Forms + Full CRUD API: Data Ka Poora Nizam

The four jobs of every app — Create, Read, Update, Delete — as real API routes: forms that send data, lists that answer, and honest status codes.

Introduction

Almost every app ever built does the same four jobs with data: **Create** it, **Read** it, **Update** it, **Delete** it — **CRUD**. A students app adds a student, shows the list, corrects a mark, removes a leaver. In an API, each job has its own HTTP method: `POST` creates, `GET` reads, `PUT` updates, `DELETE` deletes. This lesson builds the complete set in Express, fed by a real HTML **form**, with data held in an array for now — lesson N.6 will swap the array for a real database without changing the shape of the routes.

Two supporting ideas matter. First, the server must be able to **read the body** of a request — the data a form sends — which Express does through built-in middleware. Second, every answer carries a **status code**: 201 when something was created, 400 when the visitor's data was wrong, 404 when the thing does not exist. Honest codes are how an API speaks clearly. *(Roman Urdu: CRUD matlab data ke chaar kaam — banana, parhna, badalna, mitana. Har kaam ka apna tareeqa aur apna status code.)*

Key points

- **Four methods, four jobs:** `app.post` (Create), `app.get` (Read), `app.put` (Update), `app.delete` (Delete). Same address, different method, different job — `/students` serves all four.
- **Forms speak `urlencoded`:** an HTML form's data arrives as text pairs. The middleware `express.urlencoded({ extended: true })` turns it into `req.body`; APIs receiving JSON need `express.json()` instead. Mount both and both kinds of visitors work.
- **`req.body` is the sent data:** after the middleware runs, `req.body.name` and `req.body.marks` hold exactly what the form fields named `name` and `marks` carried.
- **Status codes are sentences:** 200 = OK, **201 = Created**, 400 = your data was wrong, 404 = not found. Set one with `res.status(201).json(...)`.

  (Roman Urdu: Status code jawab ke sath ek chhota sa pegham hai — kaam hua, banna, ya ghalti hui.)
- **New ids must be unique:** when creating, give the record an id nobody else has — here, one more than the highest existing id. Real databases automate this (lesson N.6).
- **Validate before you save:** an empty name or marks outside 0–100 must be refused with 400 *before* touching the data. Lesson N.8 turns this habit into a full system.
- **Update changes only what exists:** `PUT /students/1` finds the record, changes its fields, answers with the updated record — and 404s honestly when the id is absent.

Code examples

Example 1 — The form and the Create route (POST)

```
<!-- form.html — open this file in the browser -->
<form action="http://localhost:3000/students" method="POST">
  <input name="name" placeholder="Student name" />
  <input name="marks" type="number" placeholder="Marks" />
  <button type="submit">Add student</button>
</form>
```

```
// server.js
const express = require("express");
const app = express();

app.use(express.urlencoded({ extended: true }));
app.use(express.json());

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

app.post("/students", (req, res) => {
  const name = req.body.name;
  const marks = Number(req.body.marks);
  if (!name || marks < 0 || marks > 100) {
    return res.status(400).json({ error: "Name and marks (0-100) are required" });
  }
  const newStudent = { id: students.length + 1, name, marks };
  students.push(newStudent);
  res.status(201).json(newStudent);
});

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

What it does: The form's `action` points at the API and its method is POST, so submitting sends the two fields as the request body. The `urlencoded` middleware unpacks them into `req.body`; the route validates, builds a record with a fresh id, `push`es it into the array (lesson 3.8) and answers **201 Created** with the new record. Bad data never gets in — it is refused with 400 and a reason.

What you will see: submitting the form with name `Hira` and marks `95` answers `{"id":4,"name":"Hira","marks":95}` with status 201; submitting with an empty name answers the 400 error

message instead.

Example 2 — Read: the list and one student (GET)

```
app.get("/students", (req, res) => {
  res.json(students);
});

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

What it does: Reading never changes data, so these routes only look and answer. The first returns the whole array — including anyone added through the form. The second reuses the parameter pattern from lesson N.3: find by id, answer the record, or answer an honest 404. Because the data lives in one shared array, creates, reads, updates and deletes all see the same truth.

What you will see: after adding Hira in Example 1, `GET /students` lists four students; `GET /students/2` answers Sara alone; `GET /students/99` answers the 404 error.

Example 3 — Update and Delete (PUT + DELETE)

```
app.put("/students/:id", (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.marks = Number(req.body.marks);
  res.json(student);
});

app.delete("/students/:id", (req, res) => {
  const index = students.findIndex((s) => s.id === Number(req.params.id));
  if (index === -1) {
    return res.status(404).json({ error: "Student not found" });
  }
  const removed = students.splice(index, 1)[0];
  res.json({ deleted: removed });
});
```

What it does: Update finds the record and changes only its fields — here Ali's marks move from 70 to 90, the same correction the video lesson makes. Delete uses `findIndex` plus `splice` (lesson 3.8) to remove

exactly that record and answers with what was removed, so the caller can be sure. Both routes 404 honestly when the id is not there. Tools like Postman — or `fetch` from a page — send PUT and DELETE; a plain HTML form cannot, which is why APIs are tested with proper tools.

What you will see: `PUT /students/1` with marks `90` answers `{"id":1,"name":"Ali","marks":90}`; `DELETE /students/3` answers `{"deleted":{"id":3,"name":"Rahim","marks":64}}`, and the list afterwards holds Ali, Sara and Hira only.

Quick recap

- **CRUD** = Create (`POST`), Read (`GET`), Update (`PUT`), Delete (`DELETE`) — one address, four methods.
- Body-reading middleware (`express.urlencoded` / `express.json`) must be mounted before the routes, or `req.body` is empty.
- Answer honestly: **201** created, **400** bad data, **404** not found.
- Validate first, then save; find first, then change or remove.

Common mistakes

1. **No body middleware, empty `req.body`** — without `app.use(express.json())` (or `urlencoded` for forms), the sent data silently never arrives and every create "works" with empty fields. The fix: mount the matching middleware **above** the routes, before anything that reads `req.body`.
2. **Answering 200 for everything** — a create that answers plain 200, or a missing student answered with an empty object, forces every caller to guess. The fix: 201 for creates, 400 for bad input, 404 for missing records — the codes are the API's honest voice.
3. **Restarting the server and "losing" the data** — the array lives in memory, so a restart resets it to the starting three students, which looks like a bug. It is not: memory is temporary by design. The fix: real storage arrives in lesson N.6 with MongoDB; until then, expect the reset.

Practice

1. Add a `city` field end to end: the form gains a city input, POST stores it, and `GET /students?city=...` (lesson N.3) filters by it. Refuse a create with an empty city using a 400.
2. Write a route `GET /students/top` that answers the student with the highest marks — and place it **above** `/students/:id` in the file. Then move it below and observe what `/students/top` answers instead; explain the difference in one sentence.

Practice quiz

1. One address, four jobs: which HTTP method does each CRUD job use?

- GET for all four
- POST for Create, GET for Read, PUT for Update, DELETE for Delete; same address, different method, different job

- PUT for Create, POST for Update
- DELETE for Read

Answer: POST for Create, GET for Read, PUT for Update, DELETE for Delete; same address, different method, different job

2. What happens if you forget the body middleware, like express.json()?

- The server crashes
- req.body is empty, the sent data silently never arrives, and every create "works" with empty fields; mount the matching middleware above the routes
- The data arrives as a string instead
- Nothing, Express reads bodies by itself

Answer: req.body is empty, the sent data silently never arrives, and every create "works" with empty fields; mount the matching middleware above the routes

3. Which status code answers a successful create, bad data, and a missing record?

- 200 for all three
- 201 Created, 400 for bad data, 404 for not found, set with res.status(201).json(...)
- 404 for creates, 201 for missing
- 500 for all three

Answer: 201 Created, 400 for bad data, 404 for not found, set with res.status(201).json(...)

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-crud-api · More free tools and notes at https://scopetreasure.com