express # guide

Routing & Middleware

TCSS 460 — Client/Server Programming

In the Intro to Express guide, you wrote route handlers that receive a request and send a response. But what happens *between* the request arriving and your handler running? That is where **middleware** comes in. Middleware lets you run code before (or after) your route handlers — logging, parsing JSON, checking authentication, validating input — without cluttering the handlers themselves. This guide also covers how to use Express `Router` to organize routes across multiple files as your API grows.

---

1 What is Middleware?

Middleware is a function that sits between the incoming request and your route handler. It can inspect the request, modify it, send a response early, or pass control to the next function in the chain.

Every middleware function has the same signature:

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

const myMiddleware = (request: Request, response: Response, next:
NextFunction) => {
    // Do something with the request
    next(); // Pass control to the next middleware or route handler
};
```

The key is `next()`. If you call it, the request continues down the chain. If you don't, the request stops here — useful for sending an error response before the handler ever runs.

---

2 The Middleware Chain

When a request arrives, Express runs functions in the order they were registered. A middleware function can modify `request` or `response` (add data, set headers), but when it

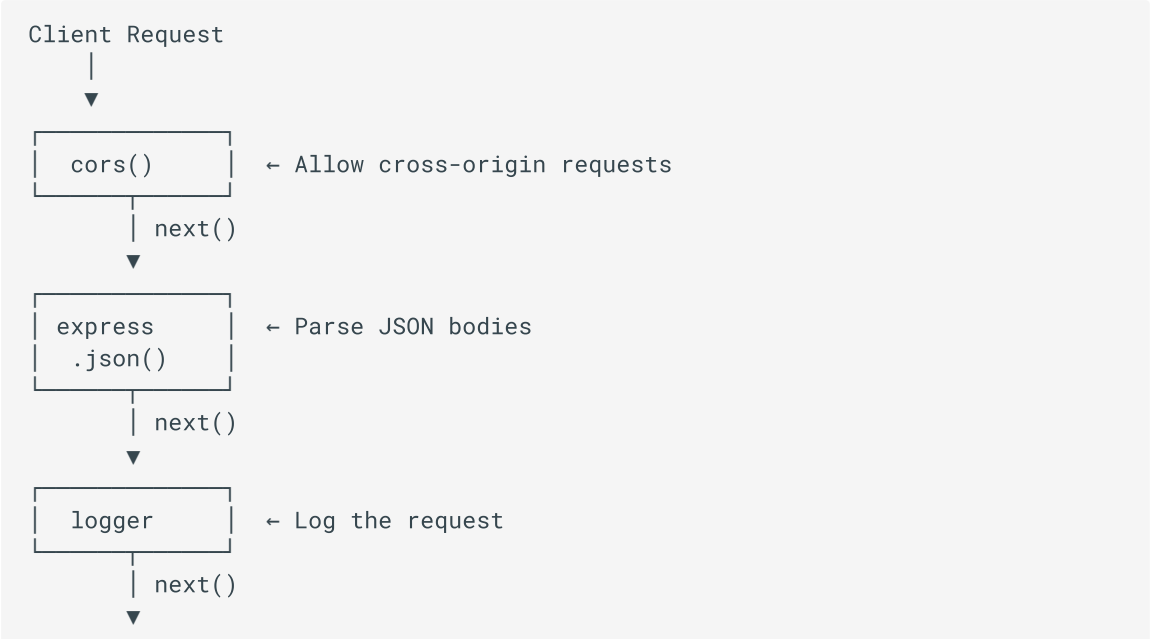
is done, it **must** do one of two things:

1. **Call** `next()` to pass control to the next function in the chain
2. **Send a response** (`response.json()` , `response.status().json()` , etc.) to end the chain

There is no third option.

> **The #1 Middleware Rule**
>
> Every code path in a middleware function must either call `next()` or send a response — never both, never neither. If you forget both, the request hangs forever — the client waits, the server does nothing, and eventually the request times out. If a middleware has branches (like an `if/else` for validation), each branch must end with `next()` or a response.
>
> ```
> const validate = (request: Request, response: Response, next:
> NextFunction) => {
>     if (!request.query.q) {
>         response.status(400).json({ error: 'Missing q' }); // ←
> sends response, stops chain
>         return;
>     }
>     next(); // ← passes control, chain continues
> };
> ```
>
> Both the `if` branch and the `else` path (after the `if`) end correctly. If you removed `return` after the response, Express would call `next()` too — sending a response AND continuing the chain, which causes the "Cannot set headers after they are sent" error.

Here is the mental model:



```
┌──────────────┐
│  Route       │  ← Your handler runs, sends response
│  Handler     │
└──────┬───────┘
       │
       ▼
Client Response
```

Try It Yourself

1. Start the lecture demo: `npm run dev`
2. Send `GET /v1/hello` — watch the terminal for the logger output
3. Now send `GET /v1/input/users/42` — notice how the logger prints the method and path for every request
4. Send `GET /v2/input/users/abc` — the validation middleware returns `400` and the handler never runs. The logger still prints the request because it runs before validation.

2.1 Order Matters

Middleware runs in the order you register it. This is a common source of bugs:

```
// CORRECT: JSON parser before routes
app.use(express.json());
app.post('/users', createUser);  // request.body is populated

// WRONG: Routes before JSON parser
app.post('/users', createUser);  // request.body is undefined!
app.use(express.json());
```

The Most Common Middleware Bug

If `request.body` is `undefined`, the JSON parser middleware is either missing or registered after the route. Always register `express.json()` before your routes.

---

3 Built-in Middleware

Express ships with a few middleware functions built in. You do not need to install anything extra.

3.1 `express.json()`

Parses incoming JSON request bodies and makes the result available as `request.body`:

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

Without this, any POST or PUT request with a JSON body will have `request.body` as `undefined`.

3.2 `express.static()`

Serves static files (HTML, CSS, images) from a directory. Less common for API servers, but useful to know:

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

You will not use this in your API projects, but it is how traditional web servers serve files.

---

4 Application-Level Middleware

When you register middleware with `app.use()` without specifying a path, it runs on every request that reaches your app. This is called **application-level middleware**.

4.1 Logging Middleware

The lecture demo includes a simple logger in `middleware/logger.ts`:

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

export const logger = (request: Request, _response: Response, next:
NextFunction) => {
    const timestamp = new Date().toISOString();
    console.log(`[${timestamp}] ${request.method} ${request.path}`);
    next();
};
```

It is registered in `app.ts` so every request is logged:

```
import { logger } from './middleware/logger';

const app = express();
```

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

When you send `GET /v1/hello`, you see in the terminal:

```
[2026-04-01T14:30:00.000Z] GET /v1/hello
```

4.2 CORS Middleware

The `cors` package is third-party middleware that handles Cross-Origin Resource Sharing — it tells browsers which domains are allowed to call your API:

```
import cors from 'cors';

app.use(cors());
```

Without CORS middleware, a front-end app running on `localhost:3001` cannot make requests to your API on `localhost:3000`. The browser blocks it. CORS middleware adds the headers that tell the browser "this is allowed."

CORS Is Configurable

Calling `cors()` with no arguments allows requests from **any origin** — fine for development, but not what you want in production. You can restrict it to specific domains:

```
app.use(cors({ origin: 'https://your-frontend.com' }));
```

This is not covered in depth in this guide, but be aware that CORS configuration is one of the things you will need to set up when you deploy your API and connect it to your front end.

---

5 Route-Level Middleware

Unlike application-level middleware that runs on every request, route-level middleware runs only on specific routes. You pass it as an argument before the route handler.

5.1 v1 vs. v2 in the Lecture Demo

The lecture demo uses versioning to show middleware in action. The v1 routes have **no middleware** — the handler runs directly. The v2 routes add **validation middleware** before the same handlers:

v1 — no middleware:

```
// routes/v1/input.ts
inputRouter.get('/search', searchByQuery);
inputRouter.get('/users/:id', getUserById);
inputRouter.post('/users', createUser);
```

v2 — with validation middleware:

```
// routes/v2/input.ts
inputRouter.get('/search', validateSearchQuery, searchByQuery);
inputRouter.get('/users/:id', validateNumericId, getUserById);
inputRouter.post('/users', validateUserBody, createUser);
```

The same controllers handle the request in both versions. The only difference is whether validation runs first.

5.2 How Route-Level Middleware Works

When you write:

```
router.get('/users/:id', validateNumericId, getUserById);
```

Express runs `validateNumericId` first. If the ID is valid, it calls `next()` and `getUserById` runs. If the ID is invalid, the middleware sends a `400` response and never calls `next()` — the handler is skipped entirely.

Here is `validateNumericId` from `middleware/validation.ts`:

```
export const validateNumericId = (request: Request, response: Response, next:
NextFunction) => {
    const id = Number(request.params.id);
    if (!Number.isInteger(id) || id <= 0) {
        response.status(400).json({ error: 'Parameter "id" must be a positive
integer' });
        return;
    }
    next();
};
```

Send `GET /v2/input/users/abc` and you get:

```
{ "error": "Parameter \"id\" must be a positive integer" }
```

The `getUserById` handler never executes.

5.3 Multiple Middleware on One Route

You can chain several middleware functions before the handler:

```
router.get('/weather', requireEnvVar('WEATHER_API_KEY'), requireCity,
getWeather);
```

Express runs them left to right: `requireEnvVar` → `requireCity` → `getWeather`. If any middleware sends a response without calling `next()`, the chain stops.

Two Ways to Chain Middleware

You can pass multiple middleware as separate arguments on the route definition (as shown above), or apply them with separate `.use()` calls on the router. These are equivalent:

```
// Option A: inline on the route
router.get('/weather', requireEnvVar('WEATHER_API_KEY'),
requireCity, getWeather);

// Option B: separate .use() calls
router.use(requireEnvVar('WEATHER_API_KEY'));
router.use(requireCity);
router.get('/weather', getWeather);
```

The difference: Option A applies the middleware only to the `/weather` GET route. Option B applies `requireEnvVar` and `requireCity` to **every** route on the router. Choose based on whether the middleware should apply to one route or all routes in the group.

---

6 Express Router

As your API grows beyond a few routes, you need a way to group related routes together. Express provides `Router` for this.

6.1 Creating a Router

A `Router` is like a mini Express app — it has its own `.get()`, `.post()`, `.use()`, and so on:

```
import { Router } from 'express';

const helloRouter = Router();

helloRouter.get('/', getHello);
helloRouter.post('/', postHello);
helloRouter.put('/', putHello);
helloRouter.patch('/', patchHello);
helloRouter.delete('/', deleteHello);

export { helloRouter };
```

6.2 Mounting a Router

You attach a router to a path prefix using `app.use()` or `router.use()`:

```
v1Routes.use('/hello', helloRouter);
```

Now all routes in `helloRouter` are prefixed with `/hello`:

- `helloRouter.get('/')` → `GET /hello`
- `helloRouter.post('/')` → `POST /hello`

6.3 Nested Routers

Routers can mount other routers. The lecture demo nests three levels deep:

```
// routes/index.ts — the top-level router
const routes = Router();
routes.use('/v1', v1Routes);
routes.use('/v2', v2Routes);
routes.use(protectedRoutes);
```

```
// routes/v1/index.ts — groups v1 feature routers
const v1Routes = Router();
v1Routes.use('/hello', helloRouter);
v1Routes.use('/input', inputRouter);
```

The result: `helloRouter.get('/')` becomes `GET /v1/hello` — the prefixes stack.

Router Level	Prefix Added	Cumulative Path
`routes` mounts `v1Routes`	`/v1`	`/v1`

Router Level	Prefix Added	Cumulative Path
`v1Routes` mounts `helloRouter`	`/hello`	`/v1/hello`
`helloRouter.get('/')`	`/`	`GET /v1/hello`

6.4 Router-Level Middleware

You can apply middleware to an entire router, so every route under it inherits the middleware:

```
// routes/protected/proxy.ts
const proxyRouter = Router();

// All proxy routes require the API key to be configured
proxyRouter.use(requireEnvVar('WEATHER_API_KEY'));

proxyRouter.get('/weather', requireCity, getWeather);
proxyRouter.get('/summary', requireCity, getWeatherSummary);
```

The `requireEnvVar` middleware runs before every route in `proxyRouter`. Individual routes can add their own middleware on top (like `requireCity`).

---

7 Organizing Routes in Files

The lecture demo shows a clean pattern for splitting routes across files. Each file exports a `Router`, and a central file mounts them all.

7.1 The Pattern

Step 1: Each feature gets its own route file that exports a router:

```
// routes/v1/hello.ts
import { Router } from 'express';
import { getHello, postHello, putHello, patchHello, deleteHello } from
'../../controllers/hello';

const helloRouter = Router();

helloRouter.get('/', getHello);
helloRouter.post('/', postHello);
helloRouter.put('/', putHello);
helloRouter.patch('/', patchHello);
```

```
helloRouter.delete('/', deleteHello);

export { helloRouter };
```

Step 2: A version index file mounts the feature routers:

```
// routes/v1/index.ts
import { Router } from 'express';
import { helloRouter } from './hello';
import { inputRouter } from './input';

const v1Routes = Router();

v1Routes.use('/hello', helloRouter);
v1Routes.use('/input', inputRouter);

export { v1Routes };
```

Step 3: The top-level routes file mounts the version routers:

```
// routes/index.ts
import { Router } from 'express';
import { v1Routes } from './v1';
import { v2Routes } from './v2';
import { protectedRoutes } from './protected';

const routes = Router();

routes.use('/v1', v1Routes);
routes.use('/v2', v2Routes);
routes.use(protectedRoutes);

export { routes };
```

Step 4: The app mounts the top-level router:

```
// app.ts
app.use(routes);
```

7.2 Why This Structure Works

Try It Yourself

1. Open the lecture demo and look at `src/routes/v1/index.ts`
2. Create a new file `src/controllers/greeting.ts` with a handler that returns `{ message: "Hi there!" }`
3. Create `src/routes/v1/greeting.ts` with a Router that maps `GET /` to your handler
4. Mount your router in `src/routes/v1/index.ts`: `v1Routes.use('/greeting', greetingRouter)`
5. Restart the server and visit `http://localhost:3000/v1/greeting`
6. Notice how the path prefix stacked: your router's `/` became `/v1/greeting`

- **Adding a new feature** = create a new route file + controller file, mount in the version index
- **Adding middleware** = create a middleware file, import it in the route file
- **Changing a handler** = edit only the controller file
- **Testing** = import the `app` from `app.ts` without starting the server

Start Simple

You do not need this full structure on day one. Start with everything in one file, then split when the file gets long or when you add a second group of routes. The lecture demo uses this structure because it demonstrates the patterns you will need in your group project.

Gen AI & Learning: AI Agents and Route Organization

When you ask an AI agent to "add a new route for movies," a well-organized project makes all the difference. If your routes follow the pattern shown here — feature routers mounted in a version index — the agent will create `routes/v1/movies.ts` and `controllers/movies.ts` without being told. It learned the pattern from your existing files. If your project has routes scattered across random files with no consistent structure, the agent will guess — and guess wrong. Good route organization is context engineering.

---

8 Error Handling Middleware

Express has a special kind of middleware for errors — it takes **four** parameters instead of three:

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

const errorHandler = (err: Error, _request: Request, response: Response,
_next: NextFunction) => {
    console.error(err.stack);
    response.status(500).json({ error: 'Something went wrong' });
};
```

> **Important**
>
> The four-parameter signature `(err, request, response, next)` is how Express knows this is an error handler, not regular middleware. All four parameters must be present even if you don't use `next`.

8.1 Registering the Error Handler

Error handling middleware must be registered **after** all your routes:

```
// app.ts
app.use(express.json());
app.use(logger);
app.use(routes);               // All routes first
app.use(errorHandler);          // Error handler last
```

If an error occurs in a route handler, Express skips all remaining regular middleware and jumps to the error handler.

8.2 Catching Async Errors

In Express 5, errors thrown inside `async` route handlers are caught automatically and forwarded to the error handler. This is a significant improvement over Express 4, where you had to wrap every async handler in a try/catch.

```
// Express 5 — async errors are caught automatically
app.get('/data', async (request, response) => {
    const data = await fetchFromDatabase(); // If this throws, Express catches
it
    response.json(data);
});
```

You can still use explicit try/catch when you want to send a specific error response instead of relying on the global handler:

```
app.get('/data', async (request, response) => {
    try {
```

```
        const data = await fetchFromDatabase();
        response.json(data);
    } catch (error) {
        response.status(502).json({ error: 'Failed to reach database' });
    }
});
```

The lecture demo's proxy controllers use this pattern — they catch fetch errors and send a `502 Bad Gateway` response instead of letting them bubble to a global handler.

---

9 Middleware Order Checklist

Order is the most common source of middleware bugs. Here is the correct order for a typical Express API:

```
// 1. CORS — must be first so preflight requests are handled
app.use(cors());

// 2. Body parsing — before any route that reads request.body
app.use(express.json());

// 3. Application-level middleware — logging, etc.
app.use(logger);

// 4. API documentation
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(spec));

// 5. Routes — with their own route-level middleware
app.use(routes);

// 6. Error handler — always last
app.use(errorHandler);
```

Gen AI & Learning: Middleware Order Bugs

Middleware ordering bugs are some of the hardest to debug because the error message rarely points to the real cause. If `request.body` is undefined, the problem is not in your route handler — it is in `app.ts` where the JSON parser is registered after the routes. AI agents are good at spotting these issues if you tell them: "request.body is undefined in my POST handler — check the middleware order in app.ts." Give the agent the right file to look at, and it will find the ordering bug quickly.

⚠ Common Mistakes

- **Forgetting `next()`** — Your middleware runs but the request hangs. The client waits forever.
- **Calling `next()` after sending a response** — Express warns "Cannot set headers after they are sent." If you send a response, don't call `next()`.
- **Wrong order** — JSON parser after routes means `request.body` is undefined. Error handler before routes means it never catches anything.

---

10 Summary

Concept	Key Point
Middleware	Functions that run between request and response: `(request, response, next)`
`next()`	Passes control to the next middleware; without it, the chain stops
Application-level	`app.use(middleware)` — runs on every request
Route-level	`router.get('/path', middleware, handler)` — runs on specific routes
`express.json()`	Built-in middleware to parse JSON bodies
`cors()`	Third-party middleware to allow cross-origin requests
`Router`	Groups related routes together with shared path prefixes
Nested routers	Routers can mount other routers — prefixes stack
Error middleware	Four parameters: `(err, request, response, next)` — must be registered last
Order matters	JSON parser → logger → routes → error handler

---

11 References

Official Documentation:

- Express.js — Using Middleware — Application, router, and error-handling middleware
- Express.js — Routing — `express.Router()` and route grouping
- Express.js — Error Handling — Error middleware and async error handling
- Express.js — API Reference: `express.json()` — JSON body parser options
- cors Package — CORS middleware configuration

TCSS 460 Lecture Demo

github.com/UWT-SET-TCSS460-LECTURE-MATERIALS/TCSS-460-Backend-1 — v1 vs. v2 middleware comparison

---

12 Further Reading

External Resources

- Express.js — Writing Middleware — Official guide to creating custom middleware
- MDN — CORS — How cross-origin resource sharing works
- Express 5 Migration Guide — Async error handling improvements in Express 5

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