**Seminarul 0 — Partea 1 (Teorie, extins)**

**Hook realist: De ce un „toolchain” bun îți salvează timpul și proiectul**

Imaginează‑ți un atelier mecanic. Ai pe masă scule fără etichete, lipsesc cheile potrivite, nu știi unde este manualul  
și fiecare rotiță se potrivește doar dacă o lovești insistent. În programare, un „atelier” prost organizat înseamnă  
timp pierdut în depanare, bug‑uri greu de reprodus, diferențe între mediul tău și al colegilor, versiuni de biblioteci  
incompatibile și, în final, o arhitectură fragilă. Seminarul introductiv este exact acel moment în care deschidem cutia cu scule,  
ordonăm uneltele, stabilim regulile de lucru și lipim etichete clare.

„Toolchain” (lanțul de unelte) înseamnă „tot ceea ce îți permite să scrii, rulezi, testezi și livrezi cod în mod repetabil”.  
În web, vorbim despre \*\*Node.js\*\* (runtime), \*\*npm/PNPM/Yarn\*\* (package managers), \*\*Git/GitHub\*\* (version control & remote),  
un editor productiv (de regulă \*\*VS Code\*\*), utilitare de linie de comandă (Windows Terminal/PowerShell), scripturi în `package.json`,  
plus o structură de directoare care face sens. De aici pornește totul. Orice seminar ulterior (HTML/CSS/JS, REST, React, SQLite, etc.)  
se sprijină pe acest minim operațional.

**Harta ecosistemului: unde se află fiecare piesă**

Într‑un proiect web minimal avem două lumi: \*\*browserul\*\* și \*\*serverul\*\*. Browserul execută JavaScript în V8/WebKit/Gecko,  
gestionează DOM, CSSOM și evenimente. Serverul rulează în \*\*Node.js\*\* (V8 + API‑uri pentru fișiere, rețea), ascultă pe un port,  
servește răspunsuri HTTP și, adesea, livrează fișiere statice (HTML/CSS/JS). Deși ambele folosesc JavaScript, capabilitățile diferă:  
în Node.js ai `fs`, `http`, `path`; în browser ai `document`, `window`, `fetch`. Această separație explică de ce scriem cod  
„isomorf” cu grijă și de ce testăm contractele (ce iese) mai degrabă decât detaliile (cum arată pixelii).

\*\*Node.js\*\* este motorul care execută aplicația noastră server. \*\*npm\*\* este administratorul de pachete implicit,  
dar în proiecte serioase vei vedea frecvent \*\*PNPM\*\* (spațiu pe disc eficient, `node\_modules` determinist) sau \*\*Yarn\*\*.  
\*\*Git\*\* ține istoria și ne ajută să colaborăm fără fricțiune: creăm \*branch‑uri\*, facem \*pull requests\*, revenim la versiuni stabile.  
\*\*GitHub\*\* este gazda distantă care sincronizează munca. \*\*VS Code\*\* pune toate acestea într‑un singur loc, cu extensii  
pentru linting (ESLint), formatare (Prettier), integrare Git și debug.

**Windows 10/11: instalări & verificări esențiale**

Pe Windows, calea de rezistență mică este instalatorul oficial Node.js \*\*LTS\*\*. Alternativ, pentru proiecte multiple  
cu versiuni diferite, `nvm-windows` este o opțiune matură (deși cu nuanțe la permisiuni). După instalare, verifică versiunile.  
În mod similar, instalează \*\*Git for Windows\*\*; configurează numele și emailul pentru commit‑uri; activează un \*credential helper\*  
ca să nu reintroduci parola mereu. Un editor actual (VS Code) și Windows Terminal îți simplifică viața de zi cu zi.

Verificări de bază (nu uita să rulezi comenzile într‑un terminal cu drepturi standard, nu administrative):

node --version

npm --version

git --version

Dacă Node raportează o versiune „Current” foarte nouă, preferă \*\*LTS\*\* pentru stabilitate în mediul educațional.  
Dacă intenționezi să folosești pachete care compilează addon‑uri native, instalează „Visual Studio Build Tools” (C++ Build Tools)  
și Python pentru `node-gyp`. Aceste detalii nu sunt obligatorii în seminarul introductiv, dar e bine să știi de ele.

**Controlul versiunilor: Git & GitHub (disciplina minimă)**

Fără Git, nu există „istoric” verosimil și nici colaborare civilizată. Practicile minime includ:  
un \*repository\* local (`git init`), un \*remote\* pe GitHub, commit‑uri mici, mesaje semnificative, `.gitignore` corect  
și \*branch‑uri\* pentru dezvoltare. Recomand un stil apropiat de \*\*Conventional Commits\*\* („feat: …”, „fix: …”, „docs: …”)  
chiar dacă nu integrăm încă \*semantic versioning\*. Îți oferă o „gramatică” a schimbărilor care va scădea entropia proiectului.

git config --global user.name "Nume Prenume"

git config --global user.email "you@example.com"

git config --global init.defaultBranch main

Inițializare repo și primul commit:

mkdir tw-seminar-0 && cd tw-seminar-0

git init

echo "# TW Seminar 0" > README.md

git add -A

git commit -m "chore: initial commit (scaffold repo)"

git branch -M main

Creezi un repo gol pe GitHub („tw-seminar-0”) și îl legi ca \*origin\*, apoi publici „main”.  
Din acest moment, fiecare laborator va începe cu un branch nou, iar integrarea are loc prin pull request.

git remote add origin https://github.com/<user>/tw-seminar-0.git

git push -u origin main

**Inițializarea proiectului Node: package.json, scripturi, ESM vs. CJS**

`package.json` este declarația de intenție: nume, versiune, dependențe, scripturi.  
În liniile generale, evităm să punem „magie” în scripturi; dorim comenzi simple și clare, ușor de rulat local și în CI.  
În 2025, \*\*ESM\*\* (ECMAScript Modules) este alegerea recomandată pentru consistență cu browserul și ecosistemul modern.  
Dacă ai dependențe mai vechi „CommonJS only”, explicăm strategii de interoperabilitate.

npm init -y

Adaugă `type` în `package.json` pentru a activa ESM în fișiere `.js` (altfel trebuie `.mjs`):

{

"name": "tw-seminar-0",

"version": "1.0.0",

"type": "module",

"scripts": {

"dev": "nodemon src/index.js",

"start": "node src/index.js",

"test:vitest": "vitest run --reporter verbose",

"test:jest": "jest --runInBand"

}

}

Instalăm framework‑ul HTTP \*\*Express\*\* pentru serverul minimal, și \*\*nodemon\*\* pentru „watch mode”.  
În partea 2 vom seta și testele (Vitest/Jest) în oglindă; aici doar pregătim terenul.

npm i express

npm i -D nodemon

**Structură de directoare și „sursa unică a adevărului”**

În proiecte solide, datele, configurația și codul sunt separate. Pentru seminarul 0, structura recomandată este minimă,  
dar clară: `src/` pentru codul serverului, `public/` pentru fișiere statice. Pune în `public/` un `index.html` simplu  
care îți confirmă că ruta statică funcționează. În `src/`, centralizează crearea aplicației Express într‑o funcție `createApp()`  
(pe care o vom testa în partea 2), iar pornirea pe port într‑un fișier small `index.js`.

tw-seminar-0/

public/

index.html

src/

app.js

index.js

package.json

README.md

**Server minimal: concepte (fără a „fura” din laborator)**

Un server minimal cu Express înseamnă trei idei: (1) declarăm aplicația; (2) adăugăm \*middleware\* pentru JSON & static;  
(3) definim rute simple și pornim ascultarea pe un port. În plus, vrem rute de „observabilitate” (health check, ping) pentru a verifica rapid  
dacă serverul trăiește. În partea 2, fiecare pas va fi implementat și testat; aici fixăm vocabularul și intențiile.

// src/app.js (schiță conceptuală)

import express from 'express'

export function createApp() {

const app = express()

app.use(express.json())

app.use(express.static('public'))

app.get('/ping', (req, res) => res.send('pong'))

app.get('/api/time', (req, res) => res.json({ now: new Date().toISOString() }))

return app

}

// src/index.js (schiță conceptuală)

import { createApp } from './app.js'

const PORT = process.env.PORT || 3000

createApp().listen(PORT, () => console.log(`Server on http://localhost:${PORT}`))

De ce această separație `createApp()`/`listen()`? Pentru testare: putem instanția aplicația fără a porni un port,  
trimite \*request‑uri\* cu utilitare (e.g., `supertest`) și aserta răspunsurile. Aceasta reduce fricțiunea la scrierea testelor și la CI.

**ESM vs. CJS: interoperabilitate la zi (2025)**

ESM este standardul modulelor în JavaScript. În Node 18+ și 20+, ESM este matur, dar pot apărea colțuri ascuțite:  
pachete vechi fără export ESM, fișiere `.cjs` care coabitează cu `.js` ESM, eroarea `ERR\_MODULE\_NOT\_FOUND` când folosești căi relative  
fără extensie. Practică: specifică extensiile (`import x from './x.js'`), declară `type: "module"` în `package.json`, iar pentru pachete  
CJS oferă \*bridging\* la nivel de import (de pildă, `const pkg = await import('old-commonjs')` sau folosești versiunea compat).

**Mediu și configurare: .env, PORT, cross‑platform**

Variabilele de mediu decuplează codul de contexte (dev/test/prod). Pentru seminar: `PORT` și, eventual,  
`NODE\_ENV`. Nu comite niciodată `.env` în Git; folosește `.env.example` pentru a documenta ce variabile există.  
Pe Windows, dacă ai nevoie să setezi variabile într‑un script, folosește `cross-env` pentru compatibilitate cu POSIX.

.gitignore:

node\_modules

.env

.env.example:

PORT=3000

**Calitate: ESLint (flat config) + Prettier (de ce acum, nu mai târziu)**

Cei mai mulți studenți amână linting/formatting „până la React”. Greșit. Începând din seminarul 0, formatarea automată  
și regulile statice te scapă de zgomot cognitiv și îți oferă \*feedback\* instantaneu. \*\*Flat config\*\* la ESLint (fișier `eslint.config.js`)  
simplifică moștenirile, iar Prettier rezolvă stilul. În partea 2 vom rula efectiv `npm run lint` și `npm run format`; aici motivăm.

**Debugging & observabilitate de bază (loguri, health, erori)**

Un server minimal trebuie să‑ți spună „ce face” și „dacă e în picioare”. Simplul `/ping` este deja un început.  
Poți adăuga `/healthz` și `/readyz` (după modelul Kubernetes) chiar dacă în acest seminar nu orchestri containere.  
Minimum: log la start cu URL, log la eroare neprevăzută și status codes corecte (404/500).

**AI‑assist: cum formulăm cereri productive (Copilot sub VSL / ChatGPT)**

„AI‑assist” nu înlocuiește înțelegerea, dar poate accelera pașii repetitivi. Practică „VSL” (Very Short Language):  
prompts scurte, precise, cu rezultat măsurabil. Verifică totul local, rulează testele, nu „lipi” cod fără să‑l înțelegi.  
Mai jos sunt câteva modele pe care le poți adapta pe locul tău de muncă.

Exemple de prompts utile (folosește‑le în editor sau în chat):

# GitHub Copilot (VSL)

"Create an Express app factory `createApp()` with JSON middleware, static('public'), /ping, /api/time."

"Convert this CommonJS entrypoint to ESM (Node 20): replace require/module.exports with import/export; add package.json type."

"Write Vitest tests for /ping and /api/time with supertest; then mirror the suite for Jest."

"Generate a .gitignore for a Node/Express project (Windows + macOS)."

"Suggest a flat ESLint config for ESM Node (no-unused-vars, import/order), plus Prettier integration."

**Capcane frecvente pe Windows (și remedii rapide)**

• \*\*Path/Permisiuni\*\*: Instalează Node pentru toți utilizatorii sau verifică PATH; rulează terminalul ca utilizator obișnuit.  
• \*\*Port ocupat\*\*: `EADDRINUSE: 3000` — oprește instanțe vechi sau schimbă portul (`PORT=3001`).  
• \*\*ESM import extins\*\*: `ERR\_MODULE\_NOT\_FOUND` — adaugă extensia `.js` în importuri.  
• \*\*`node-gyp`\*\*: pachete cu addon‑uri native cer Build Tools + Python; evită‑le în seminarul 0.  
• \*\*CRLF vs. LF\*\*: setează „End of Line” consistent în editor; Prettier poate normaliza automat.

**Ce înseamnă „ready” după Partea 1 (criterii de acceptare)**

La finalul acestei părți ar trebui să poți: (1) instala și verifica Node/npm/Git; (2) inițializa un repo și un proiect Node;  
(3) alege ESM și să‑l configurezi corect; (4) explica rolul `public/` vs. `src/`; (5) scrie și lansa un server minimal (fără teste încă);  
(6) înțelege cum și de ce vom adăuga testele în Partea 2; (7) formula prompts eficiente pentru a obține asistență de la AI.

**Punte către Partea 2 (Laborator)**

În Partea 2 vei construi efectiv proiectul „StudentHub Minisite”: inițializezi repo/proiect, creezi serverul Express,  
servești `public/index.html`, adaugi rutele `/ping` și `/api/time`, configurezi `nodemon`, definești scripturi, apoi scrii și rulezi  
teste \*\*Vitest\*\* și \*\*Jest\*\* în paralel. Vei avea un \*\*worksheet\*\* clar și un \*\*checklist\*\* pentru fiecare pas.