Project 2: URL Shortener API

A second complete backend: long links become short codes, visitors are redirected, and every click is counted — plan, data, routes and testing.

Introduction

Time to prove the whole series on a fresh problem. A **URL shortener** turns a long, ugly link — a maps address, a long article URL — into a short code like `/aZ3`, and anyone visiting the short address is **redirected** to the real one. Services like this are real businesses, and the backend is pure Series 6: a create route, a read-and-redirect route, stored data, counting, honest errors — plus two new tricks, generating a short code and answering with a **redirect** instead of JSON.

Plan before code, as always. The data is one record per link: the original URL, its short code, and a **click count** starting at zero. The jobs: `POST /links` creates a record from a long URL; `GET /:code` finds the record, adds one click, and redirects; `GET /links/stats/:code` reports the count. Everything else — validation, status codes, structure — you already own. *(Roman Urdu: Lamba link do, chhota code lo — aur har click ki ginti bhi server rakhega.)*

Key points

- **A redirect is a different kind of answer:** `res.redirect(url)` answers status **302** with "the thing you want lives over there", and the browser follows automatically. APIs send data; a shortener sends travellers onward.
- **Short codes must be unique and unguessable enough:** build one from random letters and digits (`aZ3x9`). Six characters from 62 symbols give billions of combinations — collisions are checked anyway: if a generated code exists, generate again.
- **Every lookup can miss:** an unknown code is a normal event, not a crash — answer an honest **404**, exactly as the students API did for unknown ids.

  (Roman Urdu: Code ghalat ho to 404 — server gire nahi, jawab de.)
- **Counting is an update like any other:** find the record, `clicks = clicks + 1`, save. The redirect must happen *after* the count, so no visit goes uncounted.
- **Validate the long URL itself:** a create request must carry text starting with `http://` or `https://`. Anything else — empty, or plain words — is refused with 400 before storage (lesson N.8's discipline).
- **The shape is the series' shape:** an array plays the database in this lesson so the new ideas stand out; swapping in the Mongoose model from N.6 changes only the storage lines, exactly as N.6 swapped N.4's.
- **Stats make it a product:** a bare redirect is a toy; knowing *how many* people used a link is the information people actually want — one small route turns the project into a tool.

Code examples

Example 1 — Creating a short link

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

const links = [];

function makeCode() {
  const chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
  let code = "";
  for (let i = 0; i < 6; i++) {
    code += chars[Math.floor(Math.random() * chars.length)];
  }
  return code;
}

app.post("/links", (req, res) => {
  const longUrl = (req.body.longUrl || "").trim();
  if (!longUrl.startsWith("http://") && !longUrl.startsWith("https://")) {
    return res.status(400).json({ error: "A full URL starting with http is required" });
  }
  let code = makeCode();
  while (links.some((l) => l.code === code)) {
    code = makeCode();
  }
  const link = { code, longUrl, clicks: 0 };
  links.push(link);
  res.status(201).json({ shortUrl: "http://localhost:3000/" + code, code });
});

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

What it does: The create route follows the house rules — validate first (a full URL or a 400), then store. `makeCode` picks six random characters from the 62-symbol alphabet using the loop and random skills from Series 3; the `while` check makes a duplicate code impossible by simply rolling again. The stored record holds the code, the long URL and a click counter born at zero, and the answer is **201** with the ready-to-share short address.

What you will see: posting `{ "longUrl": "https://www.youtube.com/@CodeScopePK" }` answers 201 with something like `{"shortUrl":"http://localhost:3000/xK7pQ2","code":"xK7pQ2"}`; posting `{ "longUrl": "hello" }` answers the 400 error.

Example 2 — The redirect: the heart of the service

```
app.get("/:code", (req, res) => {
  const link = links.find((l) => l.code === req.params.code);
  if (!link) {
    return res.status(404).json({ error: "Short link not found" });
  }
  link.clicks += 1;
  res.redirect(link.longUrl);
});
```

What it does: One route parameter (lesson N.3) carries the code. The record is found, the counter rises by one **before** anything else happens, and `res.redirect` sends the visitor to the stored long URL — the browser's address bar changes to the real destination as if the short address were merely a signpost. A mistyped or invented code receives the honest 404 instead of a wild guess.

What you will see: visiting `http://localhost:3000/xK7pQ2` lands you on the Code Scope YouTube channel — the browser shows the channel's address, not localhost. Visiting `/nope99` answers `{"error":"Short link not found"}` with status 404.

Example 3 — Stats: proving the clicks are counted

```
app.get("/links/stats/:code", (req, res) => {
  const link = links.find((l) => l.code === req.params.code);
  if (!link) {
    return res.status(404).json({ error: "Short link not found" });
  }
  res.json({ code: link.code, longUrl: link.longUrl, clicks: link.clicks });
});
```

What it does: A quiet read-only route reports a link's record — where it points and how many travellers it has carried. Note its position in the file: it must be declared **before** the `/:code` route from Example 2, because `/:code` would otherwise swallow the address `links` as if it were a code — lesson N.3's order rule, earning its keep in a real project.

What you will see: after visiting the short link three times, `GET /links/stats/xK7pQ2` answers `{"code":"xK7pQ2","longUrl":"https://www.youtube.com/@CodeScopePK","clicks":3}` — the counter kept its promise.

Quick recap

- A shortener stores **code → long URL + click count**; creating, redirecting and reporting are its three routes.
- `res.redirect` answers 302 and the browser follows — a new kind of response in your toolkit.
- Codes are random, checked for collisions; unknown codes are honest **404**s.

- Count first, redirect second — and mind the route order around `/:code`.

Common mistakes

1. **Redirecting before counting (or in the wrong order entirely)** — if the redirect line runs first, the count line never executes and every link reports zero clicks. The fix: count, save, *then* redirect — order inside the route matters as much as order between routes.
2. **Accepting any text as a URL** — storing `hello` and later redirecting to it sends visitors to a broken relative address on your own server. The fix: validate at creation — only text beginning `http://` or `https://` may become a link (Example 1's 400).
3. **Placing `/:code` above specific routes** — a greedy parameter route swallows `/links/...` addresses as "codes", and stats mysteriously 404. The fix: specific routes first, the parameter catch-all **last** — the same order discipline as `/students/top` in lesson N.4.

Practice

1. Grow the project with a list route, `GET /links`, that answers every record with its code, long URL and clicks, sorted by clicks with the most-clicked first (the array `sort` from Series 3). Create three links, click them different numbers of times, and verify the order.
2. Move the storage into MongoDB using lesson N.6's pattern: a `Link` model with `code`, `longUrl` and `clicks`, created through Mongoose, counted with an update, and surviving a restart. Then add the JWT lock from N.5 so only a logged-in owner can create links, while redirects stay public.

Practice quiz

1. What does res.redirect(url) answer, and who follows it?

- Status 200 with the page content
- Status 302 with "the thing you want lives over there", and the browser follows automatically
- A JSON object with the URL
- Nothing, the server fetches the page itself

Answer: Status 302 with "the thing you want lives over there", and the browser follows automatically

2. In the shortener, what order must counting and redirecting happen in?

- Redirect first, then count
- Count first, save, then redirect, or the count line never executes and every link reports zero clicks
- Order does not matter
- Counting happens automatically

Answer: Count first, save, then redirect, or the count line never executes and every link reports zero clicks

3. Where must the /:code catch-all route sit, and why?

- First, so it catches everything quickly
- Last, after specific routes, because a greedy parameter route above them swallows /links/... addresses as "codes" and stats mysteriously 404
- In the middle
- It must be in a separate file

Answer: Last, after specific routes, because a greedy parameter route above them swallows /links/... addresses as "codes" and stats mysteriously 404

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