

Node.js & Express: The Backend

The Product Catalog REST API — and the Review Routes that Extend It

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. This session takes you from the React frontend (Session 3) to the backend that powers it: the Express product-catalog REST API, the MySQL data layer, and the review routes that turn a plain catalog into a multi-source review aggregator.

Table of contents

0.1	Node.js & Express: The Backend ☒ . . .	2
0.2	🕒 Where you are & what is new . . .	2
0.2.a	Covered before (do NOT re-teach)	2
0.2.b	New in this session	2
0.2.c	What this session does <i>not</i> do . . .	2
0.3	📖 The system at a glance	3
0.4	📦 RECALL / if needed — Node & Express basics	4
0.5	📦 RECALL / if needed — what is a connection <i>pool</i> ?	4
1	☒ The mandatory core	5
1.1	1 · The Express app — <code>src/index.js</code>	5
1.2	2 · Routing — <code>src/routes/products.js</code> and <code>src/routes/categories.js</code>	6
1.2.a	<code>src/routes/categories.js</code> (read this one first — it is small)	7
1.2.b	<code>src/routes/products.js</code> (the CRUD router)	7
1.3	3 · The <code>{ data, meta }</code> envelope and pagination	9
1.4	4 · The data layer — <code>src/db.js</code> + parameterized queries	10
1.4.a	Parameterized queries = SQL-injection safety	10
1.5	5 · Input validation — <code>src/middleware/validators.js</code>	10
1.6	6 · Centralized error handling — <code>src/middleware/errorHandler.js</code>	11
1.7	7 · CORS	11
1.8	☒ How the real app does it	11
1.9	🧩 The “extend it” — the review routes	12
1.9.a	The addition you build	13

1.9.b	The hard rules (Pitfall #6 and friends)	15
1.9.c	What you can test in S4 (and what moves to S5)	15
1.10	🔧 AI-generate → You verify (the Express beat)	17
1.11	📄 Prompt templates (copy-paste)	17
1.12	📌 Key takeaways	19

0.1 Node.js & Express: The Backend 📌

0.2 🕒 Where you are & what is new

Sessions 1–3 built the **frontend**: the language, the tooling, and (Session 3) a **React** app — React 19.1 + Vite 7.1 + Tailwind 3.4 (build-time) + React Router 7.9 — running in the browser on **:5173**. This session meets the **backend** that gives that frontend real data.

The central theme of this course is a multi-source product review aggregator.

- **Base (understand it fully):** the *Minimal Product Catalog* REST API — a Node + Express service backed by MySQL that serves products and categories over HTTP.
- **Extension / capstone (you build it):** a set of **review routes** on the backend — GET `/api/products/:id/reviews` and GET `/api/products/:id/aggregate` — that serve a product’s reviews and their aggregated statistics.

This is the exact shape of the course’s illustration project, `fullstack-minimal-app` (referenced **by URL**, not as a submodule). You **understand** the base API, then **extend** it.

The real stack (do not deviate):

Piece	Version / value				Port
Back-end	Node	22	+	Express 4.18	+ 4000
	mysql2 (pool)				
Data-base	MySQL	8.0	—	db	3306
	<code>minimal_app_db</code> , user <code>appuser</code>				
Front-end	React	19.1	+	Vite 7.1 + Tailwind 3.4	+ 5173
	+ React Router 7.9				
Scraper service	Shipped	<code>scraper-service/</code> (mock review data) — S5 home stretch only , never called in S4			5000

0.2.a Covered before (do NOT re-teach)

- JavaScript fundamentals (Sessions 1–2)
- **React + Vite** frontend on **:5173** (Session 3)
- The frontend calls the backend through `VITE_BACKEND_URL` (default `http://localhost:4000`)

0.2.b New in this session

- The **Express app** and its **middleware order** (`src/index.js`)
- **Routing** for products and categories (`src/routes/products.js`, `src/routes/categories.js`)
- The **{ data, meta } envelope** and **pagination**
- The **mysql2 pool** (`src/db.js`) + **parameterized queries** (SQL-injection safety)
- **Validation** (`src/middleware/validators.js`), **centralized errors** (`src/middleware/errorHandler.js`), and **CORS**
- The **review routes** — the “extend it” addition (list + aggregate; in-memory in S4, MySQL-backed in S5)

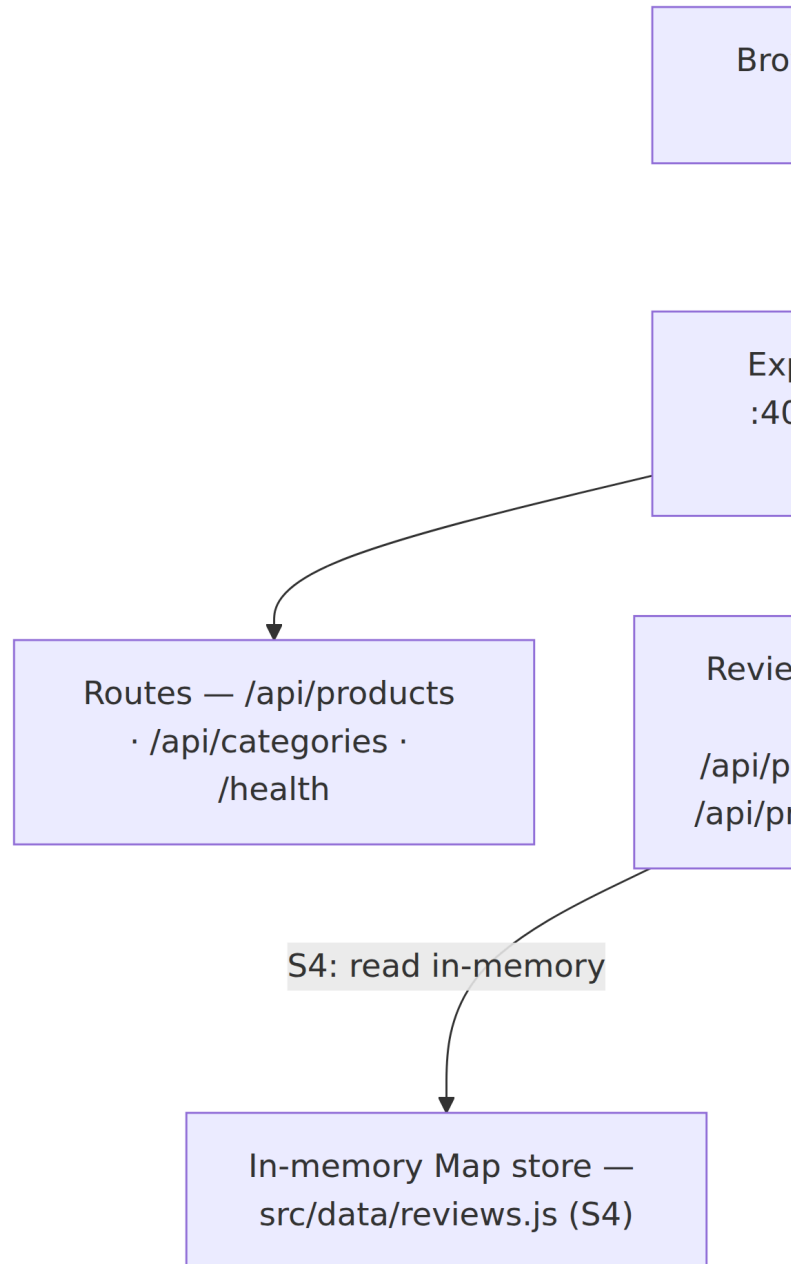
0.2.c What this session does *not* do

- It does **not** re-teach React.
- It does **not** go deep on database design and seeding — that is **Session 5 (the database)**.

Here you learn *how the app talks to MySQL*; Session 5 makes you a database person.

Land the **theme** before the code: “everything we build serves one goal — showing a product with reviews gathered from multiple sources.” The catalog is the base; the review routes are the payoff. Reinforce **understand** → **extend**: we are reading a small, well-structured code-base, then adding one feature. Foreshadow **Session 5**: “we touch MySQL just enough to run today; next session we build the database.”

0.3 📖 The system at a glance



Two facts to internalize:

1. **The browser only ever talks to the back-end (:4000).** It never talks to the database, and it never talks to any other service.

2. In S4 the review routes read an in-memory store (`src/data/reviews.js`); in S5 they read the MySQL `reviews` table. The product-existence check in S4 already uses the existing product database. The shipped `scraper-service/ (:5000)` is the mock review source for the S5 home stretch — S4 never calls it.

0.4 📦 RECALL / if needed — Node & Express basics

RECALL / if needed — what is a request handler, and what is middleware? **Node.js** runs JavaScript outside the browser, so you can build a web **server**. **Express** is a thin framework that makes routing and middleware pleasant.

The whole framework is one idea: *a function runs for every request*.

```
app.METHOD(PATH, HANDLER)
  |       |       |
  |       |       └─ function that
  |       └─ handles the request
  └─ URL path (the "address")
      └─ HTTP method (GET, POST, PUT, DELETE)
```

Two objects flow into every handler:

- **req** — the incoming request: `req.params` (URL parts like `/products/:id`), `req.query` (query string like `?page=2`), `req.body` (POST/PUT payload), `req.headers`.
- **res** — the outgoing response: `res.json(obj)`, `res.status(code)`, `res.send(text)`.

Middleware runs *before* the final handler, in registration order:

```
app.use(express.json()); // parses
                           the JSON body so req.body exists
app.use(morgan('dev'));  // logs
                           each request
app.use(cors());         // allows the
                           React frontend (a different origin)
```

HTTP methods (REST conventions):

Method	Meaning	Typical success
GET	read (safe, cacheable)	200
POST	create a new resource	201
PUT	replace a resource	200
DELETE	remove a resource	200 / 204

0.5 📦 RECALL / if needed — what is a connection pool?

RECALL / if needed — why a `mysql2` pool instead of one connection? Opening a MySQL connection is expensive (TCP handshake + authentication). A **new** connection per request wastes time and quickly hits the server's connection limit. A **pool** keeps a small set of reusable open connections: acquire one, run a query, release it; the next request reuses it.

```
import mysql from 'mysql2/promise';
const pool = mysql.createPool({
  host: process.env.DB_HOST ||
'localhost',
  user: process.env.DB_USER ||
'appuser',
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME ||
'minimal_app_db',
  waitForConnections: true,
```

```

    connectionLimit: 10,
    queueLimit: 0
  });

```

The call signature you will see everywhere:

```

const [rows, fields] = await
pool.query(sql, [ /* values */ ]);
//      ^--- what you want ^---
//      usually ignored

```

`pool.query` resolves to an array: the first element is the **rows** (for `SELECT`) or a **result object** (for `INSERT/UPDATE/DELETE`). The **? placeholders** in the SQL are filled from the second argument — that is *parameterization*, and it is what makes you safe from SQL injection.

1 ☒ The mandatory core

Nothing in this section is hidden — it is the base you must understand fully before you extend it.

1.1 1 · The Express app — `src/index.js`

`src/index.js` is the **single place the application is assembled**: it creates the app, wires middleware in the right order, mounts the routers, registers the health check, and — last — registers the centralized error handler.

i The module system is ESM

The app's `package.json` sets `"type": "module"`, so all backend code is **ESM**: `import/export` — never `require/module.exports` — and relative imports must end in `.js` (bare specifiers such as `express` or `mysql2/promise` take no extension). For brevity this lecture keeps error handling inline (`try/catch` → `next(err)`) and the SQL inline in the route; the full `fullstack-minimal-app` centralizes these in `asyncHandler/AppError` (`src/middleware/errorHandler.js`) and a `queries.js` builder. The **module system**, however, is always ESM.

```

// src/index.js — application entry
point (Node 22 + Express 4.18)
import express from 'express';
import cors from 'cors';
import morgan from 'morgan';
import helmet from 'helmet';

import db from './db.js';
import errorHandler from './
middleware/errorHandler.js';
import staticServing from './
middleware/staticServing.js';
import productRoutes from './routes/
products.js';
import categoryRoutes from './routes/
categories.js';

const app = express();

// — 1. Middleware (ORDER MATTERS)

app.use(express.json()); // parse
JSON bodies → req.body exists
app.use(morgan('dev')); // log each
request

```

```

app.use(cors()); // allow
the React frontend (different origin)
app.use(helmet()); // sensible
security headers

// — 2. Routes —————
app.use('/
api/products', productRoutes); //
CRUD + (next) the review routes
app.use('/api/categories',
categoryRoutes); // categories
app.use(staticServing); //
serve the built frontend, if present

// — 3. Health check (reports DB
status) —————
app.get('/health', async (req, res)
=> {
  let database = 'unknown';
  try {
    const conn = await
db.getConnection();
    await conn.ping();
    database = 'connected';
    conn.release();
  } catch {
    database = 'disconnected';
  }

  res.json({ status:
'ok', database, timestamp: new
Date().toISOString() });
});

// — 4. Centralized error handler —
MUST be registered LAST —
app.use(errorHandler);

const PORT = process.env.PORT ||
4000;
app.listen(PORT, () =>
console.log(`Backend running on
http://localhost:${PORT}`));

```

Why the order matters (the point of this section):

- **express.json() before the routes** — otherwise `req.body` is `undefined` and every `POST/PUT` sees an empty body.
- **morgan, cors, helmet early** — they wrap every request.
- **errorHandler last** — it catches anything `next(err)` throws; if it ran *before* the routes it would never see their errors.

💡 AI beat — the app shell

Concept → “assemble an Express app: correct middleware order, mounted routers, a DB health check, and a last-registered error handler.”

AI generates → see *Prompt template 1*.

You verify (do not skip): - Is `express.json()` registered **before** any route? - Is the error handler registered **after** the routes? - Does `/health` actually **ping the pool** and report it, not return a constant?

You integrate — start it, hit `GET /health`, then commit:

```

curl http://localhost:4000/health
git add src/index.js
git commit -m "feat(backend):
express app shell — middleware
order, routes, health, error
handler"

```

1.2 2 · Routing — `src/routes/products.js` and `src/routes/categories.js`

A **router** is a self-contained bundle of routes. The product router owns every `/api/products...` path; the category router owns `/`

api/categories. Keeping them separate is what lets you *extend* products.js (the review routes) without touching anything else.

The exact endpoints this API exposes:

Method	Path	Purpose
GET	/api/products	List (query: page, per_page, category_id)
GET	/api/products/:id	Get one product
POST	/api/products	Create a product
PUT	/api/products/:id	Update a product
DELETE	/api/products/:id	Delete a product
GET	/api/categories	List categories ({ data: [{id, name}] })
GET	/health	Liveness + DB status

1.2.a src/routes/categories.js (read this one first — it is small)

```
import express from 'express';
import pool from '../db.js';
const router = express.Router();

// GET /api/categories → { data: [ {id, name} ] }
router.get('/', async (req, res, next) => {
  try {
    const [rows] = await pool.query('SELECT id, name FROM categories ORDER BY name');
    res.json({ data: rows });
  } catch (err) {
    next(err);
  }
});

export default router;
```

Three habits to copy: (1) wrap async work in try/catch and forward to next(err); (2) return the { data, ... } envelope, never a bare array; (3) export default router;.

1.2.b src/routes/products.js (the CRUD router)

```
import express from 'express';
import pool from '../db.js';
import { validateProduct } from '../middleware/validators.js';
const router = express.Router();

// — GET /api/products (query: page, per_page, category_id) —
router.get('/', async (req, res, next) => {
  try {
    const page = Math.max(1, parseInt(req.query.page, 10) || 1);
    const perPage = Math.min(100, Math.max(1, parseInt(req.query.per_page, 10) || 10));
    const offset = (page - 1) * perPage;

    let where = '1=1';
    const params = [];
    if (req.query.category_id) { where = 'p.category_id = ?'; params.push(req.query.category_id); }

    const [[{ total }]] = await pool.query(
      `SELECT COUNT(*) AS total FROM products p WHERE ${where}`, params
    );
    const [rows] = await pool.query(
      `SELECT p.*, c.name AS category_name FROM products p LEFT JOIN categories c ON`
    );
  }
});
```

```

c.id = p.category_id
  WHERE ${where}
  ORDER BY p.id DESC
  LIMIT ? OFFSET ?`,
  [...params, perPage, offset]
);

res.json({
  data: rows,
  meta: { total,
page, per_page: perPage, total_pages:
Math.ceil(total / perPage) }
});
} catch (err) { next(err); }
});

// — GET /api/products/:id —
router.get('/:id', async (req, res,
next) => {
  try {
    const [rows] = await pool.query(
      `SELECT p.*, c.name AS
category_name
      FROM products p LEFT JOIN
categories c ON c.id = p.category_id
      WHERE p.id = ?`,
      [req.params.id]
    );
    if (rows.length === 0)
return res.status(404).json({ error:
'Product not found' });
    res.json({ data: rows[0] });
  } catch (err) { next(err); }
});

// — POST /api/products —
router.post('/', validateProduct,
async (req, res, next) => {
  try {
    const { name, description, price,
image_url, category_id } = req.body;
    const [result] = await pool.query(
      'INSERT INTO products

```

```

(name, description, price, image_url,
category_id) VALUES (?, ?, ?, ?, ?)',
      [name, description ??
null, price, image_url ?? null,
category_id ?? null]
    );
    const [rows] = await
pool.query('SELECT * FROM products
WHERE id = ?', [result.insertId]);
    res.status(201).json({ data:
rows[0] });
  } catch (err) { next(err); }
});

// — PUT /api/products/:id —
router.put('/:id', validateProduct,
async (req, res, next) => {
  try {
    const { name, description, price,
image_url, category_id } = req.body;
    const [result] = await pool.query(
      'UPDATE products SET name = ?,
description = ?, price = ?, image_url
= ?, category_id = ? WHERE id = ?',
      [name, description,
price, image_url, category_id,
req.params.id]
    );
    if (result.affectedRows === 0)
return res.status(404).json({ error:
'Product not found' });
    const [rows] = await
pool.query('SELECT * FROM products
WHERE id = ?', [req.params.id]);
    res.json({ data: rows[0] });
  } catch (err) { next(err); }
});

// — DELETE /api/products/:id —
router.delete('/:id', async (req,
res, next) => {
  try {
    const [result] = await

```

```
pool.query('DELETE FROM products
WHERE id = ?', [req.params.id]);
    if (result.affectedRows === 0)
return res.status(404).json({ error:
'Product not found' });
    res.status(204).send();
  } catch (err) { next(err); }
});

export default router;
```

💡 AI beat — the CRUD router

Concept → “a parameterized CRUD router returning the { data, meta } envelope, with pagination and a category_id filter on the list endpoint.”

AI generates → see *Prompt template 2*.

You verify: - Every SELECT/INSERT/UPDATE/DELETE uses **? placeholders**, never concatenation. - 404 is returned **early** (with return) when a resource is missing. - POST returns **201**; the list computes meta.total_pages from total.

You integrate — exercise all four verbs, then commit:

```
curl "http://localhost:4000/api/
products?page=1&per_page=10"
git add src/routes/products.js
git commit -m "feat(backend):
product CRUD router with
pagination + { data, meta }
envelope"
```

1.3 3 · The { data, meta } envelope and pagination

The whole API speaks **one** response shape, so the frontend writes *one* parser for every route.

```
// List endpoint → { data, meta }
{
  "data": [
    {
      "id": 1,
      "name": "Mechanical Keyboard",
      "description": "RGB,
hot-swappable switches",
      "price": "89.99",
      "image_url": "https://.../
keyboard.jpg",
      "category_id": 3,
      "category_name":
"Peripherals", // ← from the LEFT
JOIN on categories
      "created_at":
"2026-09-01T10:15:00Z"
    }
  ],
  "meta": { "total": 20, "page": 1,
"per_page": 10, "total_pages": 2 }
}
```

- **data** — the payload (an array for lists, a single object for GET /:id).
- **meta** — pagination bookkeeping: total, page, per_page, total_pages.
- **category_name** is *not* a column in products; it is produced by the LEFT JOIN categories in the query (a LEFT join keeps products with no category).

A single-resource endpoint has no meta — just { data: product }:

```
{
  "data": {
    "id": 1,
    "name": "Mechanical Keyboard",
    "category_id": 3, "category_name":
"Peripherals" } }
}
```

Note: this is the **real** envelope of this project. The from-scratch recall box used

a `{ success, data }` shape purely as a teaching device — do **not** mix the two.

1.4 4 · The data layer — `src/db.js` + parameterized queries

```
// src/db.js — a mysql2 connection
POOL (see the RECALL box above)
import mysql from 'mysql2/promise';

const pool = mysql.createPool({
  host: process.env.DB_HOST ||
'localhost',
  user: process.env.DB_USER ||
'appuser',
  password: process.env.DB_PASSWORD,
  neverHardcoded
  database: process.env.DB_NAME ||
'minimal_app_db',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});

export default pool;
```

1.4.a Parameterized queries = SQL-injection safety

The **only** way this codebase talks to MySQL is through `?` placeholders:

```
const id = req.params.id;

// ✗ DANGEROUS — string
concatenation. If id = "1 OR 1=1 --",
the whole table is returned.
const [bad] = await
pool.query(`SELECT * FROM products
WHERE id = ${id}`);

// ☐ SAFE — parameterized. The driver
sends SQL and value separately;
// the value is NEVER interpreted
```

as SQL.

```
const [good] = await
pool.query('SELECT * FROM products
WHERE id = ?', [id]);
```

When you can see a `?` next to a user-supplied value, the query is safe. When you can see a `${}` inside a SQL string, stop and fix it. That single habit separates an API from a vulnerability.

💡 AI beat — the pool + one safe query

Concept → “a mysql2 pool configured from environment variables, plus one parameterized query.”

You verify: - No connection string or password appears as a **literal** — all from `process.env`. - Every query touching user input uses `?`, never `${}`.

You integrate — confirm `/health` reports database: “connected”, then commit:

```
git add src/db.js
git commit -m "feat(backend):
mysql2 pool from env +
parameterized queries"
```

1.5 5 · Input validation — `src/middleware/validators.js`

Never trust `req.body`. A small, reusable middleware validates *before* the route handler runs:

```
// src/middleware/validators.js
function validateProduct(req, res,
next) {
  const { name, price } = req.body;
  const errors = [];
  if (!name || !String(name).trim())
```

```

errors.push('name is required');
    if (price === undefined
||      isNaN(Number(price)) ||
Number(price) < 0)

errors.push('price must be a
non-negative number');
    if (errors.length)
return res.status(400).json({ error:
errors.join('; ') }); // reject early
next(); // clean → proceed to the
route handler
}

export { validateProduct };

```

It is mounted **only on the routes that need it** (router.post('/', validateProduct, handler)), so GET/DELETE are not slowed by validation they do not need.

1.6 6 · Centralized error handling — src/middleware/errorHandler.js

Every handler forwards failures with `next(err)`. One place turns them into clean JSON:

```

// src/middleware/errorHandler.js
// Express treats a function with
FOUR arguments as error-handling
middleware.
function errorHandler(err, req, res,
next) {
  console.error(err.stack);
  log the real error server-side
  const status = err.status ||
500; // let handlers set 4xx;
default 500
  res.status(status).json({ error:
err.message || 'Internal server
error' });
}

```

```
export default errorHandler;
```

Two rules: handlers **never** `res.status(500)` themselves — they `next(err)`; and the four-argument signature is what makes Express route errors here, not into your routes.

1.7 7 · CORS

The React app is served from `http://localhost:5173`; the API lives on `http://localhost:4000`. Different **origins** (scheme + host + port) → the browser blocks the cross-origin request unless the server opts in:

```
app.use(cors()); // this teaching
app: allow the frontend origin
```

For production you would restrict it (`cors({ origin: [ allowedOrigins ] })`). The principle: **CORS is enforced by the browser, configured by the server.**

1.8 ☒ How the real app does it

Everything above maps to the real `fullstack-minimal-app/backend` layout:

```

backend/
├─ db/
│   └─ init.sql          ← schema
+ seed (3 categories, 20 products)
├─ src/
│   └─ index.js          ← app
assembly (middleware, routes, health,
error handler)
│   └─ db.js            ←
mysql2 connection POOL
│   └─ middleware/
│       └─ validators.js ← input
validation (validateProduct)
│           └─ errorHandler.js ←
centralized 4-arg error handler

```

```

| | └─ staticServing.js ← serve
the built frontend, if present
| └─ routes/
|   └─ products.js ←
product CRUD (+ the review routes
you will add)
|   └─ categories.js ←
categories
└─ package.json

```

The data model (from db/init.sql; Session 5 seeds it):

```

categories ( id PK, name VARCHAR(255)
NOT NULL )

products (
  id          INT PK,
  name        VARCHAR(255) NOT NULL,
  description  TEXT,
  price       DECIMAL(10,2) NOT
NULL CHECK (price >= 0 AND price
<= 999999.99),
  image_url   VARCHAR(1024),
  category_id INT FK → categories
ON DELETE SET NULL,
  created_at  TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
)
-- indexes: idx_products_category,
idx_products_created,
idx_products_category_created
-- seed: 3 categories (Electronics,
Books, Home) · 20 products

```

Run it and confirm the contract yourself:

```

cd backend && npm install && npm run
dev          # → http://localhost:4000
curl http://localhost:4000/health
curl "http://localhost:4000/api/
products?page=1&per_page=5&category_
id=1"

```

```

curl http://localhost:4000/api/
products/1
curl http://localhost:4000/api/
categories

```

Walk the file tree left-to-right and tie each file to a concept already taught. Emphasize that the `LEFT JOIN` is the only place `category_name` appears. Ask: “why `ON DELETE SET NULL` and not `CASCADE`?” → deleting a category must not delete its products; the product simply becomes uncategorized.

1.9 🧩 The “extend it” — the review routes

The base app has **no reviews**. In S4 you add **two review routes** to `src/routes/products.js`, backed by a small **in-memory** store in `src/data/reviews.js` (a Map seeded with review objects). S4 introduces **no new database table** for reviews — they live in memory — while the product existence check keeps using the **existing MySQL-backed product catalog**. In S5 the store is swapped for the **MySQL reviews table** (only the data-access lines change — the route signatures and responses stay identical).

The canonical review API across the course:

Method	Path	When
GET	/api/products/:id/reviews	S4 — you build it
GET	/api/products/:id/aggregate	S4 — you build it
POST	/api/products/:id/fetch	S5 home

Method	Path	When
		<pre>"verified_purchase": true }</pre>
		stretch } — wires the
		1.9.a The addition you build shipped Put the store in <code>src/data/reviews.js</code> and scrape have the routes import its two helpers. In service S4 the store is a <code>Map</code> keyed by product (<code>:5000</code>)id, seeded with a few reviews. In S5 you out replace the store's two helper functions with of scope <code>pool.query(...)</code> against the MySQL <code>reviews</code> in S4 table; the route signatures and responses stay identical.

The two routes you build in S4:

Method	Returns
GET /api/products/:id/ reviews	the product's reviews → { data: [review], meta: { count } } (404 if the product is missing)
GET /api/products/:id/ aggregate	overall average, per- source averages, 1–5 histogram → { data: {...} } (404 if the prod- uct is missing)

A **review** has exactly 8 fields (note: no helpful; rating is an **integer 1–5**):

```
{
  "review_id": "AMZ_0001",
  "source": "Amazon",
  Amazon | BestBuy | Walmart
  "author": "J. Rivera",
  "rating": 4,
  integer, 1–5
  "title": "Great value",
  "body": "Solid build, ships fast,
  battery lasts as advertised.",
  "created_at":
  "2026-08-20T09:30:00Z",
```

```
//
```

```
//
```

```
// src/data/reviews.js — the in-
// memory store (S4 only, ESM)
// In S5, replace the
// bodies of getProductReviews /
// aggregateProductReviews with
// pool.query('SELECT ... FROM reviews
// ...') calls — the routes below do NOT
// change.
const reviewsByProduct = new Map([
  ['1', [
    { review_id: 'AMZ_0001', source:
    'Amazon', author: 'J. Rivera',
    rating: 4,
    title: 'Great value',
    body: 'Solid build, ships fast.',
    created_at: '2026-08-20T09:30:00Z',
    verified_purchase: true },
    { review_id: 'BB_0002',
    source: 'BestBuy', author: 'M. Chen',
    rating: 5,
    title: 'Love it',
    body: 'Battery lasts as advertised.',
    created_at: '2026-08-21T14:10:00Z',
    verified_purchase: true },
    { review_id: 'WM_0003', source:
    'Walmart', author: 'P. Okafor',
    rating: 3,
    title: 'Okay', body:
    'Does the job, packaging was thin.',
```

```

    created_at: '2026-08-22T08:00:00Z',
    verified_purchase: false },
    ],
    ['2', [
      { review_id: 'AMZ_0010', source:
        'Amazon', author: 'S. Diallo',
        rating: 2,
          title: 'Underwhelming',
        body: 'Smaller than expected.',
        created_at: '2026-08-23T11:45:00Z',
        verified_purchase: true },
    ]],
  ]);

export function
getProductReviews(productId) {
  return reviewsByProduct.get(String(productId));
}

```

```

export function
aggregateProductReviews(productId) {
  const reviews =
    getProductReviews(productId);
  // Empty guard: never divide by
  reviews.length when it is 0.
  // Deterministic histogram: all five
  buckets 1..5 are always present.
  if (reviews.length === 0) {
    return { overall_avg: null,
      per_source: {}, histogram: { 1: 0, 2:
    0, 3: 0, 4: 0, 5: 0 }, count: 0 };
  }
  const histogram = { 1: 0, 2: 0, 3:
    0, 4: 0, 5: 0 };
  const bySource = {};
  let sum = 0;
  for (const r of reviews) {
    sum += r.rating;
    histogram[r.rating] += 1;
    (bySource[r.source] ??= { sum: 0,
      count: 0 });
    bySource[r.source].sum +=

```

```

    r.rating;
    bySource[r.source].count += 1;
  }
  const per_source =
    Object.fromEntries(
      Object.entries(bySource).map(([src,
        v]) => [src, Number((v.sum /
        v.count).toFixed(2))])
    );
  return {
    overall_avg: Number((sum /
      reviews.length).toFixed(2)),
    per_source,
    histogram,
    count: reviews.length,
  };
}

```

```

// src/routes/products.js - ADD the
two review routes (after the CRUD
routes)
import { getProductReviews,
  aggregateProductReviews } from '../
data/reviews.js';

// GET /api/products/:id/reviews →
the product's reviews (404 early if
the product is missing)
router.get('/:id/reviews', async
(req, res, next) => {
  try {
    const [productRows] = await
      pool.query('SELECT id FROM products
        WHERE id = ?', [req.params.id]);
    if (productRows.length === 0) {
      return
        res.status(404).json({ error:
          'Product not found' });
    }
    const data =
      getProductReviews(req.params.id);
    res.json({ data, meta: { count:

```

```

data.length } });
  } catch (err) {
    next(err);
  }
});

// GET /api/products/:id/aggregate →
// overall avg, per-source averages, 1-
// 5 histogram
router.get('/:id/aggregate', async
(req, res, next) => {
  try {
    const [productRows] = await
pool.query('SELECT id FROM products
WHERE id = ?', [req.params.id]);
    if (productRows.length === 0) {
      return
res.status(404).json({ error:
'Product not found' });
    }
    res.json({ data: aggregateProductReviews(req.params.id) });
  } catch (err) {
    next(err);
  }
});

```

In Session 5 you swap the in-memory store for the MySQL reviews table — replace the bodies of `getProductReviews` / `aggregateProductReviews` with `pool.query(...)`; the two route handlers, their paths, and their `{ data, meta } / { data }` responses do **not** change.

The shipped `scraper-service/ (:5000)` is the S5 home stretch — the future source of mock review data behind `POST /api/products/:id/fetch`. It is *not* called in S4.

1.9.b The hard rules (Pitfall #6 and friends)

The route 404s first, before touching the store. Confirm the product exists (`SELECT id FROM products WHERE id = ?`) and return 404 early — never let a missing product fall through to an empty review list that the frontend mistakes for “no reviews yet.”

Why it matters: - **Early 404** — a product that doesn’t exist is an error, not an empty result. The frontend shows a 404 page, not a silently-empty review grid. - **One envelope** — the list route returns `{ data, meta }` (with `meta.count`); the aggregate route returns `{ data }`. Mixing these up is the #1 silent bug (see the S3 checklist). - **Exactly 8 fields** — never invent a helpful field on a review; its 8-field shape carries no `product_id` (in S4 the store is keyed by product id; S5’s table adds a `product_id` join column).

1.9.c What you can test in S4 (and what moves to S5)

- **S4 (in-memory):** both routes are fully testable with the seeded `src/data/reviews.js` store — **no new database table** in S4 (the product-existence check still runs against the existing MySQL product DB).
- **404 path:** `GET /api/products/999/reviews → 404 { error: 'Product not found' }`.
- **S5 (MySQL):** the store is replaced by the reviews table; the handlers, paths, and responses stay identical, so nothing on the frontend changes.
- **S5 table adds `product_id`:** the MySQL reviews table stores the same 8 fields plus a `product_id` (FK → `products.id`) so `GET /:id/reviews` can run `WHERE product_id = ?`. The review’s 8-field shape

is unchanged — `product_id` is a join detail, not a review field.

💡 AI beat — the review routes

Concept → “two Express routes on the products router — `GET /:id/reviews` and `GET /:id/aggregate` — that read a product’s reviews from an in-memory store in S4 (MySQL in S5), 404 early on a missing product, and return the `{ data, meta }` / `{ data }` envelopes.”

AI generates → see *Prompt template 3*.

You verify: - Both routes **404 first** on a missing product, **before** reading the store. - The list route returns `{ data, meta }` with `meta.count`; the aggregate route returns `{ data }`. - The aggregate has a **1–5 histogram** (buckets 1..5) plus `overall_avg` and `per_source`. - Every review has **exactly 8 fields**; `review_id` is a **string** (e.g. `AMZ_0001`); `rating` is an integer **1–5**; no `helpful`, no `product_id`. - The store lives in `src/data/reviews.js` and the routes **import** its helpers — no inline data. - The aggregate handles a **product with zero reviews**: `overall_avg: null`, `count: 0`, all five histogram buckets present at 0 — never `NaN` or a divide-by-zero. - In S4 the **in-memory store is the only** review data source (no `POST /api/products/:id/fetch`, no `scraper-service/` call).

You integrate — test the happy path and the 404 path, then commit:

```
curl http://localhost:4000/api/
products/1/reviews          # →
{ data:[...], meta:{ count } }
curl http://localhost:4000/api/
products/1/aggregate        # →
{ data:{ overall_avg, per_source,
histogram, count } }
curl http://localhost:4000/api/
products/999/reviews        # → 404
{ error: 'Product not found' }
```

```
curl http://localhost:4000/api/
products/3/aggregate        #
real product, zero reviews →
{ overall_avg: null, count: 0,
histogram all 0 }
```

Make the **404-early** rule a **live** moment: call `GET /api/products/999/reviews` and show the clean 404 `{ error: 'Product not found' }`, then contrast it with a route that silently returns an empty list. Then preview the one-line S5 swap — replace the body of `getProductReviews` with a single `pool.query('SELECT ... FROM reviews ...')` — and emphasize that **nothing on the frontend changes** because the route signature and the `{ data, meta }` envelope stay identical.

1.10 🤖 AI-generate → You verify (the Express beat)

The workflow you just used, every time: **Concept** → **AI generates (a CRISP prompt)** → **You verify (against the checklist below)** → **You integrate (checkpoint + git commit)**. AI writes fast; the *verify* step is where correctness lives. Here is the standing checklist for Express work in this course.

Common AI mistake	Why it is wrong	Fix
SQL string concatenation (... <code>WHERE id = \${id}</code>)	SQL injection; <code>id = "1 OR 1=1 --"</code> dumps the table	Always parameterize with <code>?</code> + a values array
No <code>res.status()</code> handling	Failed upstream/DB call looks like success	Check <code>res.ok</code> / recall result rows; return the right code
Forgetting the error handler	Async errors crash the process or hang the request	<code>next(err)</code> in every handler; 4-arg handler registered last

Common AI mistake	Why it is wrong	Fix
Wrong response envelope	Frontend parser breaks per route	Always <code>{ data, meta }</code> (lists) / <code>{ data }</code> (single)
Missing CORS	Browser blocks the <code>:5173</code> → <code>:4000</code> request	Register <code>cors()</code> early in <code>src/index.js</code>
Hardcoding URLs / credentials	Breaks in Docker; leaks secrets	Read from <code>process.env</code> (<code>PORT, DB_*</code>)
Missing the 404-early product check	An empty list masquerades as “no reviews yet”	<code>SELECT id FROM products WHERE id = ?</code> → 404 before reading the store
Wrong review shape (invents helpful, or 0–5 rating)	Diverges from the real 8-field, int 1–5 review	Keep exactly the 8 fields; <code>rating</code> is an integer 1–5

1.11 📄 Prompt templates (copy-paste)

Prompt template 1 — the app shell

Context: Express 4.18 + Node 22 backend for a product catalog, port 4000.
Task: Write `src/index.js` that

assembles the app.
 Requirements:

- Middleware in THIS order: `express.json()`, `morgan('dev')`, `cors()`, `helmet()`.
- Mount routers: `app.use('/api/products', ...)` and `app.use('/api/categories', ...)`.
- GET `/health` that pings the `mysql2` pool (`src/db.js`) and reports `{ status, database, timestamp }`.
- A centralized 4-argument error handler registered LAST.
- `app.listen` on `process.env.PORT || 4000`.

Output: the complete `src/index.js` file only.

Prompt template 2 — the CRUD router with envelope + pagination

Context: Express router for a `products` table (columns: `id`, `name`, `description`, `price`, `image_url`, `category_id`, `created_at`) joined to `categories` for `category_name`. Use a `mysql2` pool.
 Task: Write `src/routes/products.js` implementing:

- GET `/` (query: `page`, `per_page`, `category_id`) → `{ data:[product], meta:{ total, page, per_page, total_pages } }`
- GET `/:id` → `{ data: product }` (404 if missing)
- POST `/` → 201, `{ data: product }`
- PUT `/:id` → `{ data: product }` (404 if missing)
- DELETE `/:id` → 204 (404 if missing)

Constraints: EVERY query uses `?` placeholders (no string concatenation). Wrap `async` work in `try/catch` and forward to `next(err)`.

`category_name` comes from a `LEFT JOIN categories`.
 Output: the complete router file, export default router.

Prompt template 3 — the review routes

Context: Express router (`src/routes/products.js`) with a `mysql2` pool. In S4 reviews live in an in-memory store in `src/data/reviews.js` (a Map keyed by product `id`, helpers exported); in S5 that store is swapped for the MySQL `reviews` table. A review has exactly 8 fields: `review_id` (a string like `"AMZ_0001"`), `source`, `author`, `rating` (integer 1–5), `title`, `body`, `created_at`, `verified_purchase`. There is NO helpful field and NO `product_id` field in a review.

Task: Add TWO routes (import the helpers from `../data/reviews.js`):

- GET `/:id/reviews` (full path `/api/products/:id/reviews`):

- 1) 404 early if the product (by `id`) does not exist (`SELECT id FROM products WHERE id = ?`);

- 2) return the product's reviews as `{ data: [review], meta: { count } }`.

- GET `/:id/aggregate` (full path `/api/products/:id/aggregate`):

- 1) 404 early if the product does not exist;

- 2) return `{ data: { overall_avg, per_source, histogram, count } }` from the store's aggregate helper — `overall_avg` is null and all five histogram buckets (1..5) are 0 when the product

has no reviews (never NaN, never divide-by-zero).

Constraints: both routes 404 before reading the store; every query is parameterized with ?; rating is an integer 1–5; no invented fields. Do NOT add POST `/:id/fetch` or call the scraper-service – that is the S5 home stretch. Output: only the added route blocks.

1.12 ☒ Key takeaways

- **One envelope, everywhere:** lists return `{ data, meta }` (with `total`, `page`, `per_page`, `total_pages`); single resources return `{ data }`. The frontend parses it once.
- **Middleware order is a contract:** `express.json()` before routes, the 4-arg error handler last. Get the order wrong and bodies or errors silently break.
- **Parameterize or die:** every user-supplied value goes through a `?` placeholder, never a `${}`. That one habit is your SQL-injection defense.
- **category_name is derived**, not stored – it comes from the `LEFT JOIN categories` in the query.
- **Errors are centralized:** handlers `next(err)`; one 4-arg handler shapes every error response.
- **The review routes are the capstone layer:** in S4 they read the **in-memory store** (`src/data/reviews.js`) – no new database table; in S5 they read the **MySQL reviews table**. Both return the app's envelopes – `{ data, meta }` (list) / `{ data }` (aggregate). The third canonical route, `POST /api/products/:id/fetch` (shipped scraper-service/ on `:5000`), is the S5 home stretch.
- **404 early:** confirm the product exists in the existing product DB before reading the

store, so a missing product is an error, not a silently-empty list.

- **A real review is exactly 8 fields** (`review_id` string like `AMZ_0001`, `source`, `author`, `rating` int 1–5, `title`, `body`, `created_at`, `verified_purchase`) – no helpful, no `product_id` field, no 0–5 rating; `product_id` only appears as S5's table join column.
- **The aggregate never divides by zero:** a product with no reviews returns `overall_avg: null`, `count: 0`, and all five histogram buckets 1–5 at 0.
- **The AI loop is the method:** Concept → AI generates → **You verify** → You integrate (commit). The verify step is where correctness actually comes from.

Close by naming what Session 5 will do: “today the app *read and wrote* MySQL through a pool and *held* reviews in an in-memory store; next session we *author* the schema, the indexes, and the seed, and swap the store for the `reviews` table.” Recap the review routes as the capstone layer that makes this a *review aggregator* rather than just a catalog.