Fetch API + Real API Project

Real data from a real API: fetch, the two-step response, building User Cards from jsonplaceholder — with loading and error states done properly.

Introduction

Until now, your code made its own data — arrays and objects typed by hand. Real apps do not work like that. They **ask a server** for data, and the server sends it back as **JSON** — the format from Lesson 3.18. The asking tool is `fetch`.

In this lesson you learn the two-step response, the success check beginners skip, and you build a real project: a page that fetches real users from a real API and shows each one as a card — loading, success and error states all done properly. *(Roman Urdu: fetch server se asal data mangta hai — data JSON mein aata hai, aur hum usay page par cards bana kar dikhate hain.)*

Key points

- **`fetch(url)` returns a promise** — calling it sends the request and gives you a promise at once. The page never freezes while the server answers.
- **The response arrives in two steps** — step one: the **response object** (status and headers, the envelope). Step two: **`response.json()`** reads the body and turns the JSON text into real JavaScript data. `response.json()` is itself a promise, so you `await` it too — or return it into the next `.then`.
  (Roman Urdu: Pehle response ka lifafa milta hai, phir response.json() andar ka asal data deta hai — do steps, dono par await.)
- **`response.ok` tells you if it worked** — `ok` is `true` for success statuses (200–299) and `false` for failures like 404 or 500.
- **A 404 still resolves** — `fetch` only rejects when the network itself fails (no internet, server unreachable). A "page not found" answer is still an answer. Checking `response.ok` — and throwing your own error when it is false — is **your** job.
- **A real app has three states** — **loading** (show a waiting message the moment the request starts), **success** (show the data), **error** (show an honest failure message). Professional code plans all three before it runs; Lesson 3.21's `async` function plus `try`/`catch` is the toolkit that does it.
- **JSON is the travelling format both ways** — data comes in as JSON and you read it with `response.json()`; data you **send** must be turned into JSON text with `JSON.stringify`.
  (Roman Urdu: Data JSON mein aata hai aur JSON mein hi jata hai — bhejne se pehle JSON.stringify lazmi hai.)
- **Sending data uses options** — `fetch(url, { method: "POST", headers: { "Content-Type": "application/json" }, body: JSON.stringify(data) })` sends instead of only reading. `GET` is the default and needs no options at all.

Code examples

Example 1 — The two-step fetch, one user

```
async function loadOneUser() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/users/1");

    if (!response.ok) {
      throw new Error("Request failed with status " + response.status);
    }

    const user = await response.json();

    console.log("Name: " + user.name);
    console.log("Email: " + user.email);
    console.log("City: " + user.address.city);
  } catch (error) {
    console.log("Could not load the user: " + error.message);
  }
}

loadOneUser();
```

What it does: It fetches one real user from the jsonplaceholder test API. First `await` gets the response envelope and checks `response.ok`. Second `await` — `response.json()` — turns the body into an object, and the code reads the name, email and city out of it, including the city nested inside `address`.

What you will see: after a short wait for the network (usually under a second), three lines in the console: `Name: Leanne Graham`, then `Email: Sincere@april.biz`, then `City: Gwenborough`. That is real data from a real server, not anything typed into the file. Disconnect the internet and run it again: instead you see one line — `Could not load the user:` followed by the browser's network error message — and the program survives.

Example 2 — The User Cards project: loading, cards, error

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>User Cards — Code Scope</title>
  <style>
    body { font-family: Arial, sans-serif; background: #faf9f6; color: #0b0f19; padding: 20px;
}
    #users { display: flex; flex-wrap: wrap; gap: 12px; }
    .card { background: white; border: 1px solid #ddd; border-radius: 10px; padding: 14px;
width: 230px; }
    .card h3 { margin: 0 0 6px; }
    .card p { margin: 4px 0; font-size: 14px; }
  </style>
</head>
<body>
  <h1>Our Users</h1>
  <p id="status">Loading users...</p>
  <div id="users"></div>

  <script>
    const statusEl = document.getElementById("status");
    const usersEl = document.getElementById("users");

    async function loadUsers() {
      try {
        const response = await fetch("https://jsonplaceholder.typicode.com/users");

        if (!response.ok) {
          throw new Error("Request failed with status " + response.status);
        }

        const users = await response.json();

        statusEl.textContent = "";
        usersEl.innerHTML = users.map((user) => `
          <div class="card">
            <h3>${user.name}</h3>
            <p>Email: ${user.email}</p>
            <p>City: ${user.address.city}</p>
          </div>
        `).join("");
      } catch (error) {
```

```
        statusEl.textContent = "Sorry, the users could not be loaded. Check your internet and
try again.";
        console.log("Error: " + error.message);
      }
    }

    loadUsers();
  </script>
</body>
</html>
```

What it does: This is the complete project page. The status paragraph starts as `Loading users...` — the loading state is on screen before the request even finishes. On success, the loading text is cleared and one card per user is built from the data with `map` and `join`, showing name, email and city. On any failure, `catch` replaces the loading text with an honest error message instead of leaving a dead page.

What you will see: the page opens with the heading `Our Users` and the text `Loading users...` underneath. After a short wait, the loading text disappears and 10 cards appear in a row-wrapping grid — the first card reads `Leanne Graham`, `Email: Sincere@april.biz`, `City: Gwenborough`, and the others follow with their own real names, emails and cities. Turn the internet off and reload: no cards appear, and the page shows `Sorry, the users could not be loaded. Check your internet and try again.` — all three states, working.

Example 3 — A POST glimpse: sending data

```
async function createPost() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/posts", {
      method: "POST",
      headers: {
        "Content-Type": "application/json"
      },
      body: JSON.stringify({
        title: "My first post",
        body: "Learning fetch in JavaScript",
        userId: 1
      })
    });

    if (!response.ok) {
      throw new Error("Request failed with status " + response.status);
    }

    const result = await response.json();
    console.log("Created post with id: " + result.id);
    console.log("Title: " + result.title);
  } catch (error) {
    console.log("Could not create the post: " + error.message);
  }
}

createPost();
```

What it does: Instead of only reading, this sends a new post to the server. The second argument of `fetch` is the options object: `method: "POST"` says "I am sending", the header says the body is JSON, and `JSON.stringify` turns the data object into the JSON text the body must be. The test API pretends to save it and answers with the post, including its new id.

What you will see: after a short wait, two lines in the console: `Created post with id: 101` then `Title: My first post`. The id 101 is the server's own answer — proof the data travelled out, not just in.

Quick recap

- **`fetch(url)`** returns a promise; the answer comes in **two steps** — the response object first, then **`await response.json()`** for the real data.
- A 404 still resolves, so always check **`response.ok`** and throw if it is false; `fetch` itself only rejects on network failure.

- Real apps show **three states** — loading, success, error — and sending data means `method: "POST"` with a `JSON.stringify` body.

Common mistakes

1. **Using the response as if it were the data** — logging `response` shows the envelope, not your users; the body is still unread text. The fix: always take step two — `const data = await response.json();`.
2. **Skipping the `response.ok` check** — a wrong URL gives a neat 404 response, your code treats it as success, and the page breaks later in a confusing place. The fix: `if (!response.ok) { throw new Error("Request failed with status " + response.status); }` right after every fetch.
3. **Forgetting `JSON.stringify` in a POST body** — sending a raw object makes the body `[object Object]`, and the server receives garbage. The fix: the body is always text — `body: JSON.stringify(data)` — with the `Content-Type: application/json` header to say what it is.

Practice

1. Change Example 2 to also show each user's **phone** on the card, under the city. Reload and confirm all 10 cards show a phone line with real data from the response.
2. Write a small page with one button: `Load a Random Post`. On click, fetch post number 5 from the same test API (`/posts/5`), show its title and body on the page, show `Loading...` while waiting, and show an error message if the fetch fails. Use an `async` function with `try` / `catch`, exactly like the project.

Practice quiz

1. What does fetch(url) give you, and in how many steps does the data arrive?

- The data at once, in one step
- A promise at once; step one is the response object, step two is await response.json() for the real data
- Nothing until the server closes the connection
- A string, never a promise

Answer: A promise at once; step one is the response object, step two is await response.json() for the real data

2. What happens when fetch gets a 404 answer?

- fetch rejects, like a network failure
- fetch still resolves, a 404 is still an answer, so you must check response.ok yourself and throw if it is false
- fetch retries automatically
- fetch returns null

Answer: fetch still resolves, a 404 is still an answer, so you must check response.ok yourself and throw if it is false

3. What three parts must a POST fetch have?

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

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

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/js-fetch-api · More free tools and notes at https://scopetreasure.com