

React Frontend Development

Building the Multi-Source Review Aggregator UI in React

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	React Frontend Development	2		
1.1	📌 Today's Mission	2		
1.2	🛠️ AI-assisted work (optional — you are the quality gate)	3		
1.3	📍 Where you are & what is new	3		
1.4	📌 The real stack (<code>fullstack-minimal-app</code>)	4		
1.5	📌 RECALL / if needed	5		
2	Part 1: React fundamentals	6		
2.1	From Session 2's review list to React	6		
2.2	1.1 Components & JSX	7		
2.3	1.2 Props: making components reusable	8		
2.4	1.3 State with <code>useState</code>	9		
2.5	1.4 Events & controlled forms (what <code>ReviewForm</code> needs)	10		
3	Part 2: Wiring real data	10		
3.1	2.1 Data fetching: <code>useEffect</code> + <code>async/await</code> against <code>:4000</code>	10		
3.2	2.2 React Router 7.9: list / detail / form	12		
3.3	2.3 Tailwind 3.4, build-time (not a CDN)	13		
4	The review layer you build in S3	14		
4.1	Shipped vs. student-created (S3 map)	14		
4.2	The sample data module (you create it)	14		
4.3	The full <code>ReviewCard</code> (the version you aim for)	15		
4.4	<code>ReviewList</code> — map + filter + states .	16		
4.5	The review shape (exactly 8 fields) .	16		
4.6	How the reviews flow (S3 sample module now · backend review routes from S4)	17		
5	🛠️ AI-generate → You verify (the React/JSX beat)	18		
6	📄 Prompt templates (CRISP, reusable) ...	19		
7	📌 Key takeaways	20		

1 React Frontend Development

1.1 ☒ Today's Mission

! What we're building

The React frontend of our illustration project — the Multi-Source Product Review Aggregator — on top of the base Minimal Product Catalog full-stack app (`fullstack-minimal-app`).

- **Part 1 — React fundamentals:** components & JSX, props, state (`useState`), events & controlled forms — all taught on **review components** (`ReviewCard`, `ReviewList`).
- **Part 2 — Wiring data:** `useEffect` + `async/await` behind a data hook — **sample data in S3**, the backend on `:4000` (the `{ data, meta }` envelope) from S4 — plus **React Router 7.9**, **Tailwind 3.4** **build-time** (not a CDN).

The arc: we convert the **Session 2 vanilla-JS review list** (`fetch` + hand-built DOM) into **React review components**. In S3 they render **sample review data you create in the frontend**; from S4 the very same components are fed by the backend review routes.

Using: the product catalog of `fullstack-minimal-app` (React 19.1 · Vite 7.1 · Tailwind 3.4 · React Router 7.9) as the foundation. **The shipped app is catalog-only** — it does *not* ship review UI, review components, review routes, or a reviews table. Building the review layer is this session. **Philosophy:** AI-assisted work is **optional from S3 onward** — but whether you generate or type the code, **you are the quality gate**: `generate` → `verify` → `integrate` → `commit`.

1.2 🤖 AI-assisted work (optional — you are the quality gate)

i The loop you will run for every concept

You are the senior developer 🤖 · a **coding agent**, if you use one, is your **junior assistant** 🤖.

Using an AI agent is **optional** in this session. What is *not* optional is the loop — for every concept below:

1. **Concept** — the idea, stated plainly.
2. **Generate** — with an agent: hand it the **CRISP** copy-paste prompt; without one: write the component yourself.
3. **Verify** — a short “common AI mistakes” checklist, run on the code either way.
4. **Integrate** — a checkpoint in the browser, then a `git commit`.

`generate` → `verify` → `integrate` → `commit`. You are learning to **verify**, **not memorize**.

1.3 📍 Where you are & what is new

You are in **Session 3**. In S1 you built the **static review card** (semantic HTML + CSS, no JS) and covered the CRISP prompting workflow; in S2 you built a working **vanilla-JS review list** (`fetch` of a local `reviews.json` + `async/await` + manual DOM manipulation) and saw why it gets painful. Today you replace that hand-built DOM with **React review components**.

Central theme: a **multi-source product review aggregator**. **Base (shipped):** the **Minimal Product Catalog** (products + categories) — the foundation you build on. It ships the catalog UI and the catalog API; it ships **no review UI**, **no review components**, **no review routes**, **no reviews table**. **This session (S3):** build the **review layer in React** —

`ReviewCard`, `ReviewList`, and where appropriate `ReviewStats` / `ReviewForm` — rendering **sample review data you create** in the frontend (`frontend/src/data/reviews.js`, a file *you* author). **Capstone (S4 / S5)**: the same components are fed by the backend review routes you build in S4 (in-memory store) and persist in S5 (MySQL `reviews` table).

☒ **Covered before (S1 / S2) — do NOT re-learn:**

- HTML + CSS (S1)
- AI-first workflow & CRISP prompting (S1)
- ES6+ JS: `const/let`, arrow functions, objects & arrays, template literals (S2)
- `async/await`, `fetch`, checking `response.ok` (S2)

☒ **New here (S3):**

- React components & JSX
- Props, state (`useState`), events & controlled forms
- `useEffect` data-fetching behind a hook (sample data now, backend in S4)
- **React Router 7.9** (list / detail / form)
- **Tailwind 3.4 build-time** (not CDN)

✗ **Not covered (yet):** Node/Express backend internals (S4), MySQL & `mysql2` (S5).

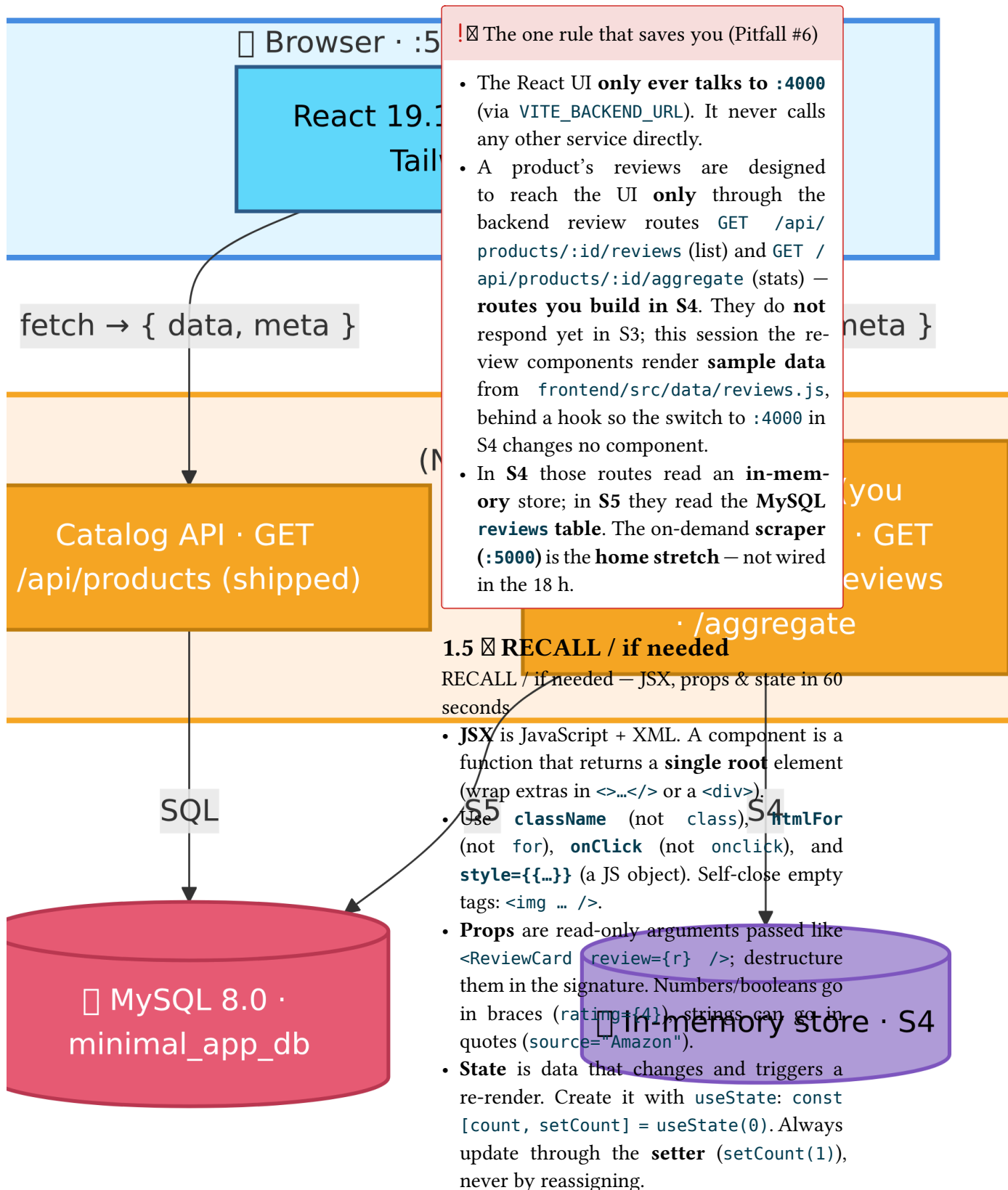
1.4 ☒ The real stack (**fullstack-minimal-app**)

This is the ground truth you will build against. **Do not substitute versions or endpoints.**

Layer	Technology	Runs on
Frontend	React 19.1 · Vite 7.1 · React Router 7.9	http://localhost: 5173
Styling	Tailwind 3.4	—

Layer	Technology	Runs on
	build-time (PostCSS), not a CDN	
Backend	Node 22 · Express 4.18 · <code>mysql2</code>	http://localhost: 4000
Database	MySQL 8.0 — <code>db_minimal_app_db</code> , user <code>appuser</code>	:3306
Mock re-view scraper	Express	:5000 — home stretch only (not wired in the 18 h)

The frontend reaches the backend through the environment variable `VITE_BACKEND_URL` (default `http://localhost:4000`).



- Embed any expression in {}: {items.map(i => <li key={i.id}>{i.name})}.

RECALL / if needed — React Router in 60 seconds

- **React Router 7.9** gives an SPA its URLs. Wrap the app in `<BrowserRouter>` and declare `<Route path="/" element={...} />` inside `<Routes>`.
- Navigate with `<Link to="/products/5">` (not `<a>` — a plain `<a>` reloads the page).
- Read URL params with `useParams()`: on the route `/products/:id`, `const { id } = useParams()` gives the id.
- Add `useNavigate()` for programmatic redirects (e.g. after a successful save).

2 Part 1: React fundamentals

2.1 From Session 2's review list to React

This is the conversion we are here to make. Same job — “get the reviews, show them” — two very different ways.

S2 style — vanilla: you fetch, then hand-build the DOM

```
async function loadReviews() {
  const res = await fetch('./reviews.json'); // S2: a local file
  if (!res.ok) throw new Error('HTTP ' + res.status);
  const reviews = await res.json();
  renderReviews(reviews);
}

function renderReviews(list) {
  const box =
document.querySelector('#review-list');

  box.innerHTML =
  ''; // you
  rebuild the DOM by hand
```

```
for (const r of list) {
  const card =
document.createElement('article');
  card.textContent = `${r.source} ·
  ${r.rating}★ — ${r.title}`;
  box.appendChild(card);
}
```

S3 style — React: declare what the UI is; React re-renders on state change

```
function ReviewSection({ productId })
{
  const [reviews, setReviews] =
useState([]);
  const [loading, setLoading] =
useState(true);

  useEffect(() => {
    let cancelled = false;
    (async () => {
      try {
        // S3: the hook resolves sample
        reviews; from S4 the same call
        // fetches GET /api/
        products/:id/reviews from the backend
        on :4000.

        const list = await
getReviewsForProduct(productId);
        if (!cancelled)
setReviews(list);
      } finally {
        if (!cancelled)
setLoading(false);
      }
    })();
    return () => { cancelled = true; };
  }, [productId]);

  return (
    <ul>
      {loading ? <li>Loading...</li> :
```

```

        reviews.map(r =>
<li key={r.review_id}>{r.source} ·
{r.rating}★ - {r.title}</li>)}
      </ul>
    );
  }

```

React is declarative: you describe *what the UI should look like* for a given state; React handles the DOM. No `innerHTML`, no `appendChild`.

2.2 1.1 Components & JSX

Concept. A component is a function returning JSX. Components are **reusable** and **composable** (you nest them). This is the pattern that keeps a growing UI maintainable — each feature is an isolated, testable piece.

```

function ReviewCard({ review }) {
  return (
    <article className="border
rounded-lg p-4 shadow-md">
      <span className="text-xs
font-semibold px-2 py-0.5 rounded bg-
orange-100 text-orange-700">
        {review.source}
      </span>
      {/* rating is an integer 1-5:
filled stars = rating, empty stars =
5 - rating */}
      <p className="text-amber-500">{'★'.re
{'★'.repeat(5 - review.rating)}</p>
      <h3 className="text-lg font-
semibold mt-2">{review.title}</h3>
      <p className="text-sm text-
gray-700">{review.body}</p>
      {review.verified_purchase && (
        <span className="text-xs text-
green-600">✓ Verified purchase</
span>
      )}
    </article>

```

```

    );
  }

```

AI generates (CRISP prompt — or write it yourself; the verify step is the same either way):

Context: React 19 frontend built with Vite, styled with Tailwind CSS (build-time).

Task: Create a presentational component `ReviewCard.jsx` that renders ONE review.

Props: a single `review` object with 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).

Constraints: single default-export function component; one root element; Tailwind

classes only (no CDN, no inline `<style>`); a 5-star scale derived from rating; the

"Verified purchase" badge renders ONLY when `verified_purchase` is true; no state, no

fetching; no extra fields (no vote counter, no "was this useful?" flag).

Output: complete `ReviewCard.jsx` file.

⚠ You verify — common AI mistakes

- ✗ **Multiple root elements** without a Fragment → wrap in `<>...</>`.
- ✗ `class=` instead of `className=` (also `for`→`htmlFor`, `onclick`→`onClick`).
- ✗ **Unclosed/self-unclosed tags** (`<img ...>` missing `/`).
- ✗ **Template literals in JSX markup** — `<p>${review.rating}</p>` renders the literal characters `$ { }`; the correct JSX interpolation is `<p>{review.rating}</p>`. Template literals belong in plain JS strings, not in JSX children.
- ✗ **Plain strings where a prop expects a value** — numbers/booleans go in braces: `rating={4}`, `verified_purchase={true}`.
- ☑ **One root**, correct JSX attribute names, props used as data.

You integrate. Render it in a scratch harness with one sample review, then commit:

```
git commit -m "feat(ui): add
presentational ReviewCard component
with props"
```

2.3 1.2 Props: making components reusable

Concept. Props are read-only arguments. Define a component **once**, use it with **different data**. Data flows **down** (parent → child); children never mutate props.

```
import ReviewCard from './
ReviewCard';
import { SAMPLE_REVIEWS } from '../
data/reviews'; // the sample module
YOU create in S3
```

```
function ReviewGrid() {
  const reviews =
SAMPLE_REVIEWS['1'] ?? []; //
sample reviews for product id 1
  return (
    <div className="grid grid-cols-1
md:grid-cols-3 gap-4">
      {reviews.map((r) => <ReviewCard
key={r.review_id} review={r} />)}
    </div>
  );
}
```

AI generates (CRISP prompt):

Context: React 19 + Tailwind (build-time). I already have `ReviewCard.jsx` and a sample data module `frontend/src/data/reviews.js` exporting `SAMPLE_REVIEWS` (an object keyed by product id, each value an array of 8-field review objects). Task: Create a `'ReviewGrid.jsx'` that renders a responsive 3-column grid of `ReviewCard` for one product's reviews. Constraints: pass each review via the `'review'` prop; do NOT modify the reviews (props and the imported array are read-only); use `key={review_id}`; Tailwind grid utilities only. Output: complete `ReviewGrid.jsx` file.

⚠ You verify — common AI mistakes

- ✗ **Treating props as state** (calling a setter on a prop, or mutating `review.rating`).
- ✗ **Hardcoding reviews** instead of threading them through props / the data module.
- ✗ **Missing or unstable key** on the `.map` — use `key={r.review_id}`.
- ☑ Each usage passes explicit props; the child only reads them.

You integrate. Confirm the grid renders several distinct review cards, then:

```
git commit -m "feat(ui): compose
ReviewCard into a responsive
ReviewGrid"
```

2.4 1.3 State with `useState`

Concept. State is data that **changes over time** and triggers a re-render. Create it with the `useState` hook. Each piece is independent and **local to the component**.

```
import { useState } from 'react';

function Counter() {
  const [count, setCount] =
    useState(0);
  return (
    <div className="text-center p-4">
      <h2 className="text-2xl
mb-4">Count: {count}</h2>
      <button
onClick={() => setCount(count - 1)}
className="px-4 py-2 bg-red-500 text-
white rounded">-1</button>
      <button
onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

```
className="px-4 py-2 bg-green-500
text-white rounded">+1</button>
    </div>
  );
}
```

Why not a plain variable? React has no way to know a regular variable changed, so it never re-renders. `useState` tells React “*this value matters — re-render when it changes.*”

AI generates (CRISP prompt):

Context: React 19 + Tailwind (build-time).

Task: Create `SourceFilter.jsx` that lets the user pick a review source and stores the choice in local state.

Props: `sources` (array of strings, e.g. `["All", "Amazon", "BestBuy", "Walmart"]`) and `onSelect` (callback).

Constraints: use `useState` for the selected value; render a `<select>` bound to that state (`value + onChange`); call `onSelect` with the new value; show the current selection as text; no fetching.

Output: complete `SourceFilter.jsx` file.

⚠ You verify — common AI mistakes

- ✗ **Reassigning state** (`selected = 'Amazon'`) instead of calling the **setter** (`setSelected('Amazon')`).
- ✗ **Forgetting import** `{ useState }` **from 'react'**.
- ☑ The setter is the only way state changes; the UI reflects it on re-render.

You integrate. Change the select, confirm the displayed selection updates, then:

```
git commit -m "feat(ui):
add SourceFilter (per-review-source
selection) with useState"
```

2.5 1.4 Events & controlled forms (what `ReviewForm` needs)

Concept. React handles events with **camel-Case** handlers (`onClick`, `onChange`, `onSubmit`). A **controlled input** binds `value` to state and updates that state in `onChange`, so React is the single source of truth for form data.

```
function ReviewSearch() {
  const [term, setTerm] =
useState('');
  const handleSubmit = (e) => {
    e.preventDefault(); //
stop the page reload
    console.log('Filter reviews by:',
term);
  };
  return (
    <form onSubmit={handleSubmit}
className="max-w-md mx-auto p-4">
      <input value={term} onChange={e
=> setTerm(e.target.value)}
className="border
rounded px-3 py-2" placeholder="Type
to filter reviews..." />
      <button type="submit"
className="bg-blue-600 text-white
px-4 py-2 rounded">Go</button>
    </form>
  );
}
```

Pattern: `value={state}` + `onChange={setState}` = a controlled input. You will reuse it in S3's practice for the

`ReviewForm` fields (`author`, `rating`, `title`, `body`).

AI generates (CRISP prompt):

Context: React 19 + Tailwind (build-time).
Task: Create a minimal controlled search form `ReviewSearch.jsx` for filtering reviews.
Constraints: one text input bound to a `useState` value; an `onSubmit` handler that calls `e.preventDefault()`; show the current term; Tailwind classes only.
Output: complete `ReviewSearch.jsx` file.

⚠ You verify — common AI mistakes

- **✗ Uncontrolled input** — no `value`, or `value` not tied to state (typing does nothing / warnings).
- **✗ Missing `e.preventDefault()`** on submit (the page reloads).
- **✗ `onclick=` / `onsubmit=`** (lowercase HTML) instead of `onClick` / `onSubmit`.

You integrate. Type and submit; confirm the term updates and the page does not reload, then:

```
git commit -m "feat(ui): add
controlled ReviewSearch form"
```

3 Part 2: Wiring real data

3.1 2.1 Data fetching: `useEffect` + `async/await` against `:4000`

Concept. `useEffect` runs side effects (like fetching) after render. It takes an effect function and a **dependency array**. Fetching

on mount = useEffect(..., []); fetching again when a value changes = useEffect(..., [value]). **Always return a cleanup** so you never set state on an unmounted component.

The envelope you must respect. The shipped catalog API does **not** return bare arrays — GET /api/products returns:

```
{
  "data": [ { "id": 1, "name": "...",
    "price": 9.9, "category_name": "...",
    "...": "..." } ],
  "meta": { "total": 42, "page": 1,
    "per_page": 6, "total_pages": 7 }
}
```

So you read **json.data** for the items and **json.meta** for pagination. GET /api/categories returns { "data": [category] }. **The review routes you build in S4 reuse this same envelope** — the fetch pattern below is exactly the one they will need.

AI generates (CRISP prompt):

```
Context: React 19 frontend. Reviews
have 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).
In this session, reviews come from
a sample module frontend/src/data/
reviews.js that
exports SAMPLE_REVIEWS, an object
keyed by product id with arrays of
review objects.
(From S4, the same hook will
instead call the backend on
import.meta.env.VITE_BACKEND_URL:
```

GET /api/products/:id/reviews, which returns the { data, meta } envelope — read json.data.)

Task: Create frontend/src/hooks/useProductReviews.js exporting useProductReviews(productId) that returns { reviews, loading, error }. Constraints: useState for reviews/loading/error; ONE useEffect keyed on [productId]

that resolves the reviews asynchronously (await a Promise so the async shape is real), with a cancelled flag and a cleanup function; never mutate the imported array; keep the public shape stable so components never change when the sample import becomes a fetch.

Output: complete useProductReviews.js file.

⚠ You verify — common AI mistakes

- **✗ Reading the wrong key later** — when S4 switches this hook to fetch, the reviews live in json.data, not json (the { data, meta } envelope).
- **✗ No cleanup** — missing return () => { cancelled = true; }.
- **✗ Async work in the render body** (outside useEffect).
- **✗ Hardcoded URL** — when the fetch arrives, it must be \${API_BASE}/... with API_BASE = import.meta.env.VITE_BACKEND_URL, never a literal http://localhost:4000.
- ☒ loading/error states are shown; the hook's public shape ({ reviews, loading, error }) matches what the components consume.

You integrate. Render `ReviewGrid` through the hook with a sample product id, confirm the cards appear (loading → list), then:

```
git commit -m "feat(ui): add
useProductReviews hook (sample data
now, :4000 in S4)"
```

3.2 2.2 React Router 7.9: list / detail / form

Concept. React Router 7.9 maps URLs to components. The shipped catalog already routes between its own views; your review layer plugs into the same router — for example a dedicated reviews view addressed by product id, reachable with `<Link>` and reading `:productId` via `useParams`.

```
import { BrowserRouter, Routes,
Route, Link, Navigate } from 'react-
router-dom';
import ProductList from './
components/ProductList'; // shipped
catalog view
import ReviewsPage from './
components/ReviewsPage'; // you
create this in S3

function App() {
  return (
    <BrowserRouter>
      <nav className="p-4 flex gap-4">
        <Link to="/">Catalog</Link>
        <Link to="/reviews/1">Reviews
(product 1)</Link>
      </nav>
      <Routes>
        <Route path="/"
element={<ProductList />} />
        <Route path="/
reviews/:productId"
element={<ReviewsPage />} />
        <Route path="*" />
```

```
element={<Navigate to="/" replace /
>} />
      </Routes>
    </BrowserRouter>
  );
}
export default App;
```

AI generates (CRISP prompt):

Context: React 19 with React Router 7.9 (react-router-dom). The catalog routes already exist in `App.jsx`. I have `ReviewGrid.jsx` and the `useProductReviews` hook.

Task: Create `ReviewsPage.jsx` — it reads `:productId` from `useParams()`, calls `useProductReviews(productId)`, and renders the review grid with loading/error states — then add the route `"/reviews/:productId" -> ReviewsPage` and a `<Link>` to `/reviews/1` in the nav, keeping the existing catalog routes untouched.

Constraints: `<Link>` (not `<a>`) for in-app navigation; `useParams` for the id; do not modify the shipped catalog routes; Tailwind for layout.

Output: complete `ReviewsPage.jsx` plus the modified `App.jsx`.

⚠ You verify — common AI mistakes

- ✗ `<a href>` for in-app navigation (causes a full page reload) — must be `<Link to>`.
- ✗ **Route path not matching the param** (e.g. `useParams().productId` but the route is `/reviews/:id`).
- ✗ **Missing the catch-all** → unknown URLs show a blank screen.
- ✗ **Overwriting the shipped catalog routes** instead of adding to them.
- ☑ Params line up; navigation is client-side; a fallback route exists.

You integrate. Click the nav link, confirm the URL changes and the reviews render without a page reload, then:

```
git commit -m "feat(ui): add /
reviews/:productId route with
ReviewsPage + Link"
```

3.3 2.3 Tailwind 3.4, build-time (not a CDN)

Concept. Tailwind is a set of **utility classes** (flex, p-4, text-2xl). In this project it runs at **build time** through **PostCSS**, not via a CDN `<script>` tag. Three files make it work:

File	Role
<code>tailwind.config.cjs</code>	Config — includes a content glob so Tailwind knows which files to scan
<code>postcss.config.cjs</code>	Wires Tailwind into the PostCSS build pipeline

File	Role
<code>src/styles.css</code>	The @tailwind directives that emit the CSS

```
/* src/styles.css — these three lines
are what make build-time Tailwind
work */
@tailwind base;
@tailwind components;
@tailwind utilities;
```

AI generates (CRISP prompt):

```
Context: Vite 7.1 + React project
using Tailwind CSS 3.4 at BUILD time.
Task: Create the build-time Tailwind
setup: tailwind.config.cjs with a
content glob covering ./index.html and ./src/
**/*.{js,jsx}; postcss.config.cjs
enabling the tailwindcss and autoprefixer plugins;
and src/styles.css with the three
@tailwind directives.
Constraints: build-time only — do NOT
add a CDN <script> to index.html.
Output: the three files' contents.
```

⚠ You verify — common AI mistakes

- ✗ **Tailwind via CDN** — a `<script src="https://cdn.tailwindcss.com">` in `index.html` (prototyping only; wrong for a build).
- ✗ **@tailwind directives missing** from `src/styles.css`, or the CSS file not imported by `main.jsx`.
- ✗ **content glob too narrow** → utilities get purged and styles vanish.
- ☑ `postcss.config.cjs` + `tailwind.config.cjs` + the three directives are all present.

You integrate. Confirm classes apply in the browser (no CDN tag in the built HTML), then:

```
git commit -m "chore(ui): configure Tailwind 3.4 at build time via PostCSS"
```

4 The review layer you build in S3

The shipped frontend gives you the **catalog** (list / detail / form views wired to `GET / api/products` on `:4000`). Everything review-related below is **created by you in this session** — it does not exist in the shipped repo. The catalog views are context, not the exercise.

4.1 Shipped vs. student-created (S3 map)

Path	Status	Role
frontend/ src/ main.jsx, App.jsx	shipped	entry point + router (you add a reviews route)

Path	Status	Role
catalog views (list / detail / form)	shipped	the product foundation — not today's exercise
frontend/ src/data/ reviews.js	you create (S3)	sample reviews, 8-field schema
frontend/ src/hooks/useProductReviews.js	you create (S3)	data hook: sample now, <code>:4000</code> in S4
frontend/ src/component/ ReviewCard.jsx	you create (S3)	presentational card for one review
frontend/ src/component/ ReviewList.jsx	you create (S3)	grid + per-source filter + loading/empty states
frontend/ src/component/ ReviewStats.jsx	you create (S3)	average + per-source + 1–5 histogram
frontend/ src/component/ ReviewForm.jsx	you create (S3)	controlled form building the canonical payload

4.2 The sample data module (you create it)

`frontend/src/data/reviews.js` is the seam that keeps S3 buildable **before the backend review routes exist** (S4) and the reviews table exists (S5). Group reviews by product id; use string `review_ids` of the form `"AMZ_0001"`:

```
// frontend/src/data/reviews.js – YOU
create this file in S3 (not in the
shipped repo)
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,
    },
    {
      review_id: "BBY_0002", source:
      "BestBuy", author: "a.martinez",
      rating: 4,
      title: "Solid, with a caveat",
      body: "Comfortable after a break-in
      period.",
      created_at: "2026-09-03T18:30:00Z",
      verified_purchase: true,
    },
    {
      review_id: "WMT_0003", source:
      "Walmart", author: "k.wong", rating:
      2,
      title: "Stopped pairing", body:
      "Worked for two weeks, then would not
      reconnect.",
      created_at: "2026-09-05T08:15:00Z",
      verified_purchase: false,
    },
  ],
  // "2": [ ... more sample reviews
  for other products ... ]
};
```

product_id on a review? No — not as a React prop. The review↔product link

is the **route path** (/api/products/:id/reviews) in S4 and a **database foreign key** in S5. Your S3 components receive *one product's reviews array* and never need a product_id field inside a review.

4.3 The full ReviewCard (the version you aim for)

```
// frontend/src/components/
ReviewCard.jsx – you create this in S3
export default function
ReviewCard({ review }) {
  const sourceColor = {
    Amazon: 'bg-orange-100 text-
orange-700',
    BestBuy: 'bg-blue-100 text-
blue-700',
    Walmart: 'bg-green-100 text-
green-700',
  }[review.source] ?? 'bg-gray-100
text-gray-700';

  // rating is an integer 1-5;
  displayed on a standard 5-star scale
  return (
    <div className="bg-white rounded-
lg shadow p-4">
      <div className="flex items-
center gap-2 mb-2">
        <span className={`text-
xs font-semibold px-2 py-0.5 rounded
${sourceColor}`}>{review.source}</
span>
        <span
          className="text-amber-500" aria-
label={`Rating ${review.rating} out
of 5`}>
          {'*'.repeat(review.rating)}
          {'☆'.repeat(5 - review.rating)}
        </span>
        {review.verified_purchase
      && <span className="text-xs text-
green-600">✓ Verified</span>}
```

```

    </div>
    <h4 className="font-
semibold">{review.title}</h4>
    <p className="text-sm text-
gray-700">{review.body}</p>
    <p className="text-xs text-
gray-400 mt-2">
    {review.author} · {new Date(review.cr
    </p>
  </div>
);
}

```

Note the two interpolation forms and when each is right: `className={`... ${sourceColor}`}` is a **plain-JS template literal inside braces** (allowed); `{review.title}` is **JSX expression interpolation** (the normal case). What is *never* right is a bare `${review.title}` sitting directly in JSX markup — that renders the literal text `${review.title}`.

4.4 ReviewList — map + filter + states

```

//      frontend/src/components/
ReviewList.jsx — you create this in S3
import { useState } from 'react';
import ReviewCard from './
ReviewCard';

const SOURCES = ['All', 'Amazon',
'BestBuy', 'Walmart'];

export default function
ReviewList({ reviews, loading }) {
  const [source, setSource] =
    useState('All');

  // derive a NEW array; never mutate
the reviews prop
  const visible = source ===
'All' ? reviews : reviews.filter((r)

```

```

=> r.source === source));

    if (loading) return <p
className="text-gray-500">Loading
reviews...</p>;
    if (reviews.length === 0)
      return <p className="text-
gray-500">No reviews available yet.</
p>;

    return (
      <div className="space-y-4">
        <div className="flex items-
center gap-3">
          <label>
            Source{' '}
            <select
value={source} onChange={(e) =>
setSource(e.target.value)}>
              {SOURCES.map((s) => <option
key={s} value={s}>{s}</option>)}
            </select>
          </label>
          <span className="text-sm
text-gray-500">{visible.length} of
{reviews.length} reviews</span>
        </div>
        <div className="grid grid-cols-1
md:grid-cols-2 gap-4">
          {visible.map((r) => <ReviewCard
key={r.review_id} review={r} />)}
        </div>
      </div>
    );
  }

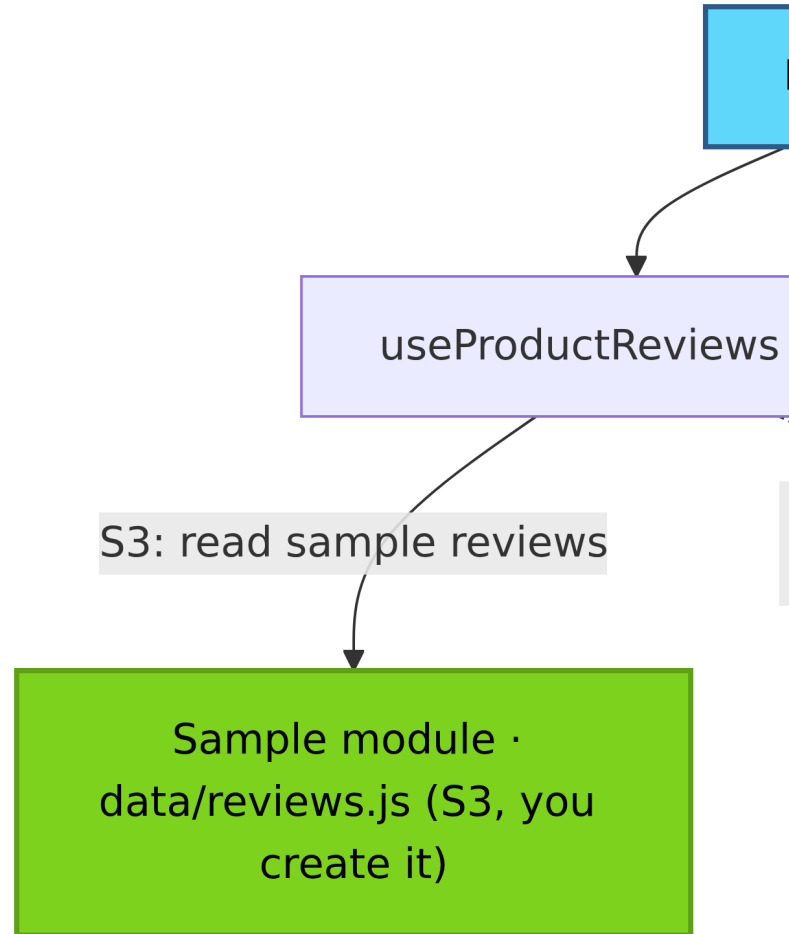
```

4.5 The review shape (exactly 8 fields)

Field	Type	Example	Notes
review_id	string	"AMZ_0001"	unique review id — use

Field	Type	Example	Notes
			as the React key
source	string	"Amazon" "BestBuy" "Walmart"	where the review came from
author	string	"j.doe"	reviewer name
rating	integer	4	1–5 (standard 5-star scale)
title	string	"Great sound"	short headline
body	string	"Bass is strong..."	full review text
created_at	string (ISO)	"2026-09-06T10:00:00Z"	time stamp
verified_purchase	boolean	true	whether the reviewer bought it

4.6 How the reviews flow (S3 sample module now · backend review routes from S4)



There is **no helpful field** in this schema — do not render or fetch one.

💡 Bridge to the practice (TP 3)

In the practice you build these review components yourself (optionally with an agent). A healthy order: render `ReviewCard` / `ReviewList` from the **sample module first**, keep every data access behind `useProductReviews`, so that in S4 — when `GET /api/products/:id/reviews` finally answers on `:4000` — you change the **hook only**, never the components.

5 🤖 AI-generate → You verify (the React/JSX beat)

Here is one **CRISP** prompt that produces the whole reviews section (use an agent if you like — then verify as written), followed by the master checklist you run on *any* generated React code.

```
Context: React 19 frontend (Vite)
for a multi-source product review
aggregator.
A review has 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).
There is NO "helpful" field. In this
session reviews come from the
useProductReviews(productId) hook
(sample module frontend/src/data/
reviews.js);
from S4 the same hook
will call the backend on
import.meta.env.VITE_BACKEND_URL
(default http://localhost:4000) via
```

```
GET /api/products/:id/reviews.
```

Task: Create `ReviewsPage.jsx` that reads `:productId` with `useParams`, uses the hook,

and renders a heading ("Reviews from N sources (M total)") plus a grid of `ReviewCard` for the product's reviews. Constraints: no fetch here (the hook owns data access); render loading/error states;

use `key={review.review_id}`; render a source badge and a 5-star scale from rating;

Tailwind (build-time) classes only.

Output: complete `ReviewsPage.jsx` file.

⚠ You verify — the master “common AI mistakes” checklist

Run these seven checks on the generated file before you trust it:

1. **Missing key on .map** → `reviews.map(r => <ReviewCard review={r}/>)` → add a stable `key={r.review_id}`.
2. **State mutation** → `reviews.push(newR)` or `form.author = x` → use the setter with a *new* value: `setForm({ ...form, author: x })`.
3. **Async work in render** → a `fetch(...)` / data call in the component body → move it inside `useEffect` (or into the hook).
4. **No `useEffect` cleanup** → missing `return () => ...` → return a cleanup that flips a *cancelled* flag (prevents setting state after unmount).
5. **Wrong response key (S4 preview)** → when the hook switches to `fetch`, `setReviews(json)` is the bug — read the envelope: `setReviews(json.data)` (and `json.meta`).
6. **Hardcoded URL** → `fetch('http://localhost:4000/...')` → use `fetch(`${API_BASE}/...`)` where `API_BASE = import.meta.env.VITE_BACKEND_URL`.
7. **Tailwind via CDN** → a `<script src="https://cdn.tailwindcss.com">` in `index.html` → keep **build-time** Tailwind (`postcss.config.cjs` + `@tailwind` directives in `src/styles.css`).

6 📄 Prompt templates (CRISP, reusable)

Save these three; swap the brackets.

1 — Generate a presentational component

Context: React 19 + Vite, Tailwind 3.4 (build-time).
Task: Create `<ComponentName>.jsx` that renders `<what it shows>`.
Props: `<name (type)>` for each piece of data.
Constraints: single default-export function component; one root element; `className` (not `class`); Tailwind classes only; no state, no fetching.
Output: complete `<ComponentName>.jsx` file.

2 — Generate a data-fetching component

Context: React 19. Backend base URL is `import.meta.env.VITE_BACKEND_URL` (default `http://localhost:4000`). `<ENDPOINT>` returns `<the exact response shape>`.
Task: Create `<ComponentName>.jsx` that fetches `<what>` and renders it.
Constraints: `useState` for data/loading/error; `useEffect` with a cancelled flag and a cleanup function; check `res.ok`; read the correct response key (`json.data`); stable key in the `.map`; Tailwind classes only.
Output: complete `<ComponentName>.jsx` file.

3 — Convert a vanilla-JS snippet to a React component

Context: I have a vanilla-JS snippet that uses `fetch` + `async/await` and builds the DOM with `innerHTML` / `createElement`. Rewrite it as a React 19 function component.

Snippet: <paste the vanilla code>

Constraints: replace manual DOM with JSX + `useState` + `useEffect`; add a `cleanup` function; keep the same data source and response shape; use `import.meta.env.VITE_BACKEND_URL` for the base URL; Tailwind classes only.

Output: the React component.

7 🚩 Key takeaways

- **React is declarative.** You describe the UI for a given state; React re-renders the DOM. This is the upgrade from S2's manual `innerHTML/appendChild`.
- **Three core ideas: components** (functions returning JSX), **props** (read-only data flowing down), **state** (data that changes, via `useState` + its setter).
- **The review layer is yours:** the shipped app is catalog-only. In S3 you *create* `data/reviews.js`, `useProductReviews`, `ReviewCard`, `ReviewList` (and the practice adds `ReviewStats` / `ReviewForm`).
- **Data access lives in the hook.** `useProductReviews` reads the sample module in S3; in S4 the very same hook reads `GET /api/products/:id/reviews` from `:4000` — the components never change.
- **Fetch inside `useEffect`, not in render** — and **always return a `cleanup`** so you never set state on an unmounted component.
- **Respect the `{ data, meta }` envelope.** The catalog API already uses it (items in `json.data`, pagination in `json.meta`); the review routes you build in S4 reuse the same

contract. Reading the wrong key is the #1 silent bug.

- **Never `hardcode` the URL.** Use `import.meta.env.VITE_BACKEND_URL` (default `http://localhost:4000`) so the frontend stays configurable.
- **In S4 the review routes read an in-memory store; in S5 the MySQL `reviews` table.** Reviews are designed to reach the UI **only** through `GET /api/products/:id/reviews`. The on-demand **scraper (:5000)** is the **home stretch**, not wired in the 18 h.
- **A review is exactly 8 fields** (`review_id`, `source`, `author`, `rating` 1–5, `title`, `body`, `created_at`, `verified_purchase`). **No helpful field.**
- **Tailwind is build-time** here: `tailwind.config.cjs` + `postcss.config.cjs` + `@tailwind` directives — **not** a CDN `<script>`.
- **The working loop (agent optional):** generate → verify against the mistakes checklist → integrate (browser checkpoint) → commit. You are the quality gate.

Next up (Session 4): the **Node 22 + Express 4.18 backend** on `:4000` — including the review routes (in-memory) — and then **Session 5**, the **MySQL 8.0** database (`reviews` table).