API Integration

Real data in a real app: fetch inside useEffect, loading and error states, rendering a user list, and sending data with a POST request.

Introduction

Until now your data lived inside the app. Real apps talk: they **fetch** lists from a server, **post** new records, and handle the waiting and the failures in between. API integration is the assembly of pieces you own — `fetch` (JavaScript series), `useEffect` (5.5), state trios (5.12) — into the one pattern every data screen follows. *(Roman Urdu: API integration matlab app ka bahar ki duniya se baat karna — data lana, bhejna, aur intezaar sambhalna.)*

Key points

- **The pattern never changes:** state for `data`, `loading`, `error` → an effect (or event) that fetches → success sets data, failure sets error, both stop the loading → render one of the three states.
- **`fetch` is two steps:** `const res = await fetch(url)` gets the *response*; `await res.json()` reads the *body* as data. Forgetting the second step is the classic first-fetch bug.
- **`res.ok` is your error gate:** `fetch` does **not** throw on a 404 or 500 — it only fails on network problems. Check `if (!res.ok) throw new Error(...)` before trusting the body.
  (Roman Urdu: fetch 404 par bhi khamosh rehta hai — res.ok check karna aap ki zimmedari hai.)
- **Sending data is a POST:** `fetch(url, { method: "POST", headers: { "Content-Type": "application/json" }, body: JSON.stringify(obj) })` — method, the JSON content-type header, and the stringified body are the three parts that must all be present.
- **`async/await` inside effects:** an effect callback cannot itself be `async`; define an async function inside (or use `.then` chains) and call it. The Weather App (5.12) already used this shape.
- **Render the list from state:** fetched arrays land in state and render with `map` + keys (5.16); new items from a POST are added to that state so the screen updates without a refetch.
- **Keys and URLs live in variables:** the base URL (`https://jsonplaceholder.typicode.com`) in one constant at the top — when the real backend replaces the practice API, one line changes.

Code examples

Example 1 — The full GET pattern: users, loaded properly

```
import { useEffect, useState } from "react";

const API = "https://jsonplaceholder.typicode.com";

function UserList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState("");

  useEffect(() => {
    async function load() {
      try {
        const res = await fetch(`${API}/users`);
        if (!res.ok) throw new Error("Server said " + res.status);
        const data = await res.json();
        setUsers(data);
      } catch (err) {
        setError("Could not load users — check your connection.");
      } finally {
        setLoading(false);
      }
    }
    load();
  }, []);

  if (loading) return <p>Loading users…</p>;
  if (error) return <p className="error">{error}</p>;

  return (
    <ul>
      {users.map((u) => (
        <li key={u.id}>{u.name} — {u.email}</li>
      ))}
    </ul>
  );
}
```

What it does: The whole discipline in one component: the mount effect loads once, `res.ok` guards the body, the try/catch/finally choreographs the three states, and the renders are the early-return pattern from 5.15 feeding a keyed list from 5.16.

What you will see: "Loading users…" briefly, then ten real users with emails, straight from the practice API — and if you unplug the network and reload, the honest error line instead.

Example 2 — POST: sending a new post to the server

```
async function createPost(title, body) {
  const res = await fetch(`${API}/posts`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ title, body, userId: 1 }),
  });
  if (!res.ok) throw new Error("Could not save the post");
  return res.json();
}

// In a form's submit handler:
const saved = await createPost("My first post", "Hello from React");
console.log("Server answered with the saved post, id:", saved.id);
```

What it does: The three POST parts line up: the method says *create*, the header tells the server the body is JSON, and `JSON.stringify` turns the object into the text that travels. The practice API answers with the saved post including a fresh `id` — proof the round trip worked.

What you will see: the console logs "Server answered with the saved post, id: 101" — the object you sent, returned with its server-given identity.

Example 3 — Add without refetch: POST, then update state

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

function PostBox() {
  const [posts, setPosts] = useState([]);
  const [title, setTitle] = useState("");
  const [saving, setSaving] = useState(false);

  async function handleSubmit(e) {
    e.preventDefault();
    setSaving(true);
    const saved = await createPost(title, "Written in the React notes");
    setPosts((list) => [saved, ...list]);
    setTitle("");
    setSaving(false);
  }

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input value={title} onChange={(e) => setTitle(e.target.value)} placeholder="Post
title" />
        <button disabled={saving}>{saving ? "Saving…" : "Add post"}</button>
      </form>
      <ul>
        {posts.map((p) => (
          <li key={p.id}>{p.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

What it does: After the POST succeeds, the returned post is **prepended to the local state** — the list updates instantly with no second fetch. The button shows its own mini loading state ("Saving…") and disables itself against double-submits — the small professional touches real apps are made of.

What you will see: type a title, press Add — "Saving…" flashes, then the new post appears at the top of the list, and the input clears for the next one.

Quick recap

- Every API screen is the same skeleton: **data / loading / error state + fetch + the three renders**.

- `fetch` is two steps with a manual `res.ok` gate; **POST** needs method + JSON header + `JSON.stringify` body, all three.
- After a successful POST, **update local state** with the server's answer instead of refetching the world.

Common mistakes

1. **Forgetting `await res.json()`** — the component renders a Promise where data should be, or crashes reading fields of it. The fix: two awaits, always — response first, body second.
2. **Trusting fetch to throw on 404** — it does not; the error JSON body gets treated as your data and the screen shows garbage or crashes. The fix: `if (!res.ok) throw …` before any `.json()` you trust.
3. **Fetching in the render body** — a fetch written directly in the component runs on *every* render, and its setState triggers another render: an infinite request storm. The fix: fetches live in effects (for loading) or handlers (for actions) — never loose in the render path.

Practice

1. Build a `TodoLoader` on the practice API (`/todos?_limit=8`): render each todo with its title and a ✓/○ for `completed`. Add a refresh button that re-runs the load (extract the load function so both the effect and the button call it), with the loading state showing during refreshes too.
2. Extend Example 3: give each listed post a Delete button that removes it from state (practice APIs accept DELETE but forget instantly — your job is the local state removal + a `deleting` indicator on that row while the call runs).

Practice quiz

1. What state does every API screen in React keep?

- Only data
- The trio: data, loading, error, and it renders one of the three states
- Only loading
- Data and error, loading is automatic

Answer: The trio: data, loading, error, and it renders one of the three states

2. Why does fetch need a res.ok check in React too?

- It does not, fetch throws on 404
- fetch does not throw on a 404 or 500, it only fails on network problems, so check if (!res.ok) throw before trusting the body
- res.ok is only for POST requests
- React checks it automatically

Answer: fetch does not throw on a 404 or 500, it only fails on network problems, so check if (!res.ok) throw before trusting the body

3. What three parts must a POST fetch have?

- Only the URL and the body
- method: "POST", headers with Content-Type: application/json, and body: JSON.stringify(obj)
- Only headers
- A GET method with a body

Answer: method: "POST", headers with Content-Type: application/json, and body: JSON.stringify(obj)

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/react-api-integration · More free tools and notes at https://scopetreasure.com