

Database Integration — Lecture

Swap the in-memory store for a MySQL reviews table (Session 5)

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
2 The mysql2 connection pool	2
3 The reviews table	2
4 Seeding the reviews table	4
5 Swapping the store for SQL	4
6 The full capstone (React to Express to MySQL)	6
7 Key takeaways	7
8 Home stretch (1 month)	7

i Note

This is the session where the app becomes **persistent**. Everything you have built so far — the React review UI (Session 3) and the two review routes (Session 4) — reads from an **in-memory** store that vanishes on every restart. Today the **data source** swaps from that in-memory store to a real **MySQL reviews table**. The **frontend** and the **route signatures are UNCHANGED** — the only thing that moves is *where* the data comes from: an in-memory `Map` → the `reviews` table. The route handlers, their paths, and their `{ data, meta } / { data }` envelopes stay identical, so the React UI now renders **DB-backed reviews with no frontend change**.

1 Overview

- **Where you are.** In **Session 4** (Node.js & Express lecture / practice) you built the two review routes on an **in-memory** `Map` store.

Today you make it **persistent**: a real MySQL `reviews` table — and you wire the full capstone (React → Express → MySQL).

- **Goal** — (1) understand the **mysql2 connection pool** the app already uses; (2) **create the reviews table** (8 fields + `product_id`); (3) **seed** it with the reviews; (4) **swap** the in-memory helpers for `pool.query`; (5) confirm the **frontend now renders DB-backed reviews with no frontend change**.

The table adds a `product_id` join column (a foreign key) but the API still exposes the 8-field shape; the route handlers, paths, and envelopes are unchanged — only the helper bodies change.

2 The mysql2 connection pool

The app already talks to MySQL through a **connection pool** in `backend/src/db.js`. You do not invent a new one — you **reuse the pool the app already creates**, exactly the way the existing catalog routes (GET `/api/products`, GET `/api/categories`) already do. The review routes will use this same pool.

```
// backend/src/db.js - a mysql2/
promise connection POOL
import mysql from 'mysql2/promise';
import dotenv from 'dotenv';

dotenv.config();

const pool = mysql.createPool({
  host: process.env.DB_HOST ||
'localhost',
  port: process.env.DB_PORT ?
Number(process.env.DB_PORT) : 3306,
  user: process.env.DB_USER ||
'appuser',
  password: process.env.DB_PASSWORD
|| 'apppassword',
```

```
  database: process.env.DB_NAME ||
'minimal_app_db',
  waitForConnections: true,
  connectionLimit: 10,
  connectTimeout: 10000, // 10 seconds
});

export default pool;
```

Pool vs a single connection. Opening a MySQL connection is expensive: a TCP handshake plus authentication on every request. Creating a *new* connection per request wastes that time and quickly exhausts the server's connection limit. A **pool** keeps a small set of reusable, already-open connections: a request acquires one, runs its query, and releases it; the next request reuses it. That is what `connectionLimit` bounds and what `waitForConnections` queues.

Why a pool matters here. The catalog already runs concurrent `pool.query` calls (list + pagination + `LEFT JOIN`) on this same pool without opening a fresh connection per hit. The review routes simply join that same pool — no new connection logic, no new file:

```
const [rows] = await
pool.query('SELECT ... FROM reviews
WHERE product_id = ?', [productId]);
// ^--- what you want (the rows)
```

`pool.query` resolves to an array whose first element is the **rows** (for a `SELECT`). The **?** **placeholders** are filled from the second argument — that is *parameterization*, and it is what keeps you safe from SQL injection.

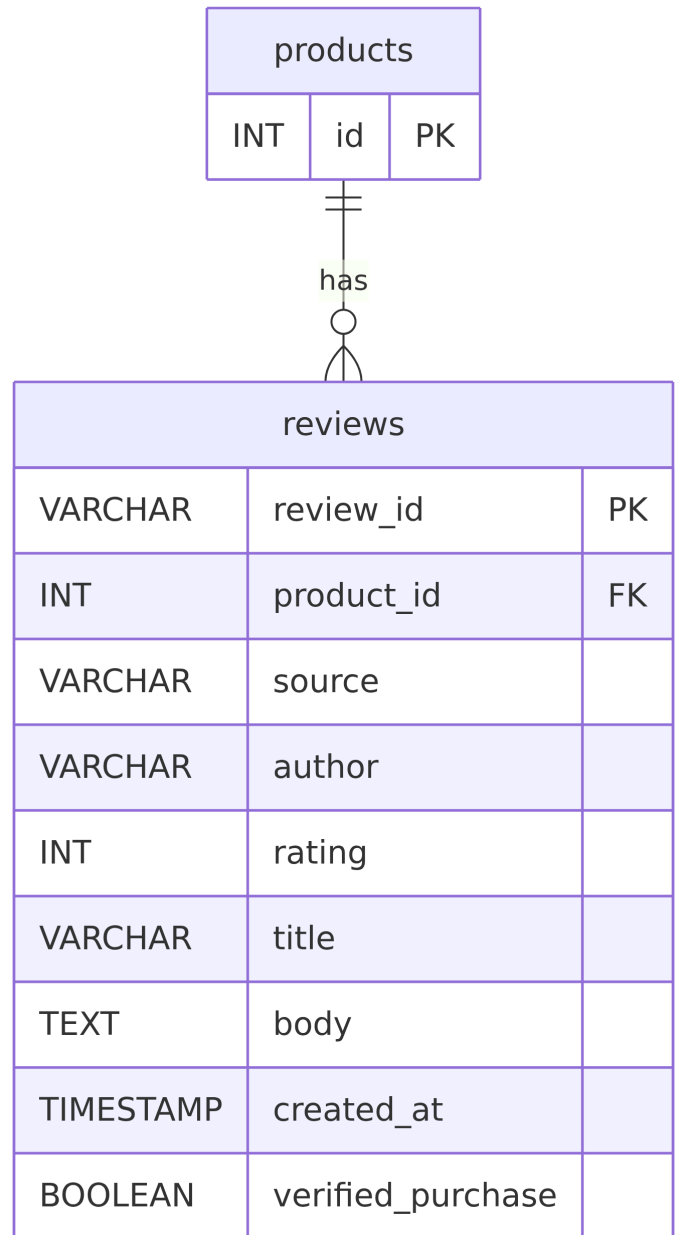
3 The reviews table

The `reviews` table stores the same **8 review fields** the API and frontend expose, **plus one `product_id` column** that links each review to

its product. This is the DDL from the course guide, reused verbatim:

```
CREATE TABLE reviews (
  review_id          VARCHAR(64)
  PRIMARY KEY,
  product_id         INT          NOT NULL,
  -- FK -> products.id (join detail,
  -- not part of the 8-field shape)
  source             VARCHAR(32)  NOT
NULL, -- Amazon | BestBuy | Walmart
  author             VARCHAR(100)
  NOT NULL,
  rating             INT          NOT
NULL CHECK (rating BETWEEN 1 AND 5),
  title              VARCHAR(255),
  body               TEXT,
  created_at         TIMESTAMP    NOT
NULL DEFAULT CURRENT_TIMESTAMP,
  verified_purchase   BOOLEAN     NOT
NULL DEFAULT FALSE,
  FOREIGN KEY (product_id) REFERENCES
products(id) ON DELETE CASCADE
);

-- Index for the dominant query
-- pattern: "all reviews for product X"
-- (WHERE product_id = ?)
CREATE INDEX idx_reviews_product ON
reviews(product_id);
```



Why **product_id** is a **JOIN column**, not a **review field**. The 8 review fields describe the review itself (review_id, source, author, rating, title, body, created_at, verified_purchase). The extra product_id does not describe the review — it **links** the review to the product it belongs to, so the review routes can run `WHERE product_id = ?`. That is exactly why `GET /api/products/:id/`

`reviews` can filter by the product in the URL. Because `product_id` is a join detail and not part of the review's data shape, the `SELECT` in `GET /:id/reviews` returns the **8 fields** — not `product_id`. (The rating `CHECK` keeps the 1–5 integer contract; `ON DELETE CASCADE` means deleting a product removes its reviews; the index serves the `WHERE product_id = ?` pattern.)

4 Seeding the reviews table

Seeding is how the `reviews` table gets its first rows. The data is *the same reviews* the in-memory store (`backend/src/data/reviews.js`) held in Session 4 — you port **every entry** of that file, so the exact row count is whatever your S4 store defines. The only difference is that each row now also carries its **`product_id`** (the join column that the in-memory `Map` key expressed implicitly). Once these rows are in MySQL, the data **persists across restarts**: stop the Node process, start it again, and the reviews are still there, because they live in the database, not in a `Map`.

The seed can live in a **db seed file** (a `.sql` script run during setup, alongside the catalog seed) or a **one-off script** you run once. The key point is not *where* it runs — it is that the data now **persists in MySQL across restarts**.

```
-- Representative seed rows: reviews
-- ported from the S4 in-memory store,
-- shown with the extra product_id
-- column (port every entry your S4 store
-- holds).
INSERT INTO reviews
  (review_id, product_id, source,
   author,    rating,  title,  body,
   created_at, verified_purchase)
VALUES
  ('AMZ_0001', 1, 'Amazon',
   'J. Rivera', 4, 'Great
```

```
value',          'Solid build, ships
fast.',          '2026-08-20 09:30:00',
TRUE),
  ('BB_0002',    1, 'BestBuy',
   'M. Chen',    5, 'Love it',
   'Battery lasts as advertised.',
   '2026-08-21 14:10:00', TRUE),
  ('WM_0003',    1, 'Walmart', 'P.
Okafor',        3, 'Okay',      'Does
the job, packaging was thin.',
   '2026-08-22 08:00:00', FALSE),
  ('AMZ_0010',   2, 'Amazon',
   'S. Diallo',  2,
   'Underwhelming', 'Smaller than
expected.',        '2026-08-23
11:45:00', TRUE),
  ('WM_0020',    3, 'Walmart',
   'L. Fontaine', 4, 'Good for the
price', 'Works well, feels solid.',
   '2026-08-24 16:20:00', TRUE);
```

The column list is the **8 review fields plus `product_id`** — exactly the table shape from the DDL above. These rows are *illustrative of the shape*: port the actual values from your `backend/src/data/reviews.js`, keeping each entry's string `review_id` (e.g. `AMZ_0001`) and using the `Map` key it lived under as the `product_id`.

5 Swapping the store for SQL

This is the whole session in one move: **replace the body of each in-memory helper with a `pool.query` call**. The function names, the returned shape, and the routes do not change — only *where* the data comes from.

`getProductReviews` — **before (S4, in-memory) vs after (S5, MySQL)**. Same name, same returned shape (an **array of 8-field objects**); only the data source changes:

```
// S4 – in-memory (what the store
gave you)
function
getProductReviews(productId) {
  return reviewsByProduct.get(String(productId));
}

// S5 – MySQL (the ONLY thing that
changed is the body + `async`)
async function
getProductReviews(productId) {
  const [rows] = await pool.query(
    'SELECT review_id, source, author,
rating, title, body, created_at,
verified_purchase FROM reviews WHERE
product_id = ?',
    [productId]
  );
  return rows;
}
```

Two things to notice: (1) the function is now **async**, because `pool.query` returns a promise; (2) the **SELECT** returns the **8 fields** — `product_id` is in the **WHERE clause** (the filter), *not* in the **SELECT list** (the shape). That is the join detail staying out of the data shape.

aggregateProductReviews — becomes async too. It fetches the rows with the **same SELECT**, then runs the **identical** histogram / `per_source` / `overall_avg` / `count` logic over them — the exact S4 loop, now DB-backed:

```
async function
aggregateProductReviews(productId) {
  const [rows] = await pool.query(
    'SELECT review_id, source, author,
rating, title, body, created_at,
verified_purchase FROM reviews WHERE
product_id = ?',
    [productId]
  );
  return aggregateProductReviews(rows);
}
```

```
);
const reviews = rows;
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,
};
}
```

The **empty case is preserved**: when a product has no rows, `reviews.length === 0` short-circuits to `overall_avg: null`, `per_source: {}` (empty), `histogram` all zeros, and `count: 0` — exactly the S4 empty result, now sourced from an empty **SELECT**.

The route handler — a one-line change.

Because the helpers are now `async`, the call site gains an `await`. That is the *only* diff in the handler — the 404-early check and the `res.json` envelope are untouched:

```
// GET /api/products/:id/reviews —
only the marked line changed (an added
`await`)
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 = await getProductReviews(req.params.id);
    ← the only change: added `await`
    res.json({ data, meta: { count:
data.length } });
  } catch (err) {
    next(err);
  }
});
```

(The `/:id/aggregate` handler changes the same way: `res.json({ data: await aggregateProductReviews(req.params.id) })`)

i Note

Nothing on the frontend changes. The route paths (GET `/:id/reviews`, GET `/:id/aggregate`), the 404-early logic, and the `{ data, meta } / { data }` envelopes are **byte-identical** to Session 4. The React UI renders exactly the same JSON it always did — it has simply started arriving from MySQL instead of an in-memory Map.

6 The full capstone (React to Express to MySQL)

With the swap in place, the **18 h capstone is wired end-to-end**. Trace the path one request takes:

1. **React (S3)** — the review components (`ReviewList`, `ReviewStats`, `SourceBreakdown`) call GET `/api/products/:id/reviews` and GET `/api/products/:id/aggregate` through `VITE_BACKEND_URL`.
2. **Express (S4)** — the two review routes run; they 404-early on a missing product, then call the (now **DB-backed**) helpers with an `await`.
3. **mysql2 (S5)** — `pool.query` reads the reviews table with `WHERE product_id = ?` and returns the 8-field rows.
4. **Back to the browser** — the data flows out as the **same** `{ data, meta } / { data }` envelopes; the React UI renders **DB-backed reviews with NO frontend change**.

That is the whole story: **React → Express → MySQL → Express → React**, with the data shape unchanged at every boundary.

Verification checklist (run these before you call the capstone done):

```
# 1. The list route — data now comes
from the reviews table
curl http://localhost:4000/api/
products/1/reviews
# → { data: [ ...8-field reviews... ],
meta: { count: N } } # N = the rows
seeded for product 1

# 2. The aggregate route — same
SELECT, DB-backed stats
curl http://localhost:4000/api/
products/1/aggregate
# → { data: { overall_avg,
```

```

per_source, histogram, count } }

# 3. Prove it is MySQL, not memory:
change a row, re-curl, see it move
# (WM_0003 is one of the example
seed rows above – substitute an id
from your own seed)
mysql -u appuser -p minimal_app_db \
  -e "UPDATE reviews SET rating = 5
WHERE review_id = 'WM_0003';"
curl http://localhost:4000/api/
products/1/reviews # rating for
WM_0003 is now 5

# 4. Confirm the UI renders it: open
the product page on :5173
# → the review list + histogram +
per-source table render the DB-backed
rows

```

If step 3 moves the value through the API and the UI, the data source is genuinely MySQL – the in-memory store is gone.

7 Key takeaways

- **A pool, not a single connection:** the app reuses the `mysql2` **connection pool** (`backend/src/db.js`); each request acquires, queries, and releases a pooled connection.
- **The `reviews` table = the 8 fields + a `product_id` join column (FK):** the table stores the canonical 8-field review **plus** `product_id` (FK → `products.id`) so the routes can filter.
- **`product_id` is a join detail that is dropped from the `SELECT`:** it lives in the `WHERE` clause, never in the returned 8-field shape.
- **The swap changes only the helper bodies (and adds `await`), not the routes or the frontend:** `getProductReviews` / `aggregateProductReviews` get a

`pool.query` body; the route handlers gain a single `await`.

- **The `{ data, meta } / { data }` envelopes are the stable contract:** lists return `{ data, meta }`, the aggregate returns `{ data }` – unchanged from S4.
- **Seeding makes the capstone persistent:** the same reviews now live in MySQL, so they survive a restart.
- **The AI loop is the method:** generate → **verify** → fix → commit; the verify step (curl both routes, confirm the data is DB-backed) is where correctness lives.
- **Agents are an optional accelerator, not a replacement:** from Session 3 onward, autonomous AI agents can speed up generation, but **you** remain the quality gate – you verify the SQL, the schema, and the rendered output before committing.

Make “**the `await` is the only route change**” a live moment: show the S4 handler next to the S5 handler and point at the single added `await` on `getProductReviews(req.params.id)`. Then prove the **frontend is agnostic to the data source** – run the UI against the in-memory store (S4) and against MySQL (S5) and show the identical review grid and histogram, so students see the data source is an implementation detail hidden behind a stable envelope. Close by previewing the home stretch – wiring the **`scraper-service/ shipped in fullstack-minimal-app`** (the standalone mock review source on `:5000`) into a **`POST / api/products/:id/fetch`** route that populates `reviews` on demand, turning the seeded capstone into a true on-demand aggregator.

8 Home stretch (1 month)

The 18 h capstone is now complete and **persistent**: the `reviews` table, the review routes, and the React review UI are wired

end-to-end, and the data survives a restart. The home stretch adds the piece that was deliberately left unwired in-book — **on-demand backfill**, using the **scraper-service/shipped in fullstack-minimal-app**. That folder is a small standalone **mock** review source: it listens on **http://localhost:5000** and answers **GET http://localhost:5000/api/scrape/reviews/:productId** with generated reviews for the product id. It is **not** part of the Express app — the app core is catalog-only plus this separate service — and nothing calls it yet; you run it on its own.

The app-side piece you build is **POST /api/products/:id/fetch**: it calls the scraper-service for that product, **normalizes** each returned review into the canonical 8-field schema (string **review_id** such as **AMZ_0001**, **source** one of **Amazon | BestBuy | Walmart**, **author**, integer **rating** 1–5, **title**, **body**, **created_at**, **verified_purchase**), inserts the rows into the **reviews** table with the URL’s **product_id**, and returns a clear result — how many reviews were fetched and stored. Because the mock regenerates its reviews on every call, the fetch handler needs a collision strategy for the **review_id** primary key: either generate fresh unique string ids per fetch, or use **INSERT IGNORE / ON DUPLICATE KEY UPDATE**.

Once it is wired, the “live” story begins: instead of only seeding, the app fetches on demand, normalizes, and persists into MySQL — and the same two GET routes serve the new rows with **no frontend change**.