**Seminarul 8 — REST cu Node/Express: rute, validări, erori  
Partea 1: Teorie (extinsă)**

**Hook realist: 48 de ore până la lansare — cum proiectezi un REST API care „ține” în fața UI‑ului și a realității**

Imaginați-vă că sunteți responsabil de infrastructura pentru „Clubs & Associations Hub”, un portal universitar care trebuie să deschidă înscrierile la cluburi studențești vineri, la ora 18:00. UI‑ul (partea client) e deja conturat — formulare accesibile, Fetch API robust, validări client‑side, micro‑cache și mesaje prietenoase, așa cum le-am dezvoltat la Seminarul 7. Lipsă: un serviciu REST care să livreze datele despre cluburi și să primească înscrierile. Constrângeri: disponibilitate rezonabilă, timp de răspuns previzibil, erori inteligibile (în format standard), reguli clare pentru duplicate, trasabilitate (un „correlation-id” pentru fiecare cerere) și o arhitectură suficient de curată încât să poată fi extinsă rapid după lansare.  
În asemenea contexte, a greși „mic” (de exemplu, să mapi un caz de input invalid la 500) nu înseamnă doar o „nota proastă” la cod — înseamnă suport supraaglomerat, încredere scăzută și costuri reputaționale. În plus, tot ce face UI‑ul pe client se reflectă în server: dacă acolo vorbim de \*\*timeout\*\*, \*\*retry\*\* și \*\*problem+json\*\*, atunci serverul trebuie să aibă contracte ferme și o gestiune a erorilor care să reducă ambiguitățile. În această parte teoretică vom construi „harta mentală” a unui API REST solid în \*\*Node.js + Express\*\*: de la modelarea resurselor și semantica HTTP până la validări stratificate, erori standardizate și politici precum ETag, CORS și idempotency‑key.

**1. Modelarea resurselor: substantivul potrivit, identități stabile, relații clare**

În REST, rutele sunt organizate în jurul \*\*resurselor\*\* (substantive), nu al acțiunilor. Pentru domeniul nostru, resursele de bază sunt \*\*clubs\*\* și \*\*registrations\*\*. Un club reprezintă o entitate cu identitate stabilă (de exemplu `id: "tech"` sau un UUID), iar o înregistrare (registration) exprimă un eveniment de interes: faptul că o persoană a solicitat intrarea într‑un club.

\*\*Nomenclatură și identitate.\*\* Este esențial să alegem chei care nu se schimbă arbitrar. Dacă folosim ID‑uri simbolice (ex. `tech`), documentăm regulile (caractere permise, stabilitate). Dacă folosim ID‑uri opace (UUID v4), păstrăm consecvența la toate rutele.

\*\*Relații.\*\* În mod tipic, `registrations` depind de `clubs`: fie ca simplă referință (câmp `club` în obiectul înscrierii), fie ca sub‑resursă (`/api/clubs/:id/registrations`). Pentru claritate și izolare, un `POST /api/registrations` cu `club: id` este deseori suficient; sub‑resursele sunt utile când vrem listări contextualizate (`GET /api/clubs/:id/registrations`).

\*\*Invariante și reguli de business.\*\* Modelarea nu înseamnă doar schema de date: presupune documentarea constrângerilor semantice (ex. un `club` poate fi „archived” și atunci `POST /api/registrations` trebuie să producă `409 Conflict` sau `422 Unprocessable Content`, în funcție de regulament). Aceste reguli se exprimă în validatori și se reflectă în codurile de stare HTTP corecte.

Recomandare pragmatică: începeți cu un minim coerent — `GET /api/clubs`, `GET /api/clubs/:id`, `POST /api/registrations`, `GET /api/registrations?club=:id`. Adăugați incremental `POST /api/clubs`, `PATCH/DELETE` atunci când guvernanța datelor o cere.

**2. Semantica HTTP (RFC 9110): metode, idempotency, coduri de stare**

\*\*Metode.\*\* `GET` este sigur (safe) și idempotent, `HEAD` este `GET` fără body, `POST` nu este idempotent (de obicei creează resurse), `PUT` este idempotent (înlocuire totală), `PATCH` exprimă modificări parțiale (cu semantica definită de tipul de patch, de ex. JSON Merge Patch, RFC 7386), `DELETE` este idempotent în sens larg (ștergerea repetată a aceleiași resurse are același efect final: resursa nu mai există).

\*\*Idempotency și retry.\*\* Pentru a depăși întreruperi de rețea sau răspunsuri 5xx, clienții pot reîncerca cereri \*\*idempotente\*\*. La `POST`, care nu este idempotent, putem introduce un \*\*Idempotency‑Key\*\*: un identificator unic pentru o intenție de creare; serverul îl folosește pentru a „coase” cererile duplicate.

\*\*Status codes.\*\* Câteva mape tipice:  
- `200 OK` la `GET`/`PATCH` reușite; `201 Created` la `POST` reușit (include un `Location` cu URL‑ul noii resurse).   
- `204 No Content` la `DELETE` reușit sau la `PATCH` fără body în răspuns;   
- `400 Bad Request` pentru input imposibil de interpretat (JSON invalid, parametri lipsă),   
- `401 Unauthorized`/`403 Forbidden` pentru probleme de autentificare/autorizare (dacă le avem),   
- `404 Not Found` când resursa lipsește,   
- `409 Conflict` pentru ciocniri (ex. încercarea de a crea un club cu un `id` existent sau de a modifica un document fără `If‑Match` potrivit),   
- `422 Unprocessable Content` când JSON‑ul este \*\*sintactic\*\* valid, dar \*\*semantic\*\* invalid (ex. email greșit, lipsă consimțământ),   
- `429 Too Many Requests` dacă aplicăm rate‑limit,   
- `5xx` pentru erori server.

\*\*Coregrafia cu UI‑ul.\*\* Dacă în UI mapăm `422` la „verifică câmpurile”, `409` la „resursa a fost schimbată între timp” și `5xx` la „încearcă mai târziu”, serverul trebuie să respecte \*\*constant\*\* aceleași semnale. Astfel evităm „mesaje generice” care nu ajută utilizatorul și nu reduc presiunea pe suport.

**3. Reprezentări JSON (RFC 8259) și convenții de modelare**

\*\*Chei și caz.\*\* Păstrați \*\*camelCase\*\* pentru chei (`createdAt`, `updatedAt`), cu unități temporale în \*\*ISO‑8601\*\* (RFC 3339). Folosiți enumerări cu valori string („`pending`”, „`approved`”), nu numere „magice”. `id` trebuie să fie stabil și opac din perspectiva clientului.

\*\*Colecții.\*\* `GET /api/clubs` poate întoarce un obiect cu `items` și `page/limit/total`, sau lista „goală” plus `Link` headers pentru navigare. Ambele sunt valide; consecvența contează. Pentru laborator vom folosi `{ items, page, limit, total }`.

\*\*Filtrare și sortare.\*\* Parametrii de query sunt adecvați (`?category=arts&sort=name,-createdAt`). Documentați clar semantica sortării multi‑cheie și ordinea (`-` pentru descrescător).

\*\*Validare la ieșire (output).\*\* Nu este obișnuință, dar e sănătos: înainte de a răspunde, treceți obiectele printr‑un strat de validare „de tip” pentru a asigura că nu scăpați date interne (PII sau câmpuri temporare). Aceasta previne „leaking by accident”.

**4. Erori standardizate cu RFC 7807: `application/problem+json`**

Formatul \*\*Problem Details for HTTP APIs\*\* (RFC 7807) aduce structură în erori:   
- `type` (URI care descrie erorile de acest tip; poate fi un URL intern de documentație),   
- `title` (rezumat scurt),   
- `status` (cod HTTP),   
- `detail` (descriere specifică instanței),   
- `instance` (URI la care se referă eroarea — opțional).   
Putem include câmpuri suplimentare (`errors` cu harta câmp‑eroare) pentru validări. Cheia este \*\*consistența\*\*: UI‑ul se poate baza pe aceste câmpuri pentru a afișa mesaje prietenoase și pentru a face mappingul uniform al erorilor.

HTTP/1.1 422 Unprocessable Content  
Content-Type: application/problem+json  
  
{  
 "type": "https://api.example.edu/problems/validation",  
 "title": "Validation failed",  
 "status": 422,  
 "detail": "Some fields are invalid.",  
 "errors": {  
 "email": "Invalid institutional email",  
 "consent": "Consent is required"  
 }  
}

Un mecanism intern de fabrică (ex. `problem({ status, title, detail, errors })`) înscrie toate erorile într‑un format uniform și reduce apariția răspunsurilor „improvizate”.

**5. Caching & concurență: ETag, condiționale, 304 și 409**

\*\*ETag\*\* este o amprentă a reprezentării. Dacă răspundeți cu `ETag: "W/abc123"` la `GET`, clientul poate trimite ulterior `If-None-Match: "W/abc123"`. Dacă reprezentarea nu s‑a schimbat, serverul răspunde \*\*`304 Not Modified`\*\* (fără body). Aceasta reduce traficul și timpul perceput de utilizator.

Pentru actualizări, folosiți `If-Match`: clientul trimite ETag‑ul versiunii pe care crede că o modifică; dacă între timp resursa s‑a schimbat, serverul răspunde \*\*`412 Precondition Failed`\*\* sau \*\*`409 Conflict`\*\*. Aceasta previne suprascrierea „oarbă”. În laborator, putem demonstra conceptul cu un ETag „weak” calculat simplu (ex. un hash al serializării).

**6. CORS și origin (RFC 6454): principiile minime**

\*\*Origin\*\* este perechea schemă + host + port. Când UI‑ul nostru rulează pe `http://localhost:5270` și back‑end‑ul pe `http://localhost:5380`, cererile sunt \*\*cross‑origin\*\* și intră sub incidența \*\*CORS\*\*. Serverul trebuie să răspundă cu antetele corespunzătoare (`Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, `Access-Control-Allow-Headers`). Anumite cereri declanșează preflight `OPTIONS`. Pentru laborator, folosim o configurație prudentă (`cors()` cu o listă de origini permise), iar pentru producție definim politici mai stricte (inclusiv `Vary: Origin`).

**7. Express: Router, middlewares și funcțiile handler — ordinea contează**

\*\*Express\*\* este un micro‑framework: oferă `app`, `Router`, `middlewares` compozabile și un mecanism pentru gestionarea erorilor (handlers cu patru argumente). Deși „mic”, Express impune o disciplină: ordinea middleware‑urilor determină comportamentul.

\*\*Structură recomandată.\*\* Separăm `routes/` (definirea URIs), `controllers/` (logica de „acceptare a cererii și returnare a răspunsului”), `services/` (reguli de business), `schemas/` (validări), `middlewares/` (CORS, correlation‑id, error handling etc.), `errors/` (fabrica de `problem`).   
Această separare reduce cuplajul: putem modifica regulile de business fără a atinge rutele, și putem schimba serializarea erorilor fără a rescrie controller‑ele.

// exemplu schematic de structură (pseudo‑cod)  
/api  
 app.js  
 routes/  
 clubs.routes.js  
 registrations.routes.js  
 controllers/  
 clubs.controller.js  
 registrations.controller.js  
 services/  
 clubs.service.js  
 registrations.service.js  
 schemas/  
 club.schema.js  
 registration.schema.js  
 middlewares/  
 validate.js  
 error.js  
 cors.js  
 correlationId.js  
 notFound.js  
 errors/  
 problem.js  
 httpErrors.js  
 repo/  
 inMemoryStore.js

**8. Validări: de la predicate simple la JSON Schema — input și output**

\*\*Validare sintactică\*\* (ex. „email are formă corectă?”, „consent este boolean?”) și \*\*validare semantică\*\* (ex. „clubul există și acceptă înscrieri?”) trebuie netezite în straturi. În laborator, folosim „predicate pure” (funcții care întorc `{ ok, errors }`) pentru claritate și testabilitate. Într‑un produs matur, putem ridica nivelul cu \*\*JSON Schema\*\* (și validatoare precum Ajv) sau cu biblioteci de tipuri la runtime (ex. Zod).

\*\*Validare de ieșire\*\*: aplicăm un filtru și la răspuns (output) pentru a evita expunerea accidentală a câmpurilor private (ex. `internalNotes`). Chiar dacă pare redundant, această plasă de siguranță reduce riscuri reale.

**9. Idempotency‑Key la POST: de ce contează și cum îl proiectăm**

La `POST`, întreruperile pot produce cereri duplicate. Dacă UI‑ul face `retry` la un 5xx, fără \*\*idempotency key\*\*, am putea crea două înregistrări identice. Soluția: clientul generează un `Idempotency-Key: <UUID>`, iar serverul memoizează rezultatul pentru (cheie, semnificativ‑după‑corp). La cereri repetate cu aceeași cheie, ori returnăm același `201` (și același `Location`/`id`), ori semnalăm că operația a fost deja procesată (`200`/`201`), fără a crea duplicate.

În laborator, demonstrăm ideea cu un \*\*in‑memory store\*\* (mapă key → rezultat). În producție, cheia trebuie păstrată într‑un storage persistent și are fereastră de valabilitate (TTL).

**10. Observabilitate: correlation‑id, loguri structurate, non‑PII**

Când apare o reclamație („Mi‑a apărut eroare după ce am apăsat Înscrie‑mă!”), vrei să poți reconstrui traseul cererii. Introducem un `X‑Correlation‑Id` (dacă nu e prezent, generăm noi un UUID), îl punem în `res.setHeader` pentru răspuns și îl includem în loguri.

\*\*Loguri structurate\*\* (JSON) fac căutarea mai eficientă; includem timestamp, ruta, metoda, statusul, durata și correlation‑id; nu includem PII în loguri. Această disciplină scade timpul de diagnostic și ridică calitatea suportului.

// middleware (schematic) pentru correlation‑id  
import { randomUUID } from 'node:crypto';  
export function correlationId(req, res, next){  
 const incoming = req.get('X-Correlation-Id');  
 const cid = incoming && /^[A-Za-z0-9-]{8,}$/.test(incoming) ? incoming : randomUUID();  
 res.setHeader('X-Correlation-Id', cid);  
 req.correlationId = cid;  
 next();  
}

**11. Versionare API: path vs. antete și principiile compatibilității**

Strategiile comune: `GET /api/v1/clubs` (în path) sau `Accept: application/vnd.example.v1+json` (în antete). Pentru laborator, varianta în path este mai simplă și mai vizibilă. \*\*Compatibilitate\*\*: schimbările care rup clienții (breaking changes) cer un nou `v2`; adăugarea de câmpuri opționale sau de rute noi este, în general, compatibilă. Documentați clar politicile de deprecieri și timeline‑urile.

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

• \*\*Coduri de stare amestecate\*\*: a returna `200 OK` cu un body de eroare te face neinteroperabil — folosește `4xx/5xx` potrivit și `problem+json`.   
• \*\*Mesaje opace\*\*: „Something went wrong” nu ajută; `title/detail` trebuie să indice remediul când e posibil.   
• \*\*Eșec la validare fără granularitate\*\*: returnezi un „invalid input” fără a indica ce câmp e greșit — adaugă harta `errors`.   
• \*\*Lipsa `Content-Type` corect\*\*: răspunzi JSON fără `application/json` — clienții pot eșua la parsing.   
• \*\*Lipsa `Location` la `201 Created`\*\*: rupe așteptările; include URL‑ul noii resurse.   
• \*\*Fără `X‑Correlation‑Id`\*\*: depanarea devine ghicit.   
• \*\*CORS „deschis” în producție\*\*: `\*` fără discernământ — restrânge la origini cunoscute și setează `Vary: Origin`.

**13. Interoperabilitate cu UI‑ul: contracte stabile și mesaje coerente**

Partea client se bazează pe:   
- \*\*coduri\*\* (`422` → evidențiere câmpuri, `409` → reîncarcă și reconciliază),   
- \*\*format\*\* (JSON, `problem+json`),   
- \*\*antete\*\* (CORS, `Content‑Type`, `Cache‑Control`),   
- \*\*performanță\*\* (răspuns previzibil, TTL‑uri moderate).   
În absența acestora, UI‑ul devine complex și fragil. Un API bine proiectat simplifică codebase‑ul client, clarifică testarea și scade costul mentenanței.

**14. Strategia de testare (în general, fără a implementa încă): unit, integrări HTTP, contracte**

\*\*Unit\*\*: validatori (email, nume, semantici club), generatorii de `problem+json`, serviciile (create/list/delete) pe un repo in‑memory.   
\*\*HTTP (Supertest)\*\*: rute `GET/POST/PATCH/DELETE`, folosind Express „în memorie”; verificăm codurile, antetele și formele JSON.   
\*\*Contracte\*\*: validarea formelor la ieșire (fie predicate simple, fie JSON Schema) pentru a asigura că nu „alunecă” câmpuri.   
\*\*Observabilitate\*\*: testăm prezența `X‑Correlation‑Id` în răspuns.   
Testele trebuie să acopere atât ferestrele de succes, cât și căile de eroare cele mai probabile (ex. `422`, `404`, `409`).

**15. Explicații de design pe exemple (micro‑fragmente de cod)**

Nu implementăm încă laboratorul, dar ilustrăm câteva puncte cheie cu micro‑fragmente. Fie ca idee de design: un router pentru `clubs`, un strat de validare, o fabrică `problem()` și handlerul de erori. Acestea vor fi elaborate complet în Partea 2.

// routes/clubs.routes.js (schematic)  
import { Router } from 'express';  
import \* as ctrl from '../controllers/clubs.controller.js';  
import { validate } from '../middlewares/validate.js';  
import { createClubSchema, patchClubSchema } from '../schemas/club.schema.js';  
  
export const router = Router();  
router.get('/', ctrl.listClubs);  
router.get('/:id', ctrl.getClub);  
router.post('/', validate(createClubSchema), ctrl.createClub);  
router.patch('/:id', validate(patchClubSchema), ctrl.patchClub);  
router.delete('/:id', ctrl.deleteClub);

\*\*Observație.\*\* `validate(schema)` este un middleware care parsează `req.body`/`req.query` și, dacă găsește erori, ridică o excepție ce va fi serializată de `errorHandler` în `problem+json` (`422`).

// middlewares/error.js (schematic)  
import { isHttpError, toProblem } from '../errors/problem.js';  
  
export function errorHandler(err, req, res, next){  
 // mapping generic → problem+json  
 const problem = toProblem(err, { instance: req.originalUrl });  
 res.status(problem.status || 500).type('application/problem+json').send(problem);  
}

\*\*Idee.\*\* `toProblem` convertește erorile „interne” (de ex. `ValidationError`, `NotFound`, `Conflict`) în obiectul standardizat. În acest fel, orice excepție iese din aplicație într‑un format uniform.

// schemas/registration.schema.js (schematic predicate)  
export function validateRegistration(input){  
 const errors = {};  
 const fullName = (input.fullName||'').trim();  
 const email = (input.email||'').trim();  
 const club = (input.club||'').trim();  
 const consent = !!(input.consent === true || input.consent === 'true' || input.consent === 'on');  
  
 if(fullName.length < 3) errors.fullName = 'Full name must have at least 3 chars';  
 if(!/^[^@\s]+@[^@\s]+\.[^@\s]+$/.test(email)) errors.email = 'Invalid email';  
 if(!club) errors.club = 'Club is required';  
 if(!consent) errors.consent = 'Consent is required';  
  
 return { ok: Object.keys(errors).length===0, data: { fullName, email, club, consent }, errors };  
}

\*\*Observație.\*\* Validatorul e \*\*pur\*\*: nu atinge IO, returnează un verdict predictibil și este testabil izolat.

**16. Securitate minimală (în contextul laboratorului)**

Deși laboratorul nu implementează autentificare/autorizație, menționăm câteva reguli sănătoase:   
- \*\*Input validation\*\*: whitelisting, limite de lungime, regex pentru email, evitarea câmpurilor „neprevăzute” (reject unknown).   
- \*\*Rate‑limit\*\* (concept): previne abuzul; dacă îl adăugăm, mapăm `429`.   
- \*\*Headers\*\*: `Content‑Type` corect, `Cache‑Control` potrivit (de ex. `no-store` pentru endpoints sensibile), `X‑Content‑Type‑Options: nosniff`.   
- \*\*CORS\*\*: lista albă pentru origini; evitați `\*` la credențiale.   
- \*\*Loguri\*\*: fără PII; corelați cu `X‑Correlation‑Id`.   
Acest „minim de igienă” face ca trecerea la un produs real să nu necesite rescrierea întregii stive.

**17. Cum ajută AI‑assist (Copilot/LLM) în VSL (Very Short Loop) fără a ceda controlul**

\*\*Scopul\*\* este accelerarea, nu substituirea judecății. Exemple de bucle scurte:   
- „Generează un `Router` pentru `/api/clubs` cu `GET/POST/PATCH/DELETE` și include importurile potrivite.”   
- „Propune o funcție `problem()` compatibilă RFC 7807 și 5 teste unitare care validează câmpurile `type/title/status/detail`.”   
- „Scrie `validateRegistration` (predicate pure) + teste pentru cazuri limită (spații, diacritice, email nevalid, lipsă consent).”   
- „Sugerează schelet pentru `Idempotency-Key` cu un in‑memory map și TTL.”   
După generare, \*\*revizuim\*\*: denumiri, separare, comentarii operaționale (concentrate, nu roman), semantica status‑urilor.

**18. Checklist teoretic — după Partea 1 ar trebui să poți:**

• Modela resursele principale (`clubs`, `registrations`) cu identități stabile și relații clare.   
• Explica și aplica semantica metodelor HTTP și a codurilor de stare (2xx/4xx/5xx) în scenarii reale.   
• Proiecta reprezentări JSON coerente (camelCase, ISO‑8601, paginare/sortare/filtrare).   
• Produce și consuma erori standardizate `application/problem+json` (RFC 7807), inclusiv hărți `errors` pentru validări.   
• Înțelege și utiliza ETag/condiționale (`If‑None‑Match`/`If‑Match`) pentru caching și concurență.   
• Configura CORS într‑o manieră prudentă, înțelegând preflight și `Origin`.   
• Organiza un proiect Express pe layere (routes/controllers/services/schemas/middlewares/errors).   
• Argumenta folosirea validării la intrare \*\*și\*\* la ieșire, cu predicat sau JSON Schema.   
• Descrie rolul `Idempotency‑Key` la `POST` și cum se implementează pragmatic.   
• Defini un minim de observabilitate (correlation‑id, loguri structurate) și reguli de igienă de securitate.

**19. Referințe (APA 7, cu DOI real)**

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

Reschke, J. (2022). HTTP/1.1 (RFC 9112). RFC Editor. https://doi.org/10.17487/RFC9112

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

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

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

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

Hunt, P., Bryan, P., & Nottingham, M. (2015). JSON Merge Patch (RFC 7386). RFC Editor. https://doi.org/10.17487/RFC7386

Ramaswamy, R., et al. (2010). HTTP Cache-Control Extensions for Stale Content (RFC 5861). RFC Editor. https://doi.org/10.17487/RFC5861

Berners‑Lee, T., Fielding, R., & Masinter, L. (2005). Uniform Resource Identifier (URI): Generic Syntax (RFC 3986). RFC Editor. https://doi.org/10.17487/RFC3986