfully');
        next(); // Pass control to the next middleware or route handler
    } else {
        res.status(401).send('Unauthorized: Invalid token');
    }
}
// Public route (no authentication needed)
app.get('/public', (req, res) => {
    res.send('This is a public route.');
});
// Protected route (requires authentication)
app.get('/protected', authenticateUser, (req, res) => {
    res.send('This is a protected route. You are authenticated!');
});
// Start the server
app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

How It Works:

Middleware Function (authenticateUser): Checks the authorization header for a token. If the token is valid (e.g., 'my-secret-token'), it calls next() to proceed to the next middleware or route handler. If the token is invalid or missing, it sends a 401 Unauthorized response to the client.

Routes:

/public: Can be accessed by anyone without authentication.

/protected: Uses the middleware to ensure only authenticated users can access it.

Testing It Out:

Send a request to /public: You'll receive "This is a public route."

Send a request to /protected:

If you include the header Authorization: my-secret-token, you'll get "This is a protected route. You are authenticated!"

If you omit the header or use an invalid token, you'll get "Unauthorized: Invalid token."

In the provided code snippet, the middleware function is defined using the `app.use()` method.

The `express.json()` middleware is used to parse incoming requests with JSON payloads. This middleware is added to the middleware stack, and it will be executed for every incoming request.

The middleware function in the `app.get('/', (req, res) => {...})` route does not have access to the next middleware function. It simply sends a response with the text "Hello World" when a GET request is made to the root ("/") route.

The middleware function in the `app.get('/api/courses', (req, res) => {...})` route also does not have

access to the next middleware function. It sends a response with the JSON representation of the `courses` array when a GET request is made to the "/api/courses" route.

In summary, middleware in Express.js is a powerful feature that allows developers to add custom functionality to the request-response cycle. It can be used for various purposes, such as authentication, logging, error handling, and more. In the provided code snippet, the `express.json()` middleware is used to parse incoming JSON payloads.

/

```
const express = require('express' );
const logger = require('./logger');
const app = express();
app.use(express.json());//returns a middleware function set req.body
app.use(express.urlencoded());
app.use(express.static('public'));
app.use(logger);

app.use(function(req, res, next){
  console.log('Authenticating...');
  next(); // pass control to the next middleware or
});

const courses = [
  { id: 1, name: 'course1' },
  { id: 2, name: 'course2' },
  { id: 3, name: 'course3' },
]
app.get('/', (req, res) => {
  res. send('Hello World');
});
app.get('/api/courses', (req, res) => {
  res.send(courses);
});
```

Static Content in NodeJS

Static content refers to files like images, CSS stylesheets, JavaScript scripts, HTML files, and other resources that don't change dynamically when served to users. These files are directly sent to the browser as they are.

How It Works Here

1. express.static() **Functionality**:
    - This middleware is used to serve static files from a directory on the server.
    - The 'public' argument specifies that the public folder in your project directory will hold static assets.
2. **Example Behavior**:
    - If your public folder has a file like logo.png, it can be accessed in the browser via http://localhost:3000/logo.png.

- Similarly, if you have an index.html file inside the public directory, navigating to http://localhost:3000/index.html will serve that file.

3. **Benefits of Static Content**:
   - Reduces server load for frequently requested files since they don't require processing logic.
   - Improves the user's experience by enabling faster loading times.

```
4.  const helmet = require('helmet'); // Importing Helmet middleware to secure HTTP headers.
5.  const morgan = require('morgan'); // Importing Morgan middleware for HTTP request logging.
6.  const express = require('express'); // Importing the Express library to create the application.
7.  const logger = require('./logger'); // Importing a custom middleware module (logger.js).
8.
9.  const app = express(); // Initializing an Express application instance.
10.
11. app.use(express.json()); // Middleware to parse incoming JSON request bodies into `req.body`.
12. app.use(express.urlencoded({ extended: true })); // Middleware to parse URL-encoded data (for
    forms) into `req.body`.
13. app.use(express.static('public')); // Middleware to serve static files from the 'public'
    directory.
14. app.use(helmet()); // Applying Helmet middleware for better security by setting HTTP headers.
15. app.use(morgan('tiny')); // Using Morgan middleware with the 'tiny' format for concise request
    logging.
16.
17. app.use(logger); // Using the custom `logger` middleware for logging additional request
    information.
18.
19. app.use(function(req, res, next) {
20.   console.log('Authenticating...'); // Custom middleware to log an "Authenticating..." message.
21.   next(); // Passing control to the next middleware in the stack.
22. });
23.
24. const courses = [ // Array representing a list of courses with unique IDs and names.
25.   { id: 1, name: 'course1' }, // Course with ID 1 and name 'course1'.
26.   { id: 2, name: 'course2' }, // Course with ID 2 and name 'course2'.
27.   { id: 3, name: 'course3' }  // Course with ID 3 and name 'course3'.
28. ];
29.
30. app.get('/', (req, res) => {
31.   res.send('Hello World'); // Route handler for the root path ('/') that sends "Hello World".
32. });
33.
34. app.get('/api/courses', (req, res) => {
35.   res.send(courses); // Route handler for '/api/courses' path that sends the `courses` array as
    the response.
36. });
37.
38. const port = process.env.PORT || 3000; // Setting the application port from environment variables
    or defaulting to 3000.
39. app.listen(port, () => console.log(`Listening on port ${port}...`)); // Starting the server and
    logging the port it's running on.
40.
```

..