Day 06: Middleware in Express.js

- Understanding middleware in Express.js
- Using built-in middleware (e.g., express.json(), express.static)
- Creating custom middleware
- Error-handling middleware

In Express.js, middleware functions are pieces of code that run during the lifecycle of an HTTP request and can handle a variety of tasks before passing control to the next middleware function or the final route handler. Think of middleware as a pipeline of steps that each request goes through until it reaches the intended endpoint.

Key Points About Middleware:

1. **Definition**: Middleware functions are functions that have access to the request (req) and response (res) objects and the next() function, which moves the request to the next middleware or route handler in the stack.
2. **Purposes of Middleware**:
   - **Request Processing**: Middleware can modify the request or response objects, parse incoming data, authenticate users, validate data, or log request details.
   - **Control Flow**: Middleware controls whether the request should proceed, return a response, or redirect based on certain conditions (e.g., authentication checks).
   - **Error Handling**: Middleware can handle errors and decide how to respond if something goes wrong.
3. **Types of Middleware**:
   - **Built-in Middleware**:
     - express.json(): Parses incoming requests with JSON payloads.
     - express.urlencoded(): Parses URL-encoded payloads, commonly used in form submissions.
     - express.static(): Serves static files like HTML, CSS, JavaScript, or images from a directory.
   - **Third-Party Middleware**: Middleware created by the community, like cors for handling cross-origin requests or morgan for logging.
   - **Custom Middleware**: Custom logic for specific requirements, such as checking authentication or data validation.
4. **Creating custom Middleware**:
   - You create middleware functions by defining a function that takes req, res, and next as parameters.
   - After processing the request, you call next() to move to the next middleware or route handler.
   - Example

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

function myMiddleware(req, res, next) {
  console.log("Request received at:", new Date());
  next(); // Pass control to the next middleware
```

```
}

app.use(myMiddleware); // Registers the middleware in the app

// Define a simple route
app.get("/", (req, res) => {
  res.send("Hello, Express!");
});
// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Why Use app.use()?

app.use() is versatile and allows you to:

- Apply middleware globally across all routes.
- Apply middleware conditionally to specific paths.
- Set up multiple middlewares in a stack, where each can process and pass the request forward.

```
app.use([path], middlewareFunction);
```

Why next() ?

```
function myMiddleware(req, res, next) {
  console.log("Request1 received at:", new Date());
  req.name = "Mahesh";
  next(); // Pass control to the next middleware
}
app.use(myMiddleware); // Registers the middleware in the app
function myMiddleware2(req, res, next) {
  console.log("Request2 received at:", new Date());
  next(); // Pass control to the next middleware
}
app.use(myMiddleware2); // Registers the middleware in the app
// Define a simple route
app.get("/", (req, res) => {
  console.log(req.name);
  res.send("Hello, Express! ");
});
```

Built-in Middleware:

Add express.json() Middleware

1. **Purpose**: express.json() parses incoming JSON payloads and makes the data available in req.body.

2. **Setup**: Add this middleware to parse JSON data

```
// Middleware to parse JSON payloads
app.use(express.json());
```

Example Route to Test: Create a POST route to test this middleware

```
app.post('/json-data', (req, res) => {
  console.log(req.body); // Access the parsed JSON data
  res.send('JSON data received');
});
```

Add express.urlencoded() Middleware

1. **Purpose**: express.urlencoded() parses URL-encoded payloads, such as data from form submissions.
2. **Setup**: Add this middleware to parse form data

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

Example Route to Test: Create another POST route to handle form data

```
app.post('/form-data', (req, res) => {
  console.log(req.body); // Access form data here
  res.send('Form data received');
});
```

Add express.static() Middleware

1. **Purpose**: express.static() serves static files (like HTML, CSS, images) from a specified directory.
2. **Setup**: Create a folder named public and add some static files (e.g., an index.html file).
3. **Configure express.static() Middleware**

```
// Middleware to serve static files from 'public' directory
app.use(express.static('public'));
```

Testing:

- Place an index.html file in the public folder.
- Start the server with node index.js.
- Open a browser and go to http://localhost:3000/index.html to see the HTML file served by the Express app.

Error-Handling Custom Middleware:

- Custom middleware can also handle errors if defined with four parameters: (err, req, res, next).
- Use this for logging or handling unexpected errors

```
function errorHandler(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something went wrong!');
  next()
}
// Define a route that triggers an error
app.get("/error", (req, res, next) => {
  const error = new Error("Intentional error");
  next(error); // Pass the error to the error-handling middleware
});
// Place the error-handling middleware after all other routes and middleware
app.use(errorHandler);
```

Interview Questions and Answers:

What is Middleware in Express.js?

Answer: Middleware in Express.js is a function that has access to the request (req), response (res), and next objects. Middleware functions execute during the request-response cycle and can modify req and res, process data, and control the flow of requests. They can handle tasks such as logging, authentication, request parsing, and error handling. In Express, middleware can be applied globally with app.use() or to specific routes.

How do you create custom middleware in Express?

Answer: To create custom middleware, define a function with parameters (req, res, next), where:

- req is the request object,
- res is the response object, and
- next is a callback to pass control to the next middleware in the stack

```
function myMiddleware(req, res, next) {
  console.log("Request received at:", new Date());
  next(); // Pass control to the next middleware
}
app.use(myMiddleware);
```

Here, myMiddleware logs the current date and time, then calls next() to continue to the next middleware or route handler.

What is the difference between app.use() and route-specific middleware (e.g., app.get())?

Answer:

- app.use() applies middleware globally or to specified paths, and it will run for all HTTP methods (GET, POST, PUT, etc.) unless restricted to a specific path.
- Route-specific middleware, like app.get() or app.post(), is attached to a particular route and HTTP method.

```
app.use('/path', myMiddleware); // Applies to all HTTP methods at `/path`
app.get('/path', myMiddleware, (req, res) => { res.send('Hello!'); }); // Only for GET requests
```

What is error-handling middleware in Express, and how is it different from regular middleware?

Answer: Error-handling middleware in Express is used to manage errors during the request-response cycle. It has four parameters: err, req, res, and next. Unlike regular middleware, error-handling middleware is triggered only when an error is passed to next(), or if an exception is thrown within synchronous code.

```
function errorHandler(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send("Something went wrong!");
}
app.use(errorHandler);
```

This middleware logs the error stack and sends a 500 Internal Server Error response.

How does Express.js decide which middleware to execute in what order?

Answer: Express executes middleware in the order in which it is registered. This order is defined by the sequence of app.use() and route handler definitions in the code. Therefore, placing logging middleware first would log every request, while placing it later would log only requests that reach that point. Middleware flow is controlled by calling next(), which passes the request to the next registered middleware.

Can you explain the next() function in middleware? What happens if next() is not called?

Answer: The next() function in middleware is used to pass control to the next middleware function in the stack. If next() is not called, the request will not proceed to the next middleware or route handler, resulting in the request hanging or timing out. For example, authentication middleware might prevent access to routes if certain conditions aren't met, calling next() only when access is granted.

What are some common use cases for middleware in Express?

Answer: Common use cases for middleware in Express include:

- **Logging**: Logging request details (e.g., `morgan`).
- **Authentication**: Verifying user credentials before accessing routes.
- **Body Parsing**: Parsing JSON or URL-encoded form data (e.g., `express.json()`).
- **Serving Static Files**: Serving HTML, CSS, and JavaScript files (e.g., `express.static()`).
- **Error Handling**: Catching and responding to errors in the request cycle

What is the purpose of `express.json()` and `express.urlencoded()` middleware?

Answer:

- `express.json()` parses incoming requests with JSON payloads, making the data accessible in `req.body`. It's often used in APIs.
- `express.urlencoded({ extended: true })` parses URL-encoded payloads (data sent from HTML forms) and supports nested objects. The `extended: true` option allows more complex data structures by using the `qs` library for parsing.