Final Project: A Complete Backend with a Real Database

Everything together: an Express API connected to a real MongoDB database with Mongoose — models, full CRUD on real data, and the JWT lock on top.

Introduction

Every lesson so far kept its data in an **array** — and every restart wiped it. This final project removes that training wheel. The same students API you know is rebuilt on a **real database**: MongoDB, reached through **Mongoose**, the package that lets JavaScript objects and database documents shake hands. You will meet the full production shape in one place: a `.env` file for the connection string, a **model** that describes a student, CRUD routes that read and write real stored data, and the JWT lock from lesson N.5 guarding the private work.

Nothing about the route *shape* changes — `POST /students`, `GET /students/:id`, the same honest status codes. That is the lesson inside the lesson: the API is a contract, and the storage underneath can grow from an array to a real database without the outside world noticing. *(Roman Urdu: Ab data array mein nahi, asal database mein mehfooz hoga — server band ho ya chaley, data waisa ka waisa.)*

Key points

- **Mongoose is the translator:** MongoDB stores documents; your code thinks in JavaScript objects. Mongoose **schemas** declare what a document must look like, and **models** give you clean methods — `save`, `find`, `findByIdAndUpdate`, `findByIdAndDelete`.
- **A schema is a promise:** `name` is a required string, `marks` is a number between 0 and 100. Mongoose checks every save against the schema, so malformed data is refused before it reaches the database (validation grows up here — lesson N.8 goes deeper).
- **The connection string is a secret:** the database address, often with a password inside, lives in the `.env` file as `MONGO_URL` — never in code. `require("dotenv").config()` loads it (lesson N.10 is devoted to this).
  (Roman Urdu: Database ka pata aur password code mein nahi — `.env` file mein, jo kabhi share nahi hoti.)
- **Connecting happens once, at the start:** `mongoose.connect(process.env.MONGO_URL)` — then every route simply uses the models. If the connection fails, the honest move is to say so in the log and stop, not to serve a broken API.
- **Database ids are not numbers:** MongoDB gives every document an `_id` like `665f1a2b9c...`. Routes read it from `req.params.id` as text — no `Number(...)` conversion like the array lessons needed.
- **Async is everywhere:** every database call takes time, so routes become `async` and use `await` (lesson 3.21). A `try/catch` around them turns database trouble into an honest 500 instead of a dead server.

- **The JWT lock fits unchanged:** the `auth` middleware from N.5 guards the write routes (POST, PUT, DELETE); reading the list can stay public — a real-world pattern.

Code examples

Example 1 — Connect, and describe a student

```
// npm install express mongoose dotenv jsonwebtoken bcrypt
const express = require("express");
const mongoose = require("mongoose");
require("dotenv").config();

const app = express();
app.use(express.json());

mongoose
  .connect(process.env.MONGO_URL)
  .then(() => console.log("Database connected"))
  .catch((err) => console.log("Database connection failed:", err.message));

const studentSchema = new mongoose.Schema({
  name: { type: String, required: true },
  marks: { type: Number, min: 0, max: 100, required: true },
  city: { type: String, default: "Sahiwal" },
});

const Student = mongoose.model("Student", studentSchema);

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

What it does: Three foundations in one file. dotenv loads `.env` so `process.env.MONGO_URL` holds the connection string; `mongoose.connect` opens the line to the real database; and the **schema + model** pair declares exactly what a student is — a required name, marks locked between 0 and 100, a city that defaults when not given. From here on, `Student` is your door to the collection.

What you will see: the terminal prints `Database connected` and `Server running on port 3000`. Start the server again tomorrow and it still prints the same — and, unlike the array, yesterday's data is still there.

Example 2 — CRUD on real, stored data

```
app.post("/students", async (req, res) => {
  try {
    const student = new Student(req.body);
    const saved = await student.save();
    res.status(201).json(saved);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

app.get("/students", async (req, res) => {
  const students = await Student.find();
  res.json(students);
});

app.put("/students/:id", async (req, res) => {
  try {
    const updated = await Student.findByIdAndUpdate(
      req.params.id,
      { marks: req.body.marks },
      { new: true, runValidators: true }
    );
    if (!updated) return res.status(404).json({ error: "Student not found" });
    res.json(updated);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

app.delete("/students/:id", async (req, res) => {
  const removed = await Student.findByIdAndDelete(req.params.id);
  if (!removed) return res.status(404).json({ error: "Student not found" });
  res.json({ deleted: removed });
});
```

What it does: Compare with lesson N.4 — the routes are the same shape, but every `students.push` and `find` is now a real database operation. `save()` writes a document that survives restarts; `findByIdAndUpdate` with `{ new: true }` answers with the *updated* record (Ali's marks, 70 becoming 90, exactly as in the video project); `findByIdAndDelete` removes Rahim's record for real. Schema rules ride along: an update trying marks of 140 is refused by `runValidators`.

What you will see: creating Ali answers 201 with the saved document including its real `_id`; after a server restart, `GET /students` **still lists him** — the moment the training wheels come off. The bad update answers 400 with the schema's complaint.

Example 3 — The finished shape: the lock on the writes

```
const jwt = require("jsonwebtoken");

function auth(req, res, next) {
  const header = req.headers.authorization;
  if (!header) return res.status(401).json({ error: "No token — login first" });
  try {
    req.user = jwt.verify(header.split(" ")[1], process.env.JWT_SECRET);
    next();
  } catch (err) {
    res.status(401).json({ error: "Invalid or expired token" });
  }
}

// Reading is public; writing needs the key.
app.post("/students", auth, async (req, res) => { /* create, as in Example 2 */ });
app.put("/students/:id", auth, async (req, res) => { /* update, as in Example 2 */ });
app.delete("/students/:id", auth, async (req, res) => { /* delete, as in Example 2 */ });
```

What it does: The final piece clicks into place. The auth middleware from lesson N.5 — now reading its secret from `.env` too — is placed as the middle argument on each *write* route. Express runs it first: valid token, the request continues into the route; anything else, a 401 and the database is never touched. Reads stay open so anyone can view the list, while changing it demands the key — precisely how real student, shop and blog systems behave.

What you will see: `POST /students` without a token answers 401 and creates nothing; the same request carrying a login token creates the student and answers 201. The complete backend — server, routes, lock and real database — is standing.

Quick recap

- **Mongoose** connects Express to MongoDB: schemas promise the shape, models do the work.
- Secrets (`MONGO_URL`, `JWT_SECRET`) live in `.env`, loaded by dotenv — never in code.
- CRUD routes keep their N.4 shape; only the storage changes, and the data now **survives restarts**.
- Database routes are `async` / `await` with `try/catch`; the **JWT guard** protects the writes.

Common mistakes

1. **Committing `.env` with the code** — sharing the project folder (or pushing it to GitHub) with `.env` inside hands everyone your database password. The fix: list `.env` in `.gitignore` from the very first day, and share only a sample file with the variable *names*, never the values.
2. **Forgetting `await` on a database call** — `const students = Student.find()` without `await` holds a promise, not students, and the API answers `{}` or nonsense. The fix: every model call inside an `async`

route gets `await` — and a `try/catch` so failures become honest 500s.

3. **Expecting numeric ids** — hunting for the bug when `req.params.id` is `665f...` and `Number(...)` turns it into `NaN`, so nothing is ever found. The fix: MongoDB `_id`s are text; pass them to the `findById...` methods exactly as they arrive.

Practice

1. Add a second model, `Course`, with `title` (required string) and `fee` (number). Build its full CRUD set next to the students routes, with the same status-code discipline, and confirm a created course survives a server restart.
2. Register-and-login, completed: add a `User` model (name, unique email, passwordHash) with `/register` and `/login` routes from lesson N.5 wired to the database instead of the array. Log in, take the token, and use it to create a student; then try the same create *without* the token and record both answers.

Practice quiz

1. What do Mongoose schemas and models each do?

- Schemas run queries, models store data
- Schemas declare what a document must look like, a promise Mongoose checks on every save; models give clean methods like save, find, findByIdAndUpdate and findByIdAndDelete
- Both do the same job
- Schemas are for SQL only

Answer: Schemas declare what a document must look like, a promise Mongoose checks on every save; models give clean methods like save, find, findByIdAndUpdate and findByIdAndDelete

2. In the final project, why must you NOT convert req.params.id with Number(...)?

- Because conversion is slow
- Because MongoDB _ids are text like 665f1a2b9c...; Number(...) turns them into NaN, so nothing is ever found; pass them to the findById... methods exactly as they arrive
- Because ids are always numbers in MongoDB
- Conversion is forbidden by Mongoose

Answer: Because MongoDB _ids are text like 665f1a2b9c...; Number(...) turns them into NaN, so nothing is ever found; pass them to the findById... methods exactly as they arrive

3. In the final project, which routes does the JWT guard protect, and which may stay public?

- All routes, including reads
- The write routes (POST, PUT, DELETE); reading the list can stay public, a real-world pattern
- Only the login route
- No routes, JWT is for display only

Answer: The write routes (POST, PUT, DELETE); reading the list can stay public, a real-world pattern

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-final-project · More free tools and notes at https://scopetreasure.com