

JavaScript modern

Laboratoare, rezultate de control și proiect final

Ediția 15 octombrie 2026. Exemple demonstrative. Citește explicația, încearcă exercițiul, apoi compară rezultatul.

Cuprins

1. Sintaxa modernă pe scurt
2. Destructuring și spread
3. Module ES
4. Promise și async/await
5. Tratarea erorilor asincrone
6. map / filter / reduce în cazuri reale
7. Organizarea codului fără framework

Sintaxa modernă pe scurt

Laborator explicat: normalizarea unui produs

Scopul acestui laborator este să transformi o înregistrare primită din exterior într-un obiect cu care interfața poate lucra predictibil. Creează un fișier `laborator.mjs` și rulează-l cu Node.js, sau folosește un script de tip module într-o pagină servită local. Exemplele sunt independente de un framework. Datele sunt fictive: un produs gratuit, unul cu preț necunoscut și unul căruia îi lipsește descrierea. Înainte de cod, scrie regula: zero este un preț valid; lipsa prețului trebuie semnalată; o descriere absentă poate primi un text de rezervă.

Separă valoarea absentă de valoarea invalidă

Operatorul pentru valoare implicită nu decide dacă datele sunt corecte. Dacă primești textul „gratuit” în câmpul numeric, înlocuirea lui cu zero ascunde o problemă a contractului. Funcția următoare acceptă numai numere finite, nenegative, și păstrează câmpul necunoscut ca null. Verificarea se face înainte de formatare. Astfel, interfața poate distinge „0 lei” de „preț indisponibil” fără să ghicească dintr-un șir deja formatat.

```
function normalizeazaProdus(raw) {
  if (raw === null || typeof raw !== 'object' || Array.isArray(raw)) {
    throw new TypeError('Produsul trebuie să fie un obiect');
  }
  const pret = raw.pret ?? null;
  if (pret !== null &&
    (typeof pret !== 'number' || !Number.isFinite(pret) || pret < 0)) {
    throw new TypeError('Preț numeric invalid');
  }
  const descriere = raw.detalii?.descriere ?? 'Fără descriere';
  if (typeof descriere !== 'string') {
    throw new TypeError('Descrierea trebuie să fie text');
  }
  return { pret, descriere };
}
const gratuit = normalizeazaProdus({ pret: 0 });
const necunoscut = normalizeazaProdus({});
console.assert(gratuit.pret === 0);
console.assert(necunoscut.pret === null);
console.assert(gratuit.descriere === 'Fără descriere');
```

Urmărește execuția pentru primul apel. Obiectul trece verificarea inițială, valoarea zero se păstrează, condiția de invaliditate este falsă, iar lipsa detaliilor produce descrierea de rezervă. Pentru al doilea apel, prețul devine null. Nu este aruncată eroare deoarece regula laboratorului permite un preț necunoscut. Dacă aplicația cere obligatoriu un preț, schimbă explicit această regulă și verificarea, în loc să introduci un preț inventat.

Alege declarația după felul în care se schimbă datele

Folosește `const` pentru o legătură care nu va fi reassignată. Un rezultat calculat o singură dată, o funcție sau o referință la un element din pagină sunt exemple obișnuite. Folosește `let` când aceeași variabilă primește succesiv alte valori, de exemplu indexul unei căutări. Nu transforma automat fiecare `let` în `const`: citește toate atribuțiile. În același timp, nu considera că un obiect declarat `const` este protejat împotriva modificării proprietăților. Aceasta este o regulă separată a programului.

Returnarea unui obiect dintr-o funcție săgeată

O funcție săgeată cu o expresie returnează acea expresie. Când expresia este un obiect literal, parantezele arată că acoladele descriu obiectul. În varianta cu un bloc de instrucțiuni, ai nevoie de `return`. Verifică rezultatul în consolă înainte de a conecta funcția la map: un `return` uitat poate produce o listă de valori `undefined`, deși numărul de elemente pare corect.

```
const eticheta = ({ pret }) => ({
  text: pret === null ? 'Preț indisponibil' : `${pret} lei`
});
console.assert(eticheta(gratuit).text === '0 lei');
```

```
console.assert(eticheta(necunoscut).text === 'Preț indisponibil');
```

Exercițiu cu rezultate verificabile

Adaugă trei apeluri: prețul -1, textul „0” și numărul Infinity. Toate trebuie să producă `TypeError` în această implementare. Testează separat descrierea goală: aceasta rămâne un șir gol deoarece ai definit fallback numai pentru absență. Decide dacă interfața ta permite această situație și scrie regula înainte să modifice funcția. Nu converți automat orice text cu `Number`: șirul gol și unele valori fără semnificație de preț pot produce rezultate surprinzătoare.

La autoevaluare, explică de ce gratuit și necunoscut au rezultate diferite, de ce descrierea este validată după extragere și de ce textul prețului este construit într-o funcție separată. Un răspuns complet include intrarea, rezultatul așteptat și motivul. Salvează și un exemplu respins, cu mesajul erorii. Această disciplină face refactorizarea verificabilă: poți schimba sintaxa fără să schimbi accidental sensul datelor.

Destructuring și spread

Laborator explicat: actualizarea unui contact fără modificarea sursei

Imaginează-ți un formular de profil în care utilizatorul poate modifica un contact și poate apoi apăsa „Anulează”. Dacă formularul schimbă direct obiectul folosit de restul paginii, anularea nu mai poate recupera valorile inițiale. Acesta este un motiv practic pentru a înțelege copierea obiectelor. Vom păstra versiunea inițială și vom construi una nouă, schimbând numai contactul ales. Exemplul poate fi rulat într-un fișier JavaScript sau în consola browserului.

Identifică fiecare nivel care se modifică

Datele au trei niveluri relevante: profilul, lista de contacte și obiectul contactului. O copie a profilului nu copiază automat lista, iar o copie a listei nu copiază automat obiectele din ea. În schimb, nu trebuie să clonezi fiecare obiect din aplicație. Construiește obiecte noi doar pe traseul proprietății schimbate. Contactele neatinse pot fi păstrate prin referință, dacă programul respectă regula de a nu le modifica direct.

```
const profil = {
  nume: 'Ana',
  contacte: [
    { id: 1, tip: 'email', valoare: 'ana@example.test' },
    { id: 2, tip: 'telefon', valoare: '0700000000' }
  ]
};

function schimbaContact(profil, id, valoare) {
  if (!profil.contacte.some(contact => contact.id === id)) {
    throw new Error('Contact inexistent');
  }
  return {
    ...profil,
    contacte: profil.contacte.map(contact =>
      contact.id === id ? { ...contact, valoare } : contact
    )
  };
}

const nou = schimbaContact(profil, 1, 'nou@example.test');
console.assert(nou !== profil);
console.assert(nou.contacte !== profil.contacte);
console.assert(nou.contacte[0] !== profil.contacte[0]);
console.assert(nou.contacte[1] === profil.contacte[1]);
console.assert(profil.contacte[0].valoare === 'ana@example.test');
```

Prima verificare confirmă un profil nou. A doua confirmă o listă nouă. A treia arată că obiectul modificat este separat, iar a patra demonstrează reutilizarea contactului neschimbat. Ultima verificare este cea care protejează comportamentul de anulare. Dacă ai testa numai valoarea din nou, ai putea rata faptul că și profilul inițial a fost alterat. Într-un test automat, folosește aserțiuni care opresc execuția la eșec; console.assert este util pentru explorare, dar nu reprezintă singur o suită de teste.

Destructuring la granița unei funcții

Un parametru obiect face apelul mai ușor de citit când există mai multe opțiuni. Totuși, un contract clar trebuie să spună ce valori sunt acceptate. Compară funcția de mai jos cu o versiune care ar primi trei argumente poziționale: la apel vezi direct ce opțiune modifici. Valoarea implicită a întregului parametru permite apelul fără argument. Ea nu validează orice intrare și nu înlocuiește null cu un obiect.

```
function configureaza({ tema = 'light', limita = 10 } = {}) {
  return { tema, limita };
}

console.assert(configureaza().limita === 10);
console.assert(configureaza({ limita: 0 }).limita === 0);
console.assert(configureaza({ limita: null }).limita === null);
```

În proiectul laboratorului, limita zero ar putea însemna că lista este ascunsă, iar null ar putea fi o intrare invalidă. Adaugă validarea în funcție de această decizie. Numele unei proprietăți poate fi redenumit în variabila locală, fără să fie modificat obiectul sursă. De exemplu, extragerea câmpului valoare ca valoareInitiala ajută când compari înainte și după. Folosește redenumirea pentru claritate, nu pentru a ascunde denumiri incoerente în date.

Combinarea opțiunilor și ordinea suprascrierii

Într-o configurare, scrie mai întâi opțiunile de bază și apoi opțiunile specifice dacă acestea trebuie să aibă prioritate. Dar definește separat ce înseamnă o proprietate prezentă cu undefined: poate suprascrie o valoare bună. De asemenea, combinarea la primul nivel a două obiecte care conțin un obiect aspect poate înlocui acel obiect întreg. Pentru a păstra culoarea când se schimbă doar dimensiunea, combină explicit și proprietățile din aspect.

Exercițiu și rezolvare de control

Adaugă un al treilea contact folosind o listă nouă, apoi modifică numai valoarea lui. Lista originală trebuie să rămână cu două contacte. Apelează funcția cu id 99 și verifică eroarea „Contact inexistent”. Pentru un rezultat complet, arată lungimile ambelor liste, valoarea veche, valoarea nouă și comparațiile de referințe. Dacă vrei ca un id absent să nu fie eroare, documentează alt contract; nu lăsa comportamentul să apară accidental din map.

Verifică salvarea și anularea ca două acțiuni diferite

Păstrează profil ca valoare confirmată și nou ca propunere de modificare. Acțiunea de anulare poate elimina propunerea și poate redesena formularul din profil. Acțiunea de salvare trebuie să trimită propunerea către serviciu și să confirme schimbarea numai după un răspuns reușit. Copierea obiectelor rezolvă independența datelor în memorie; nu garantează că serverul a salvat corect sau că alt utilizator nu a modificat între timp același profil.

Simulează o salvare eșuată și verifică dacă ai păstrat ambele valori. Utilizatorul trebuie să poată reîncerca fără să reintroducă toate câmpurile. Pentru acest experiment, este suficient să păstrezi o variabilă pentru profilul confirmat, una pentru propunere și un mesaj de eroare. Nu amesteca în același obiect datele confirmate și modificările temporare fără să precizezi care versiune este afișată. Compararea celor două versiuni este și punctul de pornire pentru evidențierea câmpurilor schimbate într-un formular.

Module ES

Laborator explicat: un catalog împărțit în fișiere

Vom organiza un catalog fictiv fără biblioteci externe. Creează un director nou, cu index.html, date.js, catalog.js și app.js. Servește acel director prin serverul local al editorului sau, dacă ai Python instalat, prin `python -m http.server 8000 --bind 127.0.0.1`. Accesează apoi `http://127.0.0.1:8000`. Nu rula serverul într-un director cu documente personale: fișierele acelui director pot fi accesate prin server. Laboratorul folosește numai fișierele demonstrative.

Datele și regula de selecție

Fiecare produs are identificator, nume, preț exprimat în bani și stare activă. Unitatea este aceeași pentru toate înregistrările. În catalog.js păstrăm funcția care selectează produsele, fără să citească pagina și fără să facă cereri. Acest contract permite verificarea regulii cu date scrise manual, înainte de orice integrare cu un serviciu extern.

```
// date.js
export const produse = [
  { id: 1, nume: 'A', pretBani: 9000, activ: true },
  { id: 2, nume: 'B', pretBani: 4000, activ: false },
  { id: 3, nume: 'C', pretBani: 12000, activ: true }
];

// catalog.js
export function selecteaza(produse, plafonBani) {
  return produse.filter(p => p.activ && p.pretBani <= plafonBani);
}
export function formatLei(bani) {
  return new Intl.NumberFormat('ro-RO', {
    style: 'currency', currency: 'RON'
  }).format(bani / 100);
}
```

În acest exemplu, datele sunt deja validate și au o structură cunoscută. Când provin dintr-un API, validarea trebuie făcută înainte de selecție. Funcția nu trebuie să decidă pe ascuns ce înseamnă un preț lipsă. Nu exportăm o variabilă fiscală cu un procent fix: formatarea sumei nu are nevoie de reguli de taxare, iar o asemenea constantă ar introduce o presupunere fără legătură cu obiectivul lecției.

Fișierul care conectează modulele

```
// app.js
import { produse } from './date.js';
import { selecteaza, formatLei } from './catalog.js';
const lista = document.querySelector('#produse');
const rezultat = selecteaza(produse, 10000);
console.assert(rezultat.length === 1 && rezultat[0].id === 1);
lista.replaceChildren(...rezultat.map(produs => {
  const rand = document.createElement('li');
  rand.textContent = `${produs.nume}: ${formatLei(produs.pretBani)}`;
  return rand;
})));

<!doctype html>
<html lang="ro">
<meta charset="utf-8">
<title>Catalog modular</title>
<ul id="produse"></ul>
<script type="module" src="./app.js"></script>
</html>
```

Pune fiecare fragment în fișierul indicat, fără marcasele de delimitare ale blocului de cod. În pagină trebuie să apară numai produsul A, cu valoarea de 90 lei formatată pentru limba română. Modulul de formatare primește bani, deci împarte la o sută exact la prezentare. Nu transmite 90 în loc de 9000: ai afișa 0,90 lei și ai investiga

greșit filtrul, deși problema este unitatea de măsură.

Depanarea încărcării, pas cu pas

Deschide panoul Network înainte de reîncărcarea paginii și caută cele trei fișiere JavaScript. Dacă unul răspunde cu 404, verifică mai întâi literele mari, extensia și directorul. Dacă răspunsul are status 200, dar conține HTML, serverul poate trimite pagina implicită în locul modulului cerut. Deschide răspunsul și corectează calea sau configurarea serverului. Nu schimba funcția selectează pentru o problemă de încărcare.

Dacă un export lipsește, verifică numele din fișierul sursă și cel din import. În laborator folosim exporturi nominale, pentru ca numele funcțiilor să fie explicite. Un export implicit poate fi potrivit în alte convenții, dar schimbarea stilului la întâmplare îngreunează căutarea. Păstrează aceeași convenție în această aplicație mică și evită importurile circulare: catalog.js nu are nevoie să importe app.js.

Exercițiu de extindere

Mută formatLei într-un fișier formatare.js, actualizează importul și verifică din nou pagina. Apoi schimbă plafonul la 13000: trebuie să apară A și C, în această ordine. Adaugă un produs inactiv cu preț zero și confirmă că este exclus. Salvează rezultatele și explică ce fișier ai modifica pentru o regulă nouă de selecție și ce fișier ai modifica pentru o altă prezentare a prețului. Separarea este utilă dacă aceste răspunsuri sunt clare.

Verificarea responsabilităților printr-o schimbare mică

Presupune că cerința devine „afișează și numărul produselor excluse”. Nu adăuga un querySelector în selectează doar ca să actualizezi un contor. Funcția de selecție poate continua să returneze lista, iar app.js poate calcula diferența dintre lungimea sursei și lungimea rezultatului. Astfel, regula rămâne utilizabilă și într-un test fără browser. Dacă rezultatul selecției devine mai complex, poți introduce un obiect cu lista și metadatele, dar actualizează explicit apelurile care îl consumă.

Acum imaginează-ți că numele produselor vine dintr-un fișier editat de altcineva. Introdu temporar numele „Promoție” și verifică afișarea. În acest laborator trebuie să vezi caracterele textului, fără un element strong nou. Alegerea.textContent este parte din contractul modulului de interfață. Funcția selectează nu are nevoie să transforme numele în HTML și nici să le curețe pentru afișare, deoarece ea decide doar ce produse trec filtrul.

Promise și async/await

Laborator explicat: ordinea operațiilor asincrone

Pentru început, eliminăm rețeaua din experiment. Un temporizator produce o valoare după un interval ales, astfel încât poți observa ordinea fără să depinzi de un serviciu extern. Creează un fișier `laborator.mjs` și rulează-l cu Node.js. Alternativ, pune codul într-un script de tip module în browser. În exemple, `await` de la nivelul principal este folosit într-un modul, nu într-un script clasic.

Prezice rezultatul înainte de execuție

```
const jurnal = [];  
function dupa(ms, valoare) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      jurnal.push(valoare);  
      resolve(valoare);  
    }, ms);  
  });  
}  
  
const rezultate = await Promise.all([  
  dupa(40, 'catalog'),  
  dupa(5, 'preferinte')  
]);  
console.log(jurnal);  
console.log(rezultate);  
console.assert(rezultate[0] === 'catalog');  
console.assert(rezultate[1] === 'preferinte');
```

Jurnalul urmărește finalizarea temporizatoarelor: în această demonstrație vei observa preferințele înainte de catalogul. Lista rezultate păstrează asocierea cu pozițiile din intrare. Folosește această diferență când conectezi două răspunsuri la două zone ale paginii. Nu presupune că primul rezultat este răspunsul cel mai rapid. Intervalele sunt orientative și pot crește într-o pagină ocupată sau suspendată; un temporizator nu este un cronometru cu execuție garantată la milisecundă.

Dependența decide ce poate porni împreună

Imaginează-ți că trebuie să încarci profilul, comenzile aceluși profil și un catalog public. Comenzile au nevoie de identificatorul profilului, dar catalogul nu. Desenează pe hârtie aceste relații înainte să scrii `await`. Poți porni catalogul independent și poți avea o funcție care citește profilul, apoi comenzile. Așteptarea celor două ramuri împreună exprimă dependențele reale și păstrează tratarea erorilor într-un loc previzibil.

Nu transforma însă orice listă de o mie de cereri într-un singur `Promise.all`. Serviciul poate limita traficul, iar aplicația poate consuma prea multe conexiuni sau memorie. Pentru un număr mare de operații, procesează loturi limitate și stabilește ce se întâmplă la eșec. O buclă cu `await` este potrivită când ai nevoie de ordine, când fiecare operație depinde de precedentă sau când limitezi intenționat solicitările. Alegerea ține de contract, nu de o regulă care interzice buclele.

Observă succesul parțial

```
const verificari = await Promise.allSettled([  
  Promise.resolve(['A', 'C']),  
  Promise.reject(new Error('Recomandări indisponibile'))  
]);  
const [catalog, recomandari] = verificari;  
console.assert(catalog.status === 'fulfilled');  
console.assert(recomandari.status === 'rejected');  
if (catalog.status === 'fulfilled') {  
  console.log('Produce:', catalog.value);  
}  
if (recomandari.status === 'rejected') {  
  console.log('Zona de recomandări nu este disponibilă.');}
```

În acest scenariu inventat, catalogul este util chiar dacă recomandările lipsesc. Decizia de a accepta succesul parțial este una de produs. Pentru o operație care are nevoie de toate datele, alege un rezultat comun care eșuează dacă o componentă eșuează. Niciuna dintre aceste metode nu face automat o tranzacție și nu inversează efectele operațiilor deja executate. Când operezi plăți sau modificări, ai nevoie de reguli ale serviciului, nu doar de combinarea promisiunilor.

Erorile trebuie așteptate în locul potrivit

O funcție async care aruncă o eroare produce o promisiune respinsă. Un try/catch în jurul unui apel fără await nu prinde respingerea viitoare. Așteaptă promisiunea în blocul try sau atașează un handler catch. Într-un handler de eveniment, asigură-te că funcția pornită are propria tratare a erorilor; browserul nu transformă automat respingerea într-un mesaj pentru utilizator.

Exercițiu și criterii de verificare

Înlocuiește valoarea „preferinte” cu o respingere după cinci milisecunde. Verifică faptul că rezultatul comun intră în catch, apoi observă că temporizatorul catalogului continuă să ruleze. Scrie de ce respingerea nu este anulare. Refă experimentul cu allSettled și arată cele două stări finale. Pentru varianta secvențială, apelează după de două ori, fiecare după await-ul precedent, și compară timpul aproximativ. Nu testa o durată exactă; testează valorile și dependențele.

La final trebuie să poți explica separat momentul pornirii, momentul finalizării, ordinea rezultatelor și locul unde ajunge eroarea. Aceste patru întrebări sunt mai utile pentru depanare decât înlocuirea mecanică a tuturor apelurilor then cu await.

Tratarea erorilor asincrone

Laborator explicat: date, listă goală, eroare și anulare

Construim un cititor de produse care poate fi verificat fără un server real. Funcția primește opțional implementarea transportului. În aplicație, aceasta este fetch; în verificări, îi oferim o funcție care produce un răspuns controlat. Astfel putem reproduce o eroare HTTP și un JSON invalid la fiecare rulare, fără să așteptăm să cadă un serviciu. Exemplul funcționează într-un browser modern sau într-un mediu Node.js care oferă fetch și Response.

Definește contractul cititorului

Funcția returnează o listă de produse validate sau respinge promisiunea. Nu returnează null la eroare și nu înlocuiește eșecul cu o listă goală. Pentru laborator, fiecare produs trebuie să aibă nume textual, un preț întreg nenegativ în bani și o stare activă booleană. Contractul este intenționat mic; un API real poate cere și identificatori unici sau alte câmpuri.

```
async function citeșteProduse(url, { signal, transport = fetch } = {}) {
  const raspuns = await transport(url, { signal });
  if (!raspuns.ok) throw new Error(`HTTP ${raspuns.status}`);
  const date = await raspuns.json();
  const valid = Array.isArray(date) && date.every(p =>
    p !== null && typeof p === 'object' &&
    typeof p.nume === 'string' &&
    Number.isSafeInteger(p.pretBani) && p.pretBani >= 0 &&
    typeof p.activ === 'boolean'
  );
  if (!valid) throw new Error('Structură de produse invalidă');
  return date;
}

const gol = await citeșteProduse('/demo', {
  transport: async () => new Response('[]', { status: 200 })
});
console.assert(gol.length === 0);
```

Lista goală este validă deoarece nu există niciun produs care să încalce structura. Un obiect cu proprietatea produse nu este însă lista cerută de această funcție. Dacă API-ul folosește acel format, adaptează contractul explicit, nu adăuga presupuneri pentru a face orice răspuns să treacă. În mod similar, statusul 204 nu oferă corp JSON: pentru un endpoint care îl permite trebuie tratat separat. Exemplul nostru cere un răspuns JSON inclusiv pentru lista goală.

Construiește o matrice mică de incidente

```
for (const [corp, status, mesaj] of [
  ['indisponibil', 503, 'HTTP 503'],
  ['{"produse":[]}'], 200, 'Structură de produse invalidă']
) {
  let eroare;
  try {
    await citeșteProduse('/demo', {
      transport: async () => new Response(corp, { status })
    });
  } catch (e) { eroare = e; }
  console.assert(eroare?.message === mesaj);
}
```

Adaugă separat un corp care conține doar acolada de început. Parsarea trebuie să eșueze înainte de validarea structurii. Aceste cazuri au cauze diferite: server indisponibil, contract greșit și document JSON corupt. În interfață poți afișa un mesaj simplu comun, dar în diagnostic păstrează categoria și contextul cererii. Nu afișa corpul integral al răspunsului ca HTML și nu trimite în jurnale date personale sau credențiale.

Limitează așteptarea și curăță temporizatorul

```
async function cuTimeout(url, transport = fetch) {
  const controller = new AbortController();
  const timer = setTimeout(() => controller.abort(), 5000);
  try {
    return await citeșteProduse(url, {
      signal: controller.signal, transport
    });
  } finally {
    clearTimeout(timer);
  }
}
```

Aici păstrăm `await` în interiorul `try` astfel încât `finally` să ruleze după încheierea citirii. Dacă ai returna direct promisiunea fără `await`, `finally` ar curăța imediat temporizatorul și limita de timp nu ar mai proteja operația. Anularea este transmisă transportului prin `signal`. O implementare de test care ignoră `signal` nu se oprește singură; de aceea verificarea anulării trebuie să folosească un transport care respectă semnalul.

Cine are dreptul să actualizeze interfața

Când utilizatorul schimbă repede un filtru, o cerere veche poate termina după una nouă. Păstrează un identificator crescător al cererii și verifică-l înainte de a aplica datele sau eroarea. Poți anula și cererea precedentă pentru a reduce lucrul inutil, dar identificatorul protejează rezultatul chiar dacă o operație nu mai poate fi anulată. Aceeași verificare trebuie făcută în `finally`: o cerere veche nu trebuie să stingă indicatorul de încărcare al celei noi.

Exercițiul final cere patru scenarii vizibile: listă cu produse, listă goală, eroare și reîncercare reușită. Adaugă un al cincilea scenariu în care două cereri termină în ordine inversă. Notează ce mesaj vede utilizatorul și ce cerere are voie să modifice starea. Butonul de reîncercare trebuie conectat o singură dată, la inițializare. Nu reinstala ascultătorul în fiecare `catch`, deoarece un singur click ar putea ajunge să pornească mai multe cereri.

Păstrează pentru fiecare verificare intrarea simulată și rezultatul observat. Dacă o eroare apare numai uneori, înlocuiește temporar serviciul cu transportul controlat și fixează ordinea răspunsurilor. Această reproducere este mai utilă decât apăsarea repetată a butonului până când problema dispare. Revino apoi la serviciul real și verifică integrarea, păstrând aceleași categorii de rezultate.

map / filter / reduce în cazuri reale

Laborator explicat: un raport verificabil al comenzilor

Un total corect întâmplător nu dovedește că raportul este corect. Vom păstra identificatorul fiecărei poziții, vom separa selecția de calcul și vom verifica lista contribuțiilor înaintea totalului. Toate sumele laboratorului sunt exprimate în bani întregi. Nu calculăm taxe și nu folosim un procent fiscal implicit. Scopul este să poți explica de unde vine fiecare număr, inclusiv când lista de intrare este goală.

Datele și verificările de intrare

```
const comenzi = [
  { id: 'A', cantitate: 2, pretBani: 1250, status: 'platit' },
  { id: 'B', cantitate: 1, pretBani: 800, status: 'anulat' },
  { id: 'C', cantitate: 3, pretBani: 100, status: 'platit' }
];

function raport(comenzi) {
  if (!Array.isArray(comenzi)) throw new TypeError('Listă necesară');
  const ids = new Set();
  for (const c of comenzi) {
    if (!c || typeof c.id !== 'string' || ids.has(c.id) ||
        !Number.isSafeInteger(c.cantitate) || c.cantitate <= 0 ||
        !Number.isSafeInteger(c.pretBani) || c.pretBani < 0 ||
        !['platit', 'anulat'].includes(c.status)) {
      throw new TypeError('Comandă invalidă sau duplicată');
    }
    ids.add(c.id);
  }
  const pozitii = comenzi
    .filter(c => c.status === 'platit')
    .map(c => ({ id: c.id, subtotalBani: c.cantitate * c.pretBani }));
  const totalBani = pozitii.reduce((s, c) => s + c.subtotalBani, 0);
  if (!pozitii.every(c => Number.isSafeInteger(c.subtotalBani)) ||
      !Number.isSafeInteger(totalBani)) {
    throw new RangeError('Totalul depășește precizia sigură');
  }
  return { pozitii, totalBani };
}

const rezultat = raport(comenzi);
console.assert(rezultat.pozitii.map(c => c.id).join(',') === 'A,C');
console.assert(rezultat.totalBani === 2800);
console.assert(raport([]).totalBani === 0);
```

Bucula de validare este intenționată: avem nevoie să verificăm duplicatele și să oprim procesarea la prima înregistrare invalidă. Nu este obligatoriu ca fiecare pas să fie scris cu reduce. Separarea face mai clară regula de acceptare. În acest model, inclusiv o comandă anulată trebuie să fie structurată corect; dacă ai filtra înainte de validare, ai ascunde înregistrări defecte. Într-o altă aplicație poți alege alt contract, dar decizia trebuie să fie explicită.

Urmărește transformările intermediare

După filter, lista conține A și C. După map, fiecare poziție are doar identificatorul și subtotalul: A contribuie cu 2500 de bani, C cu 300 de bani. Reduce pornește de la zero, adaugă primul subtotal, apoi al doilea. Rezultatul 2800 înseamnă 28 lei la prezentare. Păstrarea id-urilor permite să explici atât cifra finală, cât și excluderea comenzii B. Un raport care păstrează numai numerele pierde această posibilitate.

Validarea numerelor întregi nu este suficientă pentru valori arbitrar de mari. De aceea verificăm și produsul, și totalul, în domeniul întregilor siguri. Pentru sisteme financiare reale, alegerea reprezentării numerice și a regulilor de rotunjire trebuie făcută la nivelul aplicației. Laboratorul rezolvă un caz delimitat, cu valori mici și o singură monedă, fără conversii valutare.

Funcția nu trebuie să modifice sursa

Salvează `JSON.stringify(comenzi)` înainte de apel și compară-l cu valoarea de după apel. Pentru această structură simplă, comparația verifică faptul că valorile nu s-au schimbat. Dacă introduci sortare, sortează o listă nouă. Dacă modifizi un obiect în callback, faptul că `map` a creat un array nou nu te protejează de modificarea obiectului original. În exemplul nostru, `map` construiește obiecte noi pentru raport și citește doar câmpurile comenzilor.

Exerciții cu răspunsuri de control

Schimbă statusul comenzii B în platit. Totalul trebuie să fie 3600 de bani, iar raportul să conțină A, B și C. Adaugă o copie a comenzii A cu același identificator: funcția trebuie să respingă intrarea, nu să dubleze venitul. Schimbă cantitatea în textul „2”: trebuie să apară eroare, deoarece laboratorul cere un număr. Pentru o listă numai cu comenzi anulate, rezultatul valid este o listă de poziții goală și total zero.

Ca extensie, grupează pozițiile pe client, păstrând separat lista id-urilor și suma fiecărui grup. Verifică faptul că suma grupurilor este egală cu totalul general. Dacă această egalitate nu ține, inspectează asocierea fiecărei comenzi, nu ajusta cifra finală. Pentru chei primite din exterior poți folosi `Map`, astfel încât numele unui client să nu se confunde cu proprietăți moștenite ale unui obiect. Predă datele de intrare, raportul intermediar și explicația pentru fiecare caz respins.

Extinderea raportului cu o perioadă

Dacă adaugi un filtru de dată, stabilește mai întâi ce dată folosești: data creării comenzii sau data plății. Cele două pot aparține unor luni diferite. Într-un raport al încasărilor, folosirea datei creării ar muta contribuția într-o perioadă greșită chiar dacă suma generală rămâne corectă. Definește intervalul cu început inclus și sfârșit exclus, ca să poți alătura două perioade fără dublarea limitei comune.

Construiește două înregistrări chiar la granița perioadei și una imediat înainte. Notează fusul orar al datelor înainte de conversie și verifică fiecare includere. Nu extrage luna prin tăierea primelor caractere dintr-un șir arbitrar fără să cunoști formatul. Pentru laborator poți utiliza date normalizate, dar în aplicație normalizarea și interpretarea perioadei trebuie să aibă un contract separat de agregarea sumelor.

Organizarea codului fără framework

Laborator explicat: un flux complet de stare și afișare

Acest laborator leagă lecțiile anterioare într-un catalog mic. Regula de selecție rămâne aceeași: produse active, cu preț cel mult egal cu plafonul. Interfața are un mesaj de stare și o listă. Vom separa transformarea datelor de efectele asupra paginii, apoi vom controla răspunsurile asincrone. Nu construim un framework propriu; avem nevoie de câteva funcții cu intrări și responsabilități clare.

Modulul care calculează lista vizibilă

```
// logic.js
export function vizibile(produse, plafonBani) {
  return produse
    .filter(p => p.activ && p.pretBani <= plafonBani)
    .sort((a, b) => a.pretBani - b.pretBani);
}
```

Filter produce o listă nouă, deci sortarea din exemplu nu reordonează lista inițială. Obiectele produselor rămân însă referințe comune: funcția le citește, nu le modifică. Verifică acest contract cu datele A, B și C din lecția despre module. La plafonul 10000 rezultatul este A; la 13000 este A, C. Logica nu are nevoie de document, de un buton sau de o conexiune la server.

Modulul care deține efectele asupra paginii

```
// view.js
export function afiseaza(stare, produse) {
  const mesaj = document.querySelector('#stare');
  const lista = document.querySelector('#produse');
  mesaj.textContent = stare.status === 'loading' ? 'Se încarcă...'
    : stare.status === 'error' ? 'Încărcare eșuată. Reîncearcă.'
    : produse.length ? `${produse.length} produse`
    : 'Niciun produs pentru filtrul ales.';
  lista.replaceChildren(...produse.map(p => {
    const li = document.createElement('li');
    li.textContent = `${p.numere}: ${(p.pretBani / 100).toFixed(2)} lei`;
    return li;
  }));
}
```

Această funcție are efecte: modifică DOM-ul. Nu o numim funcție pură. Separarea utilă este că nu pornește cereri și nu schimbă starea aplicației. Numele produsului este introdus prin textContent, astfel încât caracterele de marcaj sunt afișate ca text. Dacă introduci în fixture un nume care conține un tag HTML, pagina nu trebuie să creeze acel element. Pentru un preț afișat conform convențiilor locale, poți reutiliza formatLei din lecția despre module.

Coordonarea și protejarea rezultatului recent

```
// main.js
import { vizibile } from './logic.js';
import { afiseaza } from './view.js';
let stare = { status: 'idle', produse: [], plafonBani: 10000 };
let ultimaCerere = 0;
let controller;
function actualizeaza(patch) {
  stare = { ...stare, ...patch };
  const lista = stare.status === 'success'
    ? vizibile(stare.produce, stare.plafonBani) : [];
  afiseaza(stare, lista);
}
export async function incarca(citeste) {
  const cerere = ++ultimaCerere;
  controller?.abort();
  controller = new AbortController();
  const signal = controller.signal;
  actualizeaza({ status: 'loading' });
  try {
    const produse = await citeste(signal);
    if (cerere !== ultimaCerere) return;
    actualizeaza({ status: 'success', produse });
  } catch (eroare) {
    if (cerere !== ultimaCerere) return;
    actualizeaza({ status: 'error' });
    console.error(eroare);
  }
}
document.querySelector('#plafon').addEventListener('change', event => {
  actualizeaza({ plafonBani: Number(event.target.value) });
});
const citeste = async signal => {
  const r = await fetch('./produse.json', { signal });
  if (!r.ok) throw new Error(`HTTP ${r.status}`);
  const date = await r.json();
  if (!Array.isArray(date) || !date.every(p =>
    p && typeof p.nume === 'string' && typeof p.activ === 'boolean' &&
    Number.isSafeInteger(p.pretBani) && p.pretBani >= 0
  )) throw new Error('Produse invalide');
  return date;
};
document.querySelector('#reincearca').addEventListener('click', () => {
  void incarca(citeste);
});
void incarca(citeste);

<!doctype html>
<html lang="ro">
<meta charset="utf-8">
<title>Catalog cu stare</title>
<label>Plafon
  <select id="plafon">
    <option value="10000">100 lei</option>
    <option value="13000">130 lei</option>
  </select>
</label>
<button id="reincearca">Reîncarcă</button>
<p id="stare" role="status"></p>
<ul id="produse"></ul>
<script type="module" src="./main.js"></script>
</html>
```

Pune în produse.json lista din date.js, în format JSON valid: fără export, fără declarația const și cu cheile între ghilimele duble. Servește directorul prin HTTP local, ca în lecția despre module. Selectul oferă doar două valori numerice cunoscute; dacă îl înlocuiești cu un câmp liber, validează valoarea înainte de actualizarea stării. La încărcare și eroare, acest exemplu golește lista. Alte interfețe pot păstra rezultatul anterior, dar trebuie să explice că acesta nu este actualizat.

De ce avem și anulare, și identificator

Anularea încearcă să oprească munca inutilă. Identificatorul decide dacă rezultatul mai este relevant pentru ecran. Dacă un transport de test ignoră anularea, verificarea identificatorului continuă să protejeze interfața. Observă că și ramura catch face aceeași verificare: altfel eroarea unei cereri vechi ar putea înlocui succesul uneia recente. În acest flux, fiecare cerere relevantă ajunge fie la success, fie la error, astfel încât nu avem un al doilea indicator loading care ar putea rămâne în urmă.

Verificarea atelierului

Testează pe rând o listă validă, lista goală, un fișier lipsă și un obiect în loc de listă. Pentru primele două cazuri, mesajele trebuie să descrie succesul cu rezultate sau fără rezultate. Pentru ultimele două, mesajul trebuie să indice eșecul și să permită reîncercarea. La schimbarea plafonului nu trebuie pornită o cerere nouă: filtrarea se face din datele deja încărcate.

Pentru cursa dintre cereri, importă incarca într-un script de test din aceeași pagină și oferă două funcții care rezolvă liste diferite după intervale diferite. Pornește mai întâi varianta lentă, apoi varianta rapidă. La final trebuie să rămână lista celei de-a doua cereri, chiar dacă prima termină ulterior. Salvează observațiile și explică separat rolul fiecărui modul. Atelierul este încheiat când poți reproduce și explica inclusiv eșecurile, nu doar când pagina arată bine cu fixture-ul inițial.