**Seminarul 7 — Fetch API + formulare + JSON + persistență minimă  
Partea 1: Teorie (extins)**

**Hook realist: înscriere la hackathon în câteva ore, doar cu „vanilla” web**

Dimineața, ora 10:00. La ora 16:00 are loc lansarea înscrierilor pentru hackathonul organizat de „Clubs & Associations Hub”. Ți se cere o pagină publică ce listează cluburile disponibile (sursă: un API simplu) și un formular prin care studenții se pot înscrie. Cerințele par „banale”: validezi câmpuri (nume, e‑mail, acord), trimiți \*\*JSON\*\* la server, afișezi mesaje de succes sau eroare și salvezi local progresul dacă pică rețeaua. Nu ai voie framework‑uri; lucrezi cu \*\*HTML\*\* semantic, \*\*CSS\*\* moderat, \*\*JavaScript\*\* modern și \*\*Fetch API\*\*. Persistența este „minimală”: \*\*localStorage\*\* pentru cache de listă și „draft autosave” pentru formular; eventual, un mic \*\*mock server\*\* (Express) în laborator. Timpul este scurt, greșelile tipice (validare ascunsă, mesaje criptice, blocaje UI, lipsa time‑out‑ului) se plătesc scump. Acest seminar este planul tău de zbor.

**Hartă mentală: dincolo de „fac un fetch și gata”**

Vom lucra în patru planuri: (1) \*\*Semantica HTTP\*\* (metode, status codes, antete, reprezentări), (2) \*\*Fetch API\*\* ca interfață promițătoare peste HTTP (cerere, răspuns, corp, erori, CORS, oprire/timeout), (3) \*\*Formular\*\* (semnatică HTML, validare nativă și extinsă, serializare spre JSON, accesibilitate) și (4) \*\*Persistență minimă\*\* (local cache, autosave, reconcilieri simple). Toate sunt legate printr‑un fir roșu: proiectezi interacțiunea ca pe o \*\*mașină de stări\*\*: `idle → loading → success | error` și documentezi decizii (praguri, retry policy, TTL).

**1. Fundamente HTTP pentru dezvoltatorii de interfețe**

\*\*HTTP\*\* este protocolul conversației dintre aplicația ta și server. Ca în corespondența clasică, există un \*\*plic\*\* (antete) și un \*\*conținut\*\* (body, de regulă JSON). Câteva noțiuni ancore:  
• \*\*Metode și proprietăți\*\*: `GET` (safe, idempotent), `HEAD` (ca `GET` fără body), `POST` (nu este idempotent), `PUT` (idempotent), `DELETE` (idempotent în sens larg), `PATCH` (modificare parțială). Idempotent înseamnă că repetarea aceleiași cereri produce același efect (util pentru \*\*retry\*\*).  
• \*\*Status codes\*\*: 2xx (succes), 3xx (redirijare), 4xx (eroare la client), 5xx (eroare la server). Granularități utile: `201 Created` (POST reușit), `204 No Content` (fără body), `400 Bad Request` (input invalid), `401 Unauthorized` (lipsă/invalid credențiale), `403 Forbidden` (ai credențiale, dar nu drepturi), `404 Not Found` (resursa lipsește), `409 Conflict` (stare concurentă), `422 Unprocessable Content` (date valide sintactic, dar invalide semantic), `429 Too Many Requests` (rate‑limit), `500/502/503/504` (server).  
• \*\*Antete\*\* relevante: `Content-Type: application/json; charset=utf-8`, `Accept: application/json`, `Authorization` (dacă există), `Cache-Control`, `ETag/If-None-Match`, `Last-Modified/If-Modified-Since`.  
• \*\*Reprezentări & negociere\*\*: clientul poate cere un anumit tip (`Accept`), iar serverul livrează cea mai potrivită reprezentare.  
• \*\*Timp și date\*\*: momentul feedback‑ului către utilizator trebuie să corespundă semnificației codurilor (de ex., `422` → „Verifică câmpul X”).

**2. Fetch API: cerere, răspuns, corp, erori, CORS, oprire**

\*\*Fetch API\*\* oferă o interfață modernă peste HTTP. Îți dă obiecte `Request` și `Response`, `Headers`, support pentru `Promise` și `async/await`. Câteva idei de design:  
• \*\*Citirea răspunsului\*\*: `Response.ok` indică 2xx, `response.json()` parsează corpul (atenție: poate arunca dacă serverul nu trimite JSON valid).  
• \*\*Erori de rețea\*\* vs. \*\*răspunsuri eronate\*\*: `fetch` aruncă doar pe erori de rețea; codurile 4xx/5xx nu aruncă automat – tu trebuie să verifici `ok`/`status`.  
• \*\*Body types\*\*: JSON (`stringify`/`json()`), `FormData`, `Blob`, `ArrayBuffer`, `ReadableStream` (streaming).  
• \*\*CORS\*\*: cererile cross‑origin respectă politica de origine (Origin/Same‑Origin). Anumite cereri declanșează \*\*preflight\*\* (OPTIONS) dacă nu sunt „simple”. Răspunsul serverului trebuie să includă antete `Access-Control-Allow-\*` adecvate.  
• \*\*Credențiale\*\*: opțiunea `credentials: 'include'` trimite cookies; `same-origin` le trimite doar pe același origin; `omit` nu trimite.  
• \*\*Oprire/timeout\*\*: `AbortController` îți permite să anulezi un `fetch` (de exemplu la \*\*timeout\*\* sau când utilizatorul schimbă pagina).  
• \*\*Retry\*\*: nu e integrat; îl compui singur, atent la \*\*idempotency\*\* (nu reîncerci orbește `POST` fără garanții).

// GET JSON (defensiv)  
async function getJSON(url, opts = {}){  
 const res = await fetch(url, { headers: { 'Accept': 'application/json' }, ...opts });  
 const contentType = res.headers.get('content-type') || '';  
 const isJSON = contentType.includes('application/json');  
 if(!res.ok){  
 let payload = null;  
 if(isJSON){ try{ payload = await res.json(); } catch {} }  
 const err = new Error(`HTTP ${res.status}`);  
 err.status = res.status; err.payload = payload;  
 throw err;  
 }  
 if(!isJSON) throw new Error('Unexpected content-type');  
 return res.json();  
}

Exemplul de mai sus separă două faze: (1) acceptă doar `application/json`, (2) tratează explicit cazul `!ok` (4xx/5xx) și încearcă să extragă un payload de eroare (de tip `application/problem+json`, dacă serverul îl oferă).

**3. Formularul corect: semnatică HTML, validare nativă & extinsă, accesibilitate**

Un \*\*formular\*\* bun începe cu HTML semantic: `<form>`, `<label for>`, controale cu `name`, ajutoare `aria-describedby`, butoane de tip `submit`. Browserul oferă \*\*Constraint Validation API\*\*: `required`, `type=email`, `minlength`, `pattern`, `setCustomValidity`. Principiul este \*\*progressive enhancement\*\*: chiar dacă JS e dezactivat, `action`/`method` pot trimite date; cu JS activ, previi comportamentul implicit, colectezi datele, creezi \*\*JSON\*\* și trimiți prin \*\*Fetch\*\*; UI‑ul devine mai bogat (stări, mesaje, spinner).

<!-- Fragment de formular semantic -->  
<form id="enrol" action="/fallback" method="post" novalidate aria-describedby="form-status">  
 <div class="field">  
 <label for="fullName">Nume complet</label>  
 <input id="fullName" name="fullName" required minlength="3" autocomplete="name" />  
 </div>  
 <div class="field">  
 <label for="email">Email instituțional</label>  
 <input id="email" name="email" type="email" required inputmode="email" autocomplete="email" />  
 </div>  
 <div class="field">  
 <label for="club">Club</label>  
 <select id="club" name="club" required></select>  
 </div>  
 <div class="field">  
 <label><input id="consent" name="consent" type="checkbox" required /> Accept prelucrarea datelor</label>  
 </div>  
 <button type="submit">Înscrie‑mă</button>  
 <p id="form-status" class="u-visually-hidden" aria-live="polite"></p>  
</form>

Atributele `required`, `type=email`, `minlength` oferă validare nativă; `aria-live` anunță mesaje dinamice. `novalidate` îți dă control programatic (dar nu dezactivează permanent regulile, ci prevenirea pop‑up‑ului nativ).

// FormData → JSON (și validare custom)  
function formDataToJSON(form){  
 const fd = new FormData(form);  
 const data = Object.fromEntries(fd.entries());  
 // Custom rules  
 const errors = {};  
 if((data.fullName||'').trim().length < 3) errors.fullName = 'Numele trebuie să aibă cel puțin 3 caractere.';  
 if(!/^[^@\s]+@[^@\s]+\.[^@\s]+$/.test(data.email||'')) errors.email = 'Email invalid.';  
 if(!data.consent) errors.consent = 'Este necesar consimțământul.';  
 return { ok: Object.keys(errors).length===0, data, errors };  
}

Separarea validării în funcții pure („\*\*data in → verdict out\*\*”) îți permite \*\*testare unitară\*\* fără DOM și îmbunătățește lizibilitatea. Pentru accesibilitate, primul câmp invalid primește \*\*focus\*\*, iar mesajele sunt anunțate prin `aria-live`.

**4. JSON: modelarea datelor, convenții, parsing defensiv**

\*\*JSON\*\* este formatul de schimb preferat. Convențiile sănătoase includ:  
• \*\*Chei în engleză\*\* (snake\_case sau camelCase consecvent), `id` numeric/string stabil, `createdAt`/`updatedAt` în \*\*ISO‑8601\*\* (RFC 3339).   
• \*\*Enumerări\*\*: valori string clare (`"pending"`, `"approved"`), nu numere magice; documentează „contractul” în README.  
• \*\*Colecții\*\*: liste de obiecte, nu dicționare cu chei dinamice, decât dacă modelul o cere.  
• \*\*Parsing defensiv\*\*: `try { await res.json() } catch { … }`; nu presupune că serverul răspunde mereu cu JSON valid.  
• \*\*`application/problem+json`\*\* (RFC 7807): un model standardizat pentru erori (câmpuri `type`, `title`, `status`, `detail`).

// Parsing defensiv + model minimal de eroare RFC 7807  
async function parseProblem(res){  
 const ct = res.headers.get('content-type') || '';  
 if(!ct.includes('application/json')) return null;  
 try{  
 const data = await res.json();  
 if(data && typeof data === 'object' && ('title' in data || 'detail' in data)) return data;  
 }catch{}  
 return null;  
}

Standardizarea erorilor scade ambiguitatea în UI (poți afișa `problem.title`/`problem.detail` pentru 4xx/5xx).

**5. Persistență minimă: localStorage, sessionStorage, micro‑cache și „draft autosave”**

Persistența „client‑side” are două direcții:  
(1) \*\*Cache pentru citire\*\* (de ex., `GET /api/clubs`), pentru reducerea latenței și protecție la intermitențe;   
(2) \*\*Draft autosave\*\* pentru formular (îți salvezi progresul la input, îl recuperezi la refresh).   
\*\*localStorage\*\* este un KV sincron, per origin; \*\*sessionStorage\*\* este per tab. Pentru payload‑uri mari sau structură complexă, \*\*IndexedDB\*\* e mai potrivit (îl menționăm, detaliem în proiectele avansate).

// localStorage cache cu TTL (mecanism simplu)  
const cacheNS = 's7:cache:';  
function lsGet(key){  
 try{ const raw = localStorage.getItem(cacheNS+key); if(!raw) return null;  
 const { value, expiresAt } = JSON.parse(raw);  
 if(expiresAt && Date.now() > expiresAt){ localStorage.removeItem(cacheNS+key); return null; }  
 return value;  
 }catch{ return null; }  
}  
function lsSet(key, value, ttlMs){  
 const payload = { value, expiresAt: ttlMs ? Date.now()+ttlMs : null };  
 localStorage.setItem(cacheNS+key, JSON.stringify(payload));  
}

// „Stale‑while‑revalidate” minimal pentru GET /api/clubs  
async function getClubs(){  
 const cached = lsGet('clubs');  
 if(cached){ // livrăm imediat „stale”  
 setTimeout(()=> refreshClubs(), 0);  
 return cached;  
 }  
 return await refreshClubs();  
}  
async function refreshClubs(){  
 const data = await getJSON('/api/clubs');  
 lsSet('clubs', data, 5\*60\*1000); // 5 minute TTL  
 return data;  
}

„Stale‑while‑revalidate” asigură \*\*timp de reacție\*\* mic (afişezi cache‑ul), dar sincronizezi în fundal; la succes, UI se actualizează. TTL‑ul se alege în funcție de dinamica datelor.„Draft autosave” pe formular reține câmpurile și îți permite o revenire fără pierdere de progres.

// Draft autosave (debounce) & restore  
const draftKey = 's7:draft:enrol';  
let t = null;  
function saveDraft(form){  
 clearTimeout(t);  
 t = setTimeout(()=>{  
 const fd = new FormData(form);  
 const data = Object.fromEntries(fd.entries());  
 try{ localStorage.setItem(draftKey, JSON.stringify(data)); }catch{}  
 }, 300);  
}  
function restoreDraft(form){  
 try{  
 const raw = localStorage.getItem(draftKey); if(!raw) return;  
 const data = JSON.parse(raw)||{};  
 for(const [k,v] of Object.entries(data)){  
 const el = form.elements.namedItem(k); if(!el) continue;  
 if(el instanceof RadioNodeList || el.type === 'checkbox'){ el.checked = !!v; }  
 else el.value = v;  
 }  
 }catch{}  
}

**6. Erori, UX și observabilitate: de la etichete la mașini de stări**

Utilizatorul are nevoie de \*\*mesaje clare\*\* și \*\*stări previzibile\*\*. Propune‑ți un „contract” UI: butonul `submit` intră în starea \*\*disabled\*\* când trimiți; înlocuiești textul cu „Se trimite…”, anunți `aria-live` la succes/eroare, returnezi focusul pe primul câmp invalid. \*\*Erorile\*\* se împart în: (a) de rețea/time‑out (îi spui „Verifică conexiunea” și oferi „Încearcă din nou”), (b) 4xx (probleme de input; marcarea câmpurilor), (c) 5xx (probleme server; „Încearcă mai târziu”). Pentru \*\*observabilitate\*\*, loghezi minimul non‑PII: `level`, `code`, `when`, `endpoint`, `durationMs`.

// Mașină de stări simplă pentru submit  
const state = { status: 'idle' }; // 'idle'|'submitting'|'success'|'error'  
async function submitJSON(url, data, onChange){  
 state.status = 'submitting'; onChange(state);  
 const t0 = performance.now();  
 try{  
 const res = await fetch(url, {  
 method: 'POST',  
 headers: { 'Content-Type': 'application/json', 'Accept': 'application/json' },  
 body: JSON.stringify(data)  
 });  
 const dt = performance.now()-t0;  
 if(!res.ok){ const problem = await (async()=>{ try{ return await res.json(); }catch{return null;} })();  
 state.status = 'error'; state.problem = problem || { title: `HTTP ${res.status}` }; state.duration = dt; onChange(state); return;  
 }  
 state.status = 'success'; state.duration = dt; onChange(state);  
 }catch(err){  
 state.status = 'error'; state.problem = { title: 'Network error/timeout' }; onChange(state);  
 }  
}

**7. Securitate & confidențialitate: XSS, CSRF, CORS, PII**

În aplicațiile client‑side, \*\*XSS\*\* este riscul dominant: \*\*nu\*\* insera HTML din date neîncredere; folosește `textContent`. Evită să stochezi \*\*tokens\*\* sensibile în `localStorage` (recuperabile de scripturi rău‑voitoare); preferă \*\*cookies\*\* `HttpOnly` + `SameSite` (`Lax/Strict`) în arhitecturi adecvate. \*\*CSRF\*\* este atenuat de `SameSite`, dar înțelege că nu este „silver bullet” (depinde de flow‑ul aplicației). \*\*CORS\*\* este un mecanism server‑side: dacă folosești un mock, configurează antetele `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods` și `Access-Control-Allow-Headers` potrivite; pe client, setezi `mode:'cors'` și `credentials` doar când este nevoie. Respectă \*\*principiul minimizării datelor\*\* (GDPR): colectezi strictul necesar pentru înscriere, specifici scopul, asiguri ștergerea draft‑urilor la `success`.

// Inserare sigură (XSS safe)  
function renderClubName(el, name){  
 el.textContent = name ?? ''; // NU innerHTML cu surse neîncredere  
}

**8. Timeouts, abort și retry cu backoff: de la teorie la practică**

O cerere care pare „înghețată” este o experiență degradantă. Setează un \*\*timeout realist\*\* (ex. 6–8 s) și oprește cererea cu \*\*AbortController\*\*. \*\*Retry\*\* are sens doar pentru erori tranzitorii (rețea, `5xx`), nu pentru `4xx`. Folosește \*\*exponential backoff\*\* (ex. 200 ms, 400 ms, 800 ms) + \*\*jitter\*\* pentru a evita sincronizarea nedorită a mai multor clienți. La `POST` fii atent la \*\*idempotency\*\*: fără un `idempotency-key` pe server, repetarea poate crea duplicate; în laborator, poți agrega o „barieră locală” (nu mai trimiți cu același hash de payload în 30 s).

// Timeout + retry(3) cu backoff + jitter  
async function fetchWithRetry(url, opts = {}, { retries = 3, timeoutMs = 8000 } = {}){  
 let attempt = 0;  
 while(true){  
 const ac = new AbortController();  
 const to = setTimeout(()=> ac.abort(), timeoutMs);  
 try{  
 const res = await fetch(url, { signal: ac.signal, ...opts });  
 clearTimeout(to);  
 if(!res.ok){  
 // retry doar pentru 5xx  
 if(res.status >= 500 && res.status < 600 && attempt < retries){ attempt++; await waitBackoff(attempt); continue; }  
 return res;  
 }  
 return res;  
 }catch(err){  
 clearTimeout(to);  
 if(attempt < retries){ attempt++; await waitBackoff(attempt); continue; }  
 throw err;  
 }  
 }  
}  
function jitter(base){ return base \* (0.8 + Math.random()\*0.4); }  
function waitBackoff(attempt){  
 const base = Math.min(1000 \* (2 \*\* (attempt-1)), 8000);  
 return new Promise(r => setTimeout(r, jitter(base)));  
}

**9. Caching & validare condițională: ETag, 304, Cache‑Control (concept)**

Deși în laborator nu configurăm server caching avansat, trebuie cunoscută esența: serverul poate trimite \*\*ETag\*\*; clientul include `If-None-Match` la `GET`; la conținut neschimbat, serverul răspunde `304 Not Modified` (fără body). `Cache-Control` (de ex., `max-age`, `no-store`, `must-revalidate`) guvernează comportamentul cache‑urilor intermediare. Pentru UI‑uri cu „date viu schimbătoare” și fără control asupra serverului, recurgem la \*\*TTL + SWR\*\* pe client (ca mai sus).

**10. Accesibilitate: aria‑live, focus management, stări vizibile**

Pe lângă semantica etichetelor și atributele `required`/`type`, respectă:  
• \*\*Mesaje live\*\*: zona `aria-live="polite"` pentru statusuri de rețea/validare;   
• \*\*Focus\*\*: la eroare, mută focusul pe primul câmp invalid; butonul de submit nu trebuie să piardă focusul „fără sens”;  
• \*\*Contrast & feedback\*\*: diferențiază clar stările (`disabled`, `loading`, `error`, `success`) cromatic și textual;   
• \*\*Text alternativ\*\* la imagini/icoane.   
Accesibilitatea nu este „extra”, ea reduce costul cognitiv și crește încrederea.

**11. Anti‑pattern‑uri frecvente (și remedii)**

1) \*\*Validare doar în JS\*\*: dacă JS cade, formularul devine „permisiv” → folosește semantica HTML + server fallback; JS doar îmbogățește.  
2) \*\*`innerHTML` din date\*\*: este vector XSS → folosește `textContent`; ornează cu CSS, nu cu „HTML injectat”.  
3) \*\*Fără timeout/abort\*\*: UI‑ul pare blocat → setează `AbortController` + un mesaj clar; oferă „Încearcă din nou”.  
4) \*\*Retry pe 4xx\*\*: nu are sens → arată feedback clar și marchează câmpurile; reîncercările se aplică doar la erori tranzitorii.  
5) \*\*Stocare token în localStorage\*\*: vulnerabil la XSS → preferă cookie `HttpOnly` + `SameSite` (în arhitecturile potrivite).  
6) \*\*Mesaje tehnice\*\* („NetworkError when attempting to fetch resource”) către utilizator → mapează în limbaj natural și acțiuni (verifică rețeaua, încearcă ulterior, contactează suport).  
7) \*\*TTL arbitrar\*\*: 24 h pentru date volatile → măsoară și setează TTL-uri scurte (ex. 5 min) + SWR.

**12. AI‑assist (Copilot/LLM) — Very Short Loop (VSL) în practică**

\*\*Scopul\*\* nu este să „delegăm” gândirea, ci să accelerăm pașii repetitivi și să verificăm alternative. Practicăm bucle \*\*foarte scurte\*\*:  
• „Generează un `fetchClient` cu `timeout + retry(3)` și helperi `json()` care aruncă pe `!res.ok` (separă network vs HTTP).”  
• „Scrie validatori puri pentru nume/email/consent; returnează `{ok,data,errors}`; include 3 teste unitare.”  
• „Propune `localStorageCache` cu TTL și `staleWhileRevalidate` minimal.”  
• „Generează `aria-live` messages și focus management la prima eroare.”  
După generare, \*\*curățăm\*\*: nume consistente, separare de responsabilități, teste curate, comentarii operaționale (nu romane).

**13. Checklist teoretic — la finalul Părții 1 ar trebui să poți:**

• Formula cereri cu \*\*Fetch API\*\* (GET/POST JSON, antete corecte), separând \*\*network errors\*\* de \*\*HTTP errors\*\*.  
• Implementa \*\*timeout/abort\*\* și \*\*retry\*\* cu backoff + jitter, respectând \*\*idempotency\*\*.  
• Proiecta formulare semantice cu \*\*Constraint Validation API\*\* + validare custom, cu \*\*aria‑live\*\* și focus management.  
• Modela \*\*JSON\*\* (chei, date, enumerări), face \*\*parsing defensiv\*\* și folosi \*\*application/problem+json\*\*.  
• Aplica \*\*persistență minimă\*\*: `localStorage` TTL + \*\*SWR\*\*, \*\*draft autosave\*\* (debounce) și curățare la succes.  
• Respecta \*\*accesibilitatea\*\* (vizibilitate stări, contrast, mesaje clar formulate) și \*\*securitatea\*\* (evitarea XSS, atent la CORS/CSRF).  
• Evita anti‑pattern‑urile tipice și utiliza \*\*AI‑assist\*\* în \*\*bucle scurte\*\* pentru cod repetitiv și teste.

**Referințe (APA 7, cu DOI real, plus standarde normative)**

Fielding, R., Nottingham, M., & Reschke, J. (2022). HTTP Semantics (RFC 9110). RFC Editor. https://doi.org/10.17487/RFC9110

Fielding, R., Nottingham, M., & Reschke, J. (2022). HTTP Caching (RFC 9111). RFC Editor. https://doi.org/10.17487/RFC9111

Bray, T. (2017). The JavaScript Object Notation (JSON) Data Interchange Format (RFC 8259). RFC Editor. https://doi.org/10.17487/RFC8259

Barth, A. (2011). The Web Origin Concept (RFC 6454). RFC Editor. https://doi.org/10.17487/RFC6454

Nottingham, M., & Wilde, E. (2016). Problem Details for HTTP APIs (RFC 7807). RFC Editor. https://doi.org/10.17487/RFC7807

Kawamura, A., & Thomson, M. (2012). Additional HTTP Status Codes (RFC 6585). RFC Editor. https://doi.org/10.17487/RFC6585

Klyne, G., & Newman, C. (2002). Date and Time on the Internet: Timestamps (RFC 3339). RFC Editor. https://doi.org/10.17487/RFC3339

WHATWG. (Living Standard). Fetch Standard. https://fetch.spec.whatwg.org/ (resursă normativă, fără DOI)

W3C. (Living Standard). HTML Standard — Forms & Constraint Validation. https://html.spec.whatwg.org/ (resursă normativă, fără DOI)

W3C. (Recommendation). Indexed Database API (IndexedDB). https://www.w3.org/TR/IndexedDB-3/ (resursă normativă, fără DOI)