**Seminarul 12 — Componente externe: tabele și grafice  
Partea 1 — Teorie extinsă (narativ + exemple)**

HOOK (poveste realistă): „ClubHub+ Analytics”. După ce ați construit coloana vertebrală a aplicației (S10 React/JSX și S11 Redux Toolkit), asociația studențească vă cere acum un dashboard real de analiză: tabele cu mii de înregistrări (membri, cluburi, evenimente), grafice care sintetizează evoluții (înrolări noi pe săptămână, participare pe club, distribuția rolurilor), filtre legate între ele și export pentru raportare. În demo‑urile inițiale totul merge, dar în producție apar provocări: filtrele aplicate în tabel nu se reflectă corect în grafic, scroll‑ul se blochează pe dataset‑uri mari, formatele de dată și număr nu respectă locale‑ul utilizatorului, temele „light/dark” modifică lizibilitatea și accesibilitatea culorilor, iar echipa QA cere teste fiabile pentru piese care, aparent, „nu pot fi testate” (grafice). Acest seminar fixează o hartă riguroasă: cum alegem biblioteca de tabele și cea de grafice, cum proiectăm datele și starea (state) pentru a evita dublări și inconsistențe, cum asigurăm performanță și a11y, cum testăm (unit și de integrare) și cum folosim AI‑assist (VSL) pentru a accelera fără a compromite rigurozitatea.

1) De ce „componente externe” și de ce acum  
Comunitatea React oferă „componente externe” pentru două zone notoriu costisitoare: tabele și grafice. Ambele sunt, de fapt, „mini‑motoare” de interacțiune cu date: tabelele rezolvă căutare, sortare, filtrare, paginare, selecții, editare; graficele rezolvă agregare, cartografiere vizuală (encoding), animații și interacțiuni (tooltip, legend, zoom/brush). A le „scrie de la zero” este rar justificat în proiecte aplicate. În schimb, deciziile bune sunt: (a) separarea clară a straturilor (server cache vs app state vs view state), (b) modelarea datelor și a derivatelor prin selectori memoizați, (c) contracte explicite între store și componente (tabel/grafic) și (d) politici de testare care verifică ceea ce contează (contractul de config și efectele observabile în DOM), nu detalii de implementare internă.

Analogie cotidiană: imaginați‑vă biblioteca (table) ca pe un „catalog” de bibliotecă universitară, în care căutați, sortați și filtrați. Graficul este „harta” campusului, care vă arată dintr‑o privire unde se concentrează anumite resurse. Catalogul și harta trebuie să fie sincronizate: când filtrați autorul „Popescu” în catalog, pe hartă se luminează clădirile în care se regăsesc cărțile sale; când faceți „brush” pe o zonă a hărții, catalogul se restrânge la rezultatele din acea zonă. Sincronizarea nu înseamnă să împărtășim „instanța” hărții cu catalogul, ci să împărtășim „starea” (filtre, selecții) într‑un store predictibil.

2) Ce taskuri rezolvăm cu tabele vs. grafice (și când le combinăm)  
Tabelele sunt excelente pentru taskuri de „lookup” și „exact value reading”: utilizatorul vrea să vadă rânduri concrete, să caute după text, să compare valori numerice exacte sau să editeze pe loc. Graficele sunt excelente pentru „pattern finding”, trenduri și raporturi; sacrifică precizia exactă (nu veți citi 237 dintr‑un bar chart cu ușurință), dar oferă „pre‑atentive cues” care scot la suprafață structuri. Literatura clasică asupra percepției grafice (Cleveland & McGill; Mackinlay; Munzner) arată că unele codificări vizuale sunt mai precise decât altele: poziția pe o axă comună > lungime > unghi > arie > volum > culoare. Consecința practică: nu folosiți 3D „decorativ” pentru cuantitative; preferați poziția sau lungimea, evitați pie charts când diferențele sunt mici, folosiți scale adecvate (linear/log).

Combinarea e firească: tabelul arată detaliile, graficul arată contextul. Cheia este „linked views”: store‑ul păstrează filtrele și selecțiile, iar atât tabelul, cât și graficul derivă din aceeași sursă de adevăr. Evitați duplicarea datelor din „server cache” în app state; dacă folosiți RTK Query pentru datele de server, lăsați cache‑ul acolo și derivați doar stări „de interacțiune” în store (filtre, selecții, sortări), ca să nu mențineți două seturi divergente ale acelorași valori.

3) Modelarea datelor și a stării (state): pipeline curat  
În aplicațiile didactice și de producție, diferențiați trei straturi:  
• Server cache (RTK Query): răspunde de sincronizare cu backend, cache, invalidări, re‑fetch la focus/reconnect; nu dublați aceleași date în slice‑uri.   
• App state (RTK slices): filtre, selecții, vizibilitate de coloane, preferințe de paginare, parametrizări ale graficelor; plus entități normalizate dacă datele nu sunt gestionate prin Query.   
• View state (local component): stări efemere (textbox în editare, meniu deschis).

Modelarea datelor: învățați „tidy data” (Wickham): fiecare variabilă într‑o coloană, fiecare observație într‑un rând, fiecare tip de unitate observațională într‑un tabel. Pentru tabele este esențial: un set curat de accessors (coloane), transformări declarative pentru derivări (selectors) și tipare stricte (TypeScript). Pentru grafice: pregătiți selectors care returnează un contract explicit `{ labels: string[]; datasets: { label: string; data: number[] }[] }` (Chart.js) sau `option` (ECharts) pornind din aceeași sursă de adevăr. Astfel testele devin naturale: nu testați „cum arată pixelii”, ci „ce config a fost cerut”.

4) Semantica tabelului și accesibilitate (a11y): structura corectă înainte de estetică  
Un tabel corect este un document HTML semnificant: `<table>` cu `<caption>` (descriere concisă a conținutului), `<thead>` cu `<tr>` și `<th scope="col">`, `<tbody>` cu `<tr>` și `<td>`. Pentru sortare, expuneți `aria-sort="ascending|descending"` pe headerul activ, mențineți focus vizibil, permiteți navigarea cu tastatura (săgeți pentru celule, `Home/End` pentru capetele rândului/coloanei). Când treceți în mod „grid” (rol `grid`) pentru editing, respectați convențiile ARIA (anunțarea celulei curente, aria‑selected pentru selecții). Evitați să ascundeți informația doar în culori; folosiți iconografie + text/atribute ARIA. Pentru capete de tabel multi‑nivel (grouped columns), marcați ierarhia prin `<th scope="colgroup">` sau alegeți componente care fac mapping corect la DOM.

Internaționalizare (i18n) și formatare: valorile numerice, monedele, procentele, datele trebuie formatate cu `Intl.NumberFormat` / `Intl.DateTimeFormat` și formatare specifică locale‑ului (ro‑RO sau en‑GB). Din considerente de audit și reproducibilitate, păstrați în store valorile numerice brute și aplicați formatarea în „view layer”, nu în datele de bază. Dacă exportați CSV, decideți clar: exportați „raw” sau „vizibil” (post‑filtrare și post‑format).

5) UX pentru tabele: sorting, filtering, pagination, visibility, selection, editing  
• Sorting. Implementați sort stabil (stable sort) și indicatori clari în header. Evitați auto‑sortarea la fiecare tastă în filtre; e mai eficient `debounce` (200–300 ms).   
• Filtering. Pentru „contains” pe text: transformați la lowercase și eliminați diacriticele doar pentru comparare (nu modificați datele stocate). Pentru comparații numerice, oferiți predicate „=, ≥, ≤, între” și marcatori vizuali că filtrul este activ.   
• Pagination vs infinite scroll. Pentru dataset‑uri mari pe server, paginați „la sursă” (query params), altfel tabelul devine „o insulă” cu stare divergentă. Pentru „client‑side”, limitați page size (ex. 25/50/100) și memorați preferința în app state.   
• Column visibility & pinning. Control vizibil explicit pentru utilizator (checkbox list), dar memorați în store ce coloane sunt ascunse.   
• Row selection. Decideți dacă se permite „select all” (inclusiv rânduri neîncărcate). Evitați confuziile de tip „select all apply only to current page”. Documentați comportamentul în UI.   
• Editing. Pentru editare inline, comutați semantic în „grid role” și expuneți tastele de commit/cancel (`Enter`, `Escape`). Logica de validare aparține „thin controller”‑ului din store (nu componentelor individuale), iar commit‑ul face \*thunk\* spre server cu optimistic update sau RTK Query mutation.

6) Grafice: alegerea tipului, codificări vizuale, palete, incertitudine  
Tipul de grafic nu este decor: este ipoteza dvs. despre sarcina cognitivă. Pentru serii temporale, `line chart` cu axă temporală; pentru categorice, `bar chart` ordonat; pentru proporții la o singură variabilă, `bar chart` este de regulă mai lizibil decât `pie`. Pentru distribuții: `histogram`/`boxplot`. Pentru corelații: `scatterplot`. Literatura de percepție grafică arată ierarhia canalelor „read‑off”; folosiți‑o ca regulă implicită. Alegeți palete lizibile (sequential/diverging) și simulați „colour‑blindness” în design. Atunci când există incertitudine (intervale, valori estimate), reprezentați‑o explicit (error bars, \*shaded confidence bands\*), altfel utilizatorii supra‑încredere graficul.

Animațiile: păstrați‑le funcționale (marking of change), cu durate scurte. Evitați tranzițiile costisitoare în buclă. În Chart.js/ECharts, opriți animațiile în testele automate pentru determinism. Pentru multe serii, afişați legende interactive (toggle pe serie) și oferți mod „focus+context” (ex.: overview mic + detaliu).

7) Integrarea bibliotecilor: Chart.js, ECharts, D3/Vega‑Lite (gramatică declarativă)  
• Chart.js. API orientat pe „config object” (`data` + `options`), cu instanță pe `<canvas>`. Ușor de pornit; pentru React, folosiți wrapper sau o componentă mică care instanțiază și distruge graficul în `useEffect`.   
• ECharts. API bogat prin `option` object, puternic la interacțiuni (zoom/pan/dataZoom/brush). Pentru React, există wrappers; altfel scrieți un adaptor minimal în `useEffect`.   
• D3/Vega‑Lite. D3 este nivelul „primordial” (scales, axes, selections). Vega‑Lite oferă o „grammar of graphics” declarativă, cu avantaj la reproducibilitate (spec JSON) și validare conceptuală.

Criterii de alegere: scala proiectului, ușurința testării (mock constructor + aserții pe config), tipurile TypeScript, licența, panta de învățare pentru echipă. În cadrul seminarului vom folosi un drum principal Chart.js (sau ECharts) + TanStack Table, iar în proiectele L3 vom prezenta și Vega‑Lite/D3 pentru scenarii declarative sau custom.

8) Sincronizare tabel ↔ grafic  
Sincronizarea nu înseamnă „să treci instanța graficului către tabel”. Înseamnă să partajezi \*\*filtrele/selețiile\*\* în store: `filters: { query, category, range, selectedIds }`. Tabelul afișează rândurile vizibile (selector memoizat), iar selectorul pentru grafic derivează `labels` și `datasets` din aceleași date vizibile. Evenimentele din tabel (click pe rând, selectare multiplă) dispecerizează acțiuni (`selectRows`, `toggleSelection`); evenimentele din grafic (legend toggle, brush) la fel (`toggleSeries`, `setRange`). Astfel: (1) testele devin simple (asertați acțiuni + stări), (2) totul e previzibil și debuggable (Redux DevTools), (3) se evită scurgeri de resurse (instanța graficului este locală și efemeră).

9) Performanță și scalabilitate  
• Virtualizare (windowing) pentru tabele (10k+ rânduri).   
• Selectori memoizați (Reselect) pentru derivări; evitați reconstrucții obiectuale în selecție (rupe memoizarea).   
• `autoBatchEnhancer` (RTK) reduce numărul de notificări către subscrieri pentru rafale de acțiuni; păstrați update‑urile în loturi, nu în bucle `dispatch` per rând.   
• Debounce/Throttle pentru filtre; folosiți `requestAnimationFrame` pentru UI smooth.   
• Offloading: calcule grele (agregări) în Web Worker; graficul primește deja seria derivată.   
• Performance budget: definește praguri (timp randare tabel < 100 ms pentru 1000 rânduri; timp update grafic < 60 ms pentru 3 serii × 500 puncte) și testați în CI.

10) Testabilitate: ce testăm și cum evităm fragilitatea  
Tabele: testați DOM observabil (aria‑sort, ordinea rândurilor după sort, numărul rândurilor după filter/pagination), nu implementarea internă a bibliotecii. Grafice: testați contractul de configurare (labels/datasets/options), mock‑uiți constructorul (Chart/ECharts) și asertați că a fost chemat cu structura corectă; testați handlers (legend toggle) ca acțiuni în store; testați că selecțiile din tabel se reflectă în selectorul pentru grafic (nu pixelii). În unit tests, animațiile trebuie dezactivate; în integration tests (RTL) folosiți „smoke” determinist. Pentru performanță, puteți măsura timpi de executare pe JSDOM ca proxy (prudență la variabilitate).

11) Theming și i18n: două orizonturi care afectează tot  
Theming: folosiți CSS variables (`--bg`, `--fg`, `--muted`, `--accent`) și un „theme provider” (Context simplu) pentru comutare. Tabelul poate citi variabilele pentru culori de rând/hover, graficul își generează paleta din aceleași variabile (evitați palete hard‑coded). i18n: separați textul static; formatați numeric/date cu `Intl.\*`. Testele ar trebui să acopere cel puțin două locale (ro‑RO, en‑GB), asertând `1.234,56` vs `1,234.56` și formate de dată.

12) AI‑assist (VSL): cum îl folosim responsabil  
Folosiți prompts scurte, centrate pe o singură sarcină: „Generate TanStack Table columns for members {id, name, club, joinedAt} with sortable headers and aria-sort” sau „Create a Chart.js bar spec for counts per category using a colour‑blind‑safe palette; disable animations for tests”. Mențineți un ciclu scurt: generezi 15–30 de linii → rulezi testele → corectezi → commit mic. Revizuiți manual encodările vizuale (ex.: palete, scări), nu le lăsați „la întâmplare”: percepția umană nu poate fi „ghicită” automat. Nu copiați orbește cod; învățați să cereți „why”, nu doar „what”.

13) Capcane tipice și cum le evităm  
• Dublarea datelor (aceleași valori și în Query, și în slice).   
• Calculul derivatelor în store; păstrați raw + selectors.   
• Păstrarea instanței de Chart/ECharts în store (non‑serializabil).   
• Filtrare cu `toLowerCase` care modifică datele persistente (pierdeți diacritice).   
• „Sort by display value”: sortați după „raw” + formatați la afișare, nu invers.   
• Export CSV „formatat” (stringuri locale) care rupe re‑importul; preferați opțiunea de export raw + o opțiune separată „what you see”.   
• Chart „tematic” dar ilizibil (contrast scăzut, saturări excesive).   
• Teste de screenshot fragile; preferați contracte pe config + DOM observabil.

14) Concluzie: ce înseamnă „calitate” într‑un dashboard didactic  
Calitate înseamnă: (a) date curate și modelate corect, (b) strat de stare coerent (store + selectors), (c) tabele/grafice integrate prin contracte testabile, (d) performanță și a11y verificate explicit, (e) reproducibilitate (export config + dataset), (f) i18n/theming proiectate „by design” și (g) o cultură de lucru cu AI care accelerează, dar nu substituie judecata de proiectare. Partea a 2‑a va traduce această hartă într‑un laborator complet („ClubHub+ Analytics”), cu worksheet și teste, iar Partea a 3‑a va oferi 45 de proiecte tematice (de la fundamental la avansat) care consolidează deprinderile.

—

Exemple orientative (codul de mai jos este ilustrativ, nu intră în numărătoarea de cuvinte).

Ex. 1 — Selectori derivând config pentru Chart.js din aceeași „surse de adevăr” (store).

// TypeScript — selector pentru Chart.js  
type Member = { id: string; club: string; joinedAt: string }  
type RootState = { members: Member[]; filters: { club?: string } }  
  
export const selectChartConfig = (s: RootState) => {  
 const items = s.members.filter(m => !s.filters.club || m.club === s.filters.club)  
 const byClub = new Map<string, number>()  
 for (const m of items) byClub.set(m.club, (byClub.get(m.club) ?? 0) + 1)  
 const labels = Array.from(byClub.keys())  
 const data = Array.from(byClub.values())  
 return {  
 type: 'bar',  
 data: { labels, datasets: [{ label: 'Members per club', data }] },  
 options: { animation: false, responsive: true }  
 }  
}

Ex. 2 — TanStack Table: coloane cu headeruri sortabile și aria‑sort gestionat de bibliotecă.

// TypeScript (pseudo) — TanStack Table column defs  
import { createColumnHelper } from '@tanstack/react-table'  
type Row = { id: string; name: string; club: string; joinedAt: string }  
const h = createColumnHelper<Row>()  
export const columns = [  
 h.accessor('name', { header: 'Name' }),  
 h.accessor('club', { header: 'Club' }),  
 h.accessor('joinedAt', { header: 'Joined', cell: (info)=> new Date(info.getValue()).toLocaleDateString() })  
]

Ex. 3 — Chart.js: inițializare deterministă (fără animații) pentru a fi testabilă.

// TypeScript — inițiere Chart.js  
import { Chart, BarController, BarElement, CategoryScale, LinearScale, Tooltip, Legend } from 'chart.js'  
Chart.register(BarController, BarElement, CategoryScale, LinearScale, Tooltip, Legend)  
  
export function buildBar(el: HTMLCanvasElement, labels: string[], values: number[]) {  
 return new Chart(el, {  
 type: 'bar',  
 data: { labels, datasets: [{ label: 'Count', data: values }] },  
 options: { animation: false, responsive: true }  
 })  
}

Ex. 4 — ECharts: „option” ca obiect declarativ.

// TypeScript — inițiere ECharts  
import \* as echarts from 'echarts'  
  
export function buildEChart(el: HTMLDivElement, labels: string[], values: number[]) {  
 const chart = echarts.init(el, undefined, { renderer: 'canvas' })  
 const option = {  
 xAxis: { type: 'category', data: labels },  
 yAxis: { type: 'value' },  
 series: [{ type: 'bar', data: values }],  
 animation: false  
 }  
 chart.setOption(option)  
 return chart  
}

Ex. 5 — Test „prin contract” (Jest/Vitest): configurarea graficului.

// Jest/Vitest — test pentru config  
import { selectChartConfig } from './selectors'  
  
test('chart config respects current filters', () => {  
 const state = {  
 members: [  
 { id: '1', club: 'Tech', joinedAt: '2025-01-02' },  
 { id: '2', club: 'Arts', joinedAt: '2025-02-03' },  
 { id: '3', club: 'Tech', joinedAt: '2025-03-04' },  
 ],  
 filters: { club: 'Tech' }  
 }  
 const cfg = selectChartConfig(state)  
 expect(cfg.data.labels).toEqual(['Tech'])  
 expect(cfg.data.datasets[0].data).toEqual([2])  
 expect(cfg.options.animation).toBe(false)  
})

Bibliografie (APA 7, doar surse cu DOI)  
  
Cleveland, W. S., & McGill, R. (1984). Graphical perception: Theory, experimentation, and application to the development of graphical methods. Journal of the American Statistical Association, 79(387), 531–554. https://doi.org/10.1080/01621459.1984.10478080  
Heer, J., & Bostock, M. (2010). Crowdsourcing graphical perception: Using Mechanical Turk to assess visualization design. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI ’10) (pp. 203–212). https://doi.org/10.1145/1753326.1753357  
Mackinlay, J. (1986). Automating the design of graphical presentations of relational information. ACM Transactions on Graphics, 5(2), 110–141. https://doi.org/10.1145/159544.159617  
Bostock, M., Ogievetsky, V., & Heer, J. (2011). D3: Data-driven documents. IEEE Transactions on Visualization and Computer Graphics, 17(12), 2301–2309. https://doi.org/10.1109/TVCG.2011.185  
Munzner, T. (2014). Visualization Analysis and Design. CRC Press. https://doi.org/10.1201/b17511  
Munzner, T. (2009). A nested model for visualization design and validation. IEEE Transactions on Visualization and Computer Graphics, 15(6), 921–928. https://doi.org/10.1109/TVCG.2009.111  
Wickham, H. (2014). Tidy data. Journal of Statistical Software, 59(10), 1–23. https://doi.org/10.18637/jss.v059.i10