PROJECTS

Website Rebuild with Express

The Most Minimal Framework in the Entire Series

Raw SQL, No ORM — A Sixth Arrival at the Same Model

The Recursive CTE: Finally Solving the Series' Own Open Question

Hand-Wired Routing, Views, Auth & Deployment

Capstone: The Website Rebuild Series, Complete — Six Frameworks

Philip Osztromok

Generated with Claude

Table of Contents

Website Rebuild with Express — 12 Chapters

This is the sixth and truly final course in the Website Rebuild series, alongside the already-complete Next.js, Django, Laravel, Rails, and Astro rebuilds. It builds on this site's own existing `express1`/`express2` courses and the Food Tracker (React + Express) project — this chapter assumes the language and the basic Express mental model are already familiar ground.

Scaffolding the Project

```
npm init -y
npm install express ejs mysql2
```

The Minimal Server

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

app.set('view engine', 'ejs');

app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

Confirmed: The Most Minimal Framework in the Series

EVEN THE TEMPLATING ENGINE ISN'T ASSUMED

Every sibling in this series ships at least some opinion: Astro owns its own routing and rendering outright; Next.js, Django, Laravel, and Rails each bundle a real templating story by default. Express does none of this. `app.set('view engine', 'ejs')` is a literal, explicit statement — Express itself has no default rendering engine at all, and would run exactly the same way if this line named Pug, Handlebars, or Nunjucks instead. Nothing about "how views render" is Express's own opinion; it's entirely the developer's choice, stated once, in one line.

No ORM, No Routing Convention, No Session Handling

`mysql2` is a plain database driver, not an ORM — Chapter 2 writes real SQL directly, continuing the same deliberate choice the Food Tracker (React + Express) course already made for this exact stack. Express's own routing is a thin method-chaining API (`app.get()`, `app.post()`) with no file-based or convention-driven routing at all — every route is registered by hand. Session handling, request validation, CSRF protection — none of it exists until a specific package is chosen and installed for it, chapter by chapter, starting here.

Six Frameworks, Compared Honestly

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO	EXPRESS
Built-in ORM?	No	Yes	Yes	Yes	No	No
Built-in rendering engine?	Yes (JSX)	Yes (DTL)	Yes (Blade)	Yes (ERB)	Yes (Astro syntax)	No — chosen explicitly
Built-in routing convention?	Yes — file-based	Explicit but built in	Explicit but built in	Explicit but built in	Yes — file-based	No — hand-registered
Built-in session handling?	No	Yes	Yes	Yes	No	No

COMING IN LATER CHAPTERS

No database schema, no routing, no session package, no validation library exists yet — every one of those gets built or chosen deliberately, starting with Chapter 2's own raw SQL schema.

Hands-On Exercises

EXERCISE 1

Scaffold a new Express project, and confirm a minimal "Hello World" route responds correctly at http://localhost:3000.

View solution

EXERCISE 2

Remove app.set('view engine', 'ejs'), attempt to call res.render() from a route, and confirm Express has no default rendering engine to fall back on.

View solution

EXERCISE 3

Inspect the freshly scaffolded project's package.json, and confirm nothing beyond express, ejs, and mysql2 exists yet — no ORM, no session library, no validation library.

View solution

CHAPTER 1 QUICK REFERENCE

- `npm install express ejs mysql2` — the entire starting stack
- `app.set('view engine', 'ejs')` — Express has no default templating engine at all
- **The most minimal framework in the series** — no ORM, no routing convention, no session handling, no built-in rendering
- `mysql2` — a plain driver, not an ORM, continuing the Food Tracker (React + Express) course's own established choice
- **Next chapter:** Designing a Flexible URL & Content Model

A sixth arrival at the same design every sibling course reached independently — a real `parent_id` foreign key plus a precomputed `full_path` string. This time, written directly in SQL, with no ORM standing between the code and the schema at all.

The Schema, By Hand

```sql
-- schema.sql
CREATE TABLE pages (
  id INT AUTO_INCREMENT PRIMARY KEY,
  slug VARCHAR(255) NOT NULL,
  full_path VARCHAR(1024) NOT NULL UNIQUE,
  title VARCHAR(255) NOT NULL,
  body TEXT,
  parent_id INT,
  FOREIGN KEY (parent_id) REFERENCES pages(id) ON DELETE RESTRICT
);
```

CLOSING THE LOOP ON ASTRO'S OWN DRIZZLE FINDING

The Astro rebuild's own Chapter 2 needed a function-based workaround — `references(():AnyMySqlColumn => pages.id, ...)` — specifically because `pages` couldn't be referenced from inside its own definition while TypeScript was still evaluating it. Raw SQL DDL has no such problem: `FOREIGN KEY (parent_id) REFERENCES pages(id)` reads perfectly naturally, with no special syntax at all. That earlier workaround wasn't a fundamental complexity of self-referencing tables — it was an artifact of representing the schema *as TypeScript code*, evaluated top to bottom. Plain SQL shows that clearly.

The Connection

```
// db.js
const mysql = require('mysql2/promise');

const pool = mysql.createPool({
  uri: process.env.DATABASE_URL,
  charset: 'utf8mb4',
});

module.exports = pool;
```

`ON DELETE RESTRICT`: No Ambiguity Left

Laravel's `restrictOnDelete()` and Astro's Drizzle `onDelete: 'restrict'` were both verified as real database-level constraints — but each still required checking, since an ORM sits between the code and the schema and could, in principle, have implemented it at either layer. There's no such question here: with no ORM present at all, `ON DELETE RESTRICT` is a raw SQL clause, enforced by the database engine directly. This is the cleanest, most unambiguous version of the same finding in the whole series.

Computing `full_path`, By Hand

```
// lib/pages.js
async function computeFullPath(pool, slug, parentId) {
  if (!parentId) return slug;
  const [rows] = await pool.query('SELECT full_path FROM pages WHERE id = ?', [parentId]);
  return `${rows[0].full_path}/${slug}`;
}

module.exports = { computeFullPath };
```

NO LIFECYCLE HOOKS — EVEN MORE OBVIOUSLY TRUE HERE

Drizzle had no lifecycle hooks, requiring an explicit helper function to compute `full_path`. With no ORM present at all, that's not even a design choice to note anymore — there's nothing that could have provided a hook in the first place. `computeFullPath` is simply called directly, every time, by whatever code creates or updates a page.

Hands-On Exercises

EXERCISE 1

Write the raw CREATE TABLE schema.sql, and confirm it applies successfully via the mysql CLI or mysql2 directly.

View solution

EXERCISE 2

Confirm the self-referencing FOREIGN KEY needs no special workaround in raw SQL, contrasting this directly with Drizzle's own TypeScript circular-reference issue from the Astro rebuild's Chapter 2.

View solution

EXERCISE 3

Write and test computeFullPath(), confirming correct behavior both for a root page (no parent) and a nested page (a real parent already in the database).

View solution

CHAPTER 2 QUICK REFERENCE

- `FOREIGN KEY (parent_id) REFERENCES pages(id)` — no workaround needed, unlike Drizzle's own TypeScript-specific issue
- `ON DELETE RESTRICT` — the cleanest, most unambiguous database-level constraint in the whole series
- `computeFullPath` — a plain parameterized query, called explicitly, with no ORM lifecycle hook to lean on
- **Next chapter:** Routing — Express's Own Wildcard, Hand-Wired

Express registers every route by hand, with no file-based convention at all (Chapter 1's own finding) — including the catch-all this course's own arbitrary-depth routing needs.

Express 5's Current Wildcard Syntax

```
// routes.js
const pool = require('./db');

app.get('/*splat', async (req, res) => {
  const fullPath = req.params.splat.join('/');
  const [rows] = await pool.query('SELECT * FROM pages WHERE full_path = ?', [fullPath]);

  if (rows.length === 0) {
    return res.status(404).render('404');
  }

  res.render('page', { page: rows[0] });
});
```

A REAL, RECENT BREAKING CHANGE WORTH GETTING RIGHT

Express 4's own catch-all used a bare `*`, with the matched value at `req.params[0]`. Express 5 upgraded its underlying path-matching library, and bare wildcards no longer work the same way — a **named** wildcard, `*splat`, is required instead. `req.params.splat` is an **array** of matched segments (e.g. `['programming', 'java']` for `/programming/java`), not a single joined string — `.join('/')` is what reconstructs the full path.

Genuinely Different From Every Sibling's Own Capture

Rails' own `*path` glob captured the whole matched segment as a single string directly. Express 5's `*splat` captures an *array* of the individual segments instead — a real, mechanical difference worth naming precisely rather than assuming every wildcard-capture syntax behaves identically just because the underlying goal is the same.

Route Order — the Same Gotcha, One Final Time

Express matches routes in registration order, exactly like every sibling framework in this series. `app.get('/*splat', ...)` has to be registered **last**, after any more specific route (like a future admin API route) — otherwise it swallows every request before the more specific route ever gets a chance to match. The identical trap, repeated honestly a final time, not presented as something new to Express.

404 Handling: Not a Gotcha Here, and Here's Why

ASTRO'S GOTCHA DOESN'T HAVE AN EXPRESS EQUIVALENT — FOR A REAL REASON

Astro's own Chapter 3 found a genuine gotcha: a global catch-all silently shadows `src/pages/404.astro`, a file that *looks* like it should work automatically but never gets reached. Express has no equivalent surprise, because it never had an automatic, file-based 404 convention to shadow in the first place — `res.status(404).render('404')` inside the catch-all handler was always going to be written by hand, the same way every other response in this framework is. Nothing silently breaks here, because nothing was ever implicit to begin with.

Six Frameworks' Wildcard Routing, Compared One Final Time

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO	EX
Mechanism	`[...path]` folder	`<path:full_path>`	`{path?}` + regex	`*path` glob	`[...path].astro`	/
Capture shape	Array	Single string	Single string	Single string	Single string	Ar

Hands-On Exercises

EXERCISE 1

Build the /*splat catch-all route, querying mysql2 directly against the pages table, and confirm a real stored page renders correctly.

View solution

EXERCISE 2

Log req.params.splat directly and confirm it's a real array of segments, not a single string — contrasting this explicitly with Rails' own *path capture.

View solution

EXERCISE 3

Register the catch-all before a more specific /admin route, observe it swallow every request to /admin, then fix the ordering.

View solution

CHAPTER 3 QUICK REFERENCE

- `/*splat` — Express 5's current named wildcard, replacing Express 4's bare `*`
- `req.params.splat` — an array of matched segments, joined manually into a full path
- **Route order still matters** — the same gotcha repeated a final time across the series
- **No 404-shadowing gotcha** — Express never had an automatic 404 file to shadow in the first place
- **Next chapter:** Views: EJS Templates

EJS's own name — **E**mbedded **J**ava**S**cript — directly mirrors ERB's: **E**mbedded **R**uby. That's not a coincidence, and the base syntax really is the same shape. Where the two genuinely diverge is worth getting precisely right.

The Base Tags: A Real Parallel

```
<!-- <% %> executes, no output -->
<% if (page.featured) { %>
  <span>★ Featured</span>
<% } %>

<!-- <%= %> outputs, HTML-escaped -->
<h1><%= page.title %></h1>
```

`<% %>` for control flow, `<%= %>` for escaped output — the identical base shape ERB already established, reused here under a deliberately parallel name.

The Raw-Output Tag: A Genuine, Precise Difference

SAME IDEA, DIFFERENT SYMBOL — WORTH STATING PRECISELY, NOT ASSUMED

Rails' own `<%== %>` — added by ActionView on top of plain ERB — outputs unescaped HTML. EJS's own equivalent is `<%- %>`, a hyphen rather than a doubled equals sign. Both frameworks independently added their own raw-output escape hatch on top of the same "Embedded X" base idea, but each picked a genuinely different symbol to represent it. The parallel between EJS and ERB is real and worth naming — but it's not a claim that every tag is identical, and this is exactly the detail where that would be wrong.

```
<!-- page.body is trusted, already-formatted HTML -->
<%- page.body %>
```

No Built-In Layout System — Composed via `include()`

```
<!-- views/partials/header.ejs -->
<!DOCTYPE html>
<html lang="en">
<head><title><%= page.title %></title></head>
<body>
```

```
<!-- views/page.ejs -->
<%- include('partials/header') %>

<nav class="breadcrumb">
  <% breadcrumb.forEach(crumb => { %>
    <a href="/<%= crumb.full_path %>"><%= crumb.title %></a>
  <% }); %>
</nav>

<h1><%= page.title %></h1>
<%- page.body %>

<%- include('partials/footer') %>
```

Astro at least had first-class `<slot />` composition. EJS has no layout concept of its own at all — `include()` is a plain function that inlines another template's own output, and building a shared page shell out of a header/footer pair is a convention the developer invents, not a feature EJS provides.

The Breadcrumb — the Sixth and Final Planted N+1

```
// lib/pages.js
async function getBreadcrumb(pool, pageId) {
  const crumbs = [];
  let currentId = pageId;
  while (currentId) {
    const [rows] = await pool.query('SELECT * FROM pages WHERE id = ?', [currentId]);
    if (rows.length === 0) break;
    crumbs.unshift(rows[0]);
    currentId = rows[0].parent_id;
  }
  return crumbs;
}
```

THE LAST TIME THIS SERIES PLANTS THIS PATTERN DELIBERATELY

One query per ancestor level — the identical pattern every sibling course built into its own breadcrumb code on purpose, planted here a sixth and final time, to be caught and fixed in Chapter 6.

Raw Output, Compared Across All Six Frameworks

FRAMEWORK	MECHANISM
Django	`{{ page.body|safe }}`
Laravel	`{!! $page->body !!}`
Rails	`<%== @page.body %>`
Astro	`<div set:html={page.body} />`
Express (EJS)	`<%- page.body %>`

Hands-On Exercises

EXERCISE 1

Build header.ejs, footer.ejs, and page.ejs, and confirm the layout composes correctly via include().

View solution

EXERCISE 2

Build getBreadcrumb() and confirm it produces the correct, correctly-ordered ancestor chain for a page stored at least three levels deep.

View solution

EXERCISE 3

Store a page body containing a real tag, and confirm <%= %> escapes it into visible text while <%- %> renders it as real HTML.

View solution

CHAPTER 4 QUICK REFERENCE

- `<% %>` / `<%= %>` — the same base shape as Rails' own ERB, under a deliberately parallel name
- `<%- %>` — EJS's own raw-output tag, a genuinely different symbol from Rails' `<%== %>`
- `include()` — plain template inlining; no first-class layout system the way Astro's `<slot />` is
- `getBreadcrumb` — the sixth and final deliberately planted N+1 pattern in the series
- **Next chapter:** Styling — Dark Theme

Every sibling course in this series had some asset story, however minimal. Express's own is the most minimal of all six.

Serving a Plain Stylesheet

```
// server.js
app.use(express.static('public'));
```

```
/* public/css/global.css */
body {
  background: #0f1117;
  color: #c9d1d9;
  font-family: system-ui, sans-serif;
  margin: 0;
}
```

```
<!-- views/partials/header.ejs -->
<link rel="stylesheet" href="/css/global.css">
```

Genuinely More Minimal Than Even Rails' Propshaft

ZERO PROCESSING, NOT JUST MINIMAL PROCESSING

Rails Rebuild's own Chapter 5 called Propshaft minimal — but Propshaft still does exactly one real thing: content-hash fingerprinting for cache-busting. `express.static()` does none of that. It's a literal raw static-file server, serving bytes from disk unchanged, with no fingerprinting, no cache-busting, no compilation step of any kind — Express's own standard middleware for a job every basic web server already knows how to do, offered here with zero enhancement on top.

A Real, Honest Operational Gap

NO CACHE-BUSTING EXISTS AT ALL

Since `express.static()` serves the file at the exact same URL every time, a visitor's browser (or an intermediate proxy) may keep serving a stale, cached copy of `global.css` after an update, with nothing in this setup to signal that the file changed. A manual versioned query string — `/css/global.css?v=2`, bumped by hand on every real change — is the honest, low-tech fix; there's no automatic equivalent to Propshaft's own fingerprinting here.

Scoped Styles Don't Apply Here at All

Astro's own scoped `<style>` blocks worked because Astro has real components with real boundaries to scope styles to. EJS partials have no such boundary — `include()` just inlines one template's own output into another's, so the entire concept of "scoping" a style to one partial doesn't have anywhere to attach. This isn't a missing feature so much as a question that doesn't arise in a template-inclusion system the way it does in a component system.

The Full Asset-Bundling Spectrum, Closed Out

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO	EXPRESS
Approach	Bundles by design	Ships nothing	Pre-wires Vite	Propshaft + Importmap	Scoped styles + plain import	`express.static()`
Cache-busting?	Yes, automatic	No	Yes, via Vite	Yes — content-hash fingerprinting	Build-time hashing	None — manual only

Hands-On Exercises

EXERCISE 1

Set up express.static('public'), link global.css from the header partial, and confirm the dark theme applies site-wide.

View solution

EXERCISE 2

Update global.css, reload without a query-string change, and confirm a browser may still serve the stale cached version — then fix it with a manual ?v= bump.

View solution

EXERCISE 3

Compare the bytes of a served CSS file directly against the source file on disk, and confirm they're identical — no fingerprinting, hashing, or transformation applied at all.

View solution

CHAPTER 5 QUICK REFERENCE

- `express.static('public')` — a raw static-file server, zero processing
- **More minimal than Rails' Propshaft** — no fingerprinting, no cache-busting at all
- **Real gap** — a manual versioned query string is the only cache-busting available
- **Scoped styles don't apply** — no component boundary exists to scope them to
- **Next chapter:** The Database: Raw SQL, No ORM

Five times now, a different ORM in this series caught the same deliberately-planted N+1 breadcrumb pattern and fixed it with a nested eager-load call — and five times, that fix admitted the identical limitation: a fixed depth, hard-coded into the nesting itself, with no way to express "however deep this chain actually goes." Every one of those chapters named the theoretical fix: a raw recursive SQL query. This chapter finally builds it.

The Recursive Query

```
WITH RECURSIVE ancestors AS (
  SELECT *, 0 AS depth FROM pages WHERE id = ?
  UNION ALL
SELECT p.*, a.depth + 1 FROM pages p
  INNER JOIN ancestors a ON p.id = a.parent_id
)
SELECT * FROM ancestors ORDER BY depth DESC;
```

The base case starts at the target page (`depth 0`). Each recursive step joins back to `pages` to find the current row's own parent, incrementing `depth` as it climbs — continuing until a row's `parent_id` no longer matches anything (the root has been reached). `ORDER BY depth DESC` puts the highest depth — the furthest-back root ancestor — first, and the target page itself (`depth 0`) last: exactly root-to-leaf order, ready for a breadcrumb.

```
// lib/pages.js
async function getBreadcrumb(pool, pageId) {
  const [rows] = await pool.query(`
    WITH RECURSIVE ancestors AS (
      SELECT *, 0 AS depth FROM pages WHERE id = ?
      UNION ALL
      SELECT p.*, a.depth + 1 FROM pages p
      INNER JOIN ancestors a ON p.id = a.parent_id
    )
    SELECT * FROM ancestors ORDER BY depth DESC;
  `, [pageId]);
  return rows;
}
```

VERIFIED: GENUINELY UNBOUNDED, NOT JUST A DEEPER FIXED LIMIT

Every prior ORM's own nested eager-load call had to state its own depth explicitly — four levels, five levels, however many the developer chose to write. This query states no depth at all. A page three levels deep and a page thirty levels deep are both resolved by the exact same unchanged SQL, in one query, with zero code changes between them. This is the concrete answer to a limitation five separate courses in this series named but never actually solved.

STATED HONESTLY: AN UNBOUNDED-DEPTH WIN, NOT A UNIVERSAL PERFORMANCE WIN

This isn't "free" or automatically faster than every prior approach at the shallow depths this site realistically has — a four-level nested `with()` call and this recursive query both resolve a four-level chain in roughly comparable cost. The genuine advantage is specifically that this query's own correctness doesn't degrade or silently break as depth grows, where every sibling's own fixed-depth call would. MySQL's own recursive CTE support also has a real, configurable safety limit (`cte_max_recursion_depth`, defaulting to 1000) — far beyond anything a real content hierarchy would ever need, but worth knowing it exists.

The Whole Series' Own Shared Limitation, Finally Closed

FRAMEWORK	FIXED-DEPTH LIMITATION?
Next.js (Prisma)	Yes — nested `include`
Django	Yes — chained `select_related`
Laravel (Eloquent)	Yes — dotted `with()`
Rails (ActiveRecord)	Yes — nested `includes()`
Astro (Drizzle)	Yes — nested `with`
Express (raw SQL)	**No** — genuinely unbounded, via `WITH RECURSIVE`

Hands-On Exercises

EXERCISE 1

Build getBreadcrumb() using the recursive CTE, and confirm it correctly returns the full, correctly-ordered ancestor chain for a page three levels deep.

View solution

EXERCISE 2

Store a page six levels deep — deeper than every prior course's own fixed nested-call limit — and confirm the identical, unchanged query correctly returns the full six-level chain with zero code changes.

View solution

EXERCISE 3

Explain why ORDER BY depth DESC produces root-to-leaf order, and confirm removing the depth column and ORDER BY clause entirely produces an unpredictable row order instead.

View solution

CHAPTER 6 QUICK REFERENCE

- `WITH RECURSIVE ancestors AS (...)` — a real MySQL 8+ recursive common table expression
- **Base case + recursive case** — starts at the target page, climbs via `parent_id` until nothing more matches
- `depth` **column +** `ORDER BY depth DESC` — produces correct root-to-leaf order
- **Genuinely unbounded** — the same unchanged query resolves any depth, closing a limitation five sibling ORMs each admitted but never solved
- **Honestly stated** — an unbounded-depth win specifically, not a universal performance win at shallow depths
- **Next chapter:** Rendering Content & the Kanji Edge Case

The lightest chapter in the course — two of its usual three questions were already answered in earlier chapters, and the third was handled proactively before it could even become a question.

Routing & Rendering: Already Solved

Kanji's two original constraints — a fixed-depth hierarchy, and no self-contained-fragment convention — never applied here. Chapter 2's raw SQL schema has no depth limit of any kind, and Chapter 4's `<%- page.body %>` renders any page's own stored markup through the identical path every other page already uses.

String Safety: Confirmed a Third Time

THE SAME RUNTIME, A THIRD TIME IN THIS SERIES

Next.js and Astro both already established this: Node.js's own V8 engine represents strings as UTF-16 code units, and common kanji characters — including 水 — are single code units within that model, with no risk of a slicing operation corrupting them. Express runs on the identical runtime. This isn't a third framework independently reaching the same conclusion; it's the third time in this series the exact same language and engine have simply been reused.

`utf8mb4`: Already Handled, Learned in Advance

```
// db.js — written back in Chapter 2
const pool = mysql.createPool({
  uri: process.env.DATABASE_URL,
  charset: 'utf8mb4',
});
```

A GAP THE ASTRO REBUILD FOUND — ALREADY CLOSED HERE FROM THE START

Astro's own Chapter 7 discovered that `mysql2` has no automatic `utf8mb4` default the way Rails 5.2+ does, and had to add `charset: 'utf8mb4'` explicitly, mid-course. This course's own Chapter 2 already included that same option from the very first line of `db.js` — not a coincidence, but a direct, deliberate application of what Astro's own Chapter 7 already taught. The series' own accumulated findings get applied proactively here, not rediscovered.

Kanji Across All Six Frameworks

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO	EXPRESS
String-slicing risk	None — same runtime as Express	None	Real — `mb_substr()` needed	None	None — same runtime as Express	None — same runtime as Next.js/Astro
`utf8mb4` handling	Manual	Manual	Manual	Automatic since Rails 5.2	Manual — discovered as a gap	Manual — applied proactively from Chapter 2

Hands-On Exercises

EXERCISE 1

Confirm the two original kanji constraints are already resolved by Chapters 2 and 4, with no new code needed for this chapter.

View solution

EXERCISE 2

In a Node REPL, confirm '水のページ'.length and slicing behavior are safe for typical kanji content, the same test already run for the Astro rebuild's own Chapter 7.

View solution

EXERCISE 3

Confirm Chapter 2's own db.js already includes charset: 'utf8mb4', and explain why this course applied it proactively rather than discovering it as a gap the way the Astro rebuild did.

View solution

CHAPTER 7 QUICK REFERENCE

- **Routing/rendering** — already solved by Chapters 2 and 4, no new work
- **String safety** — the same Node.js/V8 runtime as Next.js and Astro, confirmed a third time
- `charset: 'utf8mb4'` — already present since Chapter 2, applied proactively from Astro's own earlier finding
- **The series' own accumulated knowledge, in action** — a gap discovered in one course, avoided entirely in a later one
- **Next chapter:** Dynamic Content & Forms

Astro's own `request.json()` needed zero setup — it's a standard Web API method on any `Request` object. Express needs one more step before `req.body` is even populated at all.

Without `express.json()`: `req.body` Is Undefined

```
// no app.use(express.json()) registered yet
app.post('/api/pages/:id/title', (req, res) => {
  console.log(req.body); // undefined — even for a real JSON request body
});
```

MORE MINIMAL THAN EVEN ASTRO, PRECISELY STATED

A modern Astro endpoint's `request.json()` works immediately, on any request, with nothing to configure — it's inherent to the standard `Request` object itself. Express requires an explicit middleware, `express.json()`, registered before any route can see a parsed body at all. Without it, `req.body` isn't an empty object — it's genuinely `undefined`, and code that assumes otherwise crashes.

Registering It, and Validating with Zod

```
// server.js
app.use(express.json());
```

```
// routes/admin.js
const { z } = require('zod');

const titleSchema = z.object({
  title: z.string().min(1).max(255),
});

app.post('/api/pages/:id/title', async (req, res) => {
  const result = titleSchema.safeParse(req.body);

  if (!result.success) {
    return res.status(400).json({ error: result.error.flatten() });
  }

  await pool.query('UPDATE pages SET title = ? WHERE id = ?', [result.data.title,
req.params.id]);

  res.json({ status: 'ok' });
});
```

Zod — the same choice Astro made — closes the same real gap: nothing in Express validates types, lengths, or required fields on its own.

The Deliberate No-Auth-Check-Yet Gap

Matching every sibling course's own Chapter 8, no authentication check exists yet, to be closed in Chapter 9. Like Astro's own version, there's no conventional place for one to go yet — Express provides nothing resembling `before_action` or `authorize()` to even have a placeholder in.

No CSRF Protection — Repeated Honestly a Final Time

THE SAME REAL, UNADDRESSED GAP AS ASTRO

Express ships nothing resembling Rails' `protect_from_forgery`, Laravel's `VerifyCsrfToken`, or Django's own CSRF middleware. A cross-origin request could hit this endpoint with no built-in defense at all, the identical gap Astro's own Chapter 8 named and left honestly unsolved.

Request Body Parsing, Compared

	ASTRO	EXPRESS
Setup required?	None — `request.json()` is a standard Web API method	Explicit — `app.use(express.json())` required
Behavior with no setup	Works by default	`req.body` is genuinely `undefined`

Hands-On Exercises

EXERCISE 1

Without registering express.json(), send a real JSON POST request and confirm req.body is undefined, causing naive destructuring code to throw.

View solution

EXERCISE 2

Add express.json() and the Zod validation, then send a malformed request (e.g. a numeric title) and confirm it's now rejected with a 400 response.

View solution

EXERCISE 3

Confirm no CSRF protection exists by successfully calling this endpoint from a genuinely different origin, with no same-origin check blocking it.

View solution

CHAPTER 8 QUICK REFERENCE

- `app.use(express.json())` — required just to get `req.body` populated at all
- **More minimal than Astro** — Astro's `request.json()` needs zero setup; Express's `req.body` is `undefined` without one
- **Zod's** `safeParse` — the same deliberate choice Astro made, closing the same real validation gap
- **No CSRF protection** — the identical unaddressed gap named in Astro's own Chapter 8
- **Next chapter:** Admin Authentication

No framework-provided auth mechanism at all — Chapter 1's own finding. This chapter builds real auth by hand, reusing what's already proven to work twice in this series.

Session Handling: `express-session`

```
// server.js
const session = require('express-session');

app.use(session({
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
  cookie: { secure: true, httpOnly: true },
}));
```

Even Astro at least had `@auth/astro`'s own session handling built into its integration. Express has none — `express-session` is a separate package, chosen and wired by hand, for a job every other framework in the series provided some form of by default.

The Login Route

```
const bcrypt = require('bcryptjs');

app.post('/admin/login', async (req, res) => {
  const { email, password } = req.body;
  const [rows] = await pool.query('SELECT * FROM admin_users WHERE email = ?', [email]);

  if (rows.length === 0) {
    return res.status(401).json({ error: 'Invalid credentials' });
  }

  const valid = await bcrypt.compare(password, rows[0].password_hash);
  if (!valid) {
    return res.status(401).json({ error: 'Invalid credentials' });
  }

  req.session.userId = rows[0].id;
  res.json({ status: 'ok' });
});
```

THE THIRD CONFIRMATION — THE IDENTICAL LIBRARY, A THIRD TIME

`bcryptjs` is the same npm package the Next.js rebuild's own Chapter 9 first used, and the Astro rebuild's own Chapter 9 confirmed again. This is the third time in this series the exact same library has verified the exact same legacy `$2y$`-tagged hash — not even a cross-ecosystem question here either, since it's still the identical JavaScript package doing the identical job.

Closing Chapter 8's Gap — a Real, General-Purpose Mechanism

```
// middleware/requireAuth.js
function requireAuth(req, res, next) {
  if (!req.session.userId) {
    return res.status(401).json({ error: 'Not authenticated' });
  }
  next();
}

module.exports = requireAuth;
```

```
// routes/admin.js — updated from Chapter 8
const requireAuth = require('../middleware/requireAuth');

app.post('/api/pages/:id/title', requireAuth, async (req, res) => {
  // ...unchanged from Chapter 8
});
```

MINIMALISM DOESN'T MEAN NO STRUCTURE — A GENUINE POSITIVE FINDING

Express provides no specific *auth* middleware, but it does provide a real, first-class, general-purpose extensibility mechanism — the `(req, res, next)` middleware signature — used here for exactly the same purpose Rails' `before_action` or Laravel's route middleware serve. `requireAuth` isn't a workaround; it's Express's own standard pattern for this job, just without a specific, pre-built version of it. Minimalism here produces something genuinely clean, not a gap.

Six Frameworks' Own Auth Stories, Closed Out

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO
Mechanism	Auth.js	Built-in `auth` app	Built-in `Auth` facade	`has_secure_password`	`@auth/astro` (same Auth.js)
Session handling	Built into Auth.js	Built in	Built in	Built in	Built into `@auth/astro`
Bcrypt match?	Yes, via `bcryptjs`	No — needed reconfiguration	Yes, zero config	Yes, with a real tag nuance	Yes — same `bcryptjs` package

Hands-On Exercises

EXERCISE 1

Set up express-session and the login route with bcryptjs, and confirm a correct login sets req.session.userId while an incorrect password returns 401.

View solution

EXERCISE 2

Build the requireAuth middleware and apply it to Chapter 8's own endpoint, and confirm an unauthenticated request is now rejected with 401.

View solution

EXERCISE 3

Confirm bcryptjs correctly verifies the legacy $2y$-tagged hash with no configuration, and name the two earlier chapters in this series where the identical package already did the same job.

View solution

CHAPTER 9 QUICK REFERENCE

- `express-session` — a separate package; Express provides no session handling of its own at all
- `bcryptjs` — the same npm package as Next.js and Astro, verifying the legacy hash a third time
- `requireAuth` — a hand-written middleware using Express's own real, general-purpose `(req, res, next)` mechanism
- **A genuine positive finding** — minimalism here produces a clean, standard pattern, not a gap
- **Next chapter:** Admin CRUD Interface

No free admin, no scaffold generator — the same most-minimal starting point Astro already reached. Everything here is hand-built, including the one place this course's own recursive SQL knowledge pays off a second time.

The Parent Picker

The same searchable flat list every sibling course reached for — every other page's own `full_path`, filterable client-side, excluding the page being edited from its own candidate-parent list.

Reparenting: Finding Every Descendant in One Query

```
// lib/pages.js
async function movePage(pool, pageId, newParentId) {
  const [pageRows] = await pool.query('SELECT * FROM pages WHERE id = ?', [pageId]);
  const page = pageRows[0];
  const newFullPath = await computeFullPath(pool, page.slug, newParentId);

  await pool.query('UPDATE pages SET parent_id = ?, full_path = ? WHERE id = ?',
[newParentId, newFullPath, pageId]);

  // Every descendant, found in ONE recursive query, ordered parent-before-child
const [descendants] = await pool.query(`
    WITH RECURSIVE descendants AS (
      SELECT *, 0 AS depth FROM pages WHERE id = ?
      UNION ALL
      SELECT p.*, d.depth + 1 FROM pages p
      INNER JOIN descendants d ON p.parent_id = d.id
    )
    SELECT * FROM descendants WHERE id != ? ORDER BY depth ASC;
  `, [pageId, pageId]);

  const pathById = { [pageId]: newFullPath };
  for (const descendant of descendants) {
    const parentPath = pathById[descendant.parent_id];
    const newDescendantPath = `${parentPath}/${descendant.slug}`;
    await pool.query('UPDATE pages SET full_path = ? WHERE id = ?', [newDescendantPath,
descendant.id]);
    pathById[descendant.id] = newDescendantPath;
  }
}
```

A REAL, VERIFIED IMPROVEMENT — PRECISELY SCOPED

Every sibling course's own cascade logic queried children *level by level*, walking down one database round-trip per level (Laravel's `updateDescendantPaths()`, Rails' `update_descendant_paths!`, Astro's own version). Here, Chapter 6's own recursive CTE finds **every** affected descendant, at any depth, in a single query — `ORDER BY depth ASC` means the results already arrive parent-before-child, so a single linear `for` loop, not a recursive function, is enough to process them in the correct dependency order.

STILL HONESTLY SEQUENTIAL WHERE IT HAS TO BE

The *discovery* step is genuinely one query instead of many. The *write* step is not — each descendant's own new `full_path` depends on its own parent's already-computed new path, so writing them still takes one `UPDATE` per row, run in order. This is a real, partial improvement, not a magic bullet: the recursive CTE removed the N+1 read pattern, not the N-write pattern, which is fundamentally unavoidable here regardless of ORM, framework, or query strategy.

Delete Refusal — the Same Ceremony as Laravel and Astro

```
app.delete('/api/pages/:id', requireAuth, async (req, res) => {
  try {
    await pool.query('DELETE FROM pages WHERE id = ?', [req.params.id]);
    res.json({ status: 'ok' });
  } catch (err) {
    res.status(409).json({ error: "Can't delete a page that still has children — move or delete them first." });
  }
});
```

Chapter 2 verified `ON DELETE RESTRICT` as the cleanest, most unambiguous database-level constraint in the series. That means the same `try`/`catch` ceremony Laravel and Astro both needed — not Rails' own smoother, non-exception path, which was ActiveRecord's own specific API design, not a universal trait.

Six Admin Interfaces, Compared One Final Time

	DJANGO	LARAVEL	RAILS	ASTRO	EXPRESS
Free admin / scaffold?	Live admin	None	Scaffold generator	Neither	Neither
Descendant discovery	Implicit `save()` cascade	Level-by-level recursion	Level-by-level recursion	Level-by-level recursion	One recursive CTE, all levels at once
Delete-with-children handling	Caught `ProtectedError`	Caught `QueryException`	Returns `false`, no exception	Caught raw exception	Caught raw exception

Hands-On Exercises

EXERCISE 1

Build the searchable flat parent picker for the admin interface, excluding the page itself from its own candidate-parent list.

View solution

EXERCISE 2

Build movePage() using the recursive CTE for descendant discovery, and confirm a real multi-level reparent operation correctly cascades full_path updates to every descendant in the right order.

View solution

EXERCISE 3

Attempt to delete a page with children via the DELETE endpoint, confirm the raw database constraint throws, and confirm the try/catch turns it into a real 409 response with a readable message.

View solution

CHAPTER 10 QUICK REFERENCE

- **No free admin, no scaffold generator** — the same most-minimal starting point as Astro
- **Recursive CTE descendant discovery** — one query finds every affected row, at any depth, in dependency order
- **The write step stays sequential** — a real, honestly-scoped limitation, not solved by the same query
- **Same delete-refusal ceremony as Laravel and Astro** — a real `try`/`catch` needed, unlike Rails' own smoother path
- **Next chapter:** Deployment

`server.js` is already a plain, standalone Node HTTP server — the identical situation Astro's own Node adapter produced in its Chapter 11. This chapter reuses that exact mechanism deliberately, rather than forcing a sixth distinct deployment story to exist for its own sake.

No Build Step at All

```
node server.js
```

Unlike every framework-based sibling in this series, there's nothing to compile, bundle, or transpile — `server.js` runs directly, exactly as written.

Secrets: One More Small Explicit Step

```
npm install dotenv
```

```
// server.js — must be the very first line
require('dotenv').config();
```

A plain `.env` file — the same simplest secrets story as Next.js, Django, Laravel, and Astro, not Rails' own reversed encrypted-commit approach. Even loading it needs one more explicit step here: `dotenv` is a separate package, where several sibling frameworks bundle some form of `.env` loading by convention.

Keeping the Process Alive: PM2, Reused Unchanged

```
pm2 start server.js --name website-express
pm2 save
pm2 startup
```

Apache's `mod_proxy_http`, Reused Unchanged

```
# Apache VirtualHost — the site's own already-running Apache
<VirtualHost *:443>
    ServerName osztromok.com

    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:3000/
    ProxyPassReverse / http://127.0.0.1:3000/

    SSLEngine on
    SSLCertificateFile /etc/letsencrypt/live/osztromok.com/fullchain.pem
    SSLCertificateKeyFile /etc/letsencrypt/live/osztromok.com/privkey.pem
</VirtualHost>
```

HONEST REUSE, NOT A FORCED SIXTH MECHANISM

Astro's own Chapter 11 used `mod_proxy_http` because its Node adapter produces a plain standalone HTTP server on a local port. Express's own `server.js` is exactly that same thing, with no framework layer in between. There's no genuine reason for a different Apache mechanism here — reusing the identical configuration is the honest answer, not a missed opportunity for a "sixth variant."

No Migration Tooling Either

A REAL, HONEST GAP — SCHEMA.SQL APPLIED BY HAND

With no ORM, there's no `drizzle-kit migrate` equivalent. Chapter 2's own `schema.sql` is applied directly via the `mysql` CLI, and any future schema change would need to be written and applied the same way — by hand, with no tracked migration history. A real production project at this scale would likely want a lightweight migration runner; this course doesn't build one, the same honest-scope-note tradition every capstone in this series already follows.

Six Deployment Stories, Compared One Final Time

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO	EXPR
App server process	Node (PM2)	gunicorn	PHP-FPM	Passenger-embedded	Node adapter (PM2)	Plain `serv` (PM2
Apache module used	N/A	N/A	`mod_proxy_fcgi`	`mod_passenger`	`mod_proxy_http`	`mod_` — reu uncha
Build step?	Yes	No	Yes (assets)	No	Yes	None

THE MOST MINIMAL DEPLOYMENT IN THE SERIES, TOO

No build step, no framework-specific adapter, no bundler output to verify — `node server.js` is genuinely the entire production artifact. Consistent with every other chapter in this course: the least is provided by default, and the least ceremony is required to run it.

Hands-On Exercises

EXERCISE 1

Run the production server directly via node server.js, with no build step, and confirm it serves the full site correctly.

View solution

EXERCISE 2

Configure PM2 to run the server, simulate a crash by killing the process directly, and confirm PM2 restarts it automatically.

View solution

EXERCISE 3

Configure the Apache VirtualHost with mod_proxy_http reverse-proxying to the Node server, and confirm the site is reachable through Apache's own domain rather than the raw Node port directly.

View solution

CHAPTER 11 QUICK REFERENCE

- `node server.js` — the entire production artifact, no build step at all
- `dotenv` — one more explicit package needed, even for the simplest possible secrets story
- **PM2 and** `mod_proxy_http` — reused unchanged from Astro's own Chapter 11, the honest answer for the identical situation
- **No migration tooling** — a real, named gap; `schema.sql` applied by hand
- **Next chapter:** Capstone — A Complete, Working Flexible-Routing Site

Website Rebuild with Express

Chapter 12 · Capstone: A Complete, Working Flexible-Routing Site

Sam — the same site admin persona from every capstone in this series, reused a sixth and truly final time — logs into this course's own deployed Express version. Every step below draws on a specific earlier chapter, closing with an attribution table, an honest scope note, and the close of the entire six-framework Website Rebuild series.

STEP 1 — VISITING THE LIVE DEPLOYMENT

Sam opens the site. Apache's own `mod_proxy_http` forwards the request to `server.js`, kept alive under PM2 — the identical mechanism Astro's own capstone already confirmed, reused honestly rather than reinvented.

STEP 2 — LOGGING IN

Sam enters the legacy admin credential. `bcryptjs` — the same package verified in this series for a third time — checks it against the stored `$2y$`-tagged hash, and `express-session` establishes a real session on success.

STEP 3 — BUILDING THE FIVE-LEVEL CHAIN

Through the hand-built admin interface — no free admin, no scaffold generator, matching Astro's own most-minimal finding — Sam creates `programming`, then `general-purpose-languages`, then `java`, then `fundamentals`, then `chapter-1`. Each save calls Chapter 2's own `computeFullPath` helper directly, with no ORM lifecycle hook anywhere in sight.

STEP 4 — VIEWING THE PAGE

Visiting the full five-segment path, Chapter 3's `/*splat` route resolves it with a single raw SQL query, and Chapter 4's EJS partials render the breadcrumb and `<%- page.body %>` content, styled by Chapter 5's plain, zero-processing stylesheet.

STEP 5 — EDITING THE TITLE, NOW ACTUALLY PROTECTED

Sam edits the Chapter 1 page's title. Chapter 8's endpoint and Zod validation handle the submission exactly as before — but Chapter 9's `requireAuth` middleware is now in place, the first genuine "place for an auth check" this course ever had. Logged out, the identical request now returns a `401` instead.

STEP 6 — REORGANIZING: THE RECURSIVE CTE CASCADE, FOR REAL

Sam decides `fundamentals` belongs under a different parent. Chapter 10's `movePage` updates `fundamentals`' own path directly, then Chapter 6's own recursive CTE finds every descendant — just `chapter-1`, here, but genuinely unbounded regardless of how deep the real chain went — in a single query, ordered so one linear pass writes each new path correctly.

STEP 7 — ATTEMPTING TO DELETE A PAGE WITH CHILDREN

Sam tries to delete `java`, which still has `fundamentals` beneath it. Chapter 2's `ON DELETE RESTRICT` constraint refuses the operation, and Chapter 10's `try`/`catch` turns the raw database exception into a real, readable `409` response — the same ceremony Laravel's and Astro's own capstones needed, not Rails' smoother built-in path.

STEP 8 — CONFIRMING THE KANJI MIGRATION HELD UP

Sam visits the kanji page. Chapter 7's resolution holds on every front: routing and rendering were already solved by Chapters 2 and 4, JavaScript's own UTF-16 string model — shared with Next.js and Astro, confirmed a third time — never risked corrupting the character, and the database's own `utf8mb4` charset, present in `db.js` since Chapter 2, stores it correctly with nothing rediscovered along the way.

STEP 9 — THE SERIES' OWN CENTRAL FINDING, CONFIRMED ONE FINAL TIME

Sam checks the query log while loading the breadcrumb-heavy Chapter 1 page. Chapter 6's `WITH RECURSIVE` query resolves the full ancestor chain in one call — genuinely unbounded, the concrete answer to a limitation five separate ORMs in this series each admitted but never actually solved.

STEP 10 — CONFIRMING THE DEPLOY ITSELF IS HEALTHY

Sam confirms PM2 shows `server.js` running cleanly, with no build artifact anywhere to verify — the last confirmation in a series that has now rebuilt the same real site six separate times.

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1. Visiting the deployment	11 — Deployment
2. Logging in	9 — Admin Authentication
3. Building the five-level chain	2 — URL & Content Model, 10 — Admin CRUD
4. Viewing the page	3 — Routing, 4 — Views, 5 — Styling
5. Editing the title	8 — Dynamic Content & Forms, 9 — Admin Authentication
6. Reorganizing	2 — deferred cascade cost, 6 — recursive CTE, 10 — `movePage`
7. Delete refusal	2 — `ON DELETE RESTRICT`, 10 — Admin CRUD
8. Kanji migration	7 — Rendering Content & the Kanji Edge Case
9. The series' own central finding	6 — The Database: Raw SQL, No ORM
10. Deploy health check	11 — Deployment

Honest Scope Note

WHAT THIS COURSE DELIBERATELY DOESN'T COVER

- No multi-admin roles or permission levels — a single legacy admin credential only
- No revision history or audit log of content edits
- No media/image upload handling
- No automated tests or CI pipeline
- No internationalization beyond the kanji character-storage edge case itself
- No rate limiting or lockout policy on the login form
- **No CSRF protection** — a real, unaddressed gap named honestly in Chapter 8, never solved
- **No migration tooling** — `schema.sql` applied by hand, a real gap named in Chapter 11

EXPRESS'S OWN HONEST CONTRIBUTIONS TO THIS SERIES

Two genuinely Express-specific findings, grounded in real, verified facts: Chapter 6's `WITH RECURSIVE` query, the concrete answer to the fixed-depth limitation five sibling ORMs each admitted but never solved, made possible specifically because no ORM stood between the code and the database; and Chapter 9's `requireAuth` middleware, a genuine positive finding that Express's own minimalism produces a real, general-purpose extensibility mechanism, not merely an absence of structure.

Hands-On Exercises

EXERCISE 1

Trace Step 6's reorganization through the code: what does movePage() update directly, and how does the recursive CTE plus the linear write pass handle everything beneath it?

View solution

EXERCISE 2

Explain what changed between Step 5 in this capstone and the identical-looking action back in Chapter 8, and name the exact line of code responsible for the difference.

View solution

EXERCISE 3

Name two genuinely Express-specific findings from this course — not shared with any sibling — and explain the real technical fact each one is grounded in.

View solution

COURSE COMPLETE

- **12/12 chapters** — Website Rebuild with Express is now complete
- **Standout findings named honestly throughout** — the recursive CTE payoff (Ch6), the real cascade limitation stated precisely (Ch10), a genuine positive minimalism finding (Ch9)
- **Real limits named honestly too** — no CSRF protection (Ch8), no migration tooling (Ch11)
- **Sixth and truly final course in the Website Rebuild series** — alongside Next.js, Django, Laravel, Rails, and Astro

★ The Website Rebuild Series Is Now Complete — Six Frameworks, 72 Chapters

One real site, rebuilt six separate times: Next.js, Django, Laravel, Ruby on Rails, Astro, and Express. Each course found its own genuine advantages and honest limits rather than declaring any single architecture simply "the best" — the same standard the Food Tracker Quartet set for comparative courses on this site. The series' own single longest-running thread — the breadcrumb's deliberately planted N+1 pattern, caught and fixed by a different mechanism in every course — reached its true conclusion here: five ORMs each admitted the same fixed-depth limitation for an arbitrary-depth ancestor chain, and it took the one course with no ORM at all to finally build the real, unbounded answer.