CURS 5

JavaScript Asincron

Note de Curs cu Exemple Practice în Visual Studio Code

Cursul 5	JavaScript Asincron - Event Loop, Promises, async/await
Cursul 6	Fetch API & REST APIs - HTTP din browser, JSON, CORS, Thunder Client
Lab 5	DevConnect: înlocuiește datele statice cu Promise- uri simulate
Lab 6	DevConnect: conectare la API real cu fetch() și gestionare erori
Proiect	js/api.js - modulul de comunicare cu serverul (baza pentru Lab 9+)

Cursul 5 și Cursul 6 sunt strâns legate: C5 explică mecanismele asincrone JavaScript (de ce avem nevoie de Promises), iar C6 le aplică concret pentru comunicarea cu servere REST. Parcurge- le în ordine.

CURSUL 5 - JavaScript Asincron

5.1 De ce avem nevoie de asincronism?

JavaScript rulează pe un singur fir de execuție (single- threaded). Asta înseamnă că poate face un singur lucru la un moment dat. Dacă o operație durează mult - citire fișier, cerere HTTP, timer - toată pagina s- ar bloca și ar deveni neresponsivă. Asincronismul este soluția.

> Analogie: ești la restaurant. Ospatarul (thread- ul JS) ia comanda ta, o duce la bucătărie, și se întoarce să ia comenzile altor clienți - NU stă lângă aragaz să aștepte. Când mâncarea e gata, bucătarul anunță (callback/resolve) și ospatarul o aduce. Astfel servește 20 de clienți simultan cu un singur angajat.

Operații sincrone vs. asincrone

	Sincron (blochează)	Asincron (nu blochează)
Execuție	Linie cu linie, în ordine strictă	Inițiată acum, rezultat livrat mai târziu
Exemplu	let x = 1 + 1 (instant)	fetch('/api') - răspuns în 200ms
Problema	O operație lentă blochează tot	- nicio blocare
Soluție JS	-	Callbacks → Promises → async/await

Event Loop - cum funcționează JavaScript asincron

Înțelegerea Event Loop-ului explică TOATE comportamentele aparent ciudate ale JavaScript-ului asincron.

- Call Stack - stiva de execuție: funcțiile apelate sunt adăugate pe stivă și scoase când se termină
- Web APIs - mediul browser- ului: setTimeout, fetch, addEventListener rulează în afara thread-ului JS
- Callback Queue (Task Queue) - coada de callback- uri gata de execuție (setTimeout, events)
- Microtask Queue - prioritate mai mare decât Task Queue: Promise callbacks (.then, .catch) și async/await
- Event Loop - verifică continuu: dacă Call Stack e gol, mută primul element din Microtask Queue (sau Task Queue) pe stivă

```
// Demo: ordinea de execuție
console.log('1 - sincron');

setTimeout(() => console.log('3 - Task Queue (setTimeout 0ms)'), 0);

Promise.resolve()
  .then(() => console.log('2 - Microtask Queue (Promise)'));

console.log('1b - sincron');

// Output: 1 - sincron
//         1b - sincron
//         2 - Microtask Queue (Promise)   ← Promise înainte de setTimeout!
//         3 - Task Queue (setTimeout 0ms)
```

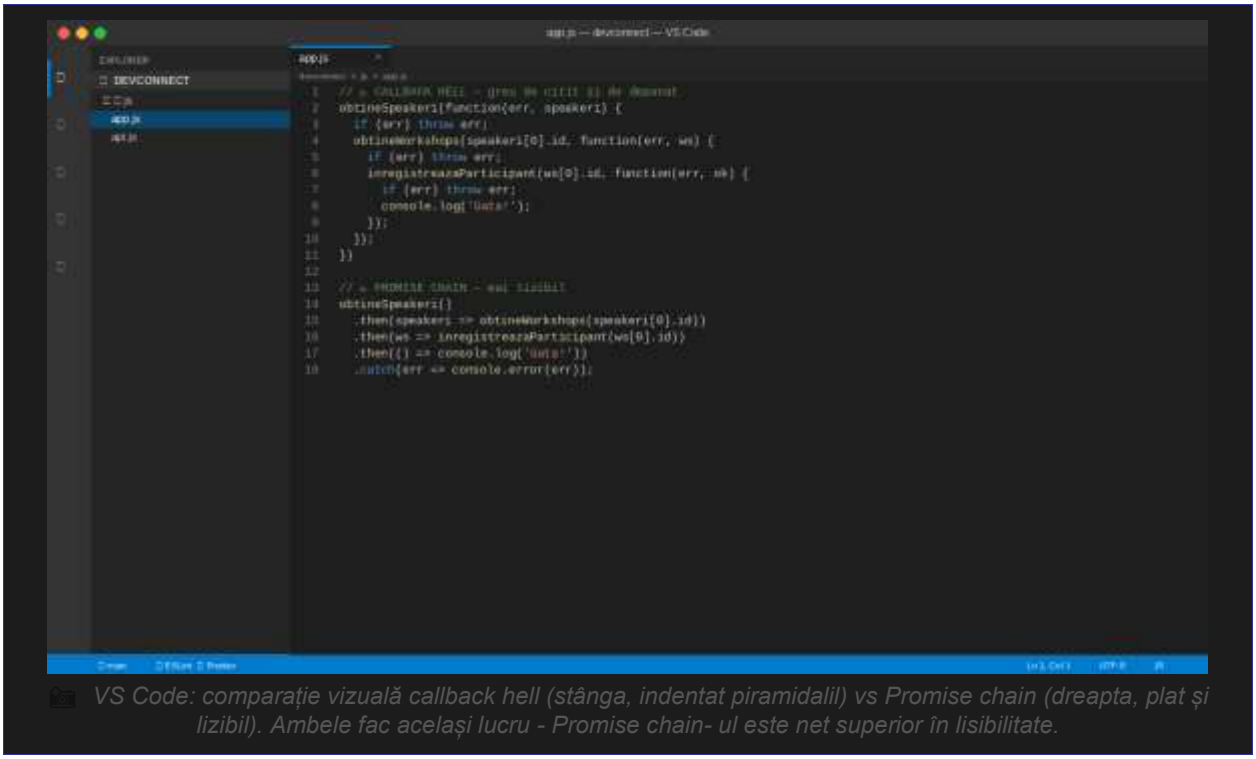
> De reținut: Promises (.then) și async/await se execută din Microtask Queue - au prioritate față de setTimeout și alte Web API callbacks din Task Queue. Aceasta explică de ce outputul nu e în ordinea în care pari să scrii codul.

5.2 Evoluția: Callbacks → Promises → async/await

JavaScript a rezolvat asincronismul în trei etape istorice. Înțelegerea evoluției explică de ce async/await arată cum arată.

Etapa 1: Callbacks (ES5, înainte de 2015)

Un callback este o funcție pasată ca argument altei funcții, care va fi apelată când operația se termină. A funcționat, dar a dus la un antipattern celebru: Callback Hell.



VS Code: comparație vizuală callback hell (stânga, indentat piramidalil) vs Promise chain (dreapta, plat și lizibil). Ambele fac același lucru - Promise chain- ul este net superior în lisibilitate.

```
// ❌ Callback Hell - 'Pyramid of Doom'
obtineSpeakeri(function(err, speakeri) {
  if (err) throw err;
  obtineWorkshops(speakeri[0].id, function(err, workshops) {
    if (err) throw err;
    inregistreaza(workshops[0].id, function(err, confirmare) {
      if (err) throw err;
      console.log('Înregistrat!', confirmare);
      // ... și tot mai adânc
    });
  });
});
```

> ⚠ Problemele cu callbacks:
>
> (1) Error handling duplicat la fiecare nivel.
>
> (2) Imposibil de urmărit fluxul de execuție.

(3) Nu poți folosi try/catch pentru gestionarea erorilor.

(4) Dificil de paralelizat operații.

Etapa 2: Promises (ES6, 2015)

O Promise este un obiect care reprezintă valoarea viitoare a unei operații asincrone. Are trei stări posibile:

Stare	Înseamnă	Ce se întâmplă
pending	În așteptare	Operația a început, nu s- a terminat încă
fulfilled	Rezolvată cu succes	Valoarea e disponibilă, .then() este apelat
rejected	Eșuată	A apărut o eroare, .catch() este apelat

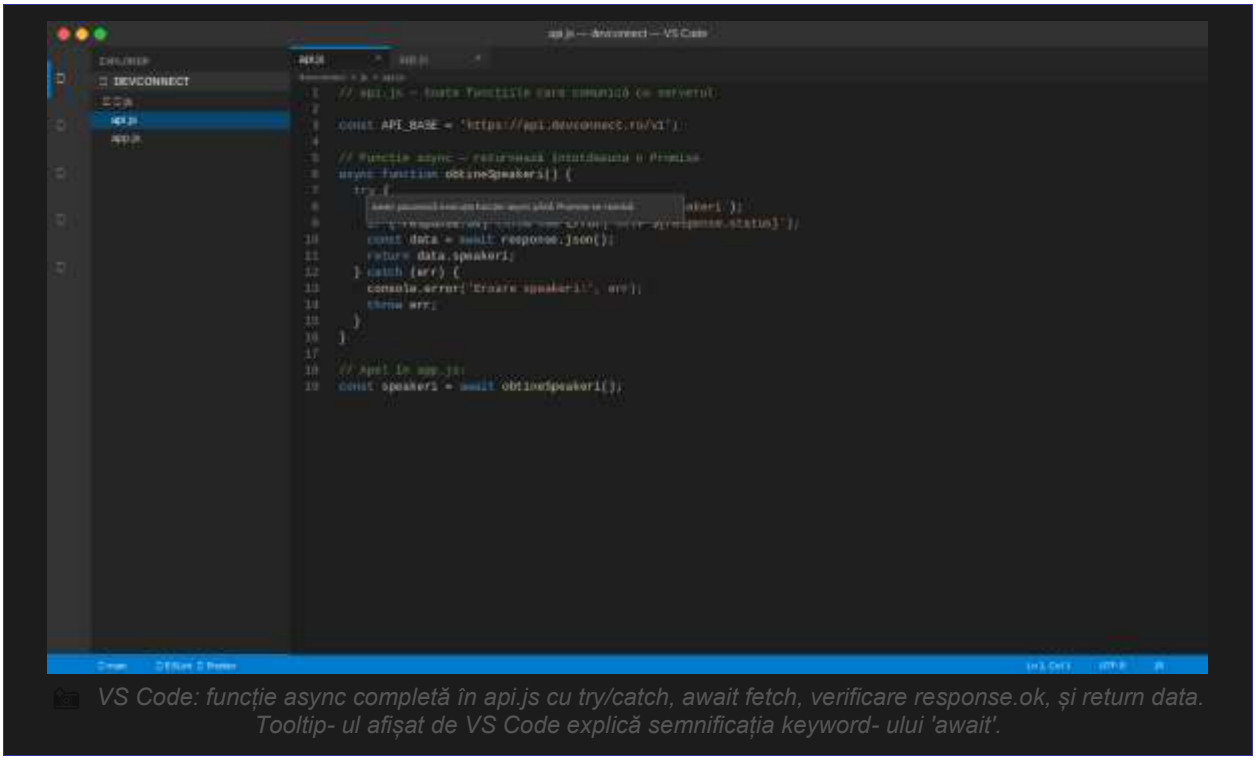
```
// Crearea unei Promise manual (de obicei nu faci asta direct)
const promise = new Promise((resolve, reject) => {
  // Operație asincronă simulată
  setTimeout(() => {
    const succes = Math.random() > 0.5;
    if (succes) resolve({ id: 1, nume: 'Ana Ionescu' });
    else reject(new Error('Server indisponibil'));
  }, 1000);
});

// Consumarea Promise- ei
promise
  .then(speaker => console.log('Speaker:', speaker.nume))
  .catch(err => console.error('Eroare:', err.message))
  .finally(() => console.log('Gata - cu succes sau eroare'));
```

Promise combinators - operații pe multiple Promise- uri

```
// Promise.all - TOATE trebuie să reușească, rulează în PARALEL
const [speakeri, workshops] = await Promise.all([
  getSpeakeri(),    // 300ms
  getWorkshops(),   // 200ms
]);
// Total: 300ms (nu 500ms) ☑

// Promise.allSettled - continuă chiar dacă unele eșuează
const rezultate = await Promise.allSettled([
  getSpeakeri(),
  getDateOptionale(),  // poate eșua
]);
rezultate.forEach(r => {
  if (r.status === 'fulfilled') console.log(r.value);
  else console.warn('Opțional eșuat:', r.reason);
});

// Promise.race - prima Promise care se termină câștigă
// Util pentru timeout:
const timeout = new Promise((_, reject) =>
  setTimeout(() => reject(new Error('Timeout!')), 5000)
);
const date = await Promise.race([getSpeakeri(), timeout]);
```

5.3 async/await - Sintaxa Modernă

async/await este syntax sugar peste Promises - codul async arată și se comportă ca și cum ar fi sincron. Aceasta este sintaxa standard în 2024 pentru tot codul asincron nou.



VS Code: funcție async completă în api.js cu try/catch, await fetch, verificare response.ok, și return data. Tooltip- ul afișat de VS Code explică semnificația keyword- ului 'await'.

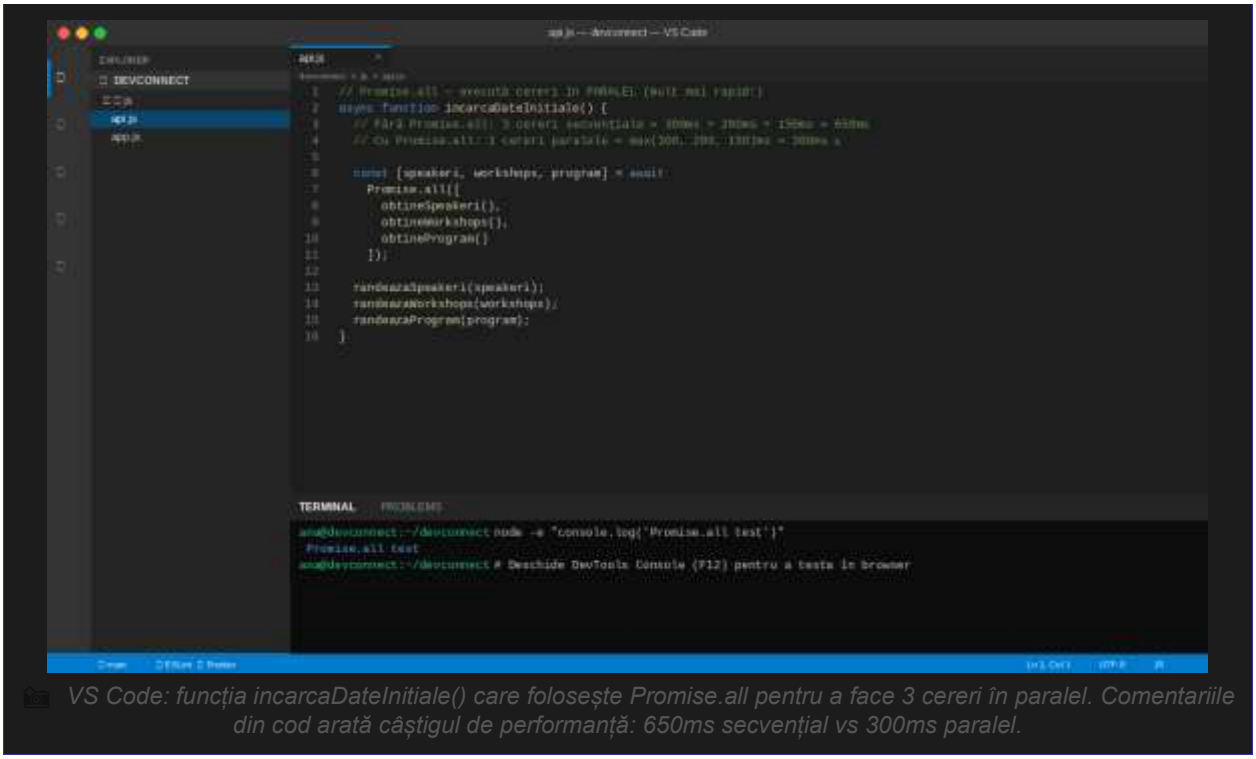
Regulile fundamentale async/await

Regulă	Explicație + Exemplu
async function → returnează Promise	Orice funcție declarată async returnează implicit o Promise, chiar dacă returnezi o valoare simplă. async function f() { return 42 } → f() returnează Promise<42>
await funcționează DOAR în async	Nu poți folosi await în afara unei funcții async (excepție: top- level await în module ES2022). Soluție: înfășoară codul în async function.
await pauzează DOAR funcția curentă	await nu blochează thread- ul - pauzează execuția funcției async curente și predă controlul Event Loop- ului. Restul paginii continuă să funcționeze.
try/catch prinde erorile	Cu async/await poți folosi try/catch nativ, ca și cum ar fi cod sincron. Mult mai clar decât .catch() înlănțuit.
await pe non- Promise e ok	await 42 returnează 42. Nu e eroare. Util când nu știi dacă o funcție returnează sync sau async.

```
// Greșeli frecvente cu async/await

// ❌ Greșeală 1: await în for loop = SECVENȚIAL (lent)
for (const id of ids) {
  const speaker = await getSpeaker(id); // 12 cereri × 200ms = 2400ms!
}
```

```
// ☑ Corect: Promise.all = PARALEL (rapid)
const speakeri = await Promise.all(ids.map(id => getSpeaker(id)));
// 12 cereri paralele = ~200ms ☑

// ✗ Greșeală 2: uitat await = primești Promise, nu valoarea
const speakeri = getSpeakeri(); // speakeri = Promise {pending}!
console.log(speakeri[0]);       // undefined

// ☑ Corect:
const speakeri = await getSpeakeri(); // speakeri = [{...}, ...]
```

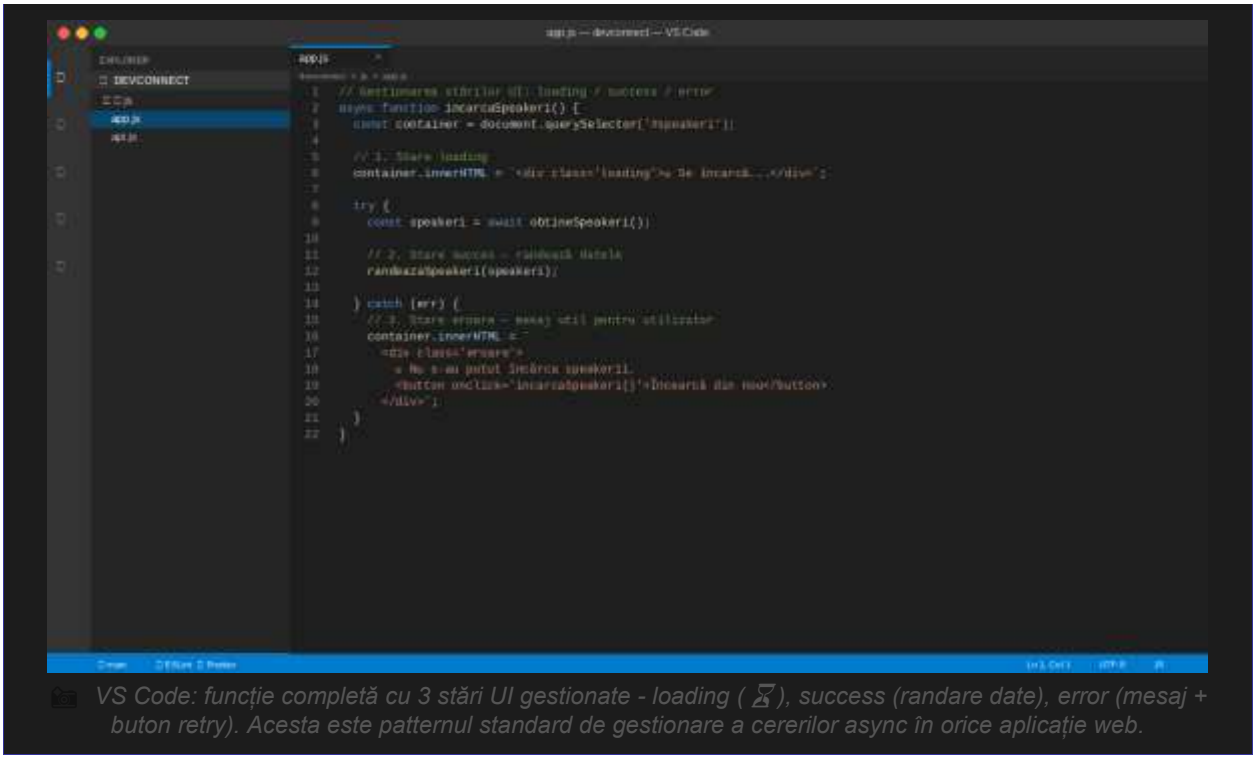
Promise.all în proiectul DevConnect



VS Code: funcția incarcaDateInitiale() care folosește Promise.all pentru a face 3 cereri în paralel. Comentariile din cod arată câștigul de performanță: 650ms secvențial vs 300ms paralel.

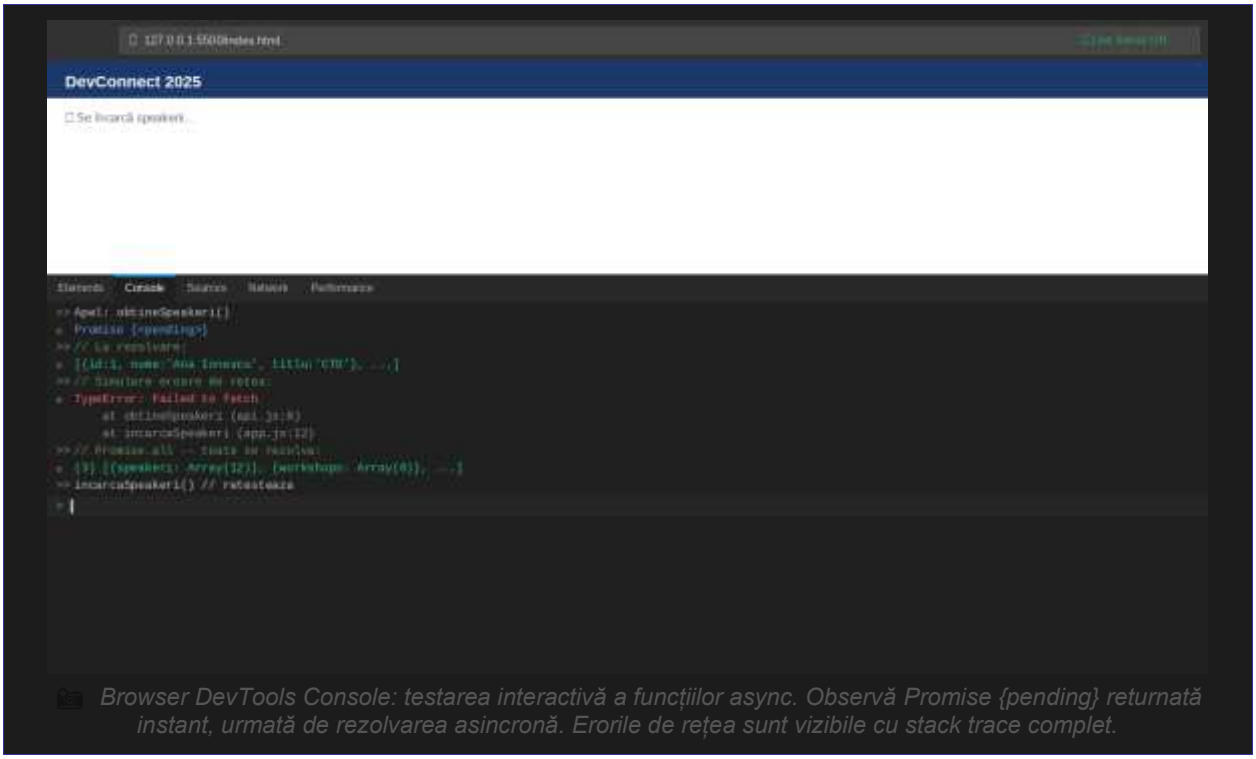
Gestionarea stărilor UI cu async/await

Un pattern esențial în orice aplicație: afișarea stărilor loading / success / error în interfață pe parcursul operațiilor asincrone.



VS Code: funcție completă cu 3 stări UI gestionate - loading (⏳), success (randare date), error (mesaj + buton retry). Acesta este patternul standard de gestionare a cererilor async în orice aplicație web.

Debugging async în DevTools Console



Browser DevTools Console: testarea interactivă a funcțiilor async. Observă Promise {pending} returnată instant, urmată de rezolvarea asincronă. Erorile de rețea sunt vizibile cu stack trace complet.

Exercițiu practic în Console (F12): tastează await getSpeakeri() direct în Console (prefixul await funcționează în Console modern). Vei vedea Promise- ul rezolvat și datele afișate direct. Aceasta este tehnica de debugging async mai rapidă.

5.5 DevConnect - Exemplificare Completă Asincronism

Această secțiune trasează evoluția completă a codului DevConnect din Lab 1-4 (sincron, date statice) la codul din Lab 5 (asincron, date simulate). Fiecare transformare este explicată pas cu pas.

Pasul 1 - Codul SINCRON original (Lab 4)

Până la Lab 4, DevConnect folosea date statice definite direct în JavaScript. Nu exista asincronism - totul se executa instantaneu:

```
// js/date.js - Lab 4 (sincron, date statice)
// Problemă: datele sunt hardcodate, nu pot veni de la server

export const SPEAKERI = [
  {
    id: 1,
    nume: 'Ana Ionescu',
    titlu: 'CTO @ TechRo',
    bio: 'Expert în arhitecturi cloud și DevOps...',
    avatar: 'img/speakers/ana- ionescu.jpg',
    workshop: 'ws- 01',
    linkedin: 'https://linkedin.com/in/ana- ionescu'
  },
  {
    id: 2,
    nume: 'Mihai Popescu',
    titlu: 'Senior React Developer @ Accenture',
    bio: 'Frontend specialist cu 8 ani experiență React...',
    avatar: 'img/speakers/mihai- popescu.jpg',
    workshop: 'ws- 02',
    linkedin: 'https://linkedin.com/in/mihai- popescu'
  },
  // ... 10 speakeri total
];

// js/speakeri.js - Lab 4 (sincron)
import { SPEAKERI } from './date.js';

export function incarcaSpeakeri() {
  // ✗ Sincron: datele sunt disponibile INSTANT, fără stare loading
  // Dacă în viitor vin de la server, tot codul trebuie refactorizat
  const grid = document.querySelector('#speakeri .grid- speakeri');
  SPEAKERI.forEach(sp => grid.appendChild(creeazaCard(sp)));
}
```

Problemele cu abordarea sincronă:

(1) Datele nu pot veni de la server - trebuie actualizate manual în cod.

(2) Nu există stare de loading - utilizatorul nu știe că se întâmplă ceva.

(3) Nu există error handling - dacă ceva merge greșit, pagina se sparge silențios.

(4) Inițializarea e instantanee - nu poți testa comportamentul cu date lente.

Pasul 2 - Transformarea în funcții ASYNC simulate (Lab 5)

Primul pas spre asincronism real este simularea latenței cu delay(). Aceasta ne permite să construim și testa UI-ul de loading înainte ca API-ul real să fie disponibil (Lab 9):

```
// js/date.js - Lab 5 (async simulat)
// Scop: mimăm comportamentul unui server real fără a folosi fetch()

// ─── Date statice (rămân neschimbate) ──────────────────────
```

```
const SPEAKERI = [
  { id: 1, nume: 'Ana Ionescu', titlu: 'CTO @ TechRo', workshop: 'ws- 01' },
  { id: 2, nume: 'Mihai Popescu', titlu: 'Senior React Dev', workshop: 'ws- 02'
},
  // ...
];

const WORKSHOPS = [
  { id: 'ws- 01', titlu: 'Kubernetes în Producție', locuri: 20, ocupate: 17 },
  { id: 'ws- 02', titlu: 'React Patterns Avansate', locuri: 25, ocupate: 25 },
  { id: 'ws- 03', titlu: 'PostgreSQL Performance', locuri: 20, ocupate: 9 },
];

// ─── Helpers pentru simulare ──────────────────────────────
// delay(ms) returnează o Promise care se rezolvă după 'ms' milisecunde
const delay = (ms) => new Promise(resolve => setTimeout(resolve, ms));

// maybeError(prob) aruncă eroare cu probabilitatea 'prob' (0.0 -  1.0)
// Scop: testăm că UI- ul gestionează corect erorile de rețea
const maybeError = (prob = 0.1) => {
  if (Math.random() < prob) {
    throw new Error('Eroare de rețea simulată (server indisponibil)');
  }
};

// ─── Funcții async exportate ──────────────────────────────
export async function getSpeakeri() {
  await delay(800);       // server real: ~300- 600ms latență tipică
  maybeError(0.1);        // 10% șansă de eroare - testăm retry UI
  return [...SPEAKERI];  // returnăm copie, nu referința originală
}

export async function getWorkshops() {
  await delay(400);
  maybeError(0.05);
  return [...WORKSHOPS];
}

// Simulează un POST (înregistrare participant)
export async function inregistreazaParticipant(date) {
  await delay(1200);      // POST = mai lent decât GET
  maybeError(0.15);       // 15% eroare - mai frecvent la scriere
  // Răspuns similar cu ce va trimite API- ul real din Lab 9
  return {
    inregistrare: {
      id: Math.floor(Math.random() * 900 + 100),
      status: 'confirmata',
      timestamp: new Date().toISOString(),
    },
    mesaj: `Înregistrare confirmată! Email trimis la ${date.email}`,
  };
}
```

Pasul 3 - Stări UI: loading → success / error

Acesta este cel mai important pattern pentru orice aplicație web. Orice operație asincronă trebuie să comunice utilizatorului în ce stare se află:

```
// js/speakeri.js - Lab 5 (async cu 3 stări UI)
import { getSpeakeri } from './date.js';

export async function incarcaSpeakeri() {
  const grid = document.querySelector('#speakeri .grid- speakeri');
  if (!grid) return;
```

```
  // STARE 1: Loading - utilizatorul vede spinner
  grid.innerHTML = `
    <div class='loading- state' role='status' aria- live='polite'>
      <div class='spinner' aria- hidden='true'></div>
      <p>Se încarcă speakerii...</p>
    </div>
  `;

  try {
    const speakeri = await getSpeakeri(); // ~800ms

    // STARE 2: Succes - randăm cardurile
    grid.innerHTML = '';
    speakeri.forEach((sp, i) => {
      const card = creeazaCard(sp);
      card.style.animationDelay = `${i * 80}ms`; // stagger animație
      card.classList.add('fade- in');
      grid.appendChild(card);
    });

  } catch (err) {
    // STARE 3: Eroare cu buton retry
    console.error('[incarcaSpeakeri]', err);
    grid.innerHTML = `
      <div class='error- state' role='alert'>
        <span>⚠</span>
        <p>${err.message}</p>
        <button class='btn- retry' type='button'>
          🔄 Încearcă din nou
        </button>
      </div>
    `;
    // Retry: reapelăm funcția - va reafișa loading și va reîncerca
    grid.querySelector('.btn- retry')
        .addEventListener('click', () => incarcaSpeakeri());
  }
}
```

Pasul 4 - Promise.all în app.js pentru init paralel

```
// js/app.js - inițializare completă DevConnect Lab 5
import { incarcaSpeakeri }     from './speakeri.js';
import { incarcaWorkshops }    from './workshops.js';
import { initCountdown }       from './countdown.js';
import { initValidare }        from './validare.js';
import { restaureazaFormular } from './persistenta.js';

async function initApp() {
  console.time('DevConnect init');

  // PARALEL: lansăm toate cererile simultan
  // Fără Promise.all: 800ms + 400ms = 1200ms secvențial
  // Cu Promise.all:   max(800ms, 400ms) = 800ms paralel ☑
  await Promise.all([
    incarcaSpeakeri(),   // ~800ms
    incarcaWorkshops(),  // ~400ms
    initCountdown(),     // ~0ms (calculat local)
  ]);

  // SECVENȚIAL: după ce datele sunt disponibile
  initValidare();        // are nevoie de lista workshops
  restaureazaFormular(); // citește din localStorage

  console.timeEnd('DevConnect init'); // ~800ms vs 1200ms fără Promise.all
```

```
    }

    document.addEventListener('DOMContentLoaded', initApp);
```

5.6 Referințe Curs 5

- javascript.info - Promises (tutorial complet, recomandat)
- MDN - async function
- Jake Archibald - Tasks, Microtasks, Queues (vizualizare interactivă Event Loop)
- JS Visualizer 9000 - Event Loop vizual
- You Don't Know JS: Async & Performance (gratuit GitHub)

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.