**Seminarul 9 — Persistență SQLite/Sequelize — Partea 2 (Laborator extins)**

Context și obiectiv general  
---------------------------  
Această parte a seminarului transformă teoria din Partea 1 într‑un laborator aplicat, integral executabil, organizat pe etape progresive. Vei porni de la un proiect Node.js + Express + Sequelize configurat pentru SQLite (fișier în modul development și „sqlite::memory:” în test), vei defini modele și asocieri, vei crea și verifica tranzacții, vei testa integritatea referențială (FK) și vei compara încărcarea „lazy” cu „eager” (anti‑N+1). Laboratorul include un „worksheet” (cerință + checklist) pentru fiecare etapă, „starter code” pregătit și \*\*teste automate\*\* în \*\*Vitest\*\* și \*\*Jest\*\* rulate side‑by‑side.

Principiul de lucru este \*\*VSL (Very Short Loop)\*\*: implementezi un pas mic, rulezi testele, verifici feedback‑ul, ajustezi. Orice bloc de cod sau comandă CLI este prezentat în blocuri separate, precedate și urmate de explicații. În plus, ofer prompturi concrete pentru \*\*GitHub Copilot\*\* (sau alt LLM) în dreptul activităților repetitive sau pretabile la completare automată (modele, migrații, teste).

Structură pe etape (rezumat):  
0) Setup proiect (repo, dependențe, scripturi, structură directoare).   
1) Modele & asocieri (Club, Member, Registration — through, Event).   
2) Validări și unicități (email, UNIQUE pe Club.name).   
3) Interogări & eager loading (include, attributes) — anti‑N+1.   
4) Tranzacții (inscriere membru + rollback/commit + invarianta contor).   
5) Migrații & seeders (separarea schemei de date de cod).   
6) PRAGMA & indexuri (WAL, foreign\_keys, indexe utile).   
7) Testare automată (Vitest/Jest, sqlite::memory:, fixtures).

\*\*Fișierele livrate\*\* (arhiva ZIP „s9p2-standalone.zip”) conțin proiectul starter, configurile de test, și scripturile npm. Poți rula: `npm i && npm test` (execută Vitest + Jest) și `npm run dev` pentru a porni un server Express minimal cu health check (`/health`).

ETAPA 0 — Setup proiect (worksheet + checklist)  
-------------------------------------------------  
\*\*Obiectiv:\*\* inițializezi un proiect Node.js pentru SQLite + Sequelize, cu testare Vitest și Jest side‑by‑side, replicând pattern‑ul didactic folosit în seminariile anterioare.

\*\*Cerință:\*\* creează structura de directoare, adaugă dependențele necesare, definește scripturile npm pentru `dev`, `test:vitest`, `test:jest` și `test`, și pregătește configuri minime (`vitest.config.ts`, `jest.config.cjs`).

\*\*Checklist minim:\*\*  
- [ ] `package.json` conține `type: "module"` și scripturile exacte.  
- [ ] Dependențe: `sequelize` (v6), `sqlite3`, `express`, `cors`; dev: `vitest`, `jest`, `supertest`, `nodemon`.  
- [ ] Structura: `src/`, `src/db/`, `src/db/models/`, `tests/vitest/`, `tests/jest/`.  
- [ ] `src/app.js` și `src/server.js` pornesc un server minimal și expun `GET /health`.  
- [ ] Testele „smoke” verifică `sequelize.authenticate()` și `PRAGMA foreign\_keys=ON`.

{  
 "name": "s9p2-sequelize-sqlite-starter",  
 "version": "1.0.0",  
 "private": true,  
 "type": "module",  
 "scripts": {  
 "dev": "nodemon src/server.js",  
 "start": "node src/server.js",  
 "test:vitest": "vitest run",  
 "test:jest": "NODE\_OPTIONS=--experimental-vm-modules jest --runInBand",  
 "test": "npm run test:vitest && npm run test:jest"  
 },  
 "dependencies": {  
 "cors": "^2.8.5",  
 "express": "^4.19.2",  
 "sequelize": "^6",  
 "sqlite3": "^5"  
 },  
 "devDependencies": {  
 "jest": "^29.7.0",  
 "nodemon": "^3.1.0",  
 "supertest": "^6.3.3",  
 "vitest": "^1.6.0"  
 }  
}

\*\*Explicație.\*\* Setarea `type: "module"` permite ESM în Node. Pentru Jest, activăm ESM prin `NODE\_OPTIONS=--experimental-vm-modules` în script (asigură încărcarea `.mjs/.js` ESM). Vitest funcționează nativ cu ESM. Dependențele sunt minimale și potrivite pentru medii didactice.

// vitest.config.ts  
import { defineConfig } from 'vitest/config'  
export default defineConfig({ test: { environment: 'node' } })

// jest.config.cjs  
module.exports = {  
 testEnvironment: 'node',  
 transform: {},  
 extensionsToTreatAsEsm: ['.js', '.mjs']  
};

ETAPA 1 — Modele & asocieri (Club, Member, Registration — through, Event)  
-----------------------------------------------------------------------------  
\*\*Obiectiv:\*\* definești modelele ORM și asocierile conform modelului relațional din Partea 1:   
- `Club` (1–N) `Event`   
- `Member` (M–N) `Club` prin `Registration` (tabel de legătură)   
- Atribute utile: `Club.name` (UNIQUE), `Member.email` (valid), `Registration.role` (opțional), `Event.startsAt` (DATE).

\*\*Pași:\*\*  
1) Creezi `src/db/connection.js` (fabrica de Sequelize), activezi `PRAGMA foreign\_keys=ON`.   
2) Definiți modelele în `src/db/models/\*.js` folosind `Model.init(...)` și `DataTypes`.   
3) Declarați asocierile în `src/db/associations.js`.   
4) În `src/db/index.js` exportați o funcție `initDb({ logging })` care construiește conexiunea, cheamă definirea modelelor, setează asocierile și rulează `sequelize.sync()` (în dev) sau lasă migrările (în etapele ulterioare).

// src/db/connection.js  
import { Sequelize } from 'sequelize';  
  
export async function makeSequelize({ logging = false, test = false } = {}) {  
 const storage = test ? ':memory:' : (process.env.SQLITE\_STORAGE || 'db.sqlite');  
 const sequelize = new Sequelize({  
 dialect: 'sqlite',  
 storage,  
 logging  
 });  
 await sequelize.authenticate();  
 await sequelize.query('PRAGMA foreign\_keys = ON;');  
 return sequelize;  
}

// src/db/models/club.js  
import { DataTypes, Model } from 'sequelize';  
export function defineClub(sequelize) {  
 class Club extends Model {}  
 Club.init({  
 id: { type: DataTypes.STRING, primaryKey: true },  
 name: { type: DataTypes.STRING, allowNull: false, unique: true, validate: { len: [3, 255] } },  
 category: { type: DataTypes.STRING, allowNull: false, defaultValue: 'general' },  
 createdAt: { type: DataTypes.DATE, allowNull: false, defaultValue: DataTypes.NOW },  
 updatedAt: { type: DataTypes.DATE, allowNull: false, defaultValue: DataTypes.NOW }  
 }, { sequelize, modelName: 'Club', tableName: 'clubs' });  
 return Club;  
}

// src/db/models/member.js  
import { DataTypes, Model } from 'sequelize';  
export function defineMember(sequelize) {  
 class Member extends Model {}  
 Member.init({  
 id: { type: DataTypes.STRING, primaryKey: true },  
 fullName: { type: DataTypes.STRING, allowNull: false, validate: { len: [3, 255] } },  
 email: { type: DataTypes.STRING, allowNull: false, unique: true, validate: { isEmail: true } },  
 createdAt: { type: DataTypes.DATE, allowNull: false, defaultValue: DataTypes.NOW },  
 updatedAt: { type: DataTypes.DATE, allowNull: false, defaultValue: DataTypes.NOW }  
 }, { sequelize, modelName: 'Member', tableName: 'members' });  
 return Member;  
}

// src/db/models/registration.js  
import { DataTypes, Model } from 'sequelize';  
export function defineRegistration(sequelize) {  
 class Registration extends Model {}  
 Registration.init({  
 id: { type: DataTypes.STRING, primaryKey: true },  
 role: { type: DataTypes.STRING, allowNull: true }  
 }, { sequelize, modelName: 'Registration', tableName: 'registrations' });  
 return Registration;  
}

// src/db/models/event.js  
import { DataTypes, Model } from 'sequelize';  
export function defineEvent(sequelize) {  
 class Event extends Model {}  
 Event.init({  
 id: { type: DataTypes.STRING, primaryKey: true },  
 title: { type: DataTypes.STRING, allowNull: false },  
 startsAt: { type: DataTypes.DATE, allowNull: false },  
 location: { type: DataTypes.STRING, allowNull: true }  
 }, { sequelize, modelName: 'Event', tableName: 'events' });  
 return Event;  
}

// src/db/associations.js  
export function setupAssociations(models){  
 const { Club, Member, Registration, Event } = models;  
  
 Club.hasMany(Event, { foreignKey: 'clubId', onDelete: 'CASCADE', as: 'events' });  
 Event.belongsTo(Club, { foreignKey: 'clubId', as: 'club' });  
  
 Club.belongsToMany(Member, { through: Registration, foreignKey: 'clubId', otherKey: 'memberId', onDelete: 'CASCADE', as: 'members' });  
 Member.belongsToMany(Club, { through: Registration, foreignKey: 'memberId', otherKey: 'clubId', onDelete: 'CASCADE', as: 'clubs' });  
  
 Registration.belongsTo(Club, { foreignKey: 'clubId' });  
 Registration.belongsTo(Member, { foreignKey: 'memberId' });  
 Club.hasMany(Registration, { foreignKey: 'clubId' });  
 Member.hasMany(Registration, { foreignKey: 'memberId' });  
}

// src/db/index.js  
import { makeSequelize } from './connection.js';  
import { defineClub } from './models/club.js';  
import { defineMember } from './models/member.js';  
import { defineRegistration } from './models/registration.js';  
import { defineEvent } from './models/event.js';  
import { setupAssociations } from './associations.js';  
import crypto from 'node:crypto';  
  
export async function initDb({ logging = false, test = false, sync = true } = {}) {  
 const sequelize = await makeSequelize({ logging, test });  
 const Club = defineClub(sequelize);  
 const Member = defineMember(sequelize);  
 const Registration = defineRegistration(sequelize);  
 const Event = defineEvent(sequelize);  
  
 setupAssociations({ Club, Member, Registration, Event });  
  
 if(sync){  
 await sequelize.sync({ force: true });  
 }  
  
 // util: small helpers  
 const uuid = () => crypto.randomUUID();  
  
 return { sequelize, Club, Member, Registration, Event, uuid };  
}

\*\*Explicație.\*\* Întâi definim toate modelele, apoi aplicăm asocierile (pentru a evita referințele nerezolvate). `sequelize.sync({ force: true })` este potrivit pentru dezvoltare/test (în producție vei folosi migrații). `PRAGMA foreign\_keys=ON` se execută după `authenticate()` pentru a asigura enforcement‑ul referențial.

ETAPA 2 — Validări și unicități  
--------------------------------  
\*\*Obiectiv:\*\* să demonstrăm că regulile de business sunt reprezentate atât la nivel de aplicație (validări Sequelize), cât și la nivel de bază (UNIQUE, NOT NULL, FK). În plus, testăm erorile relevante (email invalid, nume de club duplicat etc.).

\*\*Cerință:\*\* adaugă `unique: true` pe `Club.name` și `Member.email`, menține `allowNull: false` pe câmpurile obligatorii, definește validatoare (ex. `isEmail`). Scrie teste care așteaptă erori Sequelize și verifică mesajele/numărul de rânduri rămase în tabele.

// tests/vitest/models.vitest.test.ts  
import { beforeAll, afterAll, test, expect, describe } from 'vitest';  
import { initDb } from '../../src/db/index.js';  
  
let db;  
beforeAll(async () => { db = await initDb({ test: true, sync: true }); });  
afterAll(async () => { await db.sequelize.close(); });  
  
test('PRAGMA foreign\_keys=ON', async () => {  
 const r = await db.sequelize.query("PRAGMA foreign\_keys;");  
 expect(r[0][0].foreign\_keys).toBe(1);  
});  
  
describe('validations & uniqueness', () => {  
 test('unique Club.name', async () => {  
 await db.Club.create({ id:'tech', name:'Tech & Coding', category:'technology' });  
 await expect(db.Club.create({ id:'tech2', name:'Tech & Coding', category:'technology' }))  
 .rejects.toThrow();  
 });  
 test('Member.email is valid & unique', async () => {  
 await db.Member.create({ id:'m1', fullName:'Ana Pop', email:'ana@uni.ro' });  
 await expect(db.Member.create({ id:'m2', fullName:'Ion', email:'not-an-email' })).rejects.toThrow();  
 await expect(db.Member.create({ id:'m3', fullName:'Ana2', email:'ana@uni.ro' })).rejects.toThrow();  
 });  
});

// tests/jest/models.jest.test.cjs  
import { initDb } from '../../src/db/index.js';  
let db;  
beforeAll(async () => { db = await initDb({ test: true, sync: true }); });  
afterAll(async () => { await db.sequelize.close(); });  
  
test('PRAGMA foreign\_keys=ON', async () => {  
 const r = await db.sequelize.query("PRAGMA foreign\_keys;");  
 expect(r[0][0].foreign\_keys).toBe(1);  
});  
  
test('unique constraints & validation', async () => {  
 await db.Club.create({ id:'c1', name:'Unique Club', category:'art' });  
 await expect(db.Club.create({ id:'c2', name:'Unique Club', category:'art' })).rejects.toThrow();  
 await db.Member.create({ id:'mem1', fullName:'Test', email:'t@t.ro' });  
 await expect(db.Member.create({ id:'mem2', fullName:'X', email:'bad' })).rejects.toThrow();  
});

\*\*Explicație.\*\* Testele pornesc pe `sqlite::memory:` (rapid, izolat) și confirmă că \*\*UNIQUE\*\* și validările blochează introducerea de înregistrări neconforme. Verificarea `PRAGMA foreign\_keys` confirmă enforcement‑ul activ la nivel de conexiune.

ETAPA 3 — Interogări & eager loading (anti‑N+1)  
-------------------------------------------------  
\*\*Obiectiv:\*\* demonstrezi diferența dintre încărcarea „lazy” (secvențială) și „eager” (`include`) și măsori simplu numărul de interogări prin \*\*logging funcțional\*\*.

\*\*Cerință:\*\* populează cu câteva cluburi + membri + înregistrări. Rulează două scenarii:   
1) fără `include` (încărcare secvențială) — așteptați mai multe interogări;   
2) cu `include` (M:N prin `Registration`) — o interogare compusă. Verifică numărul de interogări și forma payload‑ului (doar atributele utile).

// tests/vitest/eager.vitest.test.ts  
import { beforeAll, afterAll, test, expect } from 'vitest';  
import { initDb } from '../../src/db/index.js';  
  
let db, logs;  
beforeAll(async () => {  
 logs = [];  
 db = await initDb({ test: true, sync: true, logging: (sql)=> logs.push(sql) });  
 const { Club, Member, Registration, uuid } = db;  
 const c1 = await Club.create({ id:'tech', name:'Tech', category:'tech' });  
 const c2 = await Club.create({ id:'arts', name:'Arts', category:'arts' });  
 const m1 = await Member.create({ id:'m1', fullName:'Ana', email:'ana@uni.ro' });  
 const m2 = await Member.create({ id:'m2', fullName:'Dan', email:'dan@uni.ro' });  
 await Registration.create({ id: uuid(), clubId:c1.id, memberId:m1.id, role:'volunteer' });  
 await Registration.create({ id: uuid(), clubId:c1.id, memberId:m2.id, role:'member' });  
 await Registration.create({ id: uuid(), clubId:c2.id, memberId:m2.id, role:'member' });  
});  
afterAll(async () => { await db.sequelize.close(); });  
  
test('lazy vs eager', async () => {  
 logs.length = 0;  
 const clubs = await db.Club.findAll(); // lazy  
 const resLazy = [];  
 for(const c of clubs){  
 const members = await c.getMembers(); // N interogări  
 resLazy.push({ id:c.id, members: members.map(m=>m.id) });  
 }  
 const lazyQueries = logs.length;  
  
 logs.length = 0;  
 const eager = await db.Club.findAll({ include: [{ association:'members', attributes:['id'] }], attributes:['id'] });  
 const eagerQueries = logs.length;  
  
 expect(eagerQueries).toBeLessThan(lazyQueries);  
 expect(eager[0].members[0].id).toBeDefined();  
});

ETAPA 4 — Tranzacții (commit/rollback) și invarianta contorului  
------------------------------------------------------------------  
\*\*Obiectiv:\*\* înscriere atomica a unui membru în club, cu incrementarea unui contor în `Club` (de ex. `memberCount`). Demonstrează atât commit (stare finală persistentă), cât și rollback (stare nemodificată).

\*\*Cerință:\*\* adaugă un câmp `memberCount` în `Club` (default 0). Scrie o funcție `enrollMember(clubId, memberInput)` care rulează într‑o tranzacție: creează `Member` dacă nu există, creează `Registration`, actualizează `memberCount`. Scrie teste pentru commit și pentru un caz care declanșează rollback (ex. e‑mail invalid).

// (fragment) src/db/models/club.js — adaugă memberCount  
memberCount: { type: DataTypes.INTEGER, allowNull: false, defaultValue: 0 },

// src/db/services/enroll.js  
export async function enrollMember(db, { clubId, member }){  
 return await db.sequelize.transaction(async (t)=>{  
 const [m] = await db.Member.findOrCreate({  
 where: { email: member.email },  
 defaults: { id: db.uuid(), fullName: member.fullName, email: member.email },  
 transaction: t  
 });  
 await db.Registration.create({ id: db.uuid(), clubId, memberId: m.id, role: member.role||'member' }, { transaction: t });  
 const c = await db.Club.findByPk(clubId, { transaction: t });  
 c.memberCount = (c.memberCount||0) + 1;  
 await c.save({ transaction: t });  
 return { clubId: c.id, memberId: m.id, memberCount: c.memberCount };  
 });  
}

// tests/jest/trx.jest.test.cjs  
import { initDb } from '../../src/db/index.js';  
import { enrollMember } from '../../src/db/services/enroll.js';  
  
let db;  
beforeAll(async()=>{ db = await initDb({ test:true, sync:true }); await db.Club.create({ id:'tech', name:'Tech', category:'tech' }); });  
afterAll(async()=>{ await db.sequelize.close(); });  
  
test('commit enrollment', async()=>{  
 const out = await enrollMember(db, { clubId:'tech', member:{ fullName:'Ana', email:'ana@uni.ro' } });  
 expect(out.memberCount).toBe(1);  
 const c = await db.Club.findByPk('tech');  
 expect(c.memberCount).toBe(1);  
});  
  
test('rollback on invalid email', async()=>{  
 await expect(enrollMember(db, { clubId:'tech', member:{ fullName:'X', email:'bad' } })).rejects.toThrow();  
 const c = await db.Club.findByPk('tech');  
 expect(c.memberCount).toBe(1); // neschimbat  
});

\*\*Explicație.\*\* Tranzacția asigură \*\*atomicitate\*\*: fie toate operațiile reușesc, fie niciuna. Cazul de eșec (e‑mail invalid) declanșează excepție în validatorul `Member`, ceea ce face rollback automat. Testul verifică invarianta `memberCount`.

ETAPA 5 — Migrații & seeders (opțional în acest laborator)  
--------------------------------------------------------------  
\*\*Obiectiv:\*\* separi schema de cod și obții reproducibilitate a schimbărilor. Pentru curs, folosim `sequelize.sync({ force: true })` în dev/test; în proiectele mai mari introducem `sequelize-cli` cu directorul `migrations/` și `seeders/`.

\*\*Cerință:\*\* ofer o schemă minimală de migrație pentru adăugarea coloanei `memberCount` și un seeder minimal pentru 2–3 cluburi. (Acestea sunt incluse în arhivă ca referință, dar laboratorul curent poate rula fără `sequelize-cli`.)

// exemplu schematic de migrație (migrations/20250914-add-memberCount.js)  
'use strict';  
module.exports = {  
 async up(queryInterface, Sequelize) {  
 await queryInterface.addColumn('clubs', 'memberCount', { type: Sequelize.INTEGER, allowNull:false, defaultValue:0 });  
 await queryInterface.addIndex('clubs', ['name'], { unique: true, name: 'clubs\_name\_uq' });  
 },  
 async down(queryInterface, Sequelize) {  
 await queryInterface.removeColumn('clubs', 'memberCount');  
 await queryInterface.removeIndex('clubs', 'clubs\_name\_uq');  
 }  
};

ETAPA 6 — PRAGMA & indexuri (WAL, foreign\_keys, indexuri utile)  
------------------------------------------------------------------  
\*\*Obiectiv:\*\* înțelegi impactul configurațiilor SQLite și stabilești indexurile minime pentru fluxul OLTP didactic.

\*\*Cerință:\*\* activează `foreign\_keys` (deja în codul de inițializare) și documentează cum se comută în WAL. Creează un index `UNIQUE` pe `Club.name` și un index compus pe `registrations(memberId, clubId)` (utile la M:N).

// exemplu de index compus (migrație separată)  
await queryInterface.addIndex('registrations', ['memberId','clubId'], { unique: true, name: 'registrations\_member\_club\_uq' });

ETAPA 7 — Testare automată (Vitest & Jest side‑by‑side)  
---------------------------------------------------------  
\*\*Obiectiv:\*\* rulezi aceleași suite logice pe ambele framework‑uri, pentru a observa convergențele de toolchain; îți configurezi rularea rapidă în „sqlite::memory:” pentru feedback imediat.

\*\*Cerință:\*\* `npm test` să ruleze ambele suite; fixurile trebuie să treacă în ambele. Introdu testele negative pentru FK și UNIQUE; adaugă un mic test de metrice de interogări pentru a confirma beneficiul `include`.

// tests/vitest/relations.vitest.test.ts  
import { beforeAll, afterAll, test, expect } from 'vitest';  
import { initDb } from '../../src/db/index.js';  
let db;  
beforeAll(async()=>{ db = await initDb({ test:true, sync:true }); });  
afterAll(async()=>{ await db.sequelize.close(); });  
  
test('FK enforcement: deleting club cascades registrations', async()=>{  
 const { Club, Member, Registration, uuid } = db;  
 const c = await Club.create({ id:'del', name:'Del Club', category:'test' });  
 const m = await Member.create({ id:'mA', fullName:'A', email:'a@x.ro' });  
 await Registration.create({ id: uuid(), clubId:c.id, memberId:m.id });  
 await c.destroy(); // CASCADE  
 const left = await Registration.findAll({ where: { clubId: c.id } });  
 expect(left.length).toBe(0);  
});

Server Express minimal (pentru integrare manuală)  
----------------------------------------------------  
\*\*Obiectiv:\*\* în plus față de testele automate, ai un server minimal pentru verificări cu `curl`/Postman. Ruta `/health` testează conexiunea și `PRAGMA foreign\_keys`.

// src/app.js  
import express from 'express';  
import cors from 'cors';  
import { initDb } from './db/index.js';  
  
export async function makeApp(){  
 const db = await initDb({ test:false, sync:true });  
 const app = express();  
 app.use(cors());  
 app.use(express.json());  
  
 app.get('/health', async (req,res)=>{  
 const fk = await db.sequelize.query('PRAGMA foreign\_keys;');  
 res.json({ ok:true, foreign\_keys: fk[0][0].foreign\_keys, now: new Date().toISOString() });  
 });  
  
 return { app, db };  
}

// src/server.js  
import { makeApp } from './app.js';  
const PORT = process.env.PORT || 5609;  
makeApp().then(({ app }) => {  
 app.listen(PORT, ()=> console.log(`Sem9 API on http://localhost:${PORT}`));  
});

\*\*Explicație.\*\* Serverul este opțional în laborator, dar util pentru testare manuală. Pentru producție, vei separa responsabilitățile (migrations în CI/CD, seedere, logs).

AI‑assist (VSL): Prompturi scurte și verificabile  
--------------------------------------------------  
- „Generează scheletul de model Sequelize pentru `Registration` cu chei FK `clubId` și `memberId`, plus `role` STRING opțional, tabel `registrations`.”   
- „Scrie `setupAssociations` pentru relațiile 1–N și M–N dintre `Club`, `Member`, `Registration`, `Event`, cu `onDelete: CASCADE` unde e cazul.”   
- „Adaugă funcția `enrollMember` cu tranzacție și teste pentru commit și rollback.”   
- „Transformă testul lazy/eager astfel încât să număr interogările folosite de Sequelize (via `logging`).”   
- „Propune o migrație `up/down` pentru `memberCount` în `Club` și un index compus pe `registrations(memberId,clubId)`.”

Cum rulezi laboratorul (în arhiva ZIP)  
-----------------------------------------  
1) `npm i` (instalează dependențele).   
2) `npm test` (rulează Vitest + Jest pe `sqlite::memory:`; toate testele ar trebui să treacă).   
3) `npm run dev` (pornește serverul Express pe http://localhost:5609, verifică `/health`).

\*\*Troubleshooting:\*\*   
• Dacă vezi erori de FK, verifică `PRAGMA foreign\_keys=ON;` în `src/db/connection.js`.   
• Dacă Jest se plânge de ESM, rulează scriptul `test:jest` din `package.json` (are `NODE\_OPTIONS=--experimental-vm-modules`).   
• În caz de blocaje la `sqlite3` (compilare pe anumite platforme), poți folosi un container sau o versiune precompilată.   
• Verifică versiunile Node (LTS) și reinstalează `sqlite3` dacă e necesar (`npm rebuild`).