Express.js Middleware Handbook: A Complete Guide

This guide breaks down the concept of Express.js middleware, explaining how it works, how to use it, and how to create your own to build powerful web applications.

1. What is Middleware in Express.js?

In simple terms, an Express.js application is just a series of functions that are executed one after another when a user makes a request to your server. **Middleware is the term for these functions.**

Think of a request traveling through an assembly line:

1. The request enters the server.
2. It passes through **Middleware Function 1** (e.g., checking the user's login status).
3. It passes through **Middleware Function 2** (e.g., logging the time).
4. It finally reaches the **Route Handler** (the function that sends the final response).

The Core Concept

A middleware function has three main parts:

1. The req (request) object.
2. The res (response) object.
3. A function called next().

Basic Code Example of a Middleware Function

A middleware function can be defined using the standard function signature (req, res, next).

```
const exampleMiddleware = (req, res, next) => {
  // 1. Execute some code (e.g., logging)
  console.log('Middleware executed: Request received.');

  // 2. IMPORTANT: Pass control to the next function in the stack
  next();
};
```

2. How Middleware Works in the Request-Response Cycle

The Flow of a Request

Every time a client (like a web browser) sends an HTTP request to an Express server, it initiates a **request-response cycle**.

1. **Request Entry:** The request hits the Express server.
2. **Middleware Chain:** The request starts passing through all middleware functions registered with app.use().
3. **Execution & Pass-Off:** Each middleware function performs its task (e.g., logging, validation) and *must* call next() to move the request along.
4. **Route Handler:** The request eventually reaches the final route handler (e.g., app.get('/')).
5. **Response End:** The route handler (or a middleware function) sends a response (res.send(), res.json(), etc.), which ends the cycle.

Chaining Multiple Middlewares

You can chain multiple middleware functions together for the same route. They execute strictly in the order they are listed.

```
const middlewareA = (req, res, next) => {
  console.log('Middleware A running...');
  req.timestamp = new Date(Date.now()).toISOString(); // Modifying req object
  next();
};

const middlewareB = (req, res, next) => {
  console.log('Middleware B running...');
  next();
};

app.get('/chained', middlewareA, middlewareB, (req, res) => {
  // This is the final route handler
  res.send(`Request processed at: ${req.timestamp}`);
});
```

3. Main Functions of Middleware

Middleware functions are incredibly versatile and are used to achieve four main goals:

Function	Explanation
Executing Code	Running setup code, like database

	connections, or simple logging.
Modifying req/res Objects	Adding custom properties to the req object (e.g., req.user, req.timestamp) or setting headers on the res object.
Ending the Cycle	If an error occurs (e.g., authentication fails), the middleware can call res.send() to stop the flow immediately. **It must NOT call next() in this case.**
Calling next()	**MANDATORY** if the request should continue to the next function.

4. What is next() and How Does it Work?

The next() function is the third argument in every middleware function and is the **key to controlling the request flow**.

How next() Passes Control

When a middleware function completes its task, calling next() tells Express: "I'm done here, please move this request to the next function in line."

If a middleware function does NOT call next(), the request stops dead in its tracks, and the client hangs forever (unless the middleware explicitly calls a res.send() or similar function to end the cycle).

next() for All Routes (Global Middleware)

When you use app.use(middlewareFunction), it applies the middleware to **every single route** that follows it in the code.

```
// This middleware runs for ALL requests (/root, /api, /random, etc.)
app.use((req, res, next) => {
  req.time = new Date(Date.now()).toISOString();
  console.log(`[${req.method}] Request at ${req.path}`);
  next();
});

// The logger runs before this route
app.get('/', (req, res) => {
  res.send(`Root accessed at: ${req.time}`); // Accesses data added by middleware
});
```

next() for Specific Routes

Middleware can be applied directly to a single route as the second (or third, fourth, etc.) argument, as seen in the chaining example above.

```
// The checkToken middleware only runs when the client hits the /api route.
app.get("/api", checkToken, (req, res)=>{
   res.send("API DATA accessed successfully!");
});
```

5. Types of Middleware in Express.js

A. Built-in Middleware

These are functions exported directly from the Express module:

Middleware	Use	Example
express.json()	Parses incoming request bodies with JSON payloads (used for handling POST data).	app.use(express.json());
express.urlencoded()	Parses incoming request bodies with URL-encoded payloads (used for form data).	app.use(express.urlencoded({ extended: true }));
express.static()	Serves static files (HTML, CSS, images) from a designated directory.	app.use(express.static('public'));

B. Third-Party Middleware

Functions installed separately via npm, used to add specific functionalities.

Middleware	Use	Example
morgan	A robust HTTP request	npm install morgan and

	logger.	app.use(morgan('tiny'));
cors	Allows cross-origin requests from different domains.	npm install cors and app.use(cors());

C. Custom Middleware

Any function you create yourself that follows the (req, res, next) signature.

6. Creating Utility Middlewares (e.g., A Logger)

A Logger Middleware is a perfect example of custom middleware. It executes code to record details about the request before passing control to the final handler.

Custom Logger Middleware Example

This logger captures the method, path, and timestamp, and attaches the timestamp to the req object for later use.

```
// custom_logger.js

const customLogger = (req, res, next) => {
  // 1. Log the incoming request details
  console.log("--- Request Log ---");
  console.log(`Method: ${req.method}`);
  console.log(`Path: ${req.path}`);

  // 2. Modify the req object by adding a custom property
  req.requestTime = new Date(Date.now()).toISOString();
  console.log(`Timestamp added: ${req.requestTime}`);
  console.log("-------------------");

  // 3. Pass control to the next middleware or route handler
  next();
};

module.exports = customLogger;
```

How to Use It: You would import this function and place it globally using app.use() near the top of your main application file.

7. Protecting API Routes using Query String Tokens

Middleware is the standard way to implement security checks like authentication and authorization. Here we use a simple token check example.

Token Check Middleware (checkToken)

This function checks the query string for a specific token. If the token is correct, it calls next(). If it's incorrect, it immediately sends a denial response, stopping the cycle.

```
// Auth Middleware for Token Check

const checkToken = (req, res, next) => {
  // Extract token from the query string (e.g., /api?token=...)
  let { token } = req.query;

  if (token === "giveaccess") {
    // Token is correct: call next() to allow access to the /api route handler
    next();
  } else {
    // Token is incorrect: send response and DO NOT call next()
    res.status(401).send("Access Denied: Invalid or missing token.");
  }
};

// Application setup using the middleware:
app.get("/api/data", checkToken, (req, res) => {
  // This code only runs IF checkToken called next()
  res.json({ message: "API DATA accessed successfully!", status: "Authorized" });
});
```

- **Test URL (Access Granted):** /api/data?token=giveaccess
- **Test URL (Access Denied):** /api/data?token=wrongkey

8. Middleware Order and Placement in Express

The placement of middleware is **CRITICAL** because Express processes requests in the exact order the functions are defined.

Rule 1: Global Middleware Goes First

Any middleware you want to run for *every* route (like a logger or a JSON parser) must be defined using app.use() **before** your route definitions.

```
const app = express();
// 1. JSON Parser runs first
app.use(express.json());
// 2. Custom Logger runs second
app.use(customLogger);

// 3. This route handler runs third
app.get('/users', (req, res) => {
  // ...
});
```

Rule 2: Order of Route-Specific Middleware Matters

If you chain multiple middleware functions to a single route, they execute in the order listed:

```
// Auth runs BEFORE RoleCheck. If Auth fails, RoleCheck never executes.
app.get('/admin', authMiddleware, roleCheckMiddleware, (req, res) => {
  // ...
});
```

Rule 3: Catching 404 Errors

Because Express processes requests sequentially, if a request reaches the very end of your file without matching any defined route (app.get, app.post, etc.), it means the resource was not found.

You can implement a final "catch-all" middleware at the very end to handle this:

```
// ... All your app.get(), app.post(), and other route definitions go here ...

// 🚨 THIS MUST BE THE LAST app.use() OR app.get() CALL IN YOUR FILE!
app.use((req, res, next) => {
  // If control reaches this point, no route above matched the request.
  res.status(404).send("Error 404: Resource Not Found");
});
```