

JavaScript & async — Practice

Vanilla JS + fetch of local JSON review data, AI-assisted (Session 2)

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	6 Task 4 (stretch) — The aggregate (per-source counts + rating histogram) ...	9
1.1 The AI-first pattern (the rhythm of every task)	2	6.1 AI generates	10
2 Setup & running the app	2	6.2 You verify — common AI mistakes for this task	10
3 Task 1 — Render the review list from a local <code>reviews.json</code>	4	6.3 You integrate	10
3.1 AI generates	5	7 Self-check	10
3.2 You verify — common AI mistakes for this task	6	8 Building toward Session 3	11
3.3 You integrate	6		
4 Task 2 — Filter the list by source	7		
4.1 AI generates	7		
4.2 You verify — common AI mistakes for this task	8		
4.3 You integrate	8		
5 Task 3 — Sort the list (by rating or by date)	8		
5.1 AI generates	8		
5.2 You verify — common AI mistakes for this task	9		
5.3 You integrate	9		

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

Philosophy — build the *core* skills and skip the deep parts (React handles those better next

session). You are writing **vanilla JavaScript**: DOM manipulation, events, and `fetch` with `async/await`. You are **not** writing React yet — that is Session 3's job.

Goal — build a small vanilla-JS **review list** prototype that:

1. renders a **list of reviews** to the DOM,
2. lets you filter it by **source** (Amazon / Best-Buy / Walmart),
3. loads the reviews from a **local reviews.json** with `fetch` + `async/await` (the same pattern you will later point at the `:4000` API),
4. (*stretch*) aggregates the reviews — per-source counts and a rating histogram.

You verify each AI-generated piece against a **common-mistakes checklist**, then integrate it with a checkpoint and a git commit.

1.1 The AI-first pattern (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.
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** (see the course Introduction for the source). 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 API response is shaped `{ data: [...], meta: {...} }`.

The **product catalog** this app ships is your **given** foundation — you are **not** building it in this session. Your deliverable is the **review layer** on top of it: a list of reviews, driven by a local `reviews.json` you fetch with the same `fetch` pattern you will later point at the `:4000` API.

⚠ Warning

`fetch` will not work from a `file://` page — open the prototype through the **Vite dev server** (`http://localhost:5173`), which serves anything under `public/` at the site root, so a relative `fetch('./reviews.json')` resolves to a real URL. (When you later point the same `fetch` at the `:4000` API, that call is cross-origin and needs the backend running with CORS enabled.)

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 (*context — the given catalog; not required for this session’s local-JSON prototype, but useful to see what you are building on top of*). 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. (Context) Peek at the given catalog. If you started the backend, run this in a terminal and confirm you see the product catalog JSON:

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

You should see an object with a `data` array (the **given** products) and a `meta` object. This catalog is your foundation, not your build target — this session’s prototype works on a local `reviews.json`, so the tasks below do **not** depend on the backend.

i Note

Your prototype files live in `frontend/public/prototypes/`. Vite serves anything under `public/` at the site root, so `frontend/public/prototypes/review-list.html` is opened at `http://localhost:5173/prototypes/review-list.html`. Your local `reviews.json` sits beside it, so a relative `fetch('./reviews.json')` from the page resolves to it. No Live Server extension is needed.

RECALL / if needed — the canonical data shapes (the only ones in this course) A **product** has exactly these fields:

Field	Type	Example
<code>id</code>	number	1
<code>name</code>	string	"Wireless Bluetooth Headphones"
<code>description</code>	string	"Premium noise-canceling headphones..."
<code>price</code>	number	199.99
<code>image_url</code>	string	"https://placeholder.co/300x200?text=Headphones"
<code>category_id</code>	number	1
<code>category_name</code>	string	"Electronics"
<code>created_at</code>	ISO string	"2024-10-01T00:00:00Z"

A **review** has exactly these fields:

Field	Type	Notes
<code>review_id</code>	string	e.g. "AMZ_0001" — a string , not a number

Field	Type	Notes
		(propagates to S4/S5)
source	string	one of Amazon, BestBuy, Walmart
author	string	the reviewer's name
rating	integer	1–5 (there is no 0)
title	string	
body	string	the review text (the field is <code>body</code> , not <code>content</code>)
created_at	ISO string	
verified_purchase	boolean	

There is **no “Helpful” button** and **no 0–5 rating** anywhere in this course. If the AI adds either, that is a mistake to catch.

RECALL / if needed — `async/await + fetch` in 30 seconds

```

async function load() {
  const res = await fetch('./
  reviews.json'); //
  a Promise<Response> (local file now;
  the :4000 API later)
  if (!res.ok) throw new Error('HTTP '
+ res.status); // 404/500 → throw,
  don't parse
  const json = await
  res.json(); //
  a Promise<value>
  return
  json; //

```

```

a local file IS the array (no { data,
meta } wrapper)
}

```

`await` only works inside an `async` function. A `fetch` you forget to `await` gives you a `Promise` object — rendering that prints `[object Promise]`.

RECALL / if needed — see the real response in DevTools Open your page at <http://localhost:5173/prototypes/review-list.html>, press **F12**, go to the **Network** tab, select **Fetch/XHR**, and reload. Click the `reviews.json` request and read the **Response** pane. This is the ground truth your `fetch` code must match — use it whenever the AI's code does not line up with the data that is actually served. (In later sessions the same tab shows the `:4000` API responses instead.)

3 Task 1 — Render the review list from a local `reviews.json`

Concept. Before wiring a network API, prove you can turn *data* into *pixels*. Start from a **local** `reviews.json` (an array of reviews using the **canonical review schema**) and render one card per review into the DOM. Fetching a local file with `fetch` is the *exact same pattern* you will later point at the `:4000` API — that is the whole point of starting local.

3.1 AI generates

💡 Tip

Copy-paste this prompt to your assistant:

```
Create three files for a vanilla-
JS review-list page (no framework,
no React, no Tailwind):
```

```
1.    frontend/public/prototypes/
reviews.json
```

```
- an array of 6 reviews, each
with EXACTLY these fields:
```

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

```
- vary the sources and the
ratings so the later filter/sort/
aggregate tasks have data to work
with
```

```
2.    frontend/public/prototypes/
review-list.html
```

```
- <h1>Multi-Source Review
List</h1>
```

```
- a short <p> subtitle: "Amazon,
BestBuy, Walmart – one list"
```

```
- an empty <div id="review-
list">
```

```
- a <style> block with a
simple responsive list (CSS grid
or flexbox) and basic card styling
```

```
- <script src="review-
list.js"></script> just before </
body>
```

```
3.    frontend/public/prototypes/
review-list.js (a plain <script>,
NOT an ES module)
```

```
- an async function
```

```
loadReviews() that:
```

```
* const res = await fetch('./
reviews.json')
```

```
* if (!res.ok) throw new
Error('HTTP ' + res.status)
```

```
* return await
```

3.2 You verify — common AI mistakes for this task

- **Made-up schema.** The AI often invents fields like `helpful`, `helpful_votes`, `comment`, `reviewer_name`, or `avgRating`. Keep **only** the canonical review fields (`review_id`, `source`, `author`, `rating`, `title`, `body`, `created_at`, `verified_purchase`). The text field is `body`, not `content/comment`; the name is `author`, not `reviewer_name`.
- **review_id treated as a number.** `review_id` is a **string** (e.g. "AMZ_0001"). If the AI emits numeric ids or drops the prefix, that breaks the schema — and it propagates to later sessions.
- **innerHTML in a loop.** `list.innerHTML += card` (or `=` inside the loop) so only the last review shows, or duplicates pile up on re-render. Clear once, then append per item.
- **Runs before the DOM is ready.** No `DOMContentLoaded` guard → `getElementById` returns `null` and the script throws.
- **Reads a { data, meta } wrapper that is not there.** A local JSON file is just the array — do **not** read `json.data` here. (The `:4000` API wraps in `{ data, meta }`; a local file does not.)
- **Rating shown raw / a 0–5 scale.** Show rating as "`N / 5`"; there is **no 0** and **no “Helpful” button** anywhere in this course.
- **Tailwind / external CSS.** Plain `<style>` only.

3.3 You integrate

Checkpoint. Open `http://localhost:5173/prototypes/review-list.html` and you should see the reviews rendered as cards, each with `source`, `author`, a `N / 5` rating, `title`, `body`, `created_at`, and a “Verified purchase” badge when `verified_purchase` is true.

```
git add frontend/public/
prototypes/reviews.json frontend/
public/prototypes/review-list.html
frontend/public/prototypes/review-
list.js
git commit -m "feat(prototype):
render review list from local
reviews.json"
```

4 Task 2 — Filter the list by source

Concept. A multi-source list is only useful if you can narrow it down. Add a **source** filter (Amazon / BestBuy / Walmart) — the canonical filter dimension for this app. This is where you practise `Array.prototype.filter()` and event listeners, the two skills that transfer directly into React next session.

4.1 AI generates



Tip

Copy-paste this prompt to your assistant:

```
Update frontend/public/
prototypes/review-list.js to add
a source filter to the EXISTING
review list
(keep the fetched `reviews` array
as the source of truth):
```

1. In the HTML, add above `#review-list`:

```
- a <select id="source-
filter"> with an "All sources"
option (value "all")
    plus one option per source:
    Amazon, BestBuy, Walmart
```

2. Add `applyFilters()` that:

```
* reads the selected source
(value "all" = no filter)
* returns a NEW array:
reviews.filter(r => source ===
"all" || r.source === source)
(do NOT mutate the original
`reviews`)
* re-renders with
renderReviews()
```

3. Wire the `<select>` to its `'change'` event, calling `applyFilters()`.

Do not introduce a backend or a second data source — keep filtering the array you already fetched.

4.2 You verify — common AI mistakes for this task

- **Mutates the source.** `reviews = reviews.filter(...)` reassigns the original, so filtering twice keeps shrinking the list. Derive a **new** array from the untouched source each time.
- **Filters the wrong field.** The canonical filter dimension is `source` (Amazon / BestBuy / Walmart) — not `category`, not `author`.
- **Wrong event.** The `<select>` uses `change`. (A free-text search input would use `input` — that is not part of this task.)
- **Forgets to re-render.** The filtered array is computed but the DOM is not updated.
- **Drops the “all” case.** Selecting “All sources” must return the full list, not an empty one.

4.3 You integrate

Checkpoint. Choosing a source narrows the list live to that source’s reviews, and selecting “All sources” brings them all back.

```
git add frontend/public/prototypes/  
git commit -m "feat(prototype):  
filter review list by source"
```

5 Task 3 — Sort the list (by rating or by date)

Concept. Once you can filter, you can order. Add a **sort** control: by **date** (`created_at`, newest first) or by **rating** (highest first). This practises `Array.prototype.sort()` with a comparator and parsing ISO date strings — both carry straight into React next session.

5.1 AI generates



Tip

Copy-paste this prompt to your assistant:

Update `frontend/public/prototypes/review-list.js` to add sorting on top of the existing source filter (keep both working together):

1. In the HTML, add next to `#source-filter`:

- a `<select id="sort-select">` with these options (value -> meaning):

`"newest" -> created_at descending`

`"oldest" -> created_at ascending`

`"highest" -> rating descending`

`"lowest" -> rating ascending`

2. Add `sortReviews(list, mode)` that returns a NEW sorted array (do NOT mutate `list`):

* `"newest": [...list].sort((a, b) => new Date(b.created_at) - new Date(a.created_at))`

* `"oldest": [...list].sort((a, b) => new Date(a.created_at) - new Date(b.created_at))`

* `"highest": [...list].sort((a, b) => b.rating - a.rating)`

* `"lowest": [...list].sort((a, b) => a.rating - b.rating)`

3. Update `applyFilters()` so the pipeline is: source filter -> sort -> `renderReviews()`.

Wire the sort `<select>` to its `'change'` event.

5.2 You verify — common AI mistakes for this task

- **Mutates / sorts in place.** `reviews.sort(...)` sorts the original array in place and reorders your source of truth. Copy first (`[...list]`), then sort the copy.
- **Sorts the wrong field.** Date sorts use `created_at` (parse with `new Date(...)`); rating sorts use `rating`. Do not sort by `review_id` or `author`.
- **Fragile date comparison.** Comparing raw ISO strings “works” but `new Date(a) - new Date(b)` is the robust form; a malformed date yields `NaN` and breaks the order.
- **Ignores the current filter.** Sorting must run *after* the source filter, so the visible (filtered) list is what gets sorted.
- **Forgets to re-render.** The sorted array is computed but the DOM is not updated.

5.3 You integrate

Checkpoint. The sort control re-orders the (already filtered) list live — newest/oldest by `created_at`, highest/lowest by `rating` — and “All sources” plus a sort still shows the full list in that order.

```
git add frontend/public/prototypes/  
git commit -m "feat(prototype): sort  
review list by rating and date"
```

`reduce` — and **guarding** your data, because real reviews are not always well-formed.

6 Task 4 (stretch) — The aggregate (per-source counts + rating histogram)

Concept. Stretch goal that previews the “read a list and summarize it” skill you will keep using. Compute two aggregates over the **currently visible** (filtered + sorted) reviews: **per-source counts** and a **rating histogram** (buckets 1–5). This is where you practise

6.1 AI generates



Tip

Copy-paste this prompt to your assistant:

```
Add an aggregate
summary to frontend/
public/prototypes/review-list.js
(optional stretch):
```

```
1. Add aggregate(list) that
returns { bySource, histogram }
computed over the CURRENTLY
VISIBLE
```

```
  (filtered + sorted) list:
    * bySource: { Amazon: n,
BestBuy: n, Walmart: n } (counts
of r.source)
    * histogram: { 1: n, 2: n, 3:
n, 4: n, 5: n } (counts
of r.rating)
```

```
2. GUARD the rating: only count a
review in the histogram when
```

```
  Number.isInteger(r.rating) &&
r.rating >= 1 && r.rating <= 5.
```

```
  Ignore anything else (0, 6, a
string like "5", NaN, missing) –
do not crash and do not add a
bucket outside 1..5.
```

```
3. Render a small summary block
above #review-list:
```

```
  - one line per source: "Amazon
– 2"
```

```
  - one line per rating bucket
1..5: "5 – 1" (word it as a count,
not a star widget)
```

```
  - recompute and update it
inside applyFilters() so it always
reflects the visible list.
```

6.2 You verify — common AI mistakes for this task

- **No rating guard.** A rating of 0, 6, "5" (a string), NaN, or missing must be **ignored** — not counted, not added to a bogus bucket, and not crash the page. The guard is `Number.isInteger(r.rating) && r.rating >= 1 && r.rating <= 5`.
- **Counts the wrong field.** `bySource` counts `r.source`; the histogram counts `r.rating`. Do not count `review_id` or `author`.
- **Aggregates the wrong list.** The summary must reflect the **visible** (filtered) list, so it updates when the source filter changes — not the full static array.
- **Missing bucket.** All five rating buckets (1–5) should appear even when a bucket is 0.
- **Recomputes in the wrong place.** The aggregate must be recomputed in the filter/render pipeline (after filtering), not once on load.

6.3 You integrate

Checkpoint. The summary shows per-source counts and a 1–5 rating histogram that update live as you change the source filter, and an invalid rating in the data is silently ignored (no crash, no bogus bucket).

```
git add frontend/public/prototypes/
git commit -m "feat(prototype): per-
source counts + rating histogram
(stretch)"
```

7 Self-check

Run through this before you finish:

- ☐ Task 1: the review list renders from a local `reviews.json` via `fetch + async/await` (8-field schema, `review_id` a **string**, `N / 5` rating, “Verified purchase” badge).

- ☐ Task 2: the source filter (Amazon / Best-Buy / Walmart) narrows the list live; “All sources” restores it.
- ☐ Task 3: the sort control orders by `created_at` (newest/oldest) or `rating` (highest/lowest) on top of the filter, without mutating the source array.
- ☐ (*stretch*) Task 4: per-source counts + a 1–5 rating histogram reflect the visible list, with the rating guarded to 1–5 (invalid values ignored).
- ☐ Every task was committed with its checkpoint message.
- ☐ No React, no hooks, no Tailwind, no `file://`, no invented endpoints, no product-list build.

8 Building toward Session 3

You just did, by hand, everything React is going to take over:

- **Manual DOM rendering** → React’s declarative components
- **Event listeners you attach yourself** → React’s event props
- **A static array / a `fetch` you trigger on `DOMContentLoaded`** → a data-fetching hook
- **Re-rendering by clearing and rebuilding `innerHTML`** → React’s diffing

Session 3 converts your review list into React components fed by the **same sample data** — the real backend arrives in Session 4. But that is tomorrow’s job. For now: vanilla JS, `fetch`, `async/await`, and the discipline of verifying what the AI wrote.