CURS 6
Fetch API & REST APIs

6.1 Ce este un REST API?

Un REST API (Representational State Transfer Application Programming Interface) este un standard arhitectural pentru comunicarea între un client (browser, aplicație mobilă) și un server. Aproape orice serviciu web modern expune un REST API.

> Analogie: REST API-ul este meniul unui restaurant. Clientul (tu, browser-ul) citește meniul (documentația API), comandă ce vrea (cerere HTTP), iar bucătarul (server-ul) pregătește și aduce comanda (răspuns JSON). Meniul definește exact ce se poate comanda și în ce format.

Principiile REST

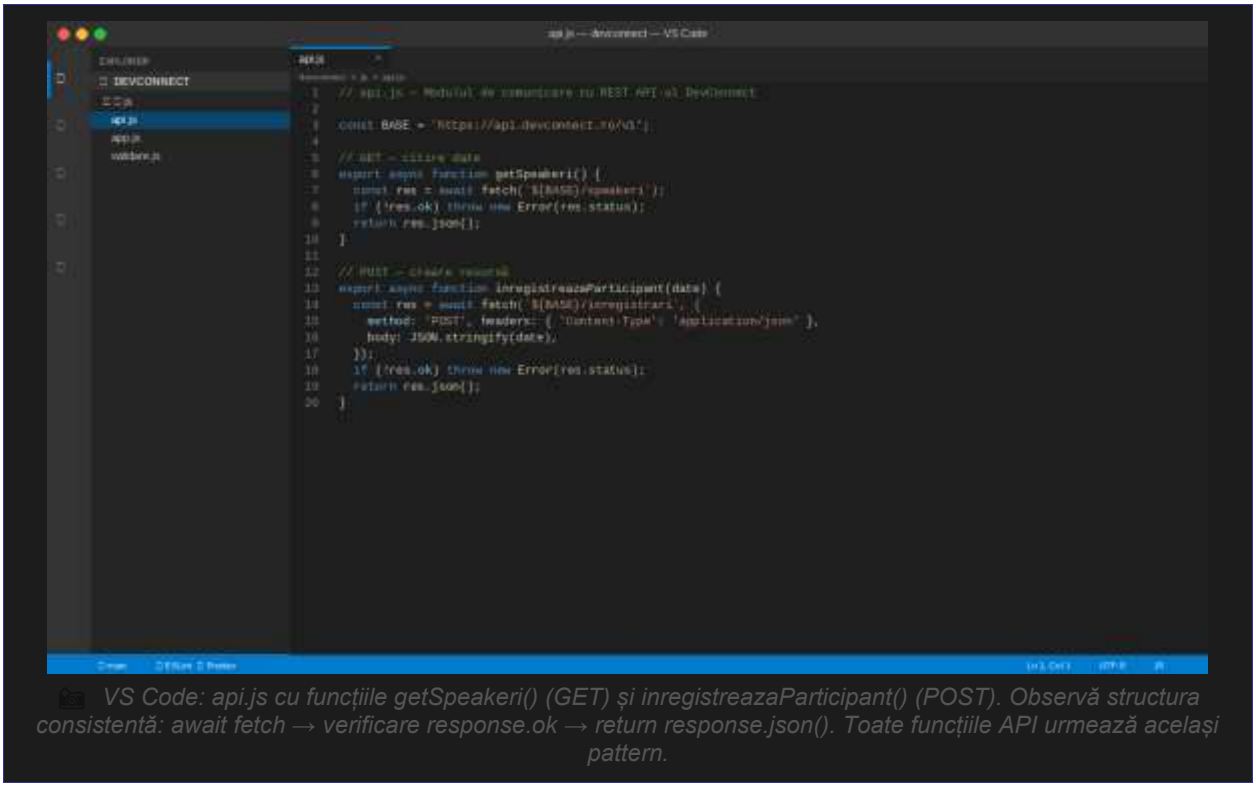
Principiu	Ce înseamnă în practică
Resurse cu URL-uri	Fiecare entitate are un URL unic: /api/speakeri, /api/speakeri/1, /api/workshops
Verbe HTTP	GET (citire), POST (creare), PUT/PATCH (modificare), DELETE (ștergere) - acțiunile sunt în verb, NU în URL
Stateless	Serverul nu ține minte starea clientului între cereri. Fiecare cerere trebuie să conțină tot contextul necesar (ex: token autentificare în header).
JSON	Format standard de date pentru request body și response body. Ușor de citit de oameni și parsabil de JavaScript.
Status codes	Serverul comunică rezultatul prin codul HTTP: 200, 201, 400, 401, 404, 500 - clientul știe ce s-a întâmplat fără să citească body-ul.

Endpoint-urile API DevConnect (ce vom construi în Lab 9-11)

```
// URL-urile REST API DevConnect:
GET    /api/speakeri          → lista tuturor speakerilor
GET    /api/speakeri/:id      → un speaker după ID
POST   /api/inregistrari      → creare înregistrare nouă
GET    /api/inregistrari/:id  → status înregistrare
PUT    /api/inregistrari/:id  → actualizare completă
PATCH  /api/inregistrari/:id  → actualizare parțială (ex: doar workshopul)
DELETE /api/inregistrari/:id  → anulare înregistrare
GET    /api/workshops         → lista workshopurilor cu locuri disponibile

// Convenții URL REST:
// ☑ /api/speakeri         (substantiv plural, nu /getSpeakeri sau
/speaker/list)
// ☑ /api/speakeri/1       (ID în URL, nu /api/speakeri?id=1 pentru resurse)
// ☑ GET /api/speakeri     (nu /api/getSpeakeri - acțiunea e în verbul HTTP)
```

6.2 Fetch API - Cererile HTTP din Browser

fetch() este API-ul nativ al browser-ului pentru cereri HTTP asincrone. A înlocuit XMLHttpRequest (XHR) - este modern, bazat pe Promises, și simplu de folosit.



VS Code: api.js cu funcțiile getSpeakeri() (GET) și inregistreazaParticipant() (POST). Observă structura consistentă: await fetch → verificare response.ok → return response.json(). Toate funcțiile API urmează același pattern.

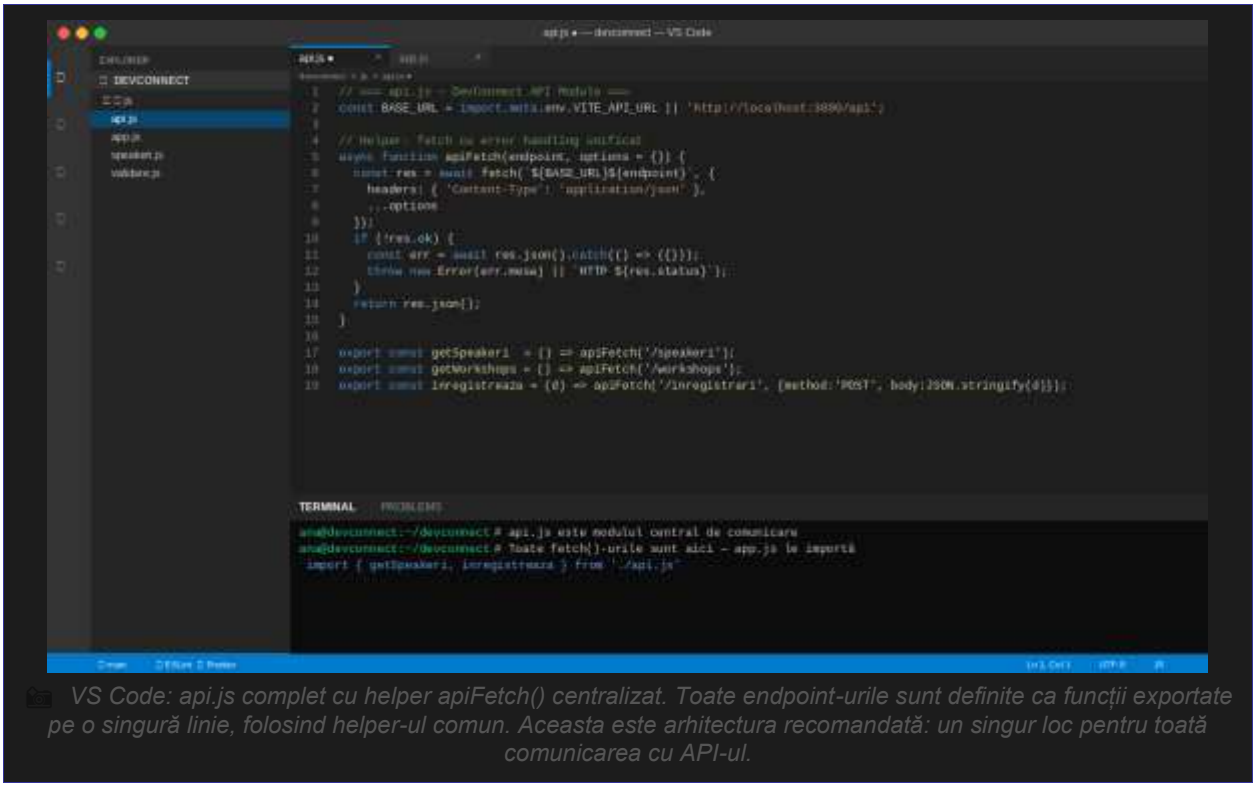
Anatomia unui apel fetch()

```
// Sintaxa completă fetch():
const response = await fetch(url, options);

// options - obiect cu configurare:
const options = {
  method: 'POST',                          // GET implicit
  headers: {
    'Content-Type': 'application/json',    // tipul datelor trimise
    'Authorization': 'Bearer token123',    // autentificare (Lab 12)
    'Accept': 'application/json',          // tipul dorit în răspuns
  },
  body: JSON.stringify({ nume: 'Ana' }),   // DOAR pentru POST/PUT/PATCH
  // body nu se trimite la GET/DELETE - e ignorat sau eroare
};

// Response object - NU este datele direct!
response.ok       // true dacă status 200-299
response.status   // 200, 201, 404, 500 etc.
response.headers  // Headers object

// Parsare body - alege metoda potrivită:
const data  = await response.json();    // JSON → obiect JS
const text  = await response.text();    // text simplu / HTML
const blob  = await response.blob();    // imagine / fișier
```

⚠ Capcana #1: fetch() NU aruncă eroare pentru statusuri HTTP 4xx/5xx! Un răspuns 404 sau 500 este considerat 'success' de fetch - Promise-ul se rezolvă. Trebuie să verifici manual

> response.ok sau response.status și să arunci eroarea tu: if (!response.ok) throw new Error(response.status).

Modulul api.js - arhitectura recomandată



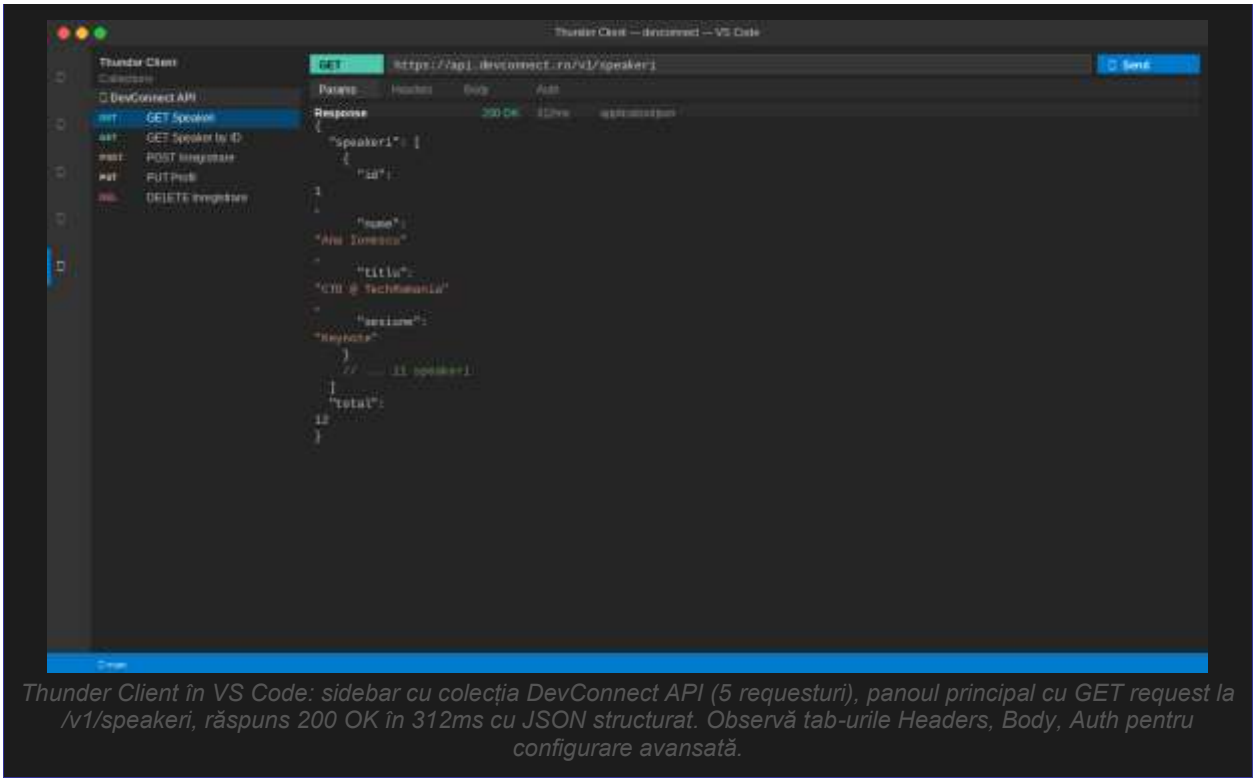
VS Code: api.js complet cu helper apiFetch() centralizat. Toate endpoint-urile sunt definite ca funcții exportate pe o singură linie, folosind helper-ul comun. Aceasta este arhitectura recomandată: un singur loc pentru toată comunicarea cu API-ul.

Avantajele arhitecturii cu modul api.js centralizat:

- Un singur loc de configurat BASE_URL - schimbi o variabilă, se actualizează tot
- Error handling unificat - orice eroare HTTP este gestionată identic
- Ușor de testat și de mock-uit în teste unitare
- Separare clară: ui/ (randare) nu știe nimic despre fetch() - doar importă funcții din api.js
- Simplu de adăugat autentificare mai târziu (header Authorization în apiFetch)

6.3 Thunder Client - Testarea API-urilor în VS Code

Thunder Client este extensia VS Code care îți permite să testezi API-uri REST direct din editor, fără a deschide Postman sau alt instrument extern. O vom folosi intens din Lab 9 când construim propriul API Express.

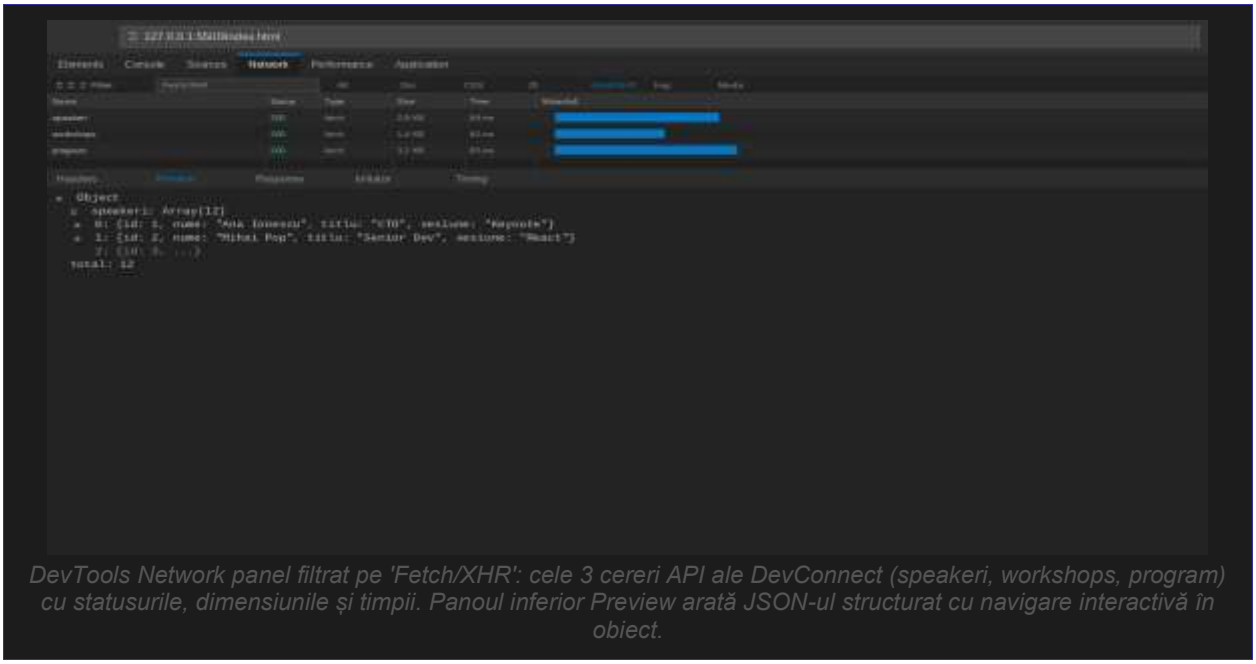


Thunder Client în VS Code: sidebar cu colecția DevConnect API (5 requesturi), panoul principal cu GET request la /v1/speakeri, răspuns 200 OK în 312ms cu JSON structurat. Observă tab-urile Headers, Body, Auth pentru configurare avansată.

Cum se creează o colecție în Thunder Client

1. Clic pe iconița Thunder Client din bara laterală stângă (sau Ctrl+Shift+P → Thunder Client)
2. Clic 'New Collection' → denumește 'DevConnect API'
3. Clic 'New Request' în colecție → selectează metoda (GET/POST/etc.)
4. Tastează URL-ul endpoint-ului
5. Configurează Headers, Body dacă e necesar
6. Clic 'Send' sau Ctrl+Enter

6.4 Analiza Răspunsurilor JSON în DevTools

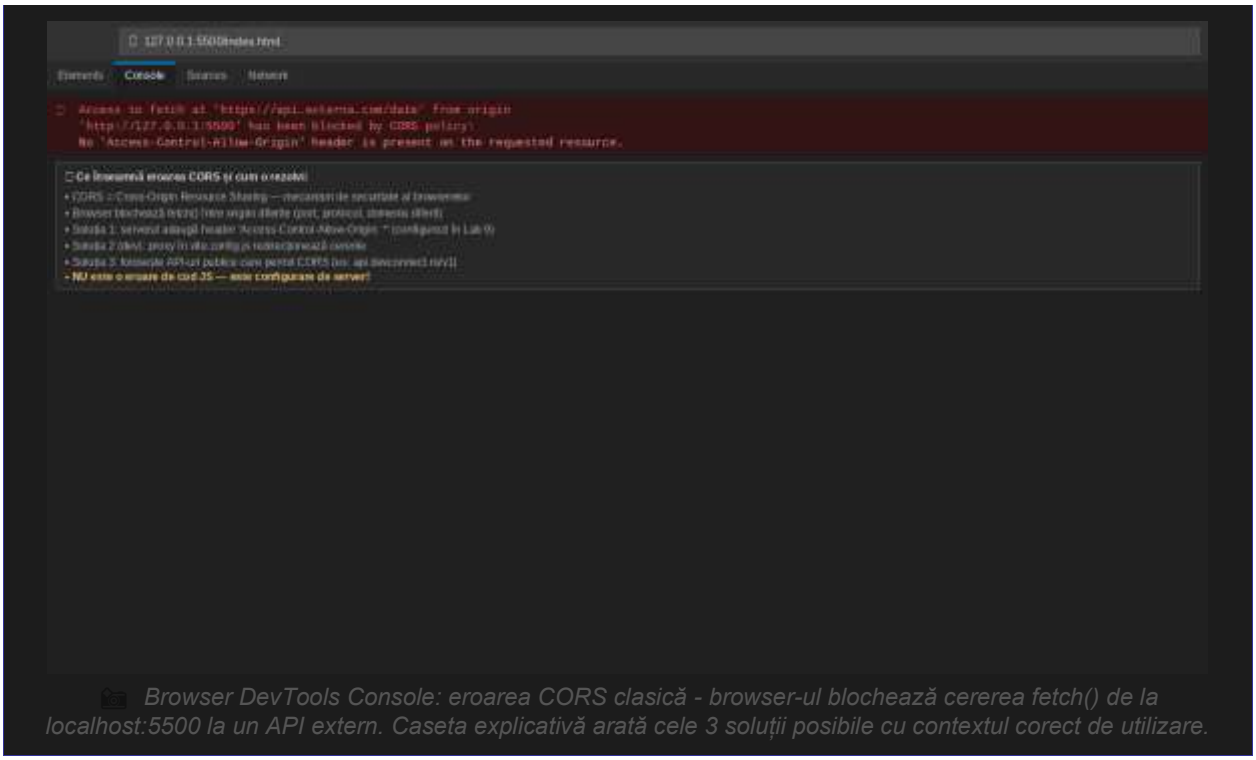


DevTools Network panel filtrat pe 'Fetch/XHR': cele 3 cereri API ale DevConnect (speakeri, workshops, program) cu statusurile, dimensiunile și timpii. Panoul inferior Preview arată JSON-ul structurat cu navigare interactivă în obiect.

Structura unui răspuns JSON tipic

```
// Convenții pentru răspunsuri API REST bine proiectate:

// ☑ Lista de resurse:
{
  "speakeri": [...],
  "total": 12,
  "pagina": 1,
  "per_pagina": 10
}

// ☑ Resursă individuală:
{
  "speaker": { "id": 1, "nume": "Ana Ionescu", "titlu": "CTO" }
}

// ☑ Eroare standardizată:
{
  "eroare": "EMAIL_DEJA_EXISTENT",
  "mesaj": "Există deja o înregistrare cu acest email.",
  "camp": "email"
}

// ☑ Creare cu succes (201 Created):
{
  "inregistrare": { "id": 247, "status": "confirmata" },
  "mesaj": "Înregistrare confirmată! Email trimis la ana@exemplu.ro"
}
```

6.5 CORS - Politica de Aceeași Origine



Browser DevTools Console: eroarea CORS clasică - browser-ul blochează cererea fetch() de la localhost:5500 la un API extern. Caseta explicativă arată cele 3 soluții posibile cu contextul corect de utilizare.

CORS este una din cele mai frecvente erori cu care se confruntă studenții. Este important de înțeles că nu este o eroare de cod JavaScript, ci un mecanism de securitate al browser-ului.

Situație	Soluție
API-ul propriu (Express, Lab 9)	Adaugă pachetul cors în Express: app.use(cors()). Configurabil pentru producție.
API extern public (dev)	Verifică documentația - dacă permite CORS public, adaugă header Accept: application/json. Dacă nu, folosește proxy.
Proxy în Vite (dev)	Configurează vite.config.js: server.proxy redirecționează /api → server real. Browser-ul crede că e același origin.
Producție	Servești front-end și API de pe același domeniu, sau configurezi CORS explicit cu lista de origini permise.

☑ Pentru Lab 5 și Lab 6, evitați problemele CORS folosind API-uri publice care permit cereri cross-origin (ex: api.devconnect.ro/v1 configurat pentru development, sau publice: jsonplaceholder.typicode.com, restcountries.com). CORS propriu configurăm în Lab 9.

6.6 DevConnect - Modulul api.js complet cu explicații

Această secțiune construiește modulul api.js al DevConnect pas cu pas, explicând fiecare decizie de design. Vom vedea și fluxul complet de la formular la server și înapoi.

De ce separăm api.js de restul codului?

O greșeală frecventă este să scrii fetch() direct în event handler-e sau în funcțiile de randare. Iată consecințele:

```
// ✗ Antipattern - fetch() împrăștiat în mai multe fișiere

// În speakeri.js:
const res = await fetch('https://api.devconnect.ro/v1/speakeri');

// În validare.js:
const res = await fetch('https://api.devconnect.ro/v1/inregistrari', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(date),
});

// Probleme generate:
// 1. URL-ul base duplicat în 5+ locuri → schimbarea URL necesită edit multiplu
// 2. Error handling diferit în fiecare loc → comportament inconsistent
// 3. Headers duplicați → uiți Content-Type undeva → erori ciudate la POST
// 4. Imposibil de testat unitar → nu poți 'mock' fetch din afara api.js

// ☑ Pattern corect: TOT fetch() în js/api.js
// speakeri.js: import { getSpeakeri } from './api.js'
// validare.js: import { inregistreazaParticipant } from './api.js'
```

api.js - versiunea completă cu error handling detaliat

```
// js/api.js - DevConnect Lab 6
// Responsabilitate unică: interfața între aplicație și server

// BASE_URL din variabilă de mediu Vite (vezi .env.local)
// Fallback pentru development fără .env configurat
const BASE_URL = import.meta.env?.VITE_API_URL ?? 'https://api.devconnect.ro/v1';

// ─── Helper central ──────────────────────────────────────
async function apiFetch(endpoint, options = {}) {
  const url = `${BASE_URL}${endpoint}`;

  const headers = {
    'Content-Type': 'application/json',
    'Accept': 'application/json',
    ...options.headers, // spread: options.headers poate suprascrie
  };

  let response;
  try {
    response = await fetch(url, { ...options, headers });
  } catch (networkError) {
    // fetch() aruncă DOAR pentru erori de rețea (offline, DNS fail)
    // NU aruncă pentru statusuri 404, 500 etc.
    throw new Error('Nu se poate conecta la server. Verifică conexiunea.');
  }

  // fetch() NU aruncă pentru 4xx/5xx - verificăm manual
  if (!response.ok) {
    let errorData = {};
    try { errorData = await response.json(); } catch { /* body non-JSON */ }

    const mesaje = {
      400: errorData.mesaj ?? 'Date incorecte. Verifică formularul.',
      409: errorData.mesaj ?? 'Emailul este deja înregistrat.',
      429: 'Prea multe cereri. Așteaptă un moment.',
      500: 'Eroare server. Încearcă din nou mai târziu.',
    };
    throw new Error(mesaje[response.status] ?? `Eroare ${response.status}`);
  }

  if (response.status === 204) return null; // DELETE → no content
  return response.json();
}

// ─── Endpoint-uri exportate ──────────────────────────────
export const getSpeakeri  = ()  => apiFetch('/speakeri');
export const getWorkshops = ()  => apiFetch('/workshops');
export const getProgram   = ()  => apiFetch('/program');

export const inregistreazaParticipant = (date) =>
  apiFetch('/inregistrari', { method: 'POST', body: JSON.stringify(date) });

export const actualizeazaInregistrare = (id, modificari) =>
  apiFetch(`/inregistrari/${id}`, { method: 'PATCH', body:
JSON.stringify(modificari) });

export const anuleazaInregistrare = (id) =>
  apiFetch(`/inregistrari/${id}`, { method: 'DELETE' });
```

Conectarea formularului de înregistrare la API - flux complet

Fluxul complet de la click pe 'Înregistrează-te' până la confirmarea serverului:

```
// js/validare.js - submit handler cu API și toate stările UI
import { inregistreazaParticipant } from './api.js';

export function initValidare() {
  const form = document.querySelector('#formular-inregistrare');
  if (!form) return;

  form.addEventListener('submit', async (event) => {
    event.preventDefault(); // previne reîncărcarea paginii

    // Colectare date formular ca obiect JavaScript
    // FormData → iterable de [key, value] → Object
    const dateFormular = Object.fromEntries(new FormData(form));
    // ↑ rezultat: { nume: 'Ana', email: 'ana@ex.ro', workshopId: 'ws-01' }

    // Normalizare: checkbox → boolean (FormData returnează 'on' sau lipsă)
    dateFormular.newsletter = form.querySelector('#newsletter')?.checked ??
false;

    // ─── STARE 1: Se trimite ──────────────────────────────
    const btnSubmit = form.querySelector('[type=submit]');
    btnSubmit.disabled = true;
    btnSubmit.textContent = '⏳ Se trimite...';
    afiseazaMesaj('loading', 'Se procesează înregistrarea...');

    try {
      const rezultat = await inregistreazaParticipant(dateFormular);
      // rezultat = { inregistrare: { id: 247, status: 'confirmata' }, mesaj:
'...' }

      // ─── STARE 2: Succes ────────────────────────────────
      form.reset();
      localStorage.removeItem('devconnect-form'); // curăță draft-ul
      afiseazaMesaj('success',
        `☑ ${rezultat.mesaj} (ID: #${rezultat.inregistrare.id})`
      );
      document.querySelector('#mesaj-feedback')
              .scrollIntoView({ behavior: 'smooth', block: 'center' });

    } catch (err) {
      // ─── STARE 3: Eroare ────────────────────────────────
      // err.message vine din apiFetch() - deja tradus în română
      afiseazaMesaj('error', `✕ ${err.message}`);

    } finally {
      // Finally rulează INDIFERENT de succes/eroare
      // Reactivăm butonul în ambele cazuri - nu duplicăm codul
      btnSubmit.disabled = false;
      btnSubmit.textContent = 'Înregistrează-te';
    }
  });
}

// Helper: afișare zonă feedback vizibilă utilizatorului
function afiseazaMesaj(tip, text) {
  const zone = document.querySelector('#mesaj-feedback');
  zone.className = `feedback feedback--${tip}`; // CSS: .feedback--success, etc.
  zone.textContent = text;
  zone.removeAttribute('hidden');
  if (tip === 'success') {
    setTimeout(() => zone.setAttribute('hidden', ''), 8000);
  }
}
```

> 💡 De ce finally? Dacă am pune btnSubmit.disabled = false separat în try și în catch, ar fi cod duplicat și riscăm să-l uităm într-un ramuri. finally garantează că butonul este reactivat ÎNTOTDEAUNA, fie că a reușit fie că a eșuat. Aceasta este folosirea canonică a try/catch/finally în JS asincron.

Diagrama fluxului complet DevConnect Lab 6

```
Utilizator completează formularul → clic 'Înregistrează-te'
  │
  ▼
validare.js: event.preventDefault()
  Object.fromEntries(new FormData(form))
  btnSubmit.disabled = true
  afiseazaMesaj('loading', ...)
  │
  ▼
api.js: inregistreazaParticipant(dateFormular)
  apiFetch('/inregistrari', { method:'POST', body: JSON.stringify(...) })
  fetch('https://api.devconnect.ro/v1/inregistrari', { ... })
  │
  ▼ (HTTP POST - ~300-600ms)
Server: primește cererea, validează, salvează în DB
  returnează 201 Created: { inregistrare: { id: 247, ... }, mesaj: '...' }
  │
  ▼
api.js: response.ok = true → return response.json()
  │
  ▼
validare.js: try { → succes
  form.reset()
  afiseazaMesaj('success', '☑ Înregistrat! ID: #247')
  finally: btnSubmit.disabled = false

─── Caz eroare (ex: email duplicat, server 409): ──────────
api.js: response.ok = false, status=409
  → throw new Error('Emailul este deja înregistrat.')
validare.js: catch { afiseazaMesaj('error', '✕ ...') }
  finally: btnSubmit.disabled = false
```

6.7 Referințe Curs 6

- MDN - Fetch API (documentație completă)
- MDN - Using Fetch (ghid practic)
- javascript.info - Fetch
- REST API Design Best Practices (freeCodeCamp)
- Thunder Client VS Code Extension
- JSONPlaceholder - API public pentru teste

Notă privind elaborarea materialelor de curs

Vreau să fiu transparent cu voi: structura și conținutul acestor note de curs au fost generate cu ajutorul unui instrument de inteligență artificială (Claude, de la Anthropic), pe baza cerințelor și direcțiilor pe care le-am formulat eu ca titular de curs.

De ce vă spun asta? Pentru că:
• Nu pot garanta că fiecare noțiune tehnică are 100% acuratete sau este actualizată
• Vă încurajez să verificați activ sursele bibliografice indicate
• Utilizarea responsabilă a AI în educație înseamnă transparență, nu ascundere

Considerați aceste materiale un ghid structurat de studiu, nu un manual definitiv. Dacă identificați o eroare sau o neclaritate, veniți cu ea la curs.