# lectures # week-2

# Express Middleware Pipeline (Thursday, April 9, 2026)

**Lecture Recording**

Watch on Panopto

## Today's Agenda

1. **Course admin** — quiz, check-off, Sprint 0, upcoming group meetings
2. **Tuesday format change** — Tuesdays become group meeting days
3. **Code quality tools** — new ESLint/Prettier guide, linters vs. formatters
4. **Postman** — why you need more than a browser for API testing
5. **Middleware pipeline** — controllers, middleware functions, and how Express builds request-handling pipelines
6. **Code walkthrough** — controllers, middleware, route building, and the Express Router pattern

---

## Plan & Admin 0:00

### What You Should Be Working On

- **Quiz 2** is live — do the readings first, then take the quiz
- **Check-off assignment** — about 10 students done so far. Peer check-offs don't have to be reciprocal — anyone who's completed the assignment can check someone off
- **Sprint 0** — the lift isn't heavy, but next week's sprint will ramp up significantly

### OpenAPI Documentation 2:02

The YAML-based OpenAPI documentation is the biggest pain point in the check-off. Advice:

- **Let the AI write it** — don't suffer through YAML by hand. Tell the AI what the endpoint does in plain English and have it generate the YAML.
- **Better workflow for future sprints:** Tell the AI your requirements → have it write the OpenAPI docs → use those docs to generate tests → use a separate agent to write the code. Keep these three steps isolated (separate folders or agents) so each one works from the spec, not from peeking at the code.

---

## Tuesday Group Meetings 5:13

Starting next week, **Tuesdays are group meeting days** — no lecture. Groups sign up for a time slot and meet with the instructor in the classroom.

- **Sprint 0 meetings** — 10 minutes each. The instructor checks in on group dynamics: Are you communicating? Any issues?
- **Sprint 1+ meetings** — 15–20 minutes. Groups must have their **MVP running on a laptop** to demonstrate. No excuses about broken pushes or Windows updates.
- **Longer meetings** may not fit in class time — some groups may need to meet before/after class or via Zoom.

---

## Code Quality Tools 8:49

A new guide is available: Code Quality Tools. It walks through setting up ESLint and Prettier. The lecture demo repo (Backend 2) has updated, stricter rules — Sprint 1 includes collapsible setup instructions for adding these to your group project.

> **In the Guides**
>
> Code Quality Tools covers ESLint configuration, Prettier setup, and IDE integration.

### Linters vs. Formatters 10:39

| Tool | What It Does | Think of it as... |
| --- | --- | --- |
| **TypeScript compiler (`tsc`)** | Checks type errors — wrong types, mismatched return values | Type-level correctness |
| **ESLint** | Checks for logic issues and code quality — unused variables, unreachable code, common bugs | Code reviewer |
| **Prettier** | Enforces consistent formatting — semicolons, quote style, brace placement | Code stylist |

**Prettier isn't about making code "pretty"** — it's about making everyone's code **consistent**. In a group project, you don't want one file with next-line braces and another with same-line braces.

## Semicolons in JavaScript 13:14

Semicolons are optional in most JavaScript contexts — `let name = 5` and `let name = 5;` are both valid. They're only strictly required when multiple statements share a line. Don't worry about the rules — configure Prettier to require semicolons and let your IDE auto-fix on save.

## IDE Integration

Whether linting shows up visually (squiggly underlines) depends on your IDE setup:

- **WebStorm** picks up ESLint automatically
- **VS Code** may need plugins configured (newer versions detect ESLint automatically)
- **No IDE?** Run `npm run lint` and `npm run format` manually from the terminal

**Student Question**

Q: Is Prettier like CheckStyle from Java?

A: Same concept, different tool. They both enforce code formatting consistency. Neither is more "efficient" than the other — they're just tools for different ecosystems, like comparing VS Code to WebStorm.

## Postman & HTTP Testing 17:21

You can test GET endpoints by typing a URL into a browser — but a browser **only sends GET requests**. You can't send POST, PUT, PATCH, or DELETE. You also can't set custom headers or an HTTP body.

**Postman** is a tool that lets you build an HTTP request exactly how you want it — choose the method, set headers, add a body, configure query strings — and send it. It also has a built-in testing framework (JavaScript-based) for writing assertions against responses.

**Why not use Postman for all testing?** Postman tests can't be integrated into a CI/CD pipeline. When you push to GitHub, you can't automatically run Postman tests as part of your build. That's why we use Jest for automated testing instead.

> **In the Guides**
>
> API Testing Tools — Postman covers using Postman for manual testing and exploration.

---

## URL Anatomy & API Versioning 23:08

### URL Structure

```
http://localhost:3000/v1/hello
|      |         |    |
|      |         |    └── Route (endpoint path)
|      |         └─── Port (which "door" on the machine)
|      └─── IP address / hostname (the building)
└── Protocol
```

- **Port** — identifies which service on the machine. `3000` is convention, not required. In production, the cloud service assigns the port — you won't type it in the URL.
- **Route** — the path to a specific endpoint

### API Versioning 24:52

Putting the version in the URL ( `/v1/` , `/v2/` ) is standard practice. **Why not just update the existing API?**

When a production API has thousands of clients — apps, websites, integrations — you can't just change how endpoints work. That would **break** every piece of software relying on the current behavior.

Instead:

1. Release **v2** with new behavior
2. **Deprecate** v1 with an end-of-life date (e.g., one year)
3. Keep v1 running until clients migrate
4. Eventually shut down v1 (usually well past the announced date)

This is the same principle as the Java standard library — methods deprecated in Java 2 are *still* available in Java 25 because removing them would break too much code.

**Student Question**

Q: Why not just merge branches instead of versioning?

A: It's about **backwards compatibility**. When you're building code that other people build their code on top of, you can't just change the API. With URL-based versioning, you can keep v1 running indefinitely while v2 evolves — it's actually easier than managing breaking changes within a single version.

---

## Routes, HTTP Methods & CRUD 26:15

An **endpoint** is defined by both a route name *and* an HTTP method. A single route can map to multiple endpoints:

| HTTP Method | Route | CRUD Operation |
|---|---|---|
| `GET` | `/v1/users` | Read all users |
| `GET` | `/v1/users/:id` | Read one user |
| `POST` | `/v1/users` | Create a user |
| `PATCH` / `PUT` | `/v1/users/:id` | Update a user |

| HTTP Method | Route | CRUD Operation |
| --- | --- | --- |
| `DELETE` | `/v1/users/:id` | Delete a user |

This differs from Java where you'd have getters and setters — here, the four CRUD operations map to five HTTP methods across shared route names.

> **In the Guides**
>
> Intro to Express — HTTP Methods in Express and Routing & Middleware — Route-Level Middleware cover method-specific route definitions.

---

## Public vs. Protected Routes 33:06

A common API pattern separates routes into **public** (open) and **protected** (requires authentication):

**Public routes** — no token required:

- **Health check** — "I'm alive" endpoint for uptime monitoring. Cloud services use this to verify the API is running.
- **Login / Register** — must be public, because the login flow is what *grants* the token. If login required a token, you'd have a chicken-and-egg problem.
- **Demo routes** — sandbox endpoints for developers evaluating the API

**Protected routes** — require a JSON Web Token (JWT):

- Everything the app charges for or restricts usage on — essentially most endpoints

The power of Express middleware: write **one** JWT-checking function, place it at the right level in the middleware pipeline, and every route behind it is protected. No need to add the check to each individual route.

We'll implement JWT authentication in **Weeks 4–5**.

---

## The Middleware Pipeline 37:51

When an HTTP request hits the Express server, `app.listen()` receives it and routes it through a **pipeline of functions**. This pipeline has two types of functions: **middleware** and a **controller**.

> **In the Guides**
>
> Routing & Middleware — The Middleware Chain covers the pipeline concept, `next()`, and middleware ordering in depth.

## Controllers 39:52

The **controller** is the last function in the pipeline. It has one absolute rule:

> **Every controller must send exactly one HTTP response.**

This means:

- **All branches** must send a response — no branch can silently exit without responding
- **No branch** can send more than one response — Express will crash your server if you try to send a second response
- The response doesn't have to be a 200 — controllers commonly branch between happy and sad paths:

```
if (x) {
    response.status(200).json({ data: result });        // Happy path
} else if (y) {
    response.status(400).json({ error: 'Bad input' });  // Sad path
} else {
    response.status(500).json({ error: 'Server error' }); // Crash path
}
```

> **Common Mistakes**
>
> 1. **Missing response on a branch** — the happy path works fine, but the sad path causes a timeout because no response is sent
> 2. **Sending two responses in one branch** — e.g., `response.status(400)` followed by `response.status(404)` in the same code path will cause a runtime error
> 3. **Async code hiding open branches** — especially with older `.then()/.catch()` promise syntax, it's easy to miss a branch that never sends a response. Using `async` / `await` makes this more visible.

## Middleware Functions 51:42

Each **middleware function** must do exactly one of two things:

| Option | When | What Happens |
|---|---|---|
| **Send an HTTP response** | Sad path — validation failed | Pipeline stops, client gets error (usually 400) |
| **Call `next()`** | Happy path — everything checks out | Request passes to the next function in the pipeline |

A middleware function **cannot do both** and **cannot do neither**. If it does neither, the request hangs and the client gets a timeout.

**Example: `express.json()` middleware:**

- Valid JSON body → parses it into a JavaScript object, calls `next()`
- Malformed JSON → sends 400
- XML body → sends 400
- Empty body → that's fine, calls `next()`

## Happy Path vs. Sad Path Summary

- **Middleware** handles *most* sad paths — validation failures, missing parameters, malformed input
- **Controllers** handle the remaining sad paths — e.g., user not found in database (404)
- **Almost always**, the happy path flows all the way through all middleware to the controller

The goal: if you build your middleware well, **controllers are mostly happy-path code**.

> ⚠ **Don't Hit the Database in Middleware**
>
> Middleware validates the *shape* of the request — "did you send me a user ID?" (if not, 400). The controller checks the *data* — "does that user ID exist in the database?" (if not, 404). Don't query the database in middleware just to check existence — wait until the controller and do it once.

## Pipeline Order Matters 1:08:16

Middleware functions execute **in the order they are added**. In `app.ts`:

```
app.use(cors());              // 1st — runs first
app.use(express.json());      // 2nd
app.use(logger);              // 3rd
```

When building pipelines across multiple files (barrel files, nested routers), you need to track the order carefully — it's less obvious than when everything is in one file.

### Route Ordering Gotcha 1:10:35

Route definition order also matters. Consider:

```
router.get('/users/:id', getUserById);   // Defined first
router.get('/users/all', getAllUsers);    // Defined second
```

A request to `GET /users/all` will match the **first** route — Express treats `"all"` as the `:id` parameter. You'll get a 404 ("no user named all") or a 400 ("id must be a number").

**Fix:** Define literal routes before parameterized routes, or restructure your route names.

---

## Code Walkthrough 1:13:24

### Controller Functions 1:13:45

Controllers are just exported functions in `src/controllers/`. They don't reference Express routes — they only define the behavior. Route wiring happens elsewhere.

**Function signature:** `(request: Request, response: Response) => { ... }`

- `_request` — the underscore prefix means "I have this parameter but I'm not using it." JavaScript requires positional parameters, so even if you only need `response`, you still need `request` in the signature because it comes first.
- `response.status(200).json({...})` — method chaining works because `.status()` returns the response object

**Naming Convention**

The lecture demo spells out `request` and `response` for clarity. Idiomatic Express code uses `req` and `res`. Both are fine — the sample code and docs for this course use the full names.

**In the Guides**

Intro to Express — Sending Responses covers `response.status()`, `.json()`, `.send()`, and method chaining.

## Middleware Functions 1:18:42

The logger middleware shows the happy-path-only pattern:

```
import { Request, Response, NextFunction } from 'express';

const logger = (_request: Request, _response: Response, next: NextFunction) =>
{
    console.log(`[${new Date().toISOString()}] Request received`);
    next();  // Always passes to the next function
};
```

**Key differences from a controller:**

- Takes a **third parameter**: `next: NextFunction` (imported from Express)
- Calls `next()` on the happy path instead of sending a response
- This is a **higher-order function** — `next` is a function passed in as an argument, and the middleware doesn't know or care what `next` actually is

**In the Guides**

Routing & Middleware — What is Middleware? and Application-Level Middleware explain the `next()` pattern and middleware signatures.

## Building Routes 1:22:17

Routes are wired up by passing functions to Express:

```
// Inline controller – function literal as argument
app.get('/charles1', (_request: Request, response: Response) =>
    response.status(200).json({ message: 'Hello Charles' })
```

```
);

// External controller + middleware — comma-separated
app.get('/charles2', mw1, controller);

// Global middleware — no route, applies to all requests
app.use(cors());
app.use(express.json());
app.use(logger);
```

**Three patterns for adding handlers:**

| Method | What It Does |
|---|---|
| `app.get(route, ...fns)` | Handles only GET requests to this route |
| `app.post(route, ...fns)` | Handles only POST requests to this route |
| `app.use(fn)` | Applies to **all** HTTP methods and **all** routes |
| `app.use(route, fn)` | Applies to all methods but only matching routes |

The handler functions are **variable-length arguments** — you can comma-separate as many middleware + controller functions as needed. They execute in order, left to right.

## Live Demo: Middleware in Action 1:24:34

A controller and middleware function built live to demonstrate the pipeline:

```
const controller = (_request: Request, response: Response) =>
    response.status(200).json({ error: 'Charles is not an error' });

const mw1 = (_request: Request, response: Response, next: NextFunction) => {
    if (/* validation fails */) {
        response.status(400).json({ error: 'Charles is an error' });
    } else {
        next();
    }
};

app.get('/charles', mw1, controller);
```

**Order matters:** If you swap to `app.get('/charles', controller, mw1)`, the controller sends a 200 immediately — the middleware never runs because controllers don't call `next()`.

**Missing response demo:** When the middleware neither sends a response nor calls `next()`, Postman sits waiting until it hits a ~30-second timeout. The server doesn't crash — it just

never responds.

### Express Router Pattern 1:35:08

The project uses **nested Router objects** to organize routes across files. This is both a JavaScript module pattern (**barrel exports**) and an Express routing pattern.

**How it works:**

1. Each route file creates its own `express.Router()` and adds handlers to it
2. Each folder's `index.ts` imports child routers and mounts them at path segments
3. The chain rolls up until `app.ts` imports one top-level router

```
hello.ts:           router.get('/', getHello)          → handles /hello
input.ts:           router.get('/', getInput)          → handles /input
v1/index.ts:        router.use('/hello', helloRouter)   → mounts at /v1/hello
                    router.use('/input', inputRouter)   → mounts at /v1/input
routes/index.ts:    router.use('/v1', v1Router)         → mounts at /v1
app.ts:             app.use(routes)                     → attaches everything
```

**The power of this pattern:** To protect all routes in the `protected/` folder with JWT authentication, add the JWT middleware **once** in the barrel file that imports `protectedRouter`. Every route inside gets the middleware automatically.

> **In the Guides**
>
> Routing & Middleware — Express Router and Organizing Routes in Files cover the nested router pattern in detail.

---

## Next Time

Tuesdays are now group meeting days — no lecture. We won't have a dedicated lecture on API testing or OpenAPI documentation — **read the guides on your own**:

- API Testing
- API Testing Tools
- OpenAPI Documentation

Next Thursday: **proxy routes** — building endpoints that call external APIs. If time permits, we'll look at some documentation and testing bits.

*This lecture outline is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.*