Validation Deep: Strict Checks for Every Field

Never trust what a form sends: one place that checks every field — required, length, shape — so bad data never reaches the database.

Introduction

Lesson N.4 checked two fields by hand: a name and a range of marks. That habit is correct — and it does not scale. Real forms carry names, emails, passwords, phone numbers, fees and dates, each with its own rules, and copy-pasting checks into every route guarantees that one route forgets one rule. This lesson builds validation as **one reusable place**: a middleware function that holds the rules for a form, checks every field, gathers **all** the problems, and either hands the route perfectly clean data or refuses the request with a 400 that lists exactly what to fix.

The principle underneath is the most important sentence in backend work: **never trust data that arrived from outside**. A browser form can be bypassed, a request can be handmade — the server is the last honest checkpoint, and whatever passes it lands in your database, possibly forever. *(Roman Urdu: Bahar se aaya data kabhi seedha mat lo — pehle har field ki sakht jaanch, phir database.)*

Key points

- **Validation lives in one place:** a middleware like `validateStudent` runs before the create and update routes. Both routes inherit every rule — fix a rule once, and it is fixed everywhere.
- **Three questions per field:** is it **present** (required)? Is its **shape** right (a number where a number belongs, an email where an email belongs)? Is it **in range** (length, minimum, maximum)?
- **Gather all errors, not the first:** checking stops too early if it quits at the first problem — the user fixes one field, resubmits, and meets the next error. Collect every problem into one list and answer with all of them at once.

  (Roman Urdu: Saari ghaltiyan aik sath batao — user aik hi baar mein sab theek kar lega.)
- **The answer format is an agreement:** refused data gets **400** and a body like `{ "errors": ["Name is required", "Marks must be between 0 and 100"] }`. Frontends learn to display exactly this list.
- **Clean the data while checking:** trim spaces from text (`" Ali "` becomes `"Ali"`), convert numeric text with `Number(...)`, and pass the route a tidy `req.clean` object. What reaches the database is already the good version.
- **Types lie in requests:** everything from a form or query arrives as **text**. `"95"` is not 95 until you convert it — and `Number("abc")` is `NaN`, which must be caught, not stored.
- **Database rules are the second wall, not the first:** schema checks (lesson N.6) still stand behind your validation, but the friendly, complete error list comes from this layer — schemas complain one field at a time, in technical language.

Code examples

Example 1 — The validator middleware

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

function validateStudent(req, res, next) {
  const errors = [];
  const name = (req.body.name || "").trim();
  const marks = Number(req.body.marks);
  const email = (req.body.email || "").trim();

  if (name.length < 3) errors.push("Name must be at least 3 characters");
  if (Number.isNaN(marks) || marks < 0 || marks > 100) {
    errors.push("Marks must be a number between 0 and 100");
  }
  if (!email.includes("@") || !email.includes(".")) {
    errors.push("Email must look like name@example.com");
  }

  if (errors.length > 0) {
    return res.status(400).json({ errors });
  }
  req.clean = { name, marks, email };
  next();
}

app.post("/students", validateStudent, (req, res) => {
  res.status(201).json({ saved: req.clean });
});

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

What it does: One function holds all the rules for the student form. It checks presence and length for the name, converts and range-checks the marks (catching `NaN` honestly), and gives the email a basic shape test. Every failure is *pushed onto a list* instead of stopping the show; at the end, either the list is empty — and the cleaned data rides along as `req.clean` — or the whole list goes back in one 400. The route itself now only ever sees good data.

What you will see: posting `{ "name": "Al", "marks": 140, "email": "ali" }` answers 400 with all three errors listed together; posting `{ "name": "Ali", "marks": 90, "email": "ali@example.com" }` answers 201 with the cleaned record.

Example 2 — The same guard on the update route

```
const students = [
  { id: 1, name: "Ali", marks: 70, email: "ali@example.com" },
  { id: 2, name: "Sara", marks: 92, email: "sara@example.com" },
];

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

What it does: The exact same middleware now stands in front of the update route — written once, guarding twice. This is the payoff of the middleware pattern from lesson N.3: the rule-maker and the rule-user are different routes, and improving the validator (say, a stricter email test) instantly improves every route it protects, with no route code touched.

What you will see: updating student 1 with marks `90` succeeds and answers the updated record; updating with an empty name answers the same tidy 400 error list as the create route — identical behaviour, zero duplicated checks.

Example 3 — A stricter email: the pattern test

```
function isEmail(text) {
  return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(text);
}

console.log(isEmail("ali@example.com"));   // true
console.log(isEmail("sara@school.pk"));    // true
console.log(isEmail("ali.example.com"));    // false — no @
console.log(isEmail("ali@example"));       // false — no dot after @
console.log(isEmail("ali @example.com"));  // false — space inside
```

What it does: The `includes` test from Example 1 passes some broken shapes, so a serious form upgrades to a **regular expression** — the pattern language from lesson 3.26. Read it in pieces: some non-space, non-@ characters, then `@`, then more characters, a real dot, and more characters to the end. Five test lines then prove the rule behaves exactly as intended before it ever guards a route — test your rules in the open, like this, before trusting them.

What you will see: two `true`s followed by three `false`s, matching the comments — the pattern accepts honest email shapes and refuses the three classic broken ones.

Quick recap

- Validation is **one middleware per form**, mounted on every route that accepts that data.
- Every field answers three questions: present? right shape? in range?
- **All** errors return together in one 400; cleaned data travels onward as `req.clean`.
- Convert text to numbers deliberately, and treat `NaN` as a failed check, never as data.

Common mistakes

1. **Checking stops at the first error** — `return` inside the first failed check forces the user into a fix-one, resubmit, meet-the-next-error loop. The fix: push every problem onto a list, answer once with the complete list.
2. **Validating only in the browser** — page JavaScript that checks the form feels sufficient, but requests can be sent without any browser, straight at the API. The fix: browser checks are for comfort; **server checks are for truth**. Both, always, with the server as the final word.
3. **Storing what arrived instead of what you cleaned** — untrimmed names, text marks and stray spaces land in the database and poison searches and reports later. The fix: the route uses **only** the validator's cleaned output (`req.clean`), never the raw `req.body`.

Practice

1. Extend the validator with a `city` rule (required, at least 3 characters) and a `fee` rule (a number, 0 or more, where a missing fee becomes 0 in the cleaned data). Test a request that breaks three rules at once and confirm all three errors return together.
2. Write a `validateLogin` middleware for the N.5 login route: email shape by the Example 3 pattern, password at least 6 characters. Mount it before `/login` and show that a short password is refused with 400 *before* bcrypt ever runs.

Practice quiz

1. Where should validation live so both create and update routes share every rule?

- Inside each route, copied twice
- In one place: a middleware like validateStudent running before the create and update routes, so fixing a rule once fixes it everywhere
- In the browser only
- In the database only

Answer: In one place: a middleware like validateStudent running before the create and update routes, so fixing a rule once fixes it everywhere

2. Why gather all validation errors instead of stopping at the first?

- It is only tidier
- So the user fixes everything in one go instead of a fix-one, resubmit, meet-the-next-error loop; refused data gets 400 with all errors at once
- First-error checking is forbidden
- All errors cannot be gathered, only one at a time

Answer: So the user fixes everything in one go instead of a fix-one, resubmit, meet-the-next-error loop; refused data gets 400 with all errors at once

3. After validation cleans the data, which object must the route use?

- req.body, the raw arrival
- Only the validator's cleaned output, req.clean, never the raw req.body, or untrimmed names, text marks and stray spaces poison searches and reports later
- Either, they are identical
- A new object built by hand

Answer: Only the validator's cleaned output, req.clean, never the raw req.body, or untrimmed names, text marks and stray spaces poison searches and reports later

---

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-validation-deep · More free tools and notes at https://scopetreasure.com