Routing & Middleware: The Depth of Express

One door for every request is not enough: separate routes for separate jobs, route parameters, and middleware — the guards every request must pass.

Introduction

Your server from lesson N.2 answers a handful of fixed addresses. Real backends need two more powers. First, **addresses with values inside them**: `/students/2` should mean "the student whose id is 2" — one route must serve every possible id, not just addresses you typed in advance. Express does this with **route parameters**, named slots in the address itself. Second, **shared checks**: before any route runs, the server may want to note the visit, or later check a login (lesson N.5). Code that must run on *every* request cannot be copied into every route — it belongs in **middleware**: functions that stand in the middle, between the arriving request and the route, like guards at a gate.

Together, routing and middleware are how Express apps stay organised as they grow from three routes to three hundred. *(Roman Urdu: Route batata hai kaam kya hai, middleware har request ko pehle jaanchta hai — phir andar jaane deta hai.)*

Key points

- **A route parameter is a named slot:** write `:id` in the address — `app.get("/students/:id", ...)` — and Express catches whatever fills that slot. Read it with `req.params.id`.
- **Parameters always arrive as text:** `/students/2` gives `req.params.id === "2"`, a string. Convert with `Number(...)` before comparing with numeric ids.
- **Query strings ride after a `?`:** `/students?city=Lahore` — read them with `req.query.city`. Routes match the address *before* the `?`, so one route serves every query.
- **Middleware is a function with three arguments:** `(req, res, next)`. It may inspect or change the request, answer it directly, or call **`next()`** to pass the request to the next guard or route.
  (Roman Urdu: `next()` ka matlab hai — jaanch mukammal, aage badho.)
- **Order is everything:** Express runs middleware and routes **top to bottom**, in the order you wrote them. A guard written *after* a route never sees that route's requests.
- **`app.use` mounts middleware:** `app.use(myLogger)` runs it for every request; `app.use("/admin", guard)` runs it only under that address.
- **Built-in middleware does common jobs:** `express.json()` reads JSON bodies sent to your API — without it, `req.body` is empty (lesson N.4 depends on it).

Code examples

Example 1 — One route, every student: route parameters

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

const students = [
  { id: 1, name: "Ali", marks: 70 },
  { id: 2, name: "Sara", marks: 92 },
  { id: 3, name: "Rahim", marks: 64 },
];

app.get("/students/:id", (req, res) => {
  const id = Number(req.params.id);
  const student = students.find((s) => s.id === id);
  if (!student) {
    return res.status(404).json({ error: "Student not found" });
  }
  res.json(student);
});

app.listen(3000, () => console.log("Server running on port 3000"));
```

What it does: The `:id` slot catches any value, so this single route serves `/students/1`, `/students/2` and every future id. `Number(...)` converts the text parameter before the `find`, and a missing student gets an honest **404** with a JSON error instead of a crash — the pattern every real API uses.

What you will see: `/students/2` answers `{"id":2,"name":"Sara","marks":92}`, while `/students/99` answers status 404 with `{"error":"Student not found"}`.

Example 2 — A guard at the gate: your first middleware

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

function logger(req, res, next) {
  console.log(req.method + " " + req.url);
  next();
}

app.use(logger);

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

app.get("/students", (req, res) => {
  res.json([{ id: 1, name: "Ali", marks: 70 }]);
});

app.listen(3000, () => console.log("Server running on port 3000"));
```

What it does: `logger` is middleware: Express runs it for **every** request before any route, because `app.use(logger)` stands above the routes. It writes the visit into the terminal — the server's diary habit that lesson N.9 grows into real logging — and calls `next()` so the request continues to its route. Delete the `next()` line and every request would hang forever: a guard that never opens the gate.

What you will see: each browser refresh adds a line like `GET /` or `GET /students` in the terminal, while the browser still receives the normal answers. One function, written once, now watches the whole app.

Example 3 — Query strings: same route, filtered answers

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

const students = [
  { id: 1, name: "Ali", marks: 70, city: "Lahore" },
  { id: 2, name: "Sara", marks: 92, city: "Karachi" },
  { id: 3, name: "Rahim", marks: 64, city: "Lahore" },
];

app.get("/students", (req, res) => {
  const city = req.query.city;
  if (city) {
    return res.json(students.filter((s) => s.city === city));
  }
  res.json(students);
});

app.listen(3000, () => console.log("Server running on port 3000"));
```

What it does: The address `/students?city=Lahore` still matches the `/students` route — the part after `?` is not part of the route. It arrives as `req.query`, ready-made instructions for *how* to answer: here, filter the list (the `filter` method from lesson 3.11) before sending it. No query, full list; a query, a filtered list.

What you will see: `/students` answers all three students; `/students?city=Lahore` answers only Ali and Rahim; `/students?city=Multan` answers an empty array `[]` — a correct answer, honestly given.

Quick recap

- **Route parameters** (`:id`, read as `req.params.id`) let one route serve endless addresses; parameters arrive as **text**.
- **Query strings** (`req.query`) carry optional instructions like filters.
- **Middleware** `(req, res, next)` runs before routes, in written order; `next()` passes the request onward.
- `app.use` mounts middleware for all requests, or only under one address.

Common mistakes

1. **Comparing the parameter without converting** — `s.id === req.params.id` compares a number with text `"2"` and never matches, so every student looks "not found". The fix: `Number(req.params.id)` first, then compare — or store ids as strings everywhere, consistently.
2. **Forgetting `next()`** — middleware that neither answers nor calls `next()` leaves the request hanging; the browser spins until it times out. The fix: every middleware path must end in exactly one of `res.something(...)` or `next()`.

3. **Writing the guard below the routes** — middleware only sees requests that arrive *after* its line in the file. A logger or login check placed at the bottom guards nothing. The fix: `app.use(...)` guards go **above** the routes they protect.

Practice

1. Extend Example 1 with a second parameterised route, `/students/:id/marks`, that answers only `{ "name": ..., "marks": ... }` for that student (and the same honest 404 when the id does not exist).
2. Write a `slowDown` practice middleware that prints `Request received at:` plus the current time for every request, mount it with `app.use`, and confirm it fires for a route above which it sits — then move it *below* one route and observe which requests it stops seeing.

Practice quiz

1. For app.get("/students/:id", ...), how do you read the id for /students/2, and what type is it?

- req.query.id, as a number
- req.params.id, and it always arrives as text, "2", so convert with Number(...) before comparing with numeric ids
- req.body.id, as a number
- It cannot be read

Answer: req.params.id, and it always arrives as text, "2", so convert with Number(...) before comparing with numeric ids

2. What happens if middleware neither answers nor calls next()?

- Express skips it automatically
- The request hangs and the browser spins until it times out; every middleware path must end in exactly one of res.something(...) or next()
- The next route runs anyway
- The server crashes at once

Answer: The request hangs and the browser spins until it times out; every middleware path must end in exactly one of res.something(...) or next()

3. Why does a guard written after a route never protect that route?

- Because guards only work on POST routes
- Because Express runs middleware and routes top to bottom, in the order you wrote them; a guard below a route never sees that route's requests
- Because app.use only works once
- Guards always protect everything, order is irrelevant

Answer: Because Express runs middleware and routes top to bottom, in the order you wrote them; a guard below a route never sees that route's requests

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/node-routing-middleware · More free tools and notes at https://scopetreasure.com