

JavaScript for React

The async + data foundation that turns a static card into a live app

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Abstract You built a static review card in Session 1. This session hands you the JavaScript that makes it *live*: modern JS in practice, async/await + Promises + fetch, and a clear answer to *why React exists* — so you can walk into Session 3 (a React component fed with sample data) with confidence.

Table of contents

1	Where you are & what is new	2	3.5	3.5 The <code>fetch</code> API, step by step	5
1.1	Covered before vs. not covered (yet)	2	3.6	3.6 Worked example — GET products from <code>:4000</code> (mirrors the app)	6
1.1.a	☑ Covered before (Session 1)	2	4	Mandatory core — how the <i>real</i> app fetches	6
1.1.b	☑ Covered in this session	2	4.1	Where the base URL comes from	7
1.1.c	➡ Not covered <i>here</i> (later sessions)	2	4.2	The essential lines (<code>frontend/src/components/ProductList.jsx</code>)	7
1.1.d	⊗ Deliberately skipped (not on the React path)	2	4.3	The rest is just “render whatever state holds”	8
2	RECALL / if needed — modern JS fundamentals	2	5	Mandatory core — why React exists	8
3	Mandatory core — async/await, Promises & <code>fetch</code>	4	5.1	5.1 The vanilla JS pain (four concrete problems)	8
3.1	3.1 The problem: the data isn’t there yet	4	5.2	5.2 React’s answer: the UI is a function of state	9
3.2	3.2 Just enough about the event loop	4	5.3	5.3 (Light) the virtual DOM	9
3.3	3.3 Promises: a placeholder for a value you’ll get later	4	6	AI-generate → verify (JS/async beat)	10
3.4	3.4 <code>async</code> / <code>await</code> : making async read top-to-bottom	5	6.1	6.1 Concept	10
			6.2	6.2 AI generates (CRISP prompt)	10
			6.3	6.3 You verify (common AI mistakes for JS/async)	10

6.4	6.4 You integrate (checkpoint + commit)	11
7	Prompt templates (reuse these)	12
7.1	T1 — Fetch a new endpoint	12
7.2	T2 — Harden an existing fetch (loading + error states)	12
7.3	T3 — Refactor callbacks / <code>.then</code> chains to <code>async/await</code>	12
8	Key takeaways	12

1 Where you are & what is new

Session 1 gave you a **static review card**: HTML for the structure, CSS for the styling. It renders, but it is a *picture* — no data comes in, nothing reacts, the values are hard-coded.

This session (Session 2) hands you the **JavaScript** you need to make that card *live*:

- fetch structured review data from a local JSON file,
- handle **async** — the fact that the data is not there yet when your code first runs,
- and see clearly **why React exists** once data starts flowing and the UI has to stay in sync.

You are *not* being made a JavaScript expert. You are getting **just enough** to walk into **Session 3** — where the card becomes a real **React component** fed with sample data — and to *verify* the JavaScript an AI assistant generates.

i The three-sentence version

- HTML/CSS = the *look* (done in Session 1).
- JavaScript = the *behavior*: get data, update the screen when it arrives.
- React = the *tool* that makes “keep the screen in sync with the data” stop being painful.

1.1 Covered before vs. not covered (yet)

1.1.a ☒ Covered before (Session 1)

- HTML document structure
- CSS styling (the review card looks right)
- Serving a static page in the browser

1.1.b ☒ Covered in this session

- Modern JS in practice (you *use* it, you do not memorize it)
- `async / await`, Promises, the `fetch` API
- How the real app fetches data
- *Why React exists*

1.1.c ➡ Not covered *here* (later sessions)

- React components & state → **Session 3**
- The backend REST API → **Session 4**
- Wiring MySQL → **Session 5 (capstone)**

1.1.d ✕ Deliberately skipped (not on the React path)

- Callbacks, `.then()` promise chains
- `Promise.all` / `Promise.race`
- POST / PUT / DELETE requests (you meet these in Session 4)
- Deep event-loop internals

2 RECALL / if needed — modern JS fundamentals

This whole section is a **refresher**, not a re-teach. If the syntax is already familiar, skip straight to the **async** core below. If you are new to JavaScript, expand only what you need. Everything here will show up in the mandatory core.

RECALL / if needed — variables, functions & strings (`let/const`, arrow functions, template literals) **Variables**. Default to `const`; use `let` only when you must reassign; never `var`.

```
const name = "Alice"; // won't be
reassigned → const
let count = 0; // will be
reassigned → let
count = 1; // OK
// name = "Bob"; // ERROR: const
```

Arrow functions. Short callbacks; implicit return when there is no body.

```
const greet = (name) => `Hello,
${name}`; // implicit return
const total = (price, tax) => price
* (1 + tax); // math
const names = products.map((p) =>
p.name); // callback (React
uses this constantly)
```

Template literals. Backticks + `${}` for interpolation and multi-line strings. (JSX uses the same idea.)

```
const line = `${p.name} - ${p.price}
`;
```

RECALL / if needed — objects, arrays, destructuring & spread **Objects** are key/value pairs; read them with dot notation (like a PHP assoc array, but `.` not `->`).

```
const product = { id: 1, name:
"Laptop", price: 999, category_name:
"Electronics" };
product.name; // "Laptop"
```

Arrays are ordered lists; `.map` transforms, `.filter` selects.

```
const names = products.map((p) =>
p.name);
const cheap = products.filter((p) =>
p.price < 50);
```

Destructuring pulls values out without writing the index or key.

```
const { name, price } =
product; // object
destructuring
const [first, ...rest] =
products; // array
destructuring + rest
function show({ title, rating = 5 })
{ ... } // destructure a function
parameter
```

Spread / rest (...) copies elements or properties.

```
const withBadge = { ...product,
onSale: true }; // spread into a
new object
const copy =
[...products]; //
shallow copy of an array
```

RECALL / if needed — a few more array & JS niceties (only when they show up) You will *recognize* these more than *write* them from scratch:

```
// reduce — fold a list into one value
(e.g. a total)
const total = prices.reduce((sum, p)
=> sum + p, 0);

// find / some / every — ask a question
about the list
const free = products.find((p) =>
p.price === 0);
const hasFiveStar = reviews.some((r)
=> r.rating === 5);
const allVerified = reviews.every((r)
=> r.verified_purchase);

// optional chaining (?.) — stop at
```

```

the first missing link instead of
crashing
const headline = review?.title; //
review not loaded yet? undefined, not
a crash

// nullish coalescing (??) – default
only for null/undefined (not for 0
or "")
const label = product.name ??
"Unknown";

```

3 Mandatory core — async/await, Promises & fetch

This is the heart of the session. Everything here is **mandatory** and written in full — no part of it is hidden.

3.1 3.1 The problem: the data isn't there yet

In Session 1 your card was a picture: the values were hard-coded in the HTML. Now the values come from a JSON data file you **fetch**. But a network request takes time — maybe 50 ms, maybe 2 s. Your JavaScript **starts running before the data arrives**.

That mismatch is what **asynchronous** code is all about: *“keep going, and here’s what to do the moment the data shows up.”*

Analogy. In a coffee shop you either order and stand there waiting, or you order, get a number, and keep reading your phone until the barista calls it. A **blocking** program behaves like the first; an **async** program behaves like the second.

3.2 3.2 Just enough about the event loop

You do **not** need the internals. You need one mental model:

- JavaScript runs **one thread** of code at a time.
- A slow operation (a network request) is **handed off** to the browser/engine so the UI keeps responding.
- When the operation finishes, the “do this next” instruction is queued and runs later.

That is why you can **await** a fetch without freezing the page — and why a missing **await** silently does something surprising (see the verify checklist in Section 6).

RECALL / if needed — the call stack, task queue & microtasks (skip if the mental model above is enough)

- **Call stack:** the synchronous calls currently executing, innermost last.
- **Web APIs:** **fetch**, **setTimeout**, etc. live here; they run outside JS and call back when done.
- **Task queue (macrotasks):** callbacks wait here and run one by one once the stack is empty.
- **Microtask queue:** **Promise** callbacks run here and are drained *before* the next macro-task — which is why promise resolution feels “immediate” once it happens.

You will reason about the *ordering* of async work with this, but you will not be tested on the details.

3.3 3.3 Promises: a placeholder for a value you’ll get later

A **Promise** is an object that stands in for a value that isn’t ready yet. It is in one of three states:

1. **pending** — the request is in flight,
2. **fulfilled (resolved)** — it succeeded and has a value,

3. **rejected** — it failed and has a reason (an error).

```
// fetch() returns a Promise
immediately — it has NOT made the
result available yet
const p = fetch("http://localhost:
4000/api/products");
console.log(p); // Promise
{ <pending> } — not the data!
```

The key insight: **fetch(url)** gives you a **Promise, not the data**. You must “unwrap” it.

3.4 3.4 **async / await**: making **async** read top-to-bottom

await only works inside an **async** function. It says: “*pause here until this Promise settles, then hand me the result.*” Wrap the body in **try/catch** to handle failures.

```
async function loadProducts() {
  try {
    const response =
    await fetch("http://localhost:4000/
api/products"); // pause until the
request settles
    const body = await response.json();
    pause until the JSON is parsed
    return body;
  } catch (err) {
    console.error("Could not load
products:", err);
    return []; // fall back to an
empty list so the UI can still render
  }
}
```

Three moving parts, every time:

Part	Where	Job
async	on the function	lets you use await inside
await	before each Promise	pauses until it settles, returns the value
try/catch	around awaits	catches network errors & thrown errors

RECALL / if needed — combining Promises (**Promise.all**) when you have several requests. If you need two things at once (say products **and** categories), you can run them in parallel and **await** the results together:

```
const [productsRes, categoriesRes] =
await Promise.all([
  fetch(`${backend}/api/products`),
  fetch(`${backend}/api/
categories`),
]);
const products = (await
productsRes.json()).data || [];
const categories = (await
categoriesRes.json()).data || [];
```

You do not need this for the next few sessions — React’s **useEffect** (Session 3) shows you the idiomatic place to fire a request. It is here so the term *is not* a surprise.

3.5 3.5 The **fetch** API, step by step

A GET request is three steps:

1. **const response = await fetch(url)** — send the request, get a **Response** object.

2. **Check the status** — `response.ok` is true for 2xx; a failed request does **not** throw, so you must check.
3. `const body = await response.json()` — parse the JSON body into a JS value.

```
async function get(url) {
  const response = await fetch(url);
  if (!response.ok)
  {
    // e.g. 404,
    500 — fetch did NOT throw
    throw new Error(`HTTP
    ${response.status} for ${url}`);
  }
  return await response.json();
}
```

Why the status check matters. `fetch` only rejects on a *network* failure (offline, bad URL, CORS block). An HTTP 404 or 500 still resolves — the status lives on the `Response`. Skipping `response.ok` is one of the most common real-world bugs.

3.6 3.6 Worked example — GET products from :4000 (mirrors the app)

Here is the exact shape of the pattern you will see everywhere in the app. The backend API runs on **port 4000**; the app calls it with a base URL (see the next section for where that base comes from).

```
// BASE: the backend origin. In the
app this is VITE_BACKEND_URL, default
http://localhost:4000
const backend = "http://localhost:
4000";

async function loadProducts() {
  try {
```

```
// 1. request
const response = await
fetch(`${backend}/api/products`);

// 2. status check (a 404/500 does
NOT throw by itself)
if (!response.ok) {
  throw new Error(`Products request
  failed: HTTP ${response.status}`);
}

// 3. parse JSON
const body = await
response.json();

// 4. the app returns { data:
[...], meta: {...} } — we want the
list
const products = body.data || [];

return products;
} catch (err) {
  console.error("Failed to load
  products:", err);
  return []; // graceful fallback:
  the UI renders "no products"
}
}
```

What to notice (this is the whole async story):

- the function is `async`,
- we `await` the `fetch` **and** the `response.json()`,
- we check `response.ok` *before* parsing,
- a `try/catch` turns a failure into a clean empty list instead of a crash.

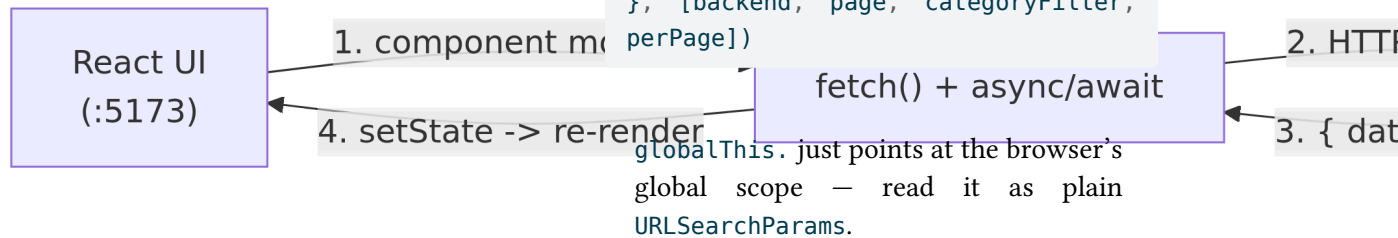
4 Mandatory core — how the *real* app fetches

You now know the pattern in the abstract. Here is the **actual code** from the app this

course builds against — fullstack-minimal-app (the develop branch of ebro/fullstack-minimal-app, referenced by URL, not a submodule). The frontend runs on :5173 and the backend API on :4000.

4.1 Where the base URL comes from

The frontend calls the backend with a base URL read from the env var VITE_BACKEND_URL, which defaults to http://localhost:4000. In the component we look at, that base is handed in as a prop named backend, so every request is built as `\${backend}/api/...`.



```

const body = await res.json()
2. parse JSON
   setProducts(body.data || [])
3. -> state
   setMeta(body.meta || { total:
0, page: p, per_page: perPage,
total_pages: 1 })
   } catch (err) {
     console.error('Fetch page failed',
err) //
4. error handled
   } finally {
     setLoading(false)
5. stop the spinner
   }
}, [backend, page, categoryFilter,
perPage])

```

4.2 The essential lines (frontend/src/components/ProductList.jsx)

This is the heart of the component — everything else around it is UI:

```

const fetchPage = useCallback(async
(p = page, c = categoryFilter) => {
  setLoading(true)
  try {
    const params = new
globalThis.URLSearchParams({ page:
String(p),
per_page: String(perPage) })
    if (c) params.set('category_id',
c)
    const res
= await fetch(`${backend}/api/
products?${params.toString()}`) //
1. request

```

Line by line — this is exactly Section 3, in the wild:

- `async + await fetch(...) + await res.json()` → the three-part pattern.
- `URLSearchParams` builds the query string ? `page=2&per_page=9&category_id=3`.
- The backend returns `{ data: [...], meta: {...} }`; the component keeps `data` (the list) and `meta` (pagination info) in `state` with `setProducts / setMeta`.
- `try / catch / finally` — the error is logged, and the `finally` always turns off the loading flag so the UI can never get stuck on a spinner.

Note: `useState`, `useEffect`, `useCallback` and `setProducts(...)` are **React** features — they are *why* the next section

exists. The **fetch/async** half of this snippet is pure JavaScript you already know.

4.3 The rest is just “render whatever state holds”

The component guards the edge cases early, then renders the list with `.map`:

```
if (loading) return <div>Loading
products...</div> // edge
case: still fetching
if (products.length === 0) return
<div>No products yet.</div> // edge
case: empty list

return (
  <ul>
    {products.map((p) => (
      <li key={p.id}>{p.name} -
{Number(p.price).toFixed(2)}</li>
    ))}
  </ul>
)
```

Two things to notice:

- **categories** is a **prop** — it is fetched the same way (`fetch → res.json()`) higher up in the app and passed down. Same pattern, different endpoint.
- **The UI is a function of state.** When `setProducts` runs, the list re-renders. No manual DOM updates — which is precisely the problem React solves.

5 Mandatory core — why React exists

You can build *everything* in Session 3 with plain JavaScript. For a static card, you already did (Session 1). So why reach for a framework at all? Because **once data flows and the UI has to stay in sync with it, the manual ap-**

proach falls over. Here is the honest version of why.

5.1 5.1 The vanilla JS pain (four concrete problems)

Imagine your card now shows a *list* of reviews fetched from the backend, and the user can add, filter, and sort them. In plain JavaScript you own every pixel:

```
// 1. MANUALLY update the DOM whenever
data changes
const list =
document.querySelector("#reviews");
list.textContent = "";
reviews.forEach((r) => {
  const el =
document.createElement("li");
  el.textContent = `${r.author}:
${r.comment}`;
  list.appendChild(el);
});

// 2. KEEP STATE IN SYNC — you track
it by hand in variables
let isLoading = false;
let error = null;
// ...and you must remember to update
the DOM every single time one of these
changes.

// 3. ASYNC COMPLEXITY — where do you
fetch? when? after what?
button.addEventListener("click",
async () => {
  isLoading = true; //
update the spinner in the DOM
  const res = await fetch(url);
  const data = await res.json();
  reviews = data; // now
go update the DOM again...
  isLoading = false; // ...
and the spinner again...
```

```
});

// 4. BOILERPLATE — select, create,
append, remove, re-listen for events,
over and over.
```

The failure mode is not any one bug — it is that **state lives in your variables and the screen lives in the DOM, and nothing keeps the two honest with each other**. Add filtering, then sorting, then a shopping cart, and every new feature means “find the right element and update it by hand.” The code does not just get longer; it gets *fragile*.

5.2 React’s answer: the UI is a function of state

React inverts the model. Instead of “*tell me how to change the screen step by step*,” you **describe what the screen looks like for a given state**, and React updates the real DOM for you:

```
// Declarative: describe the result,
not the mutations
function ReviewList({ reviews,
  isLoading }) {
  if (isLoading) return <div>Loading...
</div>;
  return (
    <ul>
      {reviews.map((r) => (
        <li key={r.id}>{r.author}:
        {r.comment}</li>
      ))}
    </ul>
  );
}
```

The mental shift:

- **State is the source of truth** (`useState`). Change state → React re-renders the component.

- **You never touch the DOM**. No `createElement`, no `appendChild`, no `textContent =`.
- **Events, lists, and conditional UI are part of the description** (`onClick`, `.map`, `if`).

That single idea — $UI = f(state)$ — is what Session 3 builds on.

Vanilla JS — imperative

```
const li
= document.createElement("li");
li.textContent = review.author;
list.appendChild(li);
// + remember to remove/rebuild on
every change
```

React — declarative

```
<li key={review.id}>{review.author}</li>
// rendered from state; React handles
the DOM
```

5.3 (Light) the virtual DOM

React keeps a lightweight in-memory copy of the UI — the **virtual DOM**. When state changes, it compares the new tree to the previous one and writes only the *minimal* set of changes to the real DOM. You do not need to understand the diffing algorithm; you only need the consequence: **you describe the target state, React computes the cheapest path to it**.

Scope check. We are *not* going to build a virtual DOM, nor to benchmark React against vanilla. We are adopting its model because it removes the sync bug class from Section 5.1.

6 AI-generate → verify (JS/async beat)

This is the uniform beat of the course: **Concept → AI generates → You verify → You integrate**. For this session the AI beat targets **JS/async** code, so the verify checklist is about the failure modes below.

6.1 6.1 Concept

We are going to have the assistant write a small, real piece of the app: **fetch the categories and make them available to render**. You already know the shape — this is the *categories* sibling of the products fetch from Section 4.

6.2 6.2 AI generates (CRISP prompt)

Send the assistant a **CRISP** prompt — Context, Role, Input, Steps, Parameters:

```
Context:    We're in the fullstack-
minimal-app frontend (React +
Vite, :5173). The backend API is on
            :4000, reached via the
VITE_BACKEND_URL base URL (default
http://localhost:4000).
Role:      You are a senior
frontend developer writing clean,
modern JavaScript.
Input:     An endpoint `${backend}/
api/categories` that returns { data:
[{ id, name }, ...] }.
Steps:    1) Write an async function
loadCategories(backend) that fetches
that URL.
           2) If !response.ok, throw new
Error(`HTTP ${response.status}`).
           3) Parse JSON and return
body.data || [].
           4) Wrap the body in try/
catch; on error, console.error and
```

```
return [].
```

5) Return data only — do NOT build any HTML or touch the DOM. Parameters: Pass `backend` in as an argument (do not hardcode the URL). No external libraries. Use const, arrow functions, and template literals.

Expected output (what you should see):

```
async                                function
loadCategories(backend) {
  try {
    const res = await
fetch(`${backend}/api/categories`);
    if (!res.ok) throw new Error(`HTTP
${res.status}`);
    const body = await res.json();
    return body.data || [];
  } catch (err) {
    console.error("Failed to load
categories:", err);
    return [];
  }
}
```

6.3 6.3 You verify (common AI mistakes for JS/async)

Do **not** paste the output into the app until it passes every line of this checklist:

#	Failure mode	What to check
1	Missing await	const res = fetch(url) then res.json() with no await → res is a Promise, not data. Every fetch and every res.json() is awaited, and the

#	Failure mode	What to check	#	Failure mode	What to check
		enclosing function is <code>async</code> .	6	innerHTML for dynamic content (XSS)	AI loves <code>el.innerHTML = data</code> — that executes injected HTML/script. React auto-escapes <code>{value}</code> in JSX, so prefer returning data + JSX over string-building with <code>innerHTML</code> . Only ever set <code>innerHTML</code> with fully trusted, static content.
2	No try/catch	Network errors crash silently. The <code>async</code> body is wrapped; the error is handled (log + graceful fallback, e.g. <code>return []</code>).			
3	Unhandled Promise rejection	A fire-and-forget <code>fetch</code> or a <code>.then</code> chain with no <code>catch</code> . Every Promise has a handler; no floating <code>await</code> sits outside a <code>try</code> .			
4	response.ok skipped / CORS assumed	A 404/500 does <i>not</i> throw, and <code>res.json()</code> on an error body can throw or parse garbage. The status is checked before parsing. Also: the request is cross-origin (<code>:5173</code> → <code>:4000</code>) — CORS is a backend concern (Session 4). Flag it; never paper over it client-side.			
5	Callback nesting / callback hell	Nested <code>.then().then()</code> instead of <code>async/await</code> . The flow reads top-to-bottom.			

If any line fails, send the assistant the specific failing line and a one-sentence correction — do not rewrite the whole thing by hand.

6.4 6.4 You integrate (checkpoint + commit)

Once the code passes the checklist:

1. **Run it.** Start the frontend (`:5173`) and the backend (`:4000`); confirm the categories actually appear — not a spinner, not a red error.
2. **Break it on purpose.** Stop the backend, reload, and confirm the app degrades gracefully (the `catch` fires, you see a clean message, not a blank page).
3. **Commit the checkpoint.**

```
git add -A
git commit -m "feat(webdev): fetch
categories with async/await + error
handling (S2)"
```

The commit is your checkpoint: if the next step (wiring it into a component in

Session 3) goes wrong, you can return to a known-good state.

7 Prompt templates (reuse these)

Keep these on hand. Swap the ... values for the task at hand. Each follows **CRISP** (Context, Role, Input, Steps, Parameters).

7.1 T1 — Fetch a new endpoint

```
Context:      fullstack-
minimal-app  frontend  (React +
Vite, :5173); backend on :4000 via
VITE_BACKEND_URL.
Role:        Senior frontend developer,
modern JS.
Input:       Endpoint `${backend}/api/...`
returning { data: [...] }.
Steps:       1) async fn load...(backend)
-> fetch(url)
              2) if (!res.ok) throw new
Error(`HTTP ${res.status}`)
              3) const body = await
res.json(); return body.data || []
              4) wrap in try/catch; on
error, console.error + return []
Parameters:  pass `backend` in as
an argument (no hardcoding); no
libraries; const + arrow + template
literals; return data only
— no DOM, no innerHTML.
```

7.2 T2 — Harden an existing fetch (loading + error states)

```
Context:      The component already
fetches `...`; it has no loading or
error state.
Role:        Senior frontend developer.
Input:       The current fetch code +
the state variables available.
Steps:       1) add a `loading`
```

```
flag: true before the request, false
in `finally`
```

```
2) add an `error` state:
set it in `catch`
```

```
3) return early in the
render for loading and for error
```

```
4) never leave the spinner
stuck (always `finally`)
```

```
Parameters: keep the existing data
shape; do not change the endpoint;
minimal diff.
```

7.3 T3 — Refactor callbacks / .then chains to async/await

```
Context:      This snippet uses
nested .then() / callbacks and is hard
to read.
```

```
Role:        Senior frontend developer,
clean modern JS.
```

```
Input:       The existing snippet
(paste it).
```

```
Steps:       1) convert to an async
function with await at each step
```

```
2) keep the same behavior
(same requests, same result)
```

```
3) add try/catch around
the awaits
```

```
4) no behavior change —
a pure readability refactor
```

```
Parameters: same function name and
return value; no new dependencies.
```

8 Key takeaways

- **Modern JS is a tool you verify, not a script you memorize** — const/let, arrow functions, template literals, destructuring, spread, and map/filter show up constantly; recognize them, do not recite them.
- **fetch(url) returns a Promise, not data** — you await it, then await response.json(),

and you check `response.ok` *before* parsing (a 404/500 does not throw).

- **async + await + try/catch is the whole async story you need** — it makes asynchronous code read top-to-bottom and handles failures gracefully (log + fallback).
- **The app's pattern is exactly this:** `fetch ${backend}/api/products → parse { data, meta } → setProducts(...)`; the async half is plain JS, the state half is React.
- **React exists because manual DOM + hand-tracked state + async is fragile at scale** — React's model, $UI = f(state)$, removes the “keep the screen in sync” bug class.
- **AI generates, you verify** — the failure modes to always check: missing `await`, no `try/catch`, unhandled rejection, skipped `response.ok` / CORS assumption, callback nesting, and `innerHTML` for dynamic content (XSS — React auto-escapes instead).