

React Review UI — Practice

Convert the vanilla-JS review list into sample-data-fed React review components, ready for the backend (Session 3)

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	2	6 Task 4 — ReviewForm (controlled form, canonical payload)	11
1.1 The working loop (the rhythm of every task)	2	6.1 AI generates	12
2 Setup & running the app	3	6.2 You verify — common AI mistakes for this task	13
3 Task 1 — ReviewCard (the presentational unit)	5	6.3 You integrate	13
3.1 Generate	6	7 Assemble the reviews section	13
3.2 Verify — common mistakes for this task	7	8 Self-check	14
3.3 Integrate	7	9 Home stretch — the live scraper (next step, not in-book)	15
4 Task 2 — <code>useProductReviews</code> hook + ReviewList (list + per-source filter)	8	10 Building toward Session 4	15
4.1 Generate	8		
4.2 Verify — common mistakes for this task	9		
4.3 Integrate	9		
5 Task 3 — ReviewStats (per-source averages + 1–5 histogram)	10		
5.1 Generate	10		
5.2 You verify — common AI mistakes for this task	11		
5.3 You integrate	11		

i Note

Using a coding agent in this session is **optional** — write the components yourself or generate them. What is not optional is the loop: **generate** → **verify** → **integrate** → **commit**. The skill you are practising is **verifying the code** (yours or the AI's) against the checklists below. You are the quality gate. Every AI prompt in this practice is a **starting point** — feel free to reword it to match your style.

1 Overview

Where you are. In **Session 2** (JavaScript & async practice) you built a vanilla-JS review list: a `fetch` of a local `reviews.json`, `async/await`, and DOM you built by hand. Today you **convert that list into React components**. The review data comes from a **sample module you create** (`frontend/src/data/reviews.js`); the same components are fed by the backend review routes from Session 4. Read the Session 3 lecture first — it covers the concepts (components, props, state, `useEffect`, React Router 7.9); this practice is where you build and verify them.

Goal — build the React **reviews section** of the Multi-Source Review Aggregator, on top of the base `fullstack-minimal-app` catalog. The shipped app is **catalog-only**: it has no review UI, no review components, no review routes, and no reviews table — every file below is **created by you** in this session:

1. **ReviewCard** — a presentational component that renders one review (your S1 card was static HTML/CSS; this is its React counterpart).

2. **ReviewList** — maps reviews to `ReviewCard`, with a **per-source filter** and loading/empty states.
3. **ReviewStats** — the **1–5 histogram** and **per-source averages** from the aggregate.
4. **ReviewForm** — a **controlled form** that composes a review and builds the canonical payload.

All of it sits behind a single hook (`useProductReviews`). In S3 the hook resolves **sample reviews**; the components read the `{ data, meta }` contract that the backend on `:4000` already uses for the catalog and that the S4 review routes will reuse, and use the **8-field review schema**. In-book, reviews only become backend-served in **S4** (in-memory routes) and **S5** (seeded MySQL `reviews` table) — there is **no** live scraping in this session (that is the home stretch at the end).

1.1 The working loop (the rhythm of every task)

Every task below follows the same four steps:

Step	What you do
Concept	1–3 lines on <i>what</i> you are building and <i>why</i> it matters.
Generate	Write the component yourself, or — if you use an agent — hand it the CRISP copy-paste prompt in the box.
Verify	Run it, then check it against the <i>common mistakes for this task</i> list. Fix what fails.
Integrate & commit	Confirm the checkpoint (what “done” looks like) and make the git commit .

💡 Tip

If generated code fails any item on a checklist, **do not just copy it** — tell the AI *which* item failed and ask it to fix only that (or fix it yourself). 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 (dev server on <http://localhost:5173>) and an Express backend on <http://localhost:4000>. The frontend calls the backend through the `VITE_BACKEND_URL` environment variable (default <http://localhost:4000>), and every list response is shaped `{ data: [...], meta: {...} }`.

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`). Check the repo's README for the exact script — the typical commands are:

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

3. Start the frontend dev server (Vite, on `:5173`):

```
cd fullstack-minimal-app/frontend
npm install
npm run dev          # → http://
localhost:5173
```

4. Verify the backend, and be clear about which routes exist. The **catalog** API answers today — confirm it:

```
curl      http://localhost:4000/api/
products      # shipped: { data:
[...], meta: {...} }
```

The review routes `GET /api/products/:id/reviews` and `GET /api/products/:id/aggregate` **do not answer yet** — you build them on the backend in **Session 4**. In this session the review data comes from a **sample module you create now**, so the components can be finished before the backend exists.

5. Create the sample data module — `frontend/src/data/reviews.js` is **your file**, not part of the shipped repo. Group reviews by product id, and give each review the exact **8 fields** (see the RECALL box below), with string ids like `"AMZ_0001"`:

```
// frontend/src/data/reviews.js — you
create this file in S3
export const SAMPLE_REVIEWS = {
  "1": [
    {
      review_id: "AMZ_0001", source:
      "Amazon", author: "j.doe", rating: 5,
      title: "Great sound", body:
      "Bass is strong and the battery lasts
      all day.",
      created_at: "2026-09-01T10:00:00Z",
      verified_purchase: true,
    },
    // ... 2-3 more for product "1"
    (mix the three sources, mix verified
    true/false)
  ],
  "2": [ /* ... reviews for product "2"
... */ ],
```

```
// ... a few more products ...
};
```

Write **6–10 reviews** across the three sources with varied ratings **1–5** — Task 2’s hook and Task 3’s stats panel will read this module. **Note the exact field names** — the tasks below depend on them.

⚠ Warning

Tailwind in this app is **build-time** (PostCSS): it comes from `tailwind.config.cjs` + `postcss.config.cjs` + the `@tailwind` directives in `src/styles.css`. **Do not** add a Tailwind CDN `<script>` tag to `index.html` — it is a prototyping shortcut and is wrong for a build.

RECALL / if needed — the canonical review shape (the only one in this course) A **review** has exactly these **8 fields** — no more, no less:

Field	Type	Notes
<code>review_id</code>	string	e.g. "AMZ_0001" — use it as the React key
<code>source</code>	string	one of Amazon, BestBuy, Walmart
<code>author</code>	string	the reviewer’s name
<code>rating</code>	integer	1–5 (there is no 0)
<code>title</code>	string	short head-line

Field	Type	Notes
<code>body</code>	string	the review text (the field is <code>body</code> , not <code>content</code>)
<code>created_at</code>	ISO string	timestamp
<code>verified_purchase</code>	boolean	whether the reviewer bought it

Two things to hold onto: the schema is **closed at these 8 fields** (there is **no** vote counter or “was this useful?” flag), and a review is tied to its product **by the route path** (`/api/products/:id/reviews`) at the API level — `product_id` is a **database foreign-key concept** (Session 5), **not** a field your React props need. In S3 the tie is simply the key of `SAMPLE_REVIEWS`.

RECALL / if needed — the aggregate shape (GET `/api/products/:id/aggregate`) The aggregate — computed by your hook from the sample reviews in S3, served by GET `/api/products/:id/aggregate` in S4 — is one object summarising all reviews for a product:

```
{
  "overall": { "average": 4.2,
    "count": 41 },
  "by_source": [
    { "source": "Amazon", "average": 4.4, "count": 20 },
    { "source": "BestBuy", "average": 3.9, "count": 11 },
    { "source": "Walmart", "average": 4.1, "count": 10 }
  ],
  "histogram": { "1": 1, "2": 3, "3": 6, "4": 14, "5": 17 }
}
```

- `overall.average / overall.count` — the headline number and total.
- `by_source` — one entry per source with its own average and count.
- `histogram` — how many reviews at each star, **1 through 5**.

(Treat the exact key names above as the contract to build against. In S3 your hook computes an aggregate with **exactly this shape** from the sample reviews; in S4 the endpoint serves the same shape — `ReviewStats` is built against the *shape*, never the transport.)

RECALL / if needed — `useEffect` + `fetch` in a React component Fetching happens **inside `useEffect`**, never in the render body, and you **always return a cleanup**. This is the exact shape Task 2’s hook uses — in S3 the “load” resolves the **sample module** instead of a `fetch`; in S4 the sample import becomes this call against `:4000`:

```
useEffect(() => {
  let cancelled = false;
  (async () => {
    const res = await
    fetch(`${API_BASE}/api/products/
    ${id}/reviews`); // S4 onward
    if (!res.ok) throw new Error('HTTP
    ' + res.status);
    const json = await res.json();
    if (!cancelled)
    setReviews(json.data); // { data,
    meta } envelope
  })();
  return () => { cancelled =
  true; }; // never set state
  after unmount
}, [id]);
```

RECALL / if needed — React Router params & navigation

- On the route `/products/:id`, `const { id } = useParams()` gives you the product id (a **string**).
- Navigate with `<Link to={...}>` (a plain `<a>` reloads the page).
- After a successful action, redirect with `useNavigate()`.

3 Task 1 — ReviewCard (the presentational unit)

Concept. A `ReviewCard` renders **one** review from its `review` prop — no fetching, no state. It is the atomic piece that `ReviewList` will reuse. Your S1 card was **static HTML + CSS**; this is its React counterpart, and it is a **new file you create now** — the shipped repo contains no review components. Generate it with the prompt below (or write it yourself), and **lock the presentational component before you wire the hook**.

3.1 Generate

💡 Tip

Prompt (starting point — copy-paste it to an agent, or use it as your own spec):

```
Create ONE file, frontend/
src/components/ReviewCard.jsx —
a presentational React 19
component.
```

It takes a single `review` prop with EXACTLY these 8 fields:

```
review_id (string, e.g.
"AMZ_0001"), source ("Amazon" |
"BestBuy" | "Walmart"),
author (string), rating
(integer 1-5), title (string),
body (string),
created_at (ISO string),
verified_purchase (boolean).
```

Render, top to bottom:

- a source badge (Amazon=orange, BestBuy=blue, Walmart=green)
- a 5-star scale derived from `rating` (filled stars = rating, empty = 5 - rating)
- a "Verified purchase" badge ONLY when verified_purchase is true
- the review title (bold)
- the review body text
- author + created_at formatted as a locale date

Constraints:

- export default function ReviewCard({ review })
- one root element; className (not class); Tailwind (build-time) classes only
- NO state, NO hooks, NO fetch, NO extra fields (no vote counter, no "was this useful?" flag)

Output: only the contents of ReviewCard.jsx.

3.2 Verify — common mistakes for this task

- **Wrong field names.** It must use exactly the 8 canonical fields. If the review text is named `content` (instead of `body`), or the reviewer is named something other than `author`, fix it back.
- **Invents an extra field.** A vote counter or a “was this useful?” flag is **not** in the schema — the 8 fields are closed. Remove anything extra.
- **0–5 rating.** A “0 stars” state or a 0–5 scale is wrong; rating is an integer 1–5.
- **Missing the verified conditional.** The “Verified purchase” badge must render **only** when `verified_purchase` is true.
- **class= instead of className=** (or unclosed tags).
- **Reaches for state/fetch.** A presentational card has neither — it only reads its `review` prop.

3.3 Integrate

Checkpoint. Render the card with **one review from your frontend/src/data/reviews.js** (a temporary sample render in a scratch spot is fine here — you remove the scratch wiring in the assembly step). Confirm you see: the source badge, a correct star count, the title, the body, author + date, and the verified badge **only** on verified reviews. Then lock it in:

```
git add frontend/src/components/
ReviewCard.jsx
git commit -m "feat(frontend): lock
presentational ReviewCard to the 8-
field review schema"
```

4 Task 2 — `useProductReviews` hook + `ReviewList` (list + per-source filter)

4.1 Generate

Concept. This is the conversion of your S2 vanilla `renderReviews` into React. A small data hook exposes the reviews (and the aggregate, which Task 3 uses) to the components. **In S3 the hook resolves your sample module** (`frontend/src/data/reviews.js`); in S4 you replace that one lookup with a `fetch` to `:4000` — the components never change, because the hook’s public shape stays `{ reviews, aggregate, loading, error }`. `ReviewList` maps the array to `ReviewCard`, adds a **per-source filter** (`All / Amazon / BestBuy / Walmart`), and handles loading and empty states.

💡 Tip

Prompt (starting point — copy-paste it to an agent, or use it as your own spec):

```
Create TWO files for a React 19
frontend (Vite, Tailwind build-
time).
```

```
Data source in THIS session:
frontend/src/data/reviews.js
exports SAMPLE_REVIEWS,
an object keyed by product id
whose values are arrays of review
objects with 8
fields: review_id (string, e.g.
"AMZ_0001"), source ("Amazon" |
"BestBuy" |
"Walmart"), author, rating
(integer 1-5), title, body,
created_at (ISO string),
verified_purchase (boolean).
(From Session 4 the same hook will
instead call
import.meta.env.VITE_BACKEND_URL
(default http://localhost:4000):
GET /api/products/:id/reviews
returning the { data, meta }
envelope — read
json.data. Keep the hook's public
shape so that swap touches ONLY
the hook.)
```

```
1. frontend/src/
hooks/useProductReviews.js
- export function
useProductReviews(productId)
- state: reviews (array),
aggregate (object|null), loading
(bool), error (string|null)
- ONE useEffect keyed on
[productId] that, when productId
is present:
```

```
* resolves the
```

```
product's reviews asynchronously
from SAMPLE_REVIEWS (await a
Promise so the async
shape is real — do not mutate the
imported array)
```

```
* computes the aggregate
```

4.2 Verify — common mistakes for this task

- **Data access in the component.** The list must not import the sample module (later: call `fetch`) — the **hook** owns data access.
- **Wrong key later.** When S4 switches this hook to `fetch`, the reviews live in **json.data** (the { data, meta } envelope) — not `json`, `json.reviews`, or `json.items`. The hook is the single place that will know this.
- **No `useEffect` cleanup.** Missing `return () => { cancelled = true; }` → state set after unmount.
- **Async work in render.** Resolving data (or a bare `fetch`) in the component body — it must live in the hook's `useEffect`.
- **Mutates the source.** `reviews = reviews.filter(...)` reassigns the prop; derive a **new** array.
- **Aggregate math mutates or mis-scales.** Compute averages and the "1".."5" histogram from the reviews without sorting/mutating the source array; ratings are **1–5** (no 0 bucket).
- **Missing key** on the `.map` — use the stable `key={review.review_id}`.
- **Drops the "All" case.** Selecting "All" must show every review, not none.
- **Hardcoded URL (S4 preview).** When the `fetch` arrives, use ``${API_BASE}/...`` with `API_BASE = import.meta.env.VITE_BACKEND_URL`, not a literal `http://localhost:4000`.

4.3 Integrate

Checkpoint. In a scratch view (or the product detail view, Step "Assemble" below) render `<ReviewList>` using `useProductReviews(<some product id>)`. On load you should see a loading message, then the **sample** reviews from your data module; switching the source filter narrows the list live,

and a product id with no reviews shows the empty state.

5.1 Generate

```
git add frontend/src/hooks/
useProductReviews.js frontend/src/
components/ReviewList.jsx
git commit -m "feat(frontend): add
useProductReviews hook + ReviewList
with per-source filter"
```

5 Task 3 — ReviewStats (per-source averages + 1–5 histogram)

Concept. Your `useProductReviews` hook already computes an **aggregate** summary: overall average, per-source averages, and a **1–5 histogram**. In S4 the backend's `GET /api/products/:id/aggregate` endpoint will serve that same shape — so build `ReviewStats` against the **shape**, not the transport. The panel shows a headline average, one row per source, and CSS bars for the histogram. **No chart library** — plain Tailwind bars.

💡 Tip

Prompt (starting point — copy-paste it to an agent, or use it as your own spec):

```
Create ONE file, frontend/
src/components/ReviewStats.jsx —
a React 19 presentational
component.
```

Props: aggregate (object|null),
loading (bool). The aggregate
shape is:

```
{
  overall: { average, count },
  by_source: [ { source, average,
count } ],
  histogram: { "1": n, "2": n,
"3": n, "4": n, "5": n }
}
```

Render:

- a loading skeleton/message when loading is true
- a "no data" message when aggregate is null
- a large overall average (e.g. "4.2") with the total count
- a per-source list: one row per by_source entry (source name, its average, its count)
- a 1-5 histogram: five horizontal CSS bars (one per star, 1..5) whose width is proportional to histogram[n] / the max bar count; label each row with the star and the count

Constraints: NO external chart library (use divs + Tailwind widths only); className (not class);

Tailwind (build-time) classes only; no state, no fetch (it reads the `aggregate` prop); a stable key on the by_source map.

Output: only the contents of ReviewStats.jsx.

5.2 You verify — common AI mistakes for this task

- **Chart library.** A charting dependency is **not** wanted — the histogram is plain Tailwind `div` bars.
- **0–5 histogram.** The bars run **1 through 5** (there is no 0).
- **Divide by zero.** The bar width is `count / maxCount`; guard against a max of 0 (no reviews) so you do not get NaN.
- **Wrong field names.** Use `overall.average`, `by_source[].average`, and `histogram` keys "1".."5" — match the shape above.
- **class= instead of className=.**

5.3 You integrate

Checkpoint.

Render `<ReviewStats aggregate={aggregate} loading={loading} />` using the same `useProductReviews` hook. The panel shows the overall average, one row per source, and five histogram bars whose lengths reflect the sample-data distribution computed by the hook.

```
git add frontend/src/components/
ReviewStats.jsx
git commit -m "feat(frontend): add
ReviewStats (per-source averages +
1-5 histogram)"
```

6 Task 4 — ReviewForm (controlled form, canonical payload)

Concept. A **controlled form** composes a new review: `author`, `rating` (an integer 1–5), `title`, and `body`. React is the single source of truth — each input's `value` is bound to state and updated in `onChange`. In-book the reviews are **seeded**, so there is **no submission POST yet**: on submit the form **validates** and

previews the canonical payload. Wiring the real persistence route is Session 4/5 backend work (and live scraping is the home stretch). The point of this task is the **controlled-form + validation** pattern, not a working endpoint.

6.1 AI generates

💡 Tip

Copy-paste this prompt to your assistant:

```
Create ONE file, frontend/src/
components/ReviewForm.jsx – a
React 19 controlled form.
```

```
Fields (all bound to useState):
- author (text input, required)
- rating (<select> of integers
1..5 only – NO 0, required)
- title (text input)
- body (textarea)
```

```
On submit:
- e.preventDefault()
- validate: author non-empty;
rating is an integer 1..5; if
invalid, show an inline message
- build the payload object with
EXACTLY these keys:
    { author, rating:
Number(rating), title, body }
(do NOT add review_id, source,
created_at, verified_purchase, or
any vote/"useful" field –
those are server-side /
seeded)
- because there is no in-book
submission endpoint, on a valid
submit do NOT call fetch:
    console.log('review payload',
payload) and show a short on-
screen confirmation that the
payload is ready
```

```
Constraints: controlled inputs
(value + onChange); className (not
class); Tailwind (build-time)
classes only; a submit button; no
extra fields.
```

```
Output: only the contents of
ReviewForm.jsx.
```

6.2 You verify — common AI mistakes for this task

- **Uncontrolled inputs.** An input with no `value` (or `value` not tied to state) — typing does nothing or warns. Each input needs `value={state} + onChange`.
- **Missing `e.preventDefault()`.** Without it the page reloads on submit.
- **Rating allows 0 or non-integers.** The `<select>` must offer **1–5 only**, and the payload must `Number()` it.
- **Invents fields in the payload.** The payload is `{ author, rating, title, body }` — no `review_id`, no `source`, no `created_at`, no `verified_purchase`, no `vote`/"useful" field.
- **Calls `fetch` on submit.** There is no in-book submission endpoint — the form previews the payload, it does not post it.

6.3 You integrate

Checkpoint. In the form, type an author + a rating + a title + a body and submit: you see the inline validation, and on success a confirmation plus the payload in the console with **exactly** the four keys `{ author, rating, title, body }`. Submitting with an empty author shows the validation message and posts nothing.

```
git add frontend/src/components/
ReviewForm.jsx
git commit -m "feat(frontend): add
controlled ReviewForm (validates +
builds canonical payload)"
```

7 Assemble the reviews section

Bring the four pieces together in a reviews section **you add** to the app. The shipped product-detail view contains **no review markup** — the reviews part is entirely your creation.

1. Mount the pieces either by extending `frontend/src/components/ProductDetail.jsx` with a reviews block, or behind a new `ReviewsPage` on a `/products/:id/reviews-style` route (see the lecture's Router section) — your choice. Everything below is fed by one `useProductReviews(id)` call:

```
import { useParams } from 'react-router-dom';
import { useProductReviews } from '../hooks/useProductReviews';
import ReviewStats from './ReviewStats';
import ReviewList from './ReviewList';
import ReviewForm from './ReviewForm';

// inside ProductDetail (or your ReviewsPage / ReviewsSection):
const { id } = useParams();
const { reviews, aggregate, loading, error } = useProductReviews(id);
// ...
<ReviewStats
  aggregate={aggregate}
  loading={loading} />
<ReviewList reviews={reviews}
  loading={loading} />
<ReviewForm />
```

2. Remove the **scratch wiring** you added for the Task 1 checkpoint (the temporary single-card render — wherever you put it, for instance a temporary `<ReviewCard>` block in `App.jsx`). The shipped files never contained review markup; only your scratch harness needs to go.

Checkpoint. On a product page you see, top to bottom: the aggregate panel, the (filterable)

review list, and the form — all served by your sample module through `useProductReviews`, with loading/empty/error states behaving.

```
git add frontend/src/components/
ProductDetail.jsx frontend/src/
App.jsx
git commit -m "feat(frontend):
assemble reviews section (stats +
list + form) in ProductDetail"
```

8 Self-check

Run through this before you finish:

- ☐ Task 1: `ReviewCard` renders **exactly** the 8 canonical fields (source badge, 1–5 stars, verified badge, title, body, author + date) — no extra field, no vote/“useful” flag.
- ☐ Task 2: `useProductReviews` resolves reviews **and** aggregate from your sample module with a cleanup (`json.data` becomes the read path in S4, when the hook switches to `fetch`); `ReviewList` filters by source and shows loading/empty states.
- ☐ Task 3: `ReviewStats` shows the overall average, per-source averages, and a 1–5 histogram (CSS bars, no chart library).
- ☐ Task 4: `ReviewForm` is fully controlled, constrains rating to 1–5, validates, and builds the payload { author, rating, title, body } — it does **not** fetch on submit.
- ☐ Assembled: the reviews section renders in your product-detail extension (or `ReviewsPage`); the scratch wiring from the Task 1 checkpoint is removed.
- ☐ Every task was committed with its checkpoint message.
- ☐ No invented endpoints, no Tailwind CDN `<script>`, no 0–5 scale, no extra review fields.

9 Home stretch — the live scraper (next step, not in-book)

In-book, reviews come from your **sample module** behind `useProductReviews` in S3, from seeded in-memory `GET` routes in S4, and from a seeded MySQL table in S5. The **live** path — a `POST /api/products/:id/fetch` that asks the backend to scrape fresh reviews from the mock scraper on `:5000` and persist them — is the **next** step, built together with the backend in Session 4. Your UI is already shaped for it: when that route exists, the **hook** starts reading the backend and `ReviewList` / `ReviewStats` render the new reviews unchanged. You do **not** build or wire the scraper in this session.

10 Building toward Session 4

You just turned a vanilla-JS review list into a real React reviews section — sample-data-fed and ready for the backend:

- A **hand-built DOM list** → a declarative `ReviewList` that React re-renders on state change.
- A **fetch you triggered yourself** → the `useProductReviews` hook (`useEffect` + `cleanup`).
- A **form you read with `addEventListener`** → a controlled form where React is the source of truth.

Next, Session 4 — the Node 22 + Express 4.18 backend on `:4000` builds the routes your hook will start calling (`GET /api/products/:id/reviews`, `GET /api/products/:id/aggregate`) from the server side — and Session 5 the MySQL 8.0 database they read.