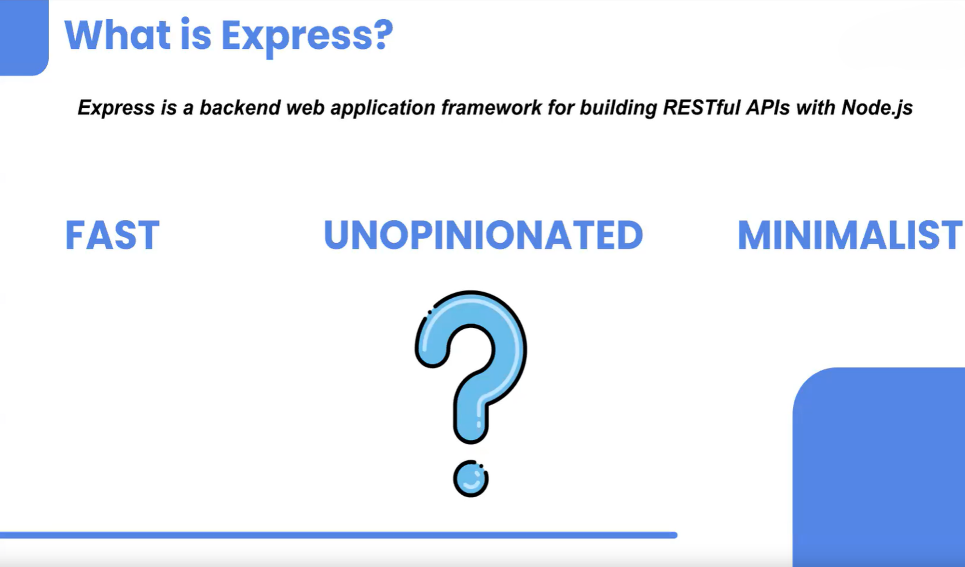
## Page:1

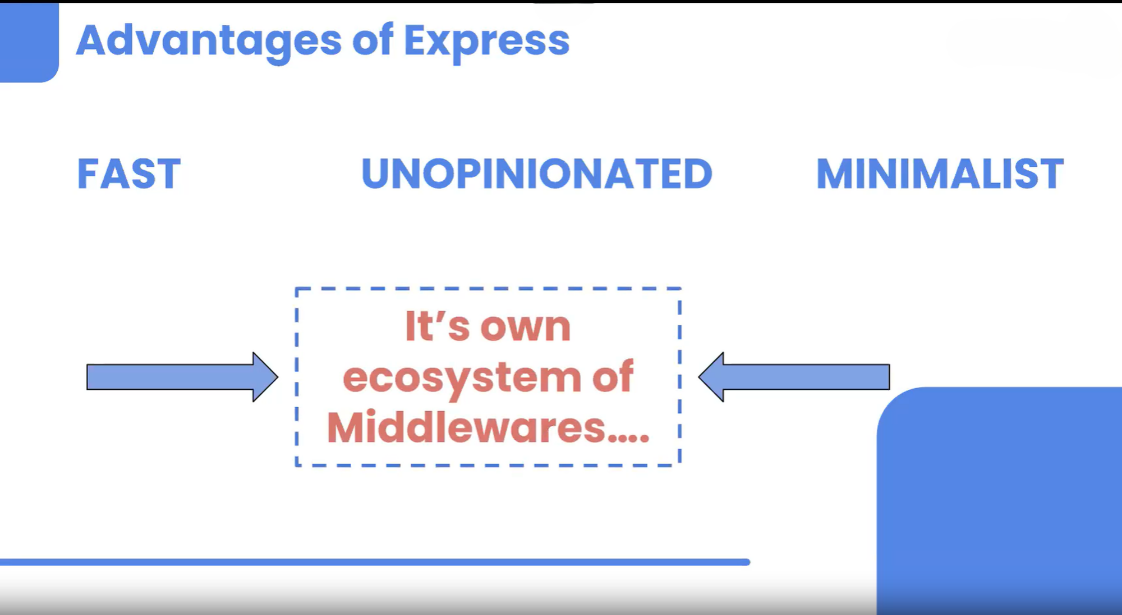
# MiddlewareNotes

## Express Middleware & Ecosystem

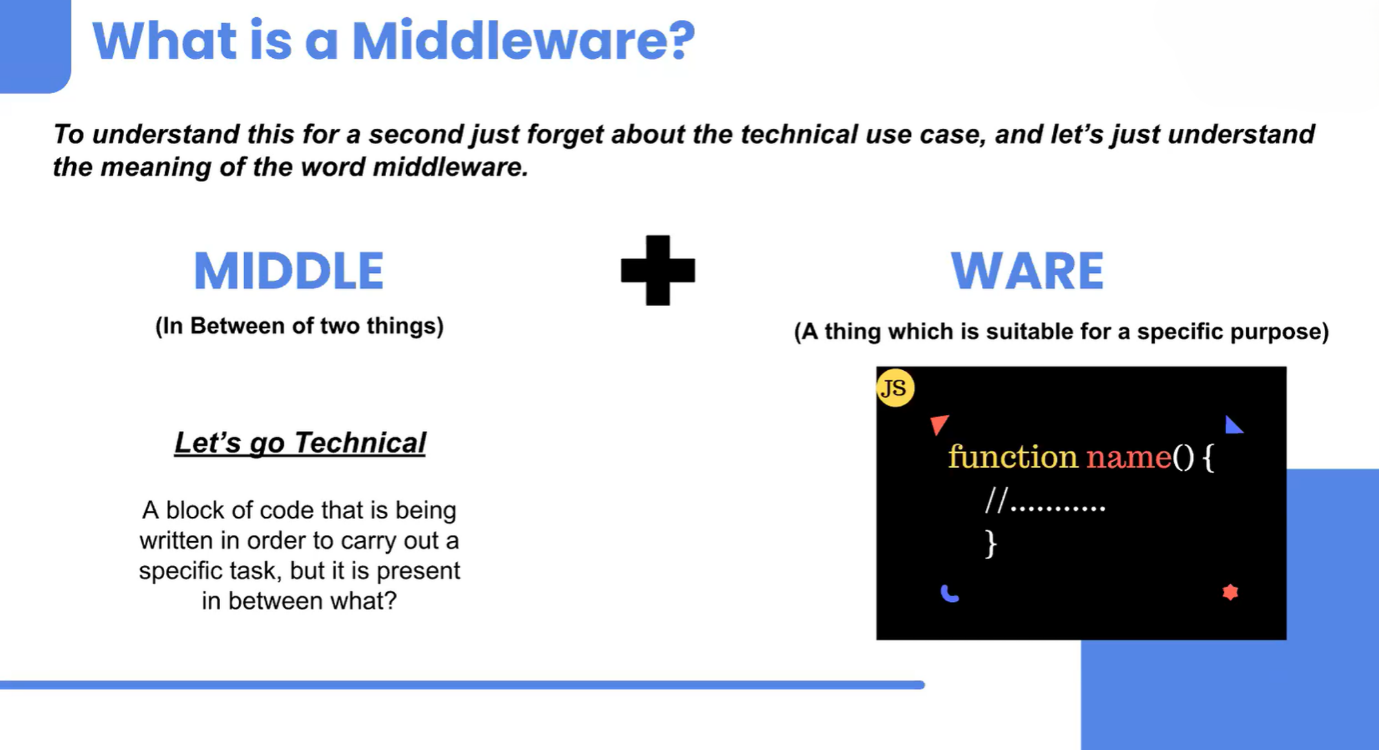


# 🌟 Advantages of Express

1. **Clean Code:** Express allows developers to write clean and organized code. This is particularly beneficial for maintaining and scaling applications.
2. **Efficient Coding:** Unlike the native HTTP module, Express enables developers to accomplish more with less code, making development faster and more efficient.
3. **Middleware Flexibility:** One of the biggest advantages of Express is its extensive collection of middleware. These are functions that execute during the lifecycle of a request to the server. Developers can also create custom middleware to perform specific tasks, offering flexibility and control over the request-response cycle.



## Page:2

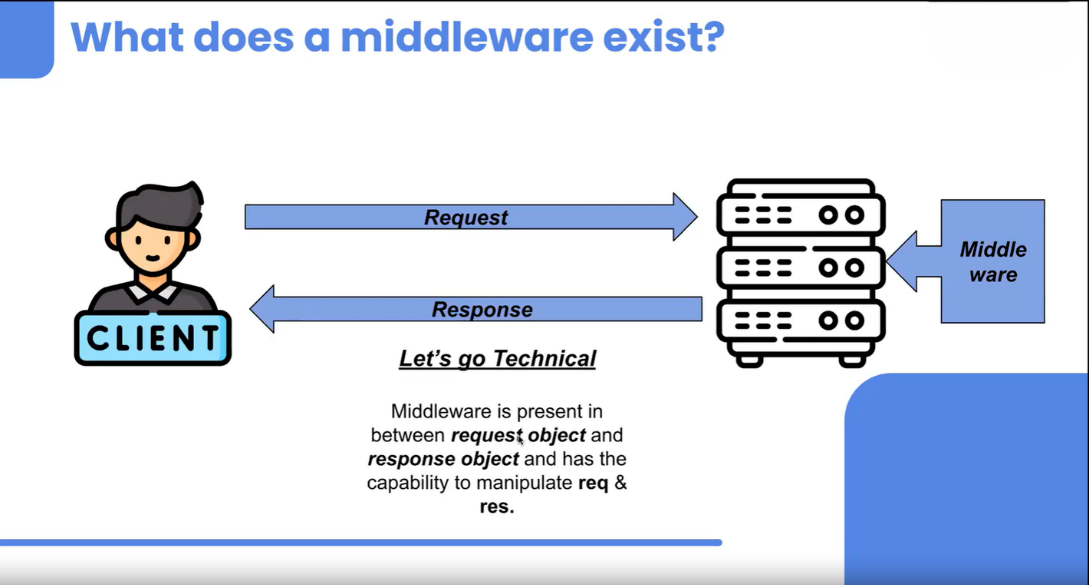


## 🛠️ Middlewares

Middleware in Node.js is a function that has access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. It is executed between the moment the server receives a request and the time the response is sent.

## Key Points:

* **Execution:** Middleware runs after a request is made but before the response is sent.
* **Positioning:** Middleware is typically placed inside the server, not between the client and the server.
* **Usage:** Use app.use() to apply middleware to all routes beneath it in the file



## Page:3

## Example Code:

javascriptCopy code

const express = require("express");

const app = express();

// Our first middleware

app.use((req, res, next) => {

console.log("Hello from Middleware");

next();

});

app.get("/", (req, res) => {

console.log("Hello from the base route");

res.send("Welcome");

});

app.get("/contacts", (req, res) => {

console.log("Hello from the contacts route");

res.send("Contacts");

});

app.get("/about", (req, res) => {

console.log("Hello from the about route");

res.send("About");

});

app.listen(3500, () => {

console.log("Running on 3500");

## Page:4

});

* **next() Function:** The next() function tells the middleware to pass control to the next middleware function. If next() is not called, the request will be left hanging, and the response will not be sent.
* **Positioning of Middleware:** Middleware that should be applied to all routes is usually placed at the top of the file.

## Example Code to Understand next():

javascriptCopy code

const express = require("express");

const app = express();

app.use((req, res, next) => {

console.log("Hello from Middleware");

next();

console.log("Bye from Middleware");

});

app.get("/", (req, res) => {

console.log("Hello from the base route");

res.send("Welcome");

});

app.listen(3500, () => {

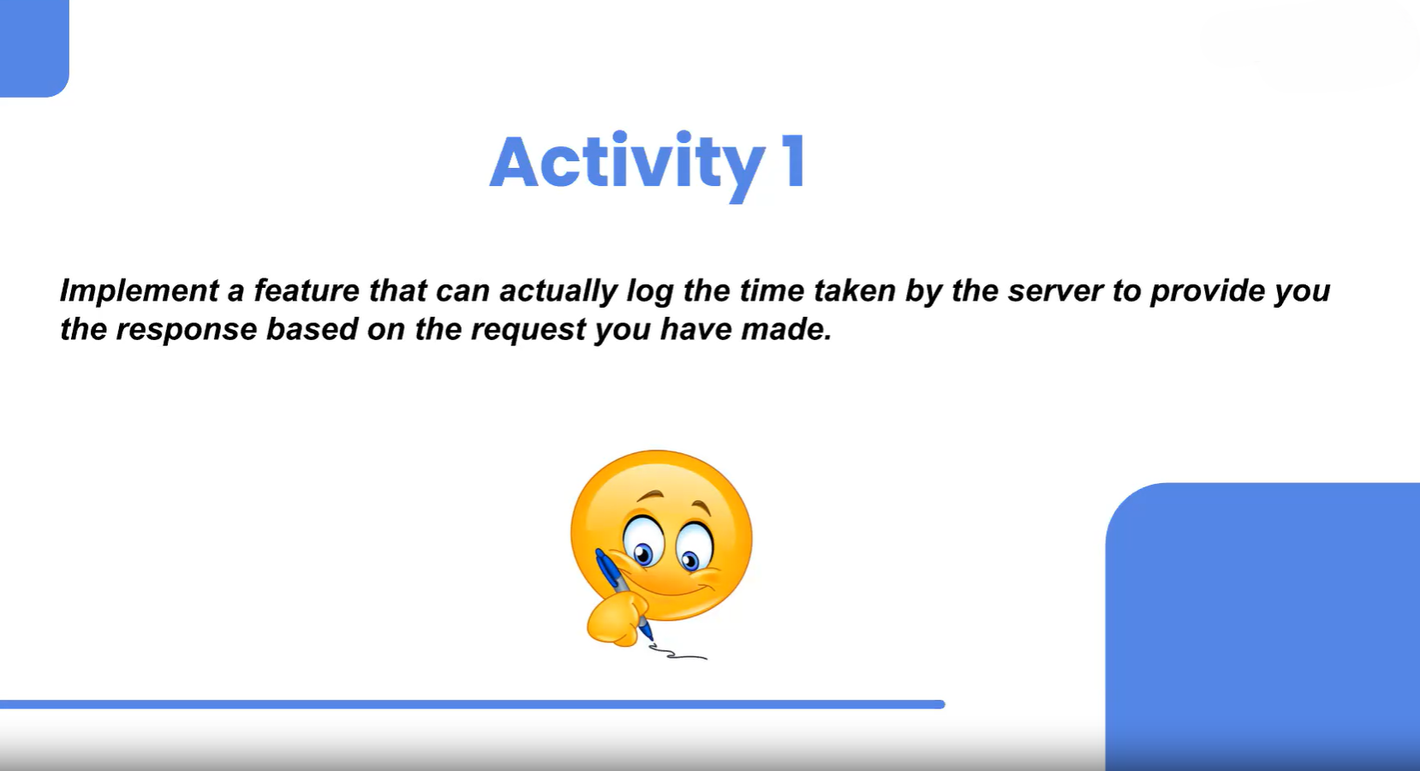
console.log("Running on 3500");

});

In the example above, the message "Bye from Middleware" will be logged after the middleware has passed control to the next function.

## Page:5

# 🌐 Real-Industry Examples



## ⏱️ TimeLogger Middleware

The TimeLogger middleware is designed to measure the time taken to process each request. This can be useful for performance monitoring and optimization.

### Implementation Steps:

1. **Create a Text File with Dummy Data:** Before setting up the middleware, create a text file named lecture.txt with some dummy data. This file will be used in one of the routes.
2. **Setup the Express Application:** Initialize your Express application and define the TimeLogger middleware.

javascriptCopy code

const express = require("express");

const fs = require("fs"); // Required for file operations

const app = express();

// TimeLogger Middleware

app.use((req, res, next) => {

const startTime = new Date().getTime(); // Capture the start time

## Page:6

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

const endTime = new Date().getTime(); // Capture the end time

console.log(`Request processing time: ${endTime - startTime}ms`);

});

// Define Routes

app.get("/", (req, res) => {

res.send("Welcome");

});

app.get("/contacts", (req, res) => {

res.send("Contacts");

});

app.get("/about", (req, res) => {

res.send("About");

});

app.get("/blogs", (req, res) => {

const data = fs.readFileSync("./lecture.txt", "utf-8");

res.send(data);

});

// Start the Server

app.listen(3500, () => {

console.log("Server is running on port 3500");

});

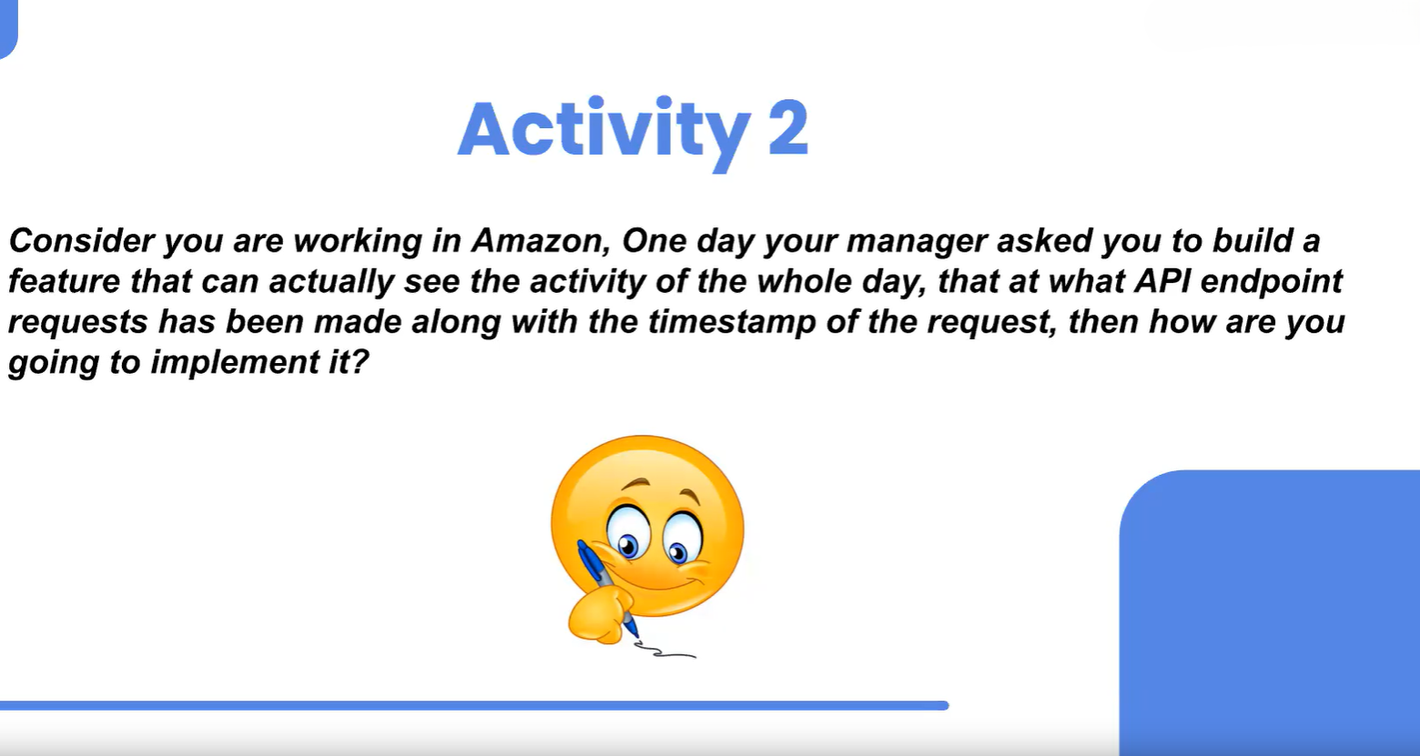
## Page:7

## Explanation:

* **Middleware Function:** The middleware captures the time before (startTime) and after (endTime) the request processing. The difference between these two timestamps gives the total time taken to handle the request.
* **Using next():** The next() function is crucial as it passes control to the next middleware or route handler. Without it, the request would not proceed further.
* **Route Handlers:** Several routes are defined (/, /contacts, /about, /blogs) to demonstrate the middleware's logging functionality. The /blogs route reads and sends the contents of the lecture.txt file.

**Note:**

Middleware functions are reusable pieces of code that can be declared outside and then applied using app.use(). This modular approach allows for better organization and reusability of middleware across different parts of the application.



## 📝 Logger Middleware

The Logger middleware is used to log information about each incoming request, such as the HTTP method and the requested URL. This is useful for monitoring and debugging purposes.

## Page:8

### Implementation:

To implement the Logger middleware, follow these steps:

1. \*\*Create the Middleware Function:\*\*

The middleware function logs the request method and URL to a file named `logs.txt`.

javascriptCopy code

const express = require("express");

const fs = require("fs");

const app = express();

// Logger Middleware

const logger = (req, res, next) => {

fs.appendFileSync("./logs.txt", `\n${req.method} ${req.url}`, "utf-8");

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

};

// Use the Logger Middleware

app.use(logger);

Define Routes:

Define routes to test the Logger middleware. Every request to these routes will be logged.

javascriptCopy code

app.get("/", (req, res) => {

res.send("Welcome");

});

app.get("/contacts", (req, res) => {

res.send("Contacts");

});

## Page:9

app.get("/about", (req, res) => {

res.send("About");

});

// Start the Server

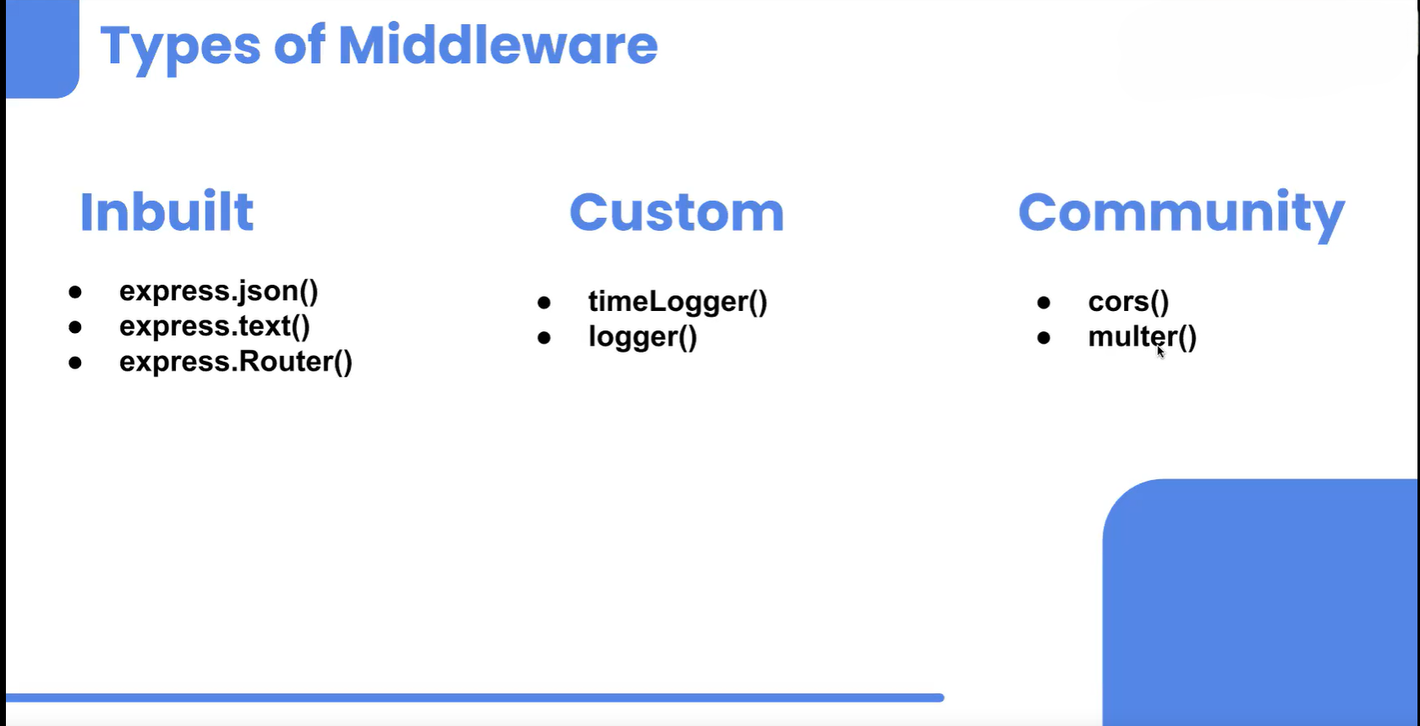
app.listen(3500, () => {

console.log("Server is running on port 3500");

});

## Explanation:

* **Middleware Function:** The logger middleware logs the HTTP method and URL of each request to the logs.txt file. This helps in keeping a record of all the routes visited.
* **Using next():** The next() function is essential as it allows the middleware to pass control to the next middleware or route handler. Without it, the request would not proceed further.



## 🔧 Inbuilt and Custom Middleware

Express provides several inbuilt middleware functions, which can be used to handle common tasks:

* **express.json():** Parses incoming requests with JSON payloads and is based on body-parser.

## Page:10

* **express.text():** Parses incoming requests with text payloads.
* **express.Router():** A modular, mountable route handler. A Router instance is a complete middleware and routing system.

## Example: Modular Routing with express.Router()

## student.router.js

javascriptCopy code

const express = require("express");

const studentRouter = express.Router();

studentRouter.get("/", (req, res) => {

res.send("All the students");

});

studentRouter.post("/addstudent", (req, res) => {

console.log(req.body);

res.send("Added the student");

});

module.exports = {

studentRouter

};

# teacher.router.js

javascriptCopy code

const express = require("express");

const teacherRouter = express.Router();

teacherRouter.get("/", (req, res) => {

res.send("All the teachers");

});

## Page:11

teacherRouter.post("/addstudent", (req, res) => {

console.log(req.body);

res.send("Added the teacher");

});

module.exports = {

teacherRouter

};

# index.js

javascriptCopy code

const express = require("express");

const { studentRouter } = require("./routes/student.router");

const { teacherRouter } = require("./routes/teacher.router");

const app = express();

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

app.use("/students", studentRouter);

app.use("/teachers", teacherRouter);

app.get("/", (req, res) => {

res.send("This is the home page");

});

app.listen(3500, () => {

console.log("Server is running on port 3500");

});

## Page:12

# Explanation:

* Modular Routing: Using express.Router(), you can separate routes into different modules (studentRouter and teacherRouter). This modular approach helps in organizing the codebase, especially in large applications.
* Middleware Application: Middleware like express.json() is applied to parse JSON bodies, while custom routers are used to handle specific sets of routes.

# 🛠️ Multiple Middlewares

In Express, you can use multiple middleware functions to handle various tasks in the request-response cycle. Each middleware can perform specific operations, and the order in which they are called matters.

## Example with Multiple Middleware Functions

The following example demonstrates how multiple middleware functions work together:

javascriptCopy code

const express = require("express");

const app = express();

// First Middleware

app.use((req, res, next) => {

console.log("1");

next();

console.log("2");

});

// Second Middleware

app.use((req, res, next) => {

console.log("3");

next();

console.log("4");

## Page:13

});

// Route Handler

app.get("/", (req, res) => {

console.log("Home");

res.send("Welcome");

});

// Start the Server

app.listen(3500, () => {

console.log("Running on port 3500");

});

# Explanation:

* **Order of Execution:** The first middleware logs "1" before calling next(), which then allows the second middleware to execute, logging "3". After the response is sent, the code after next() in the first and second middleware logs "2" and "4" respectively.
* **Control Flow:** The next() function is crucial as it controls the flow of execution. Without it, the subsequent middleware or route handlers won't be called.

## ⏲️ TimeLogger and 👁️ WatchMan Middlewares

To illustrate how to modularize and use multiple middlewares, let's define two separate middleware functions, timeLogger and watchMan, and then use them in the Express app.

## Defining the Middlewares

## timeLogger.js

javascriptCopy code

const timeLogger = (req, res, next) => {

const startTime = new Date().getTime();

next();

const endTime = new Date().getTime();

console.log(`Request time: ${endTime - startTime}ms`);

};

## Page:14

module.exports = timeLogger;

# watchMan.js

javascriptCopy code

const watchMan = (req, res, next) => {

console.log(`Request method: ${req.method}, URL: ${req.url}`);

next();

};

module.exports = watchMan;

# Using the Middlewares in the App

# index.js

javascriptCopy code

const express = require("express");

const timeLogger = require("./timeLogger");

const watchMan = require("./watchMan");

const app = express();

// Applying Middlewares

app.use(timeLogger);

app.use(watchMan);

// Route Handler

app.get("/", (req, res) => {

console.log("Home");

res.send("Welcome");

});

// Start the Server

## Page:15

app.listen(3500, () => {

console.log("Running on port 3500");

});

# Explanation:

* **Modularizing Middlewares:** The timeLogger and watchMan middlewares are defined in separate files and then imported into the main application file (index.js). This modular approach helps in organizing and managing middleware functions better.
* **Order Matters:** The order in which middlewares are applied (app.use(timeLogger); app.use(watchMan);) affects their execution sequence. In this case, timeLogger will execute first, followed by watchMan.

# Combining Multiple Middlewares in One app.use()

You can also pass multiple middleware functions to a single app.use() call. The order of the functions matters:

javascriptCopy code

app.use(timeLogger, watchMan);

This will execute timeLogger first and then watchMan for each incoming request. This method can be useful for applying a chain of related middleware functions that need to be executed in a specific order.