

Database Integration — Practice

Make the review routes persistent with MySQL (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
1.1 The AI-first pattern (CRISP)	2
2 Setup & running the app	2
3 Task 1 — create the reviews table	3
3.1 AI generates	3
3.2 You verify	4
3.3 You integrate	4
4 Task 2 — seed the reviews	5
4.1 AI generates	5
4.2 You verify	5
4.3 You integrate	6
5 Task 3 — swap getProductReviews to pool.query	6
5.1 AI generates	6
5.2 You verify	6
5.3 You integrate	7
6 Task 4 — swap aggregateProductReviews to pool.query	7
6.1 AI generates	7
6.2 You verify	8
6.3 You integrate	8
7 Task 5 (optional) — confirm the full capstone in the browser	8

7.1 You do	8
7.2 You integrate	8
8 Common AI mistakes	8
9 Key takeaways	10
10 Home stretch (beyond the 18 h)	10

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.

1 Overview

Where you are. In **Session 4** (Node.js & Express practice) you built the two review routes on an **in-memory** store. Read the Session 5 lecture first — today you make it **persistent**: create and seed a MySQL `reviews` table, then swap the in-memory helpers for `pool.query`,

and confirm the full capstone (React → Express → MySQL) renders DB-backed reviews with no frontend change.

Goal — (1) **create** the `reviews` table (8 fields + `product_id`); (2) **seed** it with the reviews; (3) **swap** `getProductReviews` to `pool.query`; (4) **swap** `aggregateProductReviews`; (5) confirm the **React UI renders DB-backed reviews** with no frontend change.

The table adds a `product_id` join column but the API still returns the **8-field** shape — the route handlers, paths, and `{ data, meta }` / `{ data }` envelopes are unchanged; only the helper bodies change (plus an `await`).

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.
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 a React/Vite frontend (on **:5173**, from S3) and an Express backend on **http://localhost:4000** — and a **MySQL 8.0** database on **:3306**. The backend already talks to MySQL through a connection **pool** in `backend/src/db.js` (ESM, `mysql2/promise`, `createPool`) — you **reuse that pool**, you do not create a new one. It uses the same defaults as the app (`localhost:3306`, user `appuser`, database `minimal_app_db` — see `backend/.env.example`).

1. Get the app (if you have not already) — run from a parent directory so you land in the repo root:

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

2. Start the database (the app’s MySQL service, exposed on **:3306**). In `docker-compose.yml` the MySQL service is named **mysql**; starting it runs the app’s migration/seed script `db/init.sql` on first start, which creates and seeds the `categories` and `products` tables (the catalog):

```
docker compose up -d mysql
```

3. Start the backend (so it listens on **:4000** and connects to the DB). Run from the repo root:

```
cd backend
npm install
npm run dev          # or `npm start`
– see the README
```

The backend reaches the database at `localhost:3306` (the compose file maps the

MySQL port to the host), which matches the pool's defaults in `src/db.js`.

4. Confirm the catalog works and the DB is reachable:

```
curl      http://localhost:4000/api/
products  # → { data:
[ ...products... ], meta: { ... } }
curl      http://localhost:4000/health
# → { status: "ok", ... }
```

- `curl http://localhost:4000/api/products` — returns the catalog in the `{ data, meta }` envelope. It reads the `products` table through the pool, so a working response also proves the DB is up.
- `curl http://localhost:4000/health` — the backend health endpoint.

The **reviews** table does NOT exist yet — the app's `db/init.sql` only creates `categories` and `products`. You create it in Task 1 and seed it in Task 2.

3 Task 1 — create the reviews table

Concept. The reviews are not persisted yet (in S4 they live in an in-memory `Map`). You persist them in **MySQL** by creating a **reviews table**: the **8 review fields** plus a **product_id join column** (a foreign key to `products.id`) so the review routes can filter with `WHERE product_id = ?`. `product_id` is a join detail — it lives in the table (and in the `WHERE` clause), but it is **not** part of the 8-field review shape the API returns.

3.1 AI generates

💡 Tip

Copy-paste this prompt to your assistant:

```
Create the `reviews` table in this
project using EXACTLY this DDL
(use it as-is):
```

```
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);
```

```
Put it in the project's migration/
seed location used by the app –
db/init.sql – appended AFTER the
existing products table and its
seed. Do not change the existing
```

```
categories/products DDL or seed,
and do
```

```
not add any extra column (no
vote/"useful" column; product_id
is the only column beyond the 8
fields).
```

3.2 You verify

The app applies its schema from `db/init.sql` on a **fresh** database volume, so after editing the file re-run the init to materialize the new table (this resets the `db_data` volume and re-seeds the catalog from `db/init.sql` – expected here while you are building the schema):

```
docker compose down -v && docker
compose up -d mysql
```

Then confirm the table exists with all 9 columns, the `product_id` foreign key, the index, and the 1–5 CHECK (the default password is `apppassword`, from `backend/.env.example`):

```
mysql -h localhost -u appuser -p
minimal_app_db -e "SHOW CREATE TABLE
reviews\G"
```

Checklist:

- `SHOW CREATE TABLE reviews;` lists **all 9 columns**: `review_id`, `product_id`, `source`, `author`, `rating`, `title`, `body`, `created_at`, `verified_purchase`.
- There is a **foreign key** `product_id` → `products(id)` with `ON DELETE CASCADE`.
- There is an **index** on `product_id` (`idx_reviews_product`).
- `rating` carries the **1–5 CHECK** (`rating BETWEEN 1 AND 5`).
- There is **no** extra column (no vote/"useful" column).

3.3 You integrate

Checkpoint. The `reviews` table exists in MySQL with the 9 columns, the `product_id` FK, the index, and the 1–5 CHECK. Lock it in:

```
git add db/init.sql && git commit -m  
"feat(db): reviews table (8 fields +  
product_id FK, 1-5 CHECK, index)"
```

4 Task 2 — seed the reviews

Concept. Load the **same reviews** the in-memory store held in S4, now as **rows** in the `reviews` table — the only difference is that each row also carries its `product_id` (the join column 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`.

4.1 AI generates



Tip

Copy-paste this prompt to your assistant:

```
Write the seed INSERTs for the  
`reviews` table. Each row has the  
8 review fields PLUS product_id:
```

```
INSERT INTO reviews  
  (review_id, product_id, source,  
   author,   rating,  title,  body,  
   created_at, verified_purchase)  
VALUES  
  (...);
```

```
Port the review rows from  
your S4 in-memory `backend/src/  
data/reviews.js` into the MySQL  
`reviews`  
table — read that file and carry  
over each review's values; the  
only new field is product_id (the  
product each review belongs to).  
rating must be an integer 1-5. Put  
it in the project's  
migration/seed location used by  
the app (db/init.sql), right after  
the reviews table DDL. Do not add  
any extra column (no vote/"useful"  
column).
```

4.2 You verify

Re-run the init so the running database picks up the new seed (resets the volume and re-seeds, as in Task 1):

```
docker compose down -v && docker  
compose up -d mysql
```

Then check the rows (default password `apppassword`):

```
mysql -h localhost -u appuser -p
minimal_app_db -e "SELECT COUNT(*) AS
total FROM reviews;"
mysql -h localhost -u appuser -
p minimal_app_db -e "SELECT * FROM
reviews WHERE product_id = 1;"
```

Checklist:

- `SELECT COUNT(*) FROM reviews;` returns the number of rows you ported from your S4 in-memory store (there is no fixed count — the 5 example rows shown in the lecture illustrate the row shape, not a required total).
- `SELECT * FROM reviews WHERE product_id = 1;` shows rows with the **8 review fields plus product_id**.
- Every rating is an **integer 1–5** (no 0, no 6, no floats).
- There is **no** vote/“useful” column in the output.

4.3 You integrate

Checkpoint. The `reviews` table holds every review ported from your S4 in-memory store (`backend/src/data/reviews.js`), each row carrying its `product_id`, and the rows persist in MySQL across restarts. Lock it in:

```
git add db/init.sql && git commit -
m "feat(db): seed reviews ported from
the S4 in-memory store"
```

5 Task 3 — swap getProductReviews to pool.query

Concept. The in-memory lookup becomes a **parameterized SQL `SELECT`** against the `reviews` table: the helper body is replaced by a `pool.query` call, the helper becomes **async**, and the route handler gains a single **await** — the path, the 404-early product check, and the `{ data, meta }` envelope are unchanged.

5.1 AI generates



Tip

Copy-paste this prompt to your assistant:

```
In
`backend/src/routes/products.js`
make `getProductReviews` `async`
and swap its in-memory body for
a `pool.query` call (the `pool` is
already imported in this file).
Run EXACTLY this query:
`pool.query('SELECT review_id,
source, author, rating,
title, body, created_at,
verified_purchase
FROM reviews WHERE product_id
= ?', [productId])` and return the
rows. The SELECT lists EXACTLY
these
8 fields and does NOT select
`product_id` (product_id is only
in the WHERE clause). Then update
the
`GET /:id/reviews` route handler
to `await getProductReviews(req.params.id)`.
Keep ESM. Do not change
the route path, the 404-early
product check, or the `{ data,
meta }` envelope.
```

5.2 You verify

With the backend running on `:4000`, run the two curls:

```
curl http://localhost:4000/api/
products/1/reviews
curl http://localhost:4000/api/
products/999/reviews
```

Checklist:

- The helper `getProductReviews` is now **async**.
- The `SELECT` lists **exactly the 8 fields** (`review_id`, `source`, `author`, `rating`, `title`, `body`, `created_at`, `verified_purchase`) — it does **not** select `product_id`.
- The `WHERE` is **parameterized**: `WHERE product_id = ?` with `[productId]` as the second argument — no string concatenation.
- The `GET /:id/reviews` handler now **awaits** the helper (`const data = await getProductReviews(req.params.id);`).
- `curl http://localhost:4000/api/products/1/reviews` returns the DB rows in the `{ data, meta }` envelope.
- `curl http://localhost:4000/api/products/999/reviews` is still **404 { error: 'Product not found' }**.

5.3 You integrate

Checkpoint. `getProductReviews` reads the reviews table through the pool (8 fields, `WHERE product_id = ?`), and the added `await` is the only route change. Lock it in:

```
git add src/routes/products.js &&
git commit -m "feat(backend):
getProductReviews -> pool.query (8
fields, WHERE product_id=?)"
```

6 Task 4 — swap aggregateProductReviews to pool.query

Concept. Compute the **same aggregate from the DB rows**: the helper body is replaced by a `pool.query` call, the helper becomes **async**, and the route handler gains a single **await** — the output shape is **identical to S4**, and the path, the 404-early product check, and the `{ data }` envelope are unchanged.

6.1 AI generates



Tip

Copy-paste this prompt to your assistant:

```
In
`backend/src/routes/products.js`
make `aggregateProductReviews`
`async` and swap its in-memory
body
for a `pool.query` call (the
`pool` is already imported in this
file). Query the database with a
parameterized `SELECT review_id,
source, author, rating,
title, body, created_at,
verified_purchase
FROM reviews WHERE product_id = ?`
(pass `[productId]` as the second
argument). Then build the SAME
`{ overall_avg, per_source,
histogram, count }` shape in JS
from those rows, mirroring the S4
in-memory aggregation exactly:
overall_avg is the mean rating
rounded to 2 decimals, per_source
is the
mean rating per source, histogram
counts the buckets 1..5 (all five
keys always present even when 0),
and count is the number of rows.
For the empty case (no rows)
return EXACTLY
`{ overall_avg: null, per_source:
{}, histogram: { 1: 0, 2: 0, 3: 0,
4: 0, 5: 0 }, count: 0 }`.
Then update the `GET /:id/
aggregate` route handler to `await aggregateProductReviews`
Keep ESM. Do not change the route
path, the 404-early product check,
or the `{ data }` envelope.
```

6.2 You verify

With the backend running on :4000, run the curl:

```
curl http://localhost:4000/api/products/1/aggregate
```

Checklist:

- The output shape is **unchanged vs S4**: { overall_avg, per_source, histogram, count }.
- A product with **no reviews** returns overall_avg: null, an **empty per_source**, a **zeroed histogram** (all five buckets 0), and count: 0.
- curl http://localhost:4000/api/products/1/aggregate returns the **same shape as S4** but **DB-backed**.

6.3 You integrate

Checkpoint. aggregateProductReviews computes the same S4 shape from DB rows, and the added await is the only route change. Lock it in:

```
git add src/routes/products.js &&
git commit -m "feat(backend):
aggregateProductReviews
->
pool.query (same shape, DB-backed)"
```

7 Task 5 (optional) — confirm the full capstone in the browser

Concept. Start the React frontend and open a product page: the **review list** and the **aggregate** (overall avg, per-source means, and the 1–5 histogram) now render from **MySQL** — with **no frontend change**. Nothing on the frontend changed because the two route paths (GET /api/products/:id/reviews and GET /api/products/:id/aggregate) and their { data,

meta } / { data } envelopes are **identical to S4** — only the data source moved from the in-memory Map to the MySQL reviews table. The frontend reads the same shapes it always read, so it renders DB-backed reviews with no line changed.

7.1 You do

1. Start the React frontend (it is on :5173 from S3). Run from the repo root:

```
cd frontend
npm run dev
```

2. Open a product page at http://localhost:5173.

3. Confirm the review list and the **aggregate** (overall avg, per-source means, and the 1–5 histogram) render from **MySQL** — with **no frontend change**.

7.2 You integrate

Checkpoint. The full capstone (React → Express → MySQL) renders DB-backed reviews and the aggregate with no frontend change. Lock it in:

```
git add -A && git commit -m
"chore: verify full capstone (React -
> Express -> MySQL) in the browser"
```

8 Common AI mistakes

Mistake	What to do instead
SQL string concatenation of product_id (... WHERE product_id = \${id})	Parameterize: WHERE [productId] as the values array of product_id (... WHERE product_id = \${id})
— SQL in-	

Mistake	What to do instead	Mistake	What to do instead
jection: a crafted <code>id</code> can break out of the query Forgetting <code>product_id</code> in the <code>WHERE</code> — returns every review for every product (no filter applied) <code>SELECT-ing product_id</code> into the response — breaks the 8-field review contract (adds a 9th field) Forgetting <code>async</code> on the helper / <code>await</code> in the handler — you get a Promise, not rows (<code>data</code> is a	Always <code>WHERE product_id</code> Select the 8 fields only; keep <code>product_id</code> in the <code>WHERE</code>, never in the <code>SELECT</code>. Make the helper <code>async</code> and <code>await</code> it in the route handler.	pending Promise) Returning a bare array instead of <code>{ data, meta }</code> on the list route — the frontend reads <code>data/ meta</code> and breaks (undefined) A rating outside 1–5 — corrupts the histogram (a value that is not a 1..5 bucket) Forgetting the empty case — <code>overall_avg</code> becomes <code>0/NaN</code> Omitting the index on <code>product_id</code> — full table	Keep the <code>{ data, meta }</code> envelope. Enforce the 1–5 CHECK and validate on write. Return <code>overall_avg: null + a zeroed histogram + count: 0</code>. Keep <code>idx_reviews_product</code>.

Mistake	What to do instead
<code>scan</code> per product page	

9 Key takeaways

- **Persistence** = swap the helper **bodies only** — `getProductReviews` and `aggregateProductReviews` become `pool.query` calls; the routes, paths, 404-early check, and envelopes are **unchanged**.
- **Always parameterize the WHERE** — `WHERE product_id = ?` with a values array; never concatenate the id into the SQL string.
- **The API contract is what keeps the frontend unchanged** — the two route paths and the `{ data, meta } / { data }` envelopes are identical to S4, so only the data source moved (memory → MySQL).
- **product_id is a join column, not a review field** — it lives in the table and in the `WHERE` clause, but it is **never** in the 8-field review shape the API returns.
- **Preserve the empty-case shape** — a product with no reviews returns `overall_avg: null`, a zeroed 1–5 histogram, and `count: 0` (never 0/NaN).
- **ESM + mysql2/promise throughout** — `export/import`, `async/await`, and the shared `pool` from `db.js`; no CommonJS.

Make the “**frontend unchanged**” point live: re-call the **same two endpoints** — `GET /api/products/1/reviews` and `GET /api/products/1/aggregate` — **before and after** the swap and show the responses are **identical in shape** (only the data source moved from memory to MySQL). Re-emphasise that **404-early still holds**: `GET /api/products/999/reviews` is still 404 `{ error: 'Product not found' }`, because the product check runs before the table is read. Preview the home

stretch in one line: beyond the 18 h, the **scraper-service/ shipped in the repo** (a standalone mock on `:5000`) can backfill the `reviews` table on demand through a **POST** `/api/products/:id/fetch` route — it is **not** wired into the 18 h.

10 Home stretch (beyond the 18 h)

Everything below is **beyond the 18 h** — none of it is part of the graded course. The natural next step is **on-demand backfill** using the **scraper-service/ shipped in fullstack-minimal-app**: a small standalone **mock** review source that listens on `http://localhost:5000` and answers `GET http://localhost:5000/api/scrape/reviews/:productId` with generated reviews for that product id. The app core is catalog-only plus this separate service — nothing in the running app calls it yet — so start it on its own (see the README inside `scraper-service/` for its run command). You do **not** install any external scraper; this is the service the repo already ships.

Step 1 — probe the service. With the scraper-service running:

```
curl http://localhost:5000/api/scrape/reviews/1
```

Inspect the JSON: note which fields each returned review carries and how they map onto your canonical 8-field schema (`review_id`, `source`, `author`, `rating`, `title`, `body`, `created_at`, `verified_purchase`). The service is a mock — each call regenerates its reviews.

Step 2 — build POST `/api/products/:id/fetch`.

💡 Tip

Copy-paste this prompt to your assistant:

```
In      `backend/src/routes/
products.js` add a `POST /:id/
fetch` route handler that
backfills the reviews
table from the local scraper-
service. Keep the existing ESM
style, the 404-early product
check, and
the already-imported `pool`. The
handler must:
```

```
1. call `fetch("http://localhost:
5000/api/scrape/reviews/" +
req.params.id)` and read the JSON;
2. normalize every returned review
into the canonical 8-field schema:
`review_id` as a string id
```

```
(VARCHAR-style, e.g. prefixed
by source like "AMZ_0001"),
`source` one of Amazon | BestBuy |
Walmart, `author`, integer
`rating` between 1 and 5, `title`,
`body`, `created_at`, and boolean
`verified_purchase`;
```

```
3. insert the normalized rows
parameterized, with the URL id as
`product_id`:
```

```
INSERT INTO reviews
(review_id, product_id, source,
author, rating, title, body,
created_at,
verified_purchase) VALUES
(?, ?, ?, ?, ?, ?, ?, ?, ?)
```

```
Because repeated fetches can
collide on the `review_id` primary
key, either generate fresh unique
string ids per fetch or use
INSERT IGNORE / ON DUPLICATE KEY
UPDATE;
```

```
4. respond with res.json({ data:
{ product_id, fetched,
inserted } }).
```

Do not change the two GET routes or their { data, meta } / { data } envelopes.

Step 3 — verify.

```
curl -X POST http://localhost:4000/
api/products/1/fetch # → { data:
{ product_id, fetched, inserted } }
curl http://localhost:4000/api/
products/1/reviews # the
fetched rows are now served
curl http://localhost:4000/api/
products/1/aggregate # stats
include the fetched rows
```

Checklist:

- The scraper-service answers on :5000 at GET /api/scrape/reviews/1 — and you installed nothing external; it is the folder shipped in the repo.
- POST /api/products/1/fetch returns a clear result with fetched / inserted counts, and a second fetch does **not** crash on a duplicate review_id primary key.
- Every stored row matches the canonical schema: string review_id, source in Amazon | BestBuy | Walmart, integer rating 1–5, plus author, title, body, created_at, verified_purchase.
- GET /api/products/:id/reviews and GET /api/products/:id/aggregate serve the new rows with the **same envelopes** — no frontend change.

Once this is in place, the “live” story begins: fresh reviews flow from the shipped mock source into MySQL on demand, and the two GET routes keep serving them unchanged.

A few more optional extensions, each self-contained:

- **Pagination** on GET /api/products/:id/reviews — add ?page=?limit= (or ?cursor=) and reflect it in meta (e.g. meta: { count, page, limit }), keeping the { data, meta } envelope.

- **POST /api/products/:id/reviews** — add a review (validate `rating` is an integer 1–5, keep the 8-field shape, return the created row), still 404-early on the product.
- **A short-lived in-memory cache**
 - cache the `getProductReviews/aggregateProductReviews` results for a few seconds to cut repeated DB hits, invalidating on write.