LECTURE 21

Middleware

The functions that sit between a request and its final response — logging, parsing, authentication, all in one pipeline.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. Understand the middleware concept and how requests flow through a pipeline
2. Learn the (req, res, next) function signature every middleware follows
3. Distinguish application-level, router-level, and error-handling middleware
4. Use built-in Express middleware (express.json, express.static) and popular third-party middleware (morgan, cors)
5. Write your own custom middleware function

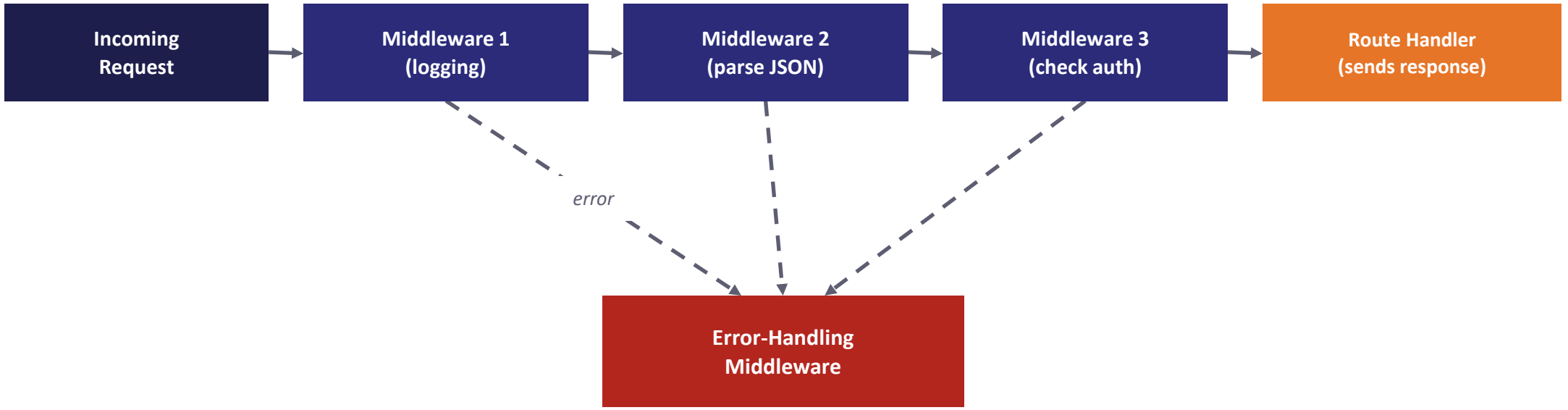
What Is Middleware?

1. A function that sits in the middle of the request and the final response
2. Express runs a chain of these functions, one after another, for every incoming request
3. Each one can run code, change req/res, end the cycle by sending a response, or pass control to the next function

The Airport Security Analogy

1. Your bag doesn't go straight from check-in to the plane — it passes through several stations
2. Ticket check, X-ray scanner, manual inspection — each one either lets it through or stops it there
3. Express middleware works the same way: every request passes through a chain of stations before it reaches its final handler

The Request/Response Pipeline



Often called the request/response pipeline. Any middleware can call next(err) to skip straight to the nearest error-handling middleware, registered last, after every other app.use() and route.

Every Middleware Gets (req, res, next)

```
function myMiddleware(req, res, next) {
  // req, res, and next -- the standard trio
  console.log(`${req.method} ${req.url}`);
  next(); // move on to the next middleware
}
```

Forgetting next() — without also sending a response — makes the client hang forever waiting for a reply.

Always Call next() or End the Response

WARNING

A middleware function must always either call next() or end the response (with res.send(), res.json(), res.end(), etc.). Doing neither is one of the most common bugs beginners hit with Express — the request just times out.

Registering Middleware with app.use()

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

app.use(function (req, res, next) {
  console.log(`${req.method} ${req.url}`);
  next();
});

app.get("/", (req, res) => {
  res.send("Home page");
});

app.listen(3000);
```

Every request — whichever route it eventually matches — passes through the logging middleware first, since it was registered before the routes.

Order Matters

TIP

Express runs middleware in the exact order you register it. A middleware registered after your routes will never run for requests an earlier route already handled.

Global vs. Path-Scoped vs. Route-Scoped

```
app.use((req, res, next) => {  // every request
  req.requestTime = Date.now();
  next();
});

app.use("/admin", (req, res, next) => { // /admin only
  console.log("Someone hit the admin area");
  next();
});

app.get("/profile", (req, res, next) => {
  next();
}, (req, res) => {
  res.send("Profile page");
});
```

A route can have multiple handler functions — each must call next() except the last, which usually sends the response.

routes/users.js

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

router.use((req, res, next) => {
  console.log("Time:", Date.now());
  next();
});

router.get("/", (req, res) => {
  res.send("List of users");
});

router.get("/:id", (req, res) => {
  res.send(`User with id ${req.params.id}`);
});

module.exports = router;
```

Mounting the Router

```
// app.js
const usersRouter = require("./routes/users");
app.use("/users", usersRouter);
```

Now the logging middleware inside users.js only fires for requests under /users/* — never for, say, /products.

Four Parameters, Not Three

1. Error-handling middleware looks almost the same, but takes FOUR parameters: (err, req, res, next)
2. Express recognizes it as an error handler purely because of that fourth parameter
3. Must be registered LAST, after all other app.use() and route calls

A Basic Error Handler

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    error: "Something went wrong on the server."
  });
});
```

Express recognizes this as an error handler purely because it takes four parameters, not three.

Reaching It with next(err)

```
app.get("/risky", (req, res, next) => {
  try {
    doSomethingThatMightFail();
    res.send("It worked!");
  } catch (err) {
    next(err); // hand off to the error handler
  }
});

// ... other routes ...

app.use((err, req, res, next) => {
  console.error(err.message);
  res.status(500).send("Internal Server Error");
});
```

next(err) skips every remaining regular middleware and jumps straight to the nearest error handler.

Chaining Error Handlers

NOTE

You can have multiple error-handling middleware functions — one that logs and calls next(err) again, and a final one that sends the response. The same chain idea applies to errors too.

Express Ships With a Few Built In

1. express.json() — parses JSON request bodies into req.body
2. express.urlencoded() — parses traditional HTML <form> submissions
3. express.static() — serves static files (HTML, CSS, images, JS) directly from a folder

express.json()

```
app.use(express.json());

app.post("/api/notes", (req, res) => {
  console.log(req.body); // parsed JSON body
  res.status(201).json({ message: "Note created" });
});
```

Without express.json(), req.body would be undefined for JSON requests.

express.static()

```
app.use(express.static("public"));

// GET /logo.png now automatically serves
// public/logo.png -- no route needed
```

Express matches the request path directly to a file in the folder.

Two You'll Use Constantly

MORGAN — REQUEST LOGGING

- Logs every incoming request: method, path, status code, response time
- npm install morgan
- app.use(morgan("dev"))
- Invaluable while developing and debugging

CORS — CROSS-ORIGIN REQUESTS

- Browsers block a page on one origin from calling an API on another (same-origin policy)
- npm install cors
- app.use(cors()) adds the headers needed to explicitly allow it
- Restrict the allowed origin before production

morgan in Practice

```
npm install morgan

const morgan = require("morgan");
app.use(morgan("dev"));
// logs: GET /users 200 12.345 ms
```

cors in Practice

```
npm install cors

const cors = require("cors");
app.use(cors()); // any origin -- fine for dev

// Production: restrict it
app.use(cors({ origin: "https://myapp.com" }));
```

Lock Down cors() Before Production

WARNING

cors() with no options allows ANY website to call your API from the browser. That's convenient during development but should usually be locked down to specific origins before you deploy to production.

A Custom Authentication Check

```
function requireLogin(req, res, next) {
  if (!req.session || !req.session.userId) {
    return res.status(401).json({
      error: "You must be logged in."
    });
  }
  next();
}

app.get("/dashboard", requireLogin, (req, res) => {
  res.send("Welcome to your dashboard!");
});
```

If the check fails, requireLogin sends 401 and does NOT call next() — the /dashboard handler never runs.

Chaining Several Small Middleware

```
function logRequest(req, res, next) {
  console.log(`${req.method} ${req.url}`);
  next();
}

function validateNoteBody(req, res, next) {
  if (!req.body.title) {
    return res.status(400).json({ error: "Title required." });
  }
  next();
}

app.post("/api/notes", logRequest, validateNoteBody, (req, res) => {
  res.status(201).json({ message: "Note created" });
});
```

Small, single-purpose middleware chained together is one of the biggest reasons Express apps stay readable as they grow.

KEY TAKEAWAYS

Lecture 21 in Seven Points

1. Middleware functions run in a chain (the request/response pipeline) for each incoming request, in the order they are registered.
2. Every middleware function receives (req, res, next); it must call next() or end the response, or the request will hang.
3. Application-level middleware (app.use()) applies globally or to a path prefix; router-level middleware applies only within an express.Router().
4. Error-handling middleware has four parameters, (err, req, res, next), must be registered last, and is reached via next(err).
5. Express ships with built-in middleware like express.json(), express.urlencoded(), and express.static().
6. Third-party middleware such as morgan (logging) and cors (cross-origin requests) are installed from npm and plugged in the same way.
7. Writing your own middleware — for logging, validation, or authentication — is one of the most common and powerful patterns in Express development.