Authentication: JWT Ka Tala (Login System)

An open API lets anyone in: passwords hashed with bcrypt, a signed JWT token as the key, and middleware that checks the key before any private data moves.

Introduction

Your CRUD API from lesson N.4 has a serious hole: **anyone** who knows the address can read, change or delete everything. Real apps first ask *who are you?* — that question is **authentication** — and only then decide what you may touch. This lesson builds the standard Node login system in three parts.

First, passwords are never stored as plain text: a package called **bcrypt** turns each password into a scrambled **hash** that cannot be read back. Second, when a user logs in with the correct password, the server issues a **JWT** (JSON Web Token) — a signed digital key. Third, a piece of **middleware** (lesson N.3) stands before the private routes and checks every visitor's key: a valid key passes, anything else gets a firm **401 — unauthorised**. The token works like a wristband at a fair: prove yourself once at the gate, then show the band at every ride. *(Roman Urdu: Password kabhi seedha mat rakho — hash karo. Login par JWT ki chaabi do, aur har private route par chaabi jaancho.)*

Key points

- **Authentication vs authorisation:** authentication asks "who are you?" (login); authorisation asks "what may you do?" (roles and permissions). This lesson builds the first; the lock pattern serves both.
- **Never store plain passwords:** if a database of plain passwords leaks, every account falls. A **hash** is one-way: you can check a guess against it, but you cannot read the original back out.
- **bcrypt adds salt:** `bcrypt.hash(password, 10)` scrambles the password together with random **salt**, so two users with the same password still get different hashes. Checking is `bcrypt.compare(guess, hash)` — answer: true or false.
- **A JWT has three parts:** header, payload (your data, like the user's id) and **signature** — joined by dots. The signature is made with a secret only the server knows, so nobody can forge or alter a token.

  (Roman Urdu: JWT aik aisi chaabi hai jis par server ke dastakhat hain — jaali chaabi nahi ban sakti.)
- **The secret lives outside the code:** the signing secret sits in an environment variable (lesson N.10 grows this topic) — never written into a file that gets shared.
- **The token travels in a header:** the client sends `Authorization: Bearer <token>` with each request. Your auth middleware reads it, verifies it with the secret, and either calls `next()` or answers **401**.
- **Login fails honestly but vaguely:** a wrong email and a wrong password get the *same* 401 message — never reveal which half was wrong, or attackers learn which emails exist.

Code examples

Example 1 — Hashing: passwords you cannot read back

```
// npm install bcrypt jsonwebtoken
const bcrypt = require("bcrypt");

async function demo() {
  const password = "ali123";
  const hash = await bcrypt.hash(password, 10);
  console.log("Stored hash:", hash);

  console.log("Correct guess:", await bcrypt.compare("ali123", hash));
  console.log("Wrong guess:", await bcrypt.compare("xyz999", hash));
}

demo();
```

What it does: `bcrypt.hash` scrambles the password with salt at cost 10 (the work level — higher is slower and safer). What you store in the database is the long hash string, never `ali123` itself. Later, `compare` answers the only question that matters — does this guess match? — without the original password existing anywhere.

What you will see: a hash like `$2b$10$N9qo8uLOickgx2ZMRZoMye...` (long and unreadable), then `Correct guess: true` and `Wrong guess: false`. Run it again and the hash differs — the salt changed — yet `compare` still works.

Example 2 — Login: checking the password, issuing the key

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

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

const SECRET = process.env.JWT_SECRET || "dev-secret-change-me";

// A tiny user store for this lesson (a database arrives in N.6).
const users = [];
users.push({
  id: 1,
  name: "Ali",
  email: "ali@example.com",
  passwordHash: bcrypt.hashSync("ali123", 10),
});

app.post("/login", async (req, res) => {
  const { email, password } = req.body;
  const user = users.find((u) => u.email === email);
  if (!user) {
    return res.status(401).json({ error: "Invalid email or password" });
  }
  const ok = await bcrypt.compare(password, user.passwordHash);
  if (!ok) {
    return res.status(401).json({ error: "Invalid email or password" });
  }
  const token = jwt.sign({ id: user.id, name: user.name }, SECRET, { expiresIn: "1h" });
  res.json({ message: "Login successful", token });
});

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

What it does: Login is a careful sequence. Find the user by email; if there is none, fail with the vague 401. Compare the sent password against the stored hash; if it fails, the *same* vague 401. Only when both pass does `jwt.sign` mint a token carrying the user's id and name, signed with the secret, set to expire in one hour. The password itself is never inside the token — tokens can be read by anyone who holds them.

What you will see: posting the correct email and password answers `{"message":"Login successful","token":"eyJhbGciOi..."}` — a long three-part token. A wrong password answers status 401 with the invalid-credentials error, and no token.

Example 3 — The guard: auth middleware before private routes

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

app.get("/profile", auth, (req, res) => {
  res.json({ message: "Welcome, " + req.user.name, id: req.user.id });
});
```

What it does: This is lesson N.3's middleware idea given its most important job. `auth` sits **between** the address and the route: no header means 401 immediately; a token is verified against the secret, and only a genuine, unexpired one reaches `next()`. On success the decoded data is attached to `req.user`, so the route itself knows *who* is visiting — the profile route greets Ali by name without asking again.

What you will see: `GET /profile` with no header answers 401 `No token — login first`; with `Authorization: Bearer <the login token>` it answers `{"message":"Welcome, Ali","id":1}`; with a tampered or expired token it answers the invalid-token 401.

Quick recap

- Store **hashes**, never plain passwords — bcrypt hashes with salt, and `compare` checks a guess.
- A successful login issues a **JWT**: signed with a server secret, carrying safe data like the user id, with an expiry.
- Clients send it as `Authorization: Bearer <token>`; **auth middleware** verifies it before private routes.
- Every failure is **401**, with a message that never reveals which half of the login was wrong.

Common mistakes

1. **Storing or logging the plain password** — saving `req.body.password` into the database (or printing it "for debugging") recreates the exact danger hashing exists to remove. The fix: hash on arrival, store only the hash, and never log credentials.

2. **Putting secrets or passwords inside the token** — a JWT payload is readable by anyone holding it (it is encoded, not locked). The fix: payload carries only safe identifiers — id, name, role — never passwords, hashes or secrets.
3. **Checking the token but not the clock** — signing without `expiresIn` makes stolen tokens valid forever. The fix: every token gets an expiry; `jwt.verify` then rejects old tokens automatically, and the user simply logs in again.

Practice

1. Add a `/register` route to Example 2: it receives name, email and password, refuses a duplicate email with 400, hashes the password with bcrypt, stores the user, and answers 201. Then log in with the new account and open `/profile` with its token.
2. Make a second private route, `GET /my-marks`, guarded by the same `auth` middleware, which answers marks for `req.user.id` from a small marks table you define. Confirm by testing all three cases: no token, a bad token, and the real token.

Practice quiz

1. Why does bcrypt add salt when hashing a password?

- To make the hash shorter
- So two users with the same password still get different hashes; checking is bcrypt.compare(guess, hash), true or false
- To make the password readable again
- Salt is only for cooking, it does nothing here

Answer: So two users with the same password still get different hashes; checking is bcrypt.compare(guess, hash), true or false

2. What are the three parts of a JWT, and what stops forgery?

- Header, body, footer, stopped by encryption
- Header, payload (your data, like the user id) and signature, joined by dots; the signature is made with a secret only the server knows, so nobody can forge or alter a token
- Username, password, and expiry, stopped by the database
- Three random strings, stopped by nothing

Answer: Header, payload (your data, like the user id) and signature, joined by dots; the signature is made with a secret only the server knows, so nobody can forge or alter a token

3. Why do a wrong email and a wrong password get the same vague 401 message?

- Because the code cannot tell them apart
- So attackers never learn which half was wrong, or they learn which emails exist
- Because 401 is the only status code

- Because login failures are never logged

Answer: So attackers never learn which half was wrong, or they learn which emails exist

---

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-auth-jwt · More free tools and notes at https://scopetreasure.com