

Error Handling + Logging: The Server's Diary

Every server meets errors: one error-handling middleware to catch them all, honest answers for users, and log files that remember what happened.

Introduction

Sooner or later every route meets trouble: the database is asleep, a file is missing, a value is the wrong shape. Without a plan, each failure behaves differently — one route crashes the whole server, another spins forever, a third answers a page of internal code that helps nobody and leaks your file paths to strangers. This lesson installs the professional pattern: **one error-handling middleware** at the very bottom of the file that catches errors from everywhere, decides what the *user* should be told, and writes what *you* need into a **log** — the server's diary.

Logging is the second half of the craft. When a user says "it broke yesterday evening", memory and guesswork are useless; the log remembers the time, the address and the error. A server without a diary is a shop without a ledger. (*Roman Urdu: Ghaltiyan hongi — sawaal ye hai ke jawab kya milega aur diary mein kya likha jayega.*)

Key points

- **Errors travel to one place:** when a route calls `next(err)` — or an `async` route's promise fails — Express skips the remaining normal routes and looks for the error handler. You stop writing `try/catch` answers in every route.
- **The error handler has four arguments:** `(err, req, res, next)`. That fourth argument is how Express recognises it as the error handler — it must be mounted **last**, after every route.
- **Two audiences, two messages:** the *user* gets a short, safe answer ("Something went wrong on the server") with the right status. The *log* gets the full truth: the real error, the address, the time. Never send internal details — file paths, database messages — to visitors.
(*Roman Urdu: User ko izzat wala mukhtasar jawab; diary mein poori sachai.*)
- **Status codes still apply:** keep errors that carry their own status (your 400s and 404s from earlier lessons) intact; only unknown failures become **500**. A tiny habit — `err.status || 500` — does this cleanly.
- **A log line has a shape:** time, method, address, and what happened: `2026-10-24 21:14:03 GET /students/99 - Student not found`. Same shape every time is what makes a diary searchable later.
- **Console is the first diary, files are the lasting one:** `console.log` vanishes when the terminal closes. Appending lines to a file with `fs.appendFile` keeps the record on disk, date by date.
- **A crashed server helps nobody:** the error handler's final job is keeping the process alive for the next visitor — answer the error, stay standing.

Code examples

Example 1 — The central error handler

```
const express = require("express");
const app = express();
app.use(express.json());

app.get("/students/:id", (req, res, next) => {
  const id = Number(req.params.id);
  if (id !== 1) {
    const err = new Error("Student not found");
    err.status = 404;
    return next(err);
  }
  res.json({ id: 1, name: "Ali", marks: 90 });
});

app.get("/crash-test", (req, res, next) => {
  next(new Error("Database is asleep"));
});

// The error handler – always LAST, always four arguments.
app.use((err, req, res, next) => {
  console.log("ERROR:", err.message);
  res.status(err.status || 500).json({
    error: err.status ? err.message : "Something went wrong on the server",
  });
});

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

What it does: Routes now *report* errors instead of answering them: build an `Error`, attach a status when you know it, hand it to `next(err)`. The single handler at the bottom logs the full truth for you, then answers the visitor — the error's own message when it is a safe, expected one (404), or a calm general sentence for unknown failures (500), whose real details stay in the log and out of the response.

What you will see: `/students/99` answers 404 with `{"error":"Student not found"}`; `/crash-test` answers 500 with the general message, while the terminal's log line shows the real `Database is asleep`; `/students/1` still answers normally, and the server never stops.

Example 2 — The diary: logging every visit to a file

```
const fs = require("fs");

function logger(req, res, next) {
  const line =
    new Date().toISOString() + " " + req.method + " " + req.url + "\n";
  fs.appendFile("server.log", line, () => {});
  next();
}

app.use(logger);
```

What it does: The logger middleware from lesson N.3 grows into its adult form. Every request appends one shaped line to `server.log` — ISO time, method, address — using `fs.appendFile`, which creates the file if absent and never erases yesterday's lines. The empty callback swallows logging's own failures: the diary must never be allowed to break the shop.

What you will see: after a few visits, `server.log` holds lines like `2026-10-24T16:14:03.000Z GET /students` and `2026-10-24T16:15:41.000Z POST /students` — a true, timed history of everything the server did, surviving restarts.

Example 3 — Async routes: errors that arrive as promises

```
app.get("/report", async (req, res, next) => {
  try {
    const data = await Promise.reject(new Error("Report file missing"));
    res.json(data);
  } catch (err) {
    next(err);
  }
});
```

What it does: Database and file calls are `async` (lesson 3.21), and a failed `await` throws *inside the promise* — Express cannot catch that by itself in this version, so the route wraps the work in `try/catch` and forwards whatever it catches with `next(err)`. The stand-in line here rejects on purpose, playing the part of a missing report file or a sleeping database: the failure lands in the same central handler, gets logged with its real message, and the visitor receives the calm 500.

What you will see: `GET /report` answers 500 with `{"error":"Something went wrong on the server"}`, the terminal logs `ERROR: Report file missing`, and the very next request to any healthy route answers as if nothing happened.

Quick recap

- One **error-handling middleware**, four arguments, mounted **last**, catches errors from every route via `next(err)`.
- Users get short, safe answers with honest statuses; unknown failures are **500** with a general message.
- The **log** gets the whole truth — time, method, address, real error — in a consistent shape.
- `fs.appendFile` turns console notes into a lasting diary; `async` routes forward failures with `try/catch` + `next(err)`.

Common mistakes

1. **Sending the real error to the visitor** — answering with the database's message or a stack trace leaks file paths and internals, and frightens users. The fix: log the truth, answer generally; only errors you *designed* (400, 404) may speak in their own words.
2. **Mounting the error handler in the middle** — an error handler only catches errors from routes written *above* it. Placed early, it silently guards nothing, and crashes return. The fix: it is the last `app.use` in the file, always.
3. **Answering an error and then calling `next()` anyway** — the request continues into more handlers after the response was sent, producing "headers already sent" chaos. The fix: after answering, return — one request, one answer.

Practice

1. Take the CRUD API from lesson N.4 and move every error answer (`400` , `404`) into the central pattern: routes build an `Error` with a status and call `next(err)`. Confirm the visible answers stay identical while the route code shrinks.
2. Extend the diary: log errors to a separate `errors.log` (time, address, real message) inside the error handler, while `server.log` keeps visits only. Trigger one 404 and one 500 and verify each lands in the correct diary.

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://toolscope.netlify.app/notes/node-error-handling-logging> · More free tools and notes at <https://toolscope.netlify.app>