AJAX, Fetch & Consuming REST APIs

Web Technologies — Lecture 7

Masoud Hamad

State University of Zanzibar (SUZA)

Semester II, 2025/2026

Outline

Why AJAX?

AJAX = Asynchronous JavaScript and XML (now mostly JSON).

- Pre-2005: every interaction reloaded the entire page
- AJAX (popularised 2005, Google Maps, Gmail) lets JS request data *without* reloading
- Page stays alive; only changed parts re-render
- Foundation of every modern web app

HTTP Methods (Verbs)

- **GET** — retrieve data, safe & idempotent, no body
- **POST** — create new resource, has body
- **PUT** — replace resource entirely, idempotent
- **PATCH** — partial update
- **DELETE** — remove resource

Status codes you'll see: 200 OK, 201 Created, 204 No Content, 301 Moved, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Server Error.

JSON — The Lingua Franca

```
{
  "id": 42,
  "name": "Amina Said",
  "courses": ["IT4001", "IT4002"],
  "active": true,
  "address": null
}
```

JS conversions:

```
JSON.stringify(obj);    // object -> string (to send)
JSON.parse(text);       // string -> object (after receiving)
```

GET Request

```
async function loadStudents() {
  const res = await fetch("/api/students");
  if (!res.ok) throw new Error(`HTTP ${res.status}`);
  const students = await res.json();
  renderList(students);
}
```

Note: `fetch` only rejects on network errors. HTTP 404/500 still resolves — always check `res.ok`.

POST Request with JSON

```
async function createStudent(data) {
  const res = await fetch("/api/students", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(data)
  });
  if (!res.ok) throw new Error(`HTTP ${res.status}`);
  return res.json();          // server typically returns the created object
}

createStudent({ name: "Hassan", course: "BSc CS" });
```

PUT, PATCH, DELETE

```
// Replace
await fetch(`/api/students/${id}`, {
  method: "PUT",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify(student)
});

// Partial update
await fetch(`/api/students/${id}`, {
  method: "PATCH",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ year: 3 })
});

// Delete
await fetch(`/api/students/${id}`, { method: "DELETE" });
```

Headers, Auth, Query Params

```
const params = new URLSearchParams({ q: "amin", limit: 10 });
const res = await fetch(`/api/search?${params}`, {
  headers: {
    "Authorization": `Bearer ${token}`,
    "Accept": "application/json"
  }
});
```

Loading / Success / Error / Empty

```
async function load() {
  setState({ status: "loading" });
  try {
    const res = await fetch("/api/items");
    if (!res.ok) throw new Error(`Server ${res.status}`);
    const items = await res.json();
    setState({ status: items.length ? "success" : "empty", items });
  } catch (err) {
    setState({ status: "error", message: err.message });
  }
}
```

Always render **four UI states**: loading, success, empty, error.

CORS — Cross-Origin Resource Sharing

Same-origin policy blocks JS from reading responses from a different origin.

- Different origin = different scheme, host, or port
- The *server* must opt-in by sending CORS headers:
- `Access-Control-Allow-Origin: https://app.example.com`
- Browsers send a **preflight OPTIONS** request for non-simple methods

During development, you'll often configure your backend to allow `http://localhost:5173` (or similar).

Centralise Your API Client

```
// api.js
const BASE = "/api";

export async function api(path, options = {}) {
  const res = await fetch(BASE + path, {
    headers: {
      "Content-Type": "application/json",
      ...(options.headers || {})
    },
    ...options
  });
  if (!res.ok) throw new Error(await res.text());
  return res.status === 204 ? null : res.json();
}

// usage
import { api } from "./api.js";
const students = await api("/students");
const created  = await api("/students", { method: "POST", body: JSON.stringify(s) })
    ;
```

Rate Limiting, Retry, Cache

- **Debounce** search inputs (e.g., wait 300ms after typing)
- **Retry** transient failures with exponential backoff
- **Cache** GET responses in `localStorage` or `IndexedDB`
- Watch for HTTP **429 Too Many Requests** from the server
- Show optimistic UI updates — but roll back on failure

Summary

- Fetch API replaces `XMLHttpRequest` for AJAX
- Send/receive JSON; check `res.ok`; handle errors gracefully
- Render four UI states: loading, success, empty, error
- CORS lets servers selectively allow cross-origin access
- Centralise HTTP logic in a small client module

Next: React — the modern frontend framework.

Questions?