

Node.js & Express — Practice

Build the review routes on an in-memory store (Session 4)

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Abstract Course lectures and practices for JavaScript full-stack web development with AI-assisted workflows.

Table of contents

1 Overview	1
1.1 The AI-first pattern (CRISP)	2
2 Setup & running the app	2
3 Task 1 — the in-memory review store	3
3.1 AI generates	3
3.2 You verify — common AI mistakes for this task	4
3.3 You integrate	4
4 Task 2 — GET /:id/reviews	5
4.1 AI generates	5
4.2 You verify — common AI mistakes for this task	6
4.3 You integrate	6
5 Task 3 — GET /:id/aggregate	6
5.1 AI generates	6
5.2 You verify	7
5.3 You integrate	7
6 Task 4 (optional) — watch the React UI light up	7
6.1 You do	7
6.2 You integrate	7
7 Common AI mistakes (review routes)	7
8 Key takeaways	8

9 Home stretch (1 month)	9
--------------------------------	---

i Note

All AI prompts in this practice are **starting points** — feel free to reword them to match your style. The skill you are practising is not “copy the prompt”; it is **verifying what the AI writes** against the checklists below. You are the quality gate on the generated code. Using a coding agent is **optional** from Session 3 onward — verifying its output is **not**.

1 Overview

Where you are. In **Session 3** (React review practice) you built the React review UI. Today you build the **backend review routes** it calls — first on an **in-memory store** (MySQL arrives in Session 5). Read the Session 4 lecture first — it covers the concepts (Express, routing, the in-memory store, 404-early, the response

envelopes); this practice is where you build and verify them.

Goal — extend the existing **fullstack-minimal-app/backend** Express app (do **not** start a new project — you extend it in place, in **ESM**) with:

1. an **in-memory review store** in `backend/src/data/reviews.js` — a Map keyed by product id, seeded with the 8-field reviews;
2. **GET** `/api/products/:id/reviews` → `{ data, meta: { count } }`, 404-early;
3. **GET** `/api/products/:id/aggregate` → `{ data: { overall_avg, per_source, histogram, count } }`, 404-early, safe on a product with zero reviews;
4. (optional) confirm the S3 React review UI now renders live data from these routes.

In Session 4 the reviews live in the **in-memory store** — **no new database table** and no live fetching (the product-existence check keeps using the existing MySQL product DB on `:3306`). The API exposes the **8-field** review shape; `rating` is an integer **1–5**; and the `product_id` join column only appears in Session 5’s table. The third canonical route, **POST** `/api/products/:id/fetch` (against the shipped `scraper-service/` on `:5000`), is the S5 home stretch — you do not build it today.

1.1 The AI-first pattern (CRISP)

Every task below follows the same four steps (CRISP):

Step	What you do
Concept	1–3 lines on <i>what</i> you are building and <i>why</i> it matters.
AI generates	Give the assistant the CRISP, copy-paste prompt in the box.

Step	What you do
You verify	Run it, then check it against the <i>common AI mistakes for this task</i> list. Fix what fails.
You integrate	Confirm the checkpoint (what “done” looks like) and make the git commit .

💡 Tip

If the AI’s output fails any item on a checklist, **do not just copy it** — tell the AI *which* item failed and ask it to fix only that. That back-and-forth **is** the skill.

2 Setup & running the app

This practice builds against the course’s real app, **fullstack-minimal-app** (develop). It has an Express backend on `http://localhost:4000` (the React/Vite frontend from S3 is on `:5173`, but you will not need it until the optional Task 4).

1. Get the app (if you have not already):

```
git clone https://github.com/ebpro/fullstack-minimal-app.git
cd fullstack-minimal-app
```

2. Start the backend (so it listens on `:4000`):

```
cd fullstack-minimal-app/backend
npm install
npm run dev          # or `npm start`
– see the README
```

3. Verify the app is up — and that the review routes do NOT exist yet (building them is this practice):

3.1 AI generates

```
curl http://localhost:4000/health
curl      http://localhost:4000/api/
products
curl      http://localhost:4000/api/
products/1/reviews
```

- `curl http://localhost:4000/health` — should respond `{ status: "ok", ... }`.
- `curl http://localhost:4000/api/products` — returns the catalog in a `{ data, meta }` envelope.
- `curl http://localhost:4000/api/products/1/reviews` — should **404** today. That route is exactly what you will add in Task 2.

3 Task 1 — the in-memory review store

Concept. The reviews are not in MySQL yet (that is Session 5 — S4 adds **no new database table**), so you hold them in a **Map** in `backend/src/data/reviews.js` — product id → array of the 8-field reviews — that the routes import. The store is the *only* thing the review routes depend on for review data; in Session 5 you swap it for the `reviews` table by changing the bodies of the two helpers, not the routes.

💡 Tip

Copy-paste this prompt to your assistant:

```
Context:      Express      4.18
backend (`fullstack-minimal-app/
backend`, ESM, Node 22). In S4
reviews live
in the in-memory store `src/data/
reviews.js`; the MySQL reviews
table arrives in S5. The product
catalog already lives in MySQL
(`src/db.js` exports the pool).
Task:      Create      `src/data/
reviews.js` exporting the store
and its two helpers.
Requirements:
- `const reviewsByProduct = new
Map()` seeded for products '1',
'2' and '3' (string keys – route
params arrive as strings), with
realistic e-commerce review text.
- Each review has EXACTLY 8
fields: review_id (a string like
'AMZ_0001'), source (one of
Amazon|BestBuy|Walmart),
author, rating (an integer 1–5),
title, body, created_at (an ISO
timestamp string),
verified_purchase (a boolean). No
"was this useful?" flag and NO
product_id
field (that join column exists
only in Session 5's table).
- Export `getProductReviews(id)`
returning `reviewsByProduct.get(String(id))`.
- Export
`aggregateProductReviews(id)`
returning `{ overall_avg,
per_source, histogram, count }`:
overall_avg is the mean rating
rounded to 2 decimals, per_source
the mean per source present,
```

```
histogram the buckets 1..5 with
all five keys always present (even
when 0), count the number
of reviews.
```

```
- Empty case: a product with no
reviews must return overall_avg
1.0, per_source {all: 1.0},
```

3.2 You verify — common AI mistakes for this task

- **Wrong keys.** The Map is keyed by the **string** ids '1', '2', '3' — route params arrive as strings, and the lookup is `reviewsByProduct.get(String(id)) ?? []`. Numeric keys 1, 2, 3 would never match.
- **Extra or missing fields.** Every review has **exactly** the 8 canonical fields — no “was this useful?” flag, no `product_id` (that join column only exists in Session 5’s table).
- **Rating out of range.** `rating` is an integer 1–5 — no 0, no 6, no floats.
- **CommonJS.** The file must be ESM (`export ...`), like the rest of `backend/src`.
- **Aggregate math.** For a product, `aggregateProductReviews` returns histogram keys 1..5 whose values sum to count; `per_source` contains each source that is present; an empty product returns `overall_avg: null, count: 0`, and all five histogram buckets at 0.

3.3 You integrate

Checkpoint. From `backend/`, print product 1’s aggregate — you should see the 1–5 histogram, the per-source means, and count; then print an unseeded product — it must show the safe empty shape (`overall_avg: null, count: 0`, all five buckets at 0, no NaN):

```
node -e "import('./src/data/
reviews.js').then(m => console.log(m.aggregateProductReviews(1)))"
node -e "import('./src/data/
reviews.js').then(m => console.log(m.aggregateProductReviews(0)))"
```

Then lock it in:

```
git add src/data/reviews.js && git
commit -m "feat(backend): in-memory
review store (8-field, 1-5) +
aggregate helper"
```

4 Task 2 — GET `/:id/reviews`

Concept. 404-early: confirm the product exists in the existing MySQL product table *before* touching the in-memory store — an unknown product id must 404 whether or not it happens to have reviews. Then return the store's reviews in the `{ data, meta }` envelope the S3 React frontend already reads.

4.1 AI generates



Tip

Copy-paste this prompt to your assistant:

```
Context: Express 4.18 router `src/
routes/products.js` (ESM, mounted
at /api/products, so the full
path matters). `pool` is already
imported from `../db.js`; Task 1's
`getProductReviews` lives in
`../data/reviews.js`. The S3
React frontend reads the `{ data,
meta }` envelope.
```

```
Task: Add one route so its full
path is GET /api/products/:id/
reviews.
```

```
Requirements:
```

```
- 404-early: first confirm
the product exists with a
parameterized SELECT on the
products table;
```

```
    if it does not, return 404
{ error: 'Product not found' }
BEFORE reading the store.
```

```
- Success: respond { data:
[review...], meta: { count } }
with the store's reviews for :id.
```

```
Constraints: inline async (req,
res, next) handler with plain try/
catch forwarding to next(err);
```

```
no handler wrapper, no
custom error class; import
getProductReviews from ../data/
reviews.js;
```

```
do not touch the existing CRUD
routes; do not query any reviews
table (S4 has none).
```

```
Output: only the import line and
the new route block.
```

4.2 You verify — common AI mistakes for this task

With the backend running on :4000, run the two curls and check each response:

- `curl http://localhost:4000/api/products/1/reviews` — returns `{ data: [...8-field reviews...], meta: { count } }`.
- `curl http://localhost:4000/api/products/999/reviews` — returns `404 { error: 'Product not found' }`. If it returns an empty list instead, the store is being read before the product check (404-early is missing).
- Every returned review has exactly the 8 fields and a rating of 1–5.
- The route keeps the inline `async (req, res, next)` handler with `next(err)` in the catch — no wrapper helper, no custom error class.

4.3 You integrate

Checkpoint. Both curls above behave as specified (200 with the envelope for a real product, 404 for an unknown one). Then lock it in:

```
git add src/routes/products.js &&
git commit -m "feat(backend):
GET /:id/reviews (in-memory, 404-
early, {data,meta})"
```

5 Task 3 — GET /:id/aggregate

Concept. Same **404-early** rule as Task 2: confirm the product exists in the existing MySQL product table *before* touching the in-memory store. Then compute the aggregate from the in-memory reviews and return it in the `{ data }` envelope — the aggregate has **no meta** (that belongs only to the list route). A real product with **zero reviews** must still return a valid, deterministic aggregate (no crash, no NaN).

5.1 AI generates



Tip

Copy-paste this prompt to your assistant:

Context: the same router ``src/routes/products.js``; Task 1's ``aggregateProductReviews`` lives in ``../data/reviews.js``.

Task: Add one route so its full path is `GET /api/products/:id/aggregate`.

Requirements:

- **404-early:** the same parameterized product-existence check as the reviews route; return 404

`{ error: 'Product not found' }` before touching the store when the product does not exist.

- Success: respond `{ data: aggregate }` where `aggregate` is what `aggregateProductReviews(:id)` returns — `{ overall_avg, per_source, histogram, count }`. No ``meta`` on this route.

- The zero-review case must be safe: `overall_avg` null, `count` 0, all five histogram buckets 0 — never NaN, never divide-by-zero.

Constraints: inline `async (req, res, next)` handler with plain `try/catch` forwarding to `next(err)`; delegate all aggregation math to `aggregateProductReviews` — do not recompute it in the route; do not query any reviews table (S4 has none).

Output: only the added route block (and its import if not already present).

5.2 You verify

With the backend running on :4000, run the curls and check each response:

- `curl http://localhost:4000/api/products/1/aggregate` — returns `{ data: { overall_avg, per_source, histogram, count } }`, where `histogram` has **all five keys 1..5** (every bucket present even when its count is 0).
- `curl http://localhost:4000/api/products/999/aggregate` — returns `404 { error: 'Product not found' }`.
- `curl http://localhost:4000/api/products/4/aggregate` (a real product the store did not seed) — returns `200` with `overall_avg: null, count: 0` and all five histogram buckets at `0` — never `NaN`, never a divide-by-zero crash.
- `overall_avg` and every value in `per_source` are **numbers rounded to 2 decimals**.

5.3 You integrate

Checkpoint. Both curls above behave as specified (200 with the `{ data }` envelope for a real product, 404 for an unknown one). Then lock it in:

```
git add src/routes/products.js && git
commit -m "feat(backend): GET /:id/
aggregate (in-memory, 404-early,
{data})"
```

6 Task 4 (optional) — watch the React UI light up

Concept. The **S3** React review components already call these two routes. Now that the routes return live in-memory data, the UI you built in Session 3 renders it — no frontend change needed.

6.1 You do

1. Open `http://localhost:5173` and go to a product page.
2. Confirm the **review list** and the **1–5 histogram** render from the two routes you just built.
3. Confirm a product with **no reviews** shows the empty state / 404 handling (the 404-early rule from Tasks 2–3).

6.2 You integrate

Checkpoint. The **S3** UI renders live data from both routes (and handles the empty/404 case). Then lock it in:

```
git add -A && git commit -m "chore:
verify React review UI against the S4
review routes"
```

7 Common AI mistakes (review routes)

Mistake	What to do instead
404 not early — an unknown product returns an empty list instead of 404	Confirm the product exists first (<code>SELECT id FROM products WHERE id = ?</code>) and return the 404 before reading the store.
Wrong envelope — the list route returns a bare array, or the aggregate route	The list route returns <code>{ data, meta: { count } }</code> ; the aggregate route returns <code>{ data }</code> (no <code>meta</code>).

Mistake	What to do instead
returns <code>{ data, meta }</code>	
Invented fields — a “was this useful?” flag or a <code>product_id</code> field on a review	Keep exactly the 8 canonical fields; <code>product_id</code> is only a join column in Session 5’s table, never a review field.
Rating out of range — <code>rating</code> as 0–5 or a float	<code>rating</code> is an integer 1–5 .
Aggregation math wrong — the histogram is missing zero buckets, or <code>per_source</code> drops a source that is present	Always emit all five keys 1..5 (even when a count is 0), and include every source that is present .
No error handling — forgetting <code>try/catch + next(err)</code>	Wrap the async work in <code>try/catch</code> and forward to <code>next(err)</code> (no wrapper, no custom error class).
Hard-coded ids — reading a fixed	Read <code>req.params.id</code> from the route; never hardcode a product id.

Mistake	What to do instead
product id instead of <code>:id</code>	
Reaching for a live/external review source — S4 is in-memory only	Use the in-memory store. The shipped <code>scraper-service/(:5000)</code> and its <code>POST /api/products/:id/fetch</code> route are the S5 home stretch , not Session 4.

8 Key takeaways

- The **in-memory store** (`backend/src/data/reviews.js`) is **S4’s only review data source**; no new database table in S4. In S5 it is swapped for the MySQL `reviews` table (which adds a `product_id` join column) — only the two helper bodies change, never the routes.
- **404-early** — confirm the product exists *before* doing any work, so a missing product is an error, not a silently-empty list.
- **One envelope per route type** — the list route returns `{ data, meta }`; the aggregate route returns `{ data }` (no `meta`).
- The **8-field shape and the 1–5 integer rating are the contract** — no invented flags, no `product_id` field, no out-of-range rating.
- **Aggregation = overall_avg + per_source** (mean per source) + a **1..5 histogram** (all five buckets) + `count`. A product with zero reviews aggregates **safely**: `overall_avg: null, count: 0`, all five histogram buckets at 0 — no NaN, no divide-by-zero.
- The **AI loop is the method**: AI generates → **you verify** → fix what fails → commit.

Make the **404-early** rule a live moment: call `GET /api/products/999/aggregate` and

show the clean `404 { error: 'Product not found' }`, then contrast it with a route that silently returns an empty list. Preview the S5 swap — the only change is the body of `getProductReviews` becoming a `pool.query(...)` on the `reviews` table filtered by `product_id`, so **nothing on the frontend changes** because the route signature and the `{ data } / { data, meta }` envelopes stay identical.

9 Home stretch (1 month)

In S5 the in-memory store becomes the persistent **MySQL `reviews` table** (which adds a `product_id` join column, FK → `products.id`, on the MySQL instance at `:3306`). The course ships a mock **scraper-service/** (Express, port `:5000`) that exposes review data at `GET /api/scrape/reviews/:productId`. In the 1-month home stretch you wire the third canonical route, **`POST /api/products/:id/fetch`**, which asks the scraper-service for a product's reviews and stores them — that is where the “live” story begins. S4 never calls the scraper.