**Seminarul 10 — React: componente și JSX — Partea 1 (Teorie)**

\*\*Context didactic şi ţintă:\*\* Această Parte 1 este o \*\*introducere riguroasă şi narativă\*\* în React (componente funcţionale şi \*\*JSX\*\*), aliniată la \*\*React 19\*\* şi la modul în care documentaţia oficială recomandă să începem proiectele moderne: cu un \*\*build tool\*\* precum \*\*Vite/Parcel/Rsbuild\*\* şi o gândire „\*\*Thinking in React\*\*” asupra structurării UI‑ului. În tot textul, termenii tehnici şi comenzile vor rămâne în \*\*English\*\* (unde se potriveşte), iar explicaţiile conceptuale sunt în \*\*română\*\*, cu analogii din lumea reală. Vom face şi trimiteri precise la documentaţia oficială acolo unde contează, pentru a evita mitologii sau practici depăşite.

# 0. Hook realist: de ce React, de ce acum (şi de ce JSX)?

Imaginează‑ţi că un \*\*asistent universitar\*\* coordonează o platformă pentru \*\*asociaţiile studenţeşti\*\* („ClubHub”). În fiecare toamnă apar cluburi noi, iar cele vechi îşi schimbă structura: pagini cu descrieri, calendare, formulare de înscriere, galerii, feed‑uri de noutăţi. La început, asistentul a folosit \*\*HTML + CSS + jQuery\*\* (imperativ) şi a legat manual multe \*\*event handlers\*\* de \*\*DOM\*\*. După un an, interfaţa a devenit greu de întreţinut: „când apăs aici se strică dincolo”.

\*\*React\*\* propune altceva: \*\*UI‑uri declarative\*\* compuse din \*\*componente\*\* (funcţii) care descriu \*\*ce\*\* afişăm pentru o \*\*stare\*\* dată şi lasă bibliotecii grija \*\*cum\*\* actualizează DOM‑ul. În loc să „împingi” instructiv fiecare nod DOM, \*\*descrii intenţia\*\* în JSX, iar React se ocupă de reconciliere. Pe româneşte, „\*\*îi spui chelnerului ce vrei\*\*” (starea finală a mesei), nu „\*\*îi mişti tu toate farfuriile\*\*” (DOM) pe masă.

Oficial, React „te lasă să construieşti interfeţe din bucăţi individuale numite \*\*componente\*\*”, pe care apoi le poţi compune în „ecrane, pagini, aplicaţii” – aceasta este exact ideea centrală de modularitate care face ca UI‑ul să rămână lizibil şi extensibil pe termen lung. citeturn0search14

# 1. Ce este React (pe scurt) şi unde se află azi (React 19)

React este o \*\*bibliotecă JavaScript\*\* pentru \*\*user interfaces\*\*. Nu este un framework complet de routing, data‑fetching şi SSR (acestea vin din ecosistem), dar oferă un model declarativ sănătos: \*\*componente\*\* + \*\*state\*\* + \*\*props\*\* + \*\*reconciliation\*\*.

În \*\*React 19\*\* (stabil din \*\*5 decembrie 2024\*\*), echipa a introdus suport pentru \*\*async functions în transitions\*\*, care pot gestiona \*\*pending states\*\*, \*\*erori\*\*, \*\*formuri\*\* şi \*\*optimistic updates\*\* \*\*automat\*\*. Acest lucru înseamnă că anumite fluxuri asincrone (trimiterea unui formular, de pildă) devin mai \*\*ergonomice\*\* şi \*\*lizibile\*\*, cu mai puţin cod „lipici” pentru încărcare/eroare. citeturn0search0

\*\*Upgrade path.\*\* În aprilie 2024, echipa a publicat şi \*\*React 18.3\*\* – identic cu 18.2, dar cu \*\*warnings\*\* pentru API‑urile deprecate care pregătesc terenul pentru 19. Recomandarea a fost: \*\*urcă mai întâi la 18.3\*\*, verifică warnings, apoi \*\*upgrade la 19\*\*. Este o practică sănătoasă în proiecte mari. citeturn0search4

\*\*Documentaţia oficială\*\* este vie şi ancorată în major‑ul curent; în momentul redactării, \*\*React 19.1\*\* apare menţionat ca „latest” pe pagina de versiuni. Pentru afirmaţii sensibile (noţiuni noi, ghid de upgrade), consultăm mereu \*\*blogul\*\* oficial şi paginile \*\*Learn/Reference\*\*. citeturn0search12turn0search9

# 2. De ce JSX (şi de ce nu e „doar HTML”)

\*\*JSX\*\* este un \*\*syntax extension\*\* pentru JavaScript care ne permite să scriem \*\*markup de tip HTML‑like\*\* în fişierele JS/TS. Spre deosebire de template‑urile tradiţionale, JSX \*\*îmbină markup‑ul cu logica\*\* la nivel de componentă – un \*\*collocation\*\* ce ajută enorm la mentenanţă: vezi în acelaşi loc datele, transformările şi rezultatul vizual. Documentaţia explică explicit că JSX este preferat de majoritatea proiectelor pentru concizie şi claritate, şi subliniază diferenţele faţă de HTML (numele atributelor, evenimente, \*escaping\*). citeturn0search6

În practică, JSX e transformat („\*\*JSX transform\*\*”) în apeluri de funcţii (proces gestionat de \*\*Babel/TypeScript\*\* ori de build tool). Dar pentru tine, ca autor de componente, JSX este \*\*limbajul de descriere\*\* a interfeţei: „\*\*scriu cum arată\*\* UI‑ul pentru această \*\*stare\*\*”.

# 3. Gândirea în React („Thinking in React”)

Abordarea canonică porneşte de la o \*\*descompunere\*\* a UI‑ului în \*\*componente\*\* (cutii conceptuale) şi o \*\*inventariere\*\* a \*\*stărilor vizuale\*\* posibile pentru fiecare. Apoi, conectezi componentele astfel încât \*\*datele\*\* să \*\*circule unidirecţional\*\* (\*\*down\*\* prin \*props\*), iar \*\*interacţiunile\*\* să \*\*propage înapoi\*\* intenţii (callback‑uri, event handlers) care \*\*actualizează state‑ul\*\*. Documentaţia oficială oferă un ghid practic exact în acest spirit. citeturn0search2

Analogic, gândeşte un \*\*panou de anunţuri\*\*: îl împarţi în „cap de pagină” (\*Header\*), „listă de anunţuri” (\*List\*), „filtru” (\*Filter\*), „item” (\*NoticeCard\*). Fiecare bucată îşi declară propriul UI pentru starea locală, iar ansamblul rămâne predictibil.

# 4. Componente: props, state, data‑flow unidirecţional

O \*\*componentă\*\* este o \*\*funcţie\*\* care primeşte \*\*props\*\* (date imutabile din afară) şi are (dacă e nevoie) un \*\*state\*\* local. \*\*Props\*\* ≈ „\*\*instrucţiuni de servire\*\*” date din \*\*bucătăria\*\* principală (părinte) către \*\*chelner\*\* (componenta copil). \*\*State\*\* ≈ „\*\*memorie\*\* internă” a componentei, schimbată prin apeluri de tip `setState` (în hook‑uri precum `useState`).

Când două componente trebuie să \*\*sincronizeze\*\* o valoare (ex.: `SearchBox` şi `List`), \*\*ridici state‑ul\*\* în cel mai apropiat \*\*părinte comun\*\* (\*lifting state up\*) şi transmiţi `value` + `onChange` în jos, prin \*props\*. React dev explică clar această mişcare („\*\*Sharing state between components\*\*”). citeturn0search23

> \*\*Anti‑pattern:\*\* duplicarea aceluiaşi „adevăr” în două locuri (stări concurente) duce la inconsistenţe.

# 5. Evenimente (events) şi actualizarea UI‑ului

În JSX, \*\*ataşezi\*\* \*event handlers\* direct ca \*\*props camelCase\*\* (`onClick`, `onChange`, `onSubmit`), cu \*\*funcţii\*\* drept valori. Aceste funcţii vor fi apelate când are loc \*\*interacţiunea\*\* (click, tastare, submit etc.). Documentaţia are un capitol explicit „\*\*Responding to Events\*\*”, cu subtilităţi (propagare, \*stopping propagation\*, \*passing handlers\*). citeturn1search1

În React 18/19, \*\*batching‑ul\*\* actualizărilor de state e \*\*automat\*\*, ceea ce duce la \*\*re‑randări\*\* mai eficiente (mai puţine). Iar \*\*transitions\*\* ajută la marcarea actualizărilor „ne‑critice” (de ex., filtrare grea) pentru a \*\*nu bloca\*\* interacţiunile. În \*\*React 19\*\*, folosirea funcţiilor \*\*asincrone\*\* în \*\*transitions\*\* poate gestiona implicit stările `pending`/`error` – revenim cu exemple în laborator. citeturn0search0

# 6. JSX în profunzime: diferenţe faţă de HTML

- \*\*Atribute\*\*: `class` → `className`, `for` → `htmlFor`.   
- \*\*Stiluri\*\*: `style={{ marginTop: 8 }}` (obiect JS), nu string CSS.   
- \*\*Evenimente\*\*: `onClick={...}`, `onChange={...}` (camelCase).   
- \*\*Expresii\*\*: `{}` pentru a injecta rezultate JS (ex. `total > 0 ? ... : ...`).   
- \*\*Escaping\*\*: JSX escape‑ază automat în text; \*dangerouslySetInnerHTML\* doar în cazuri controlate.

Toate acestea apar detaliate în „\*\*Writing markup with JSX\*\*”. citeturn0search6

Exemplu minim (controlat):

jsx  
function SearchBox({ value, onChange }) {  
 return (  
 <label>  
 Search:  
 <input value={value} onChange={e => onChange(e.target.value)} />  
 </label>  
 );  
}

# 7. Liste & keys: identitate stabilă, reconciliere previzibilă

Când afişezi colecţii (`Array.map`), fiecare element trebuie să aibă un \*\*`key` stabil\*\* (de regulă un \*\*ID\*\* din date). \*Keys\* îl ajută pe React să \*\*păstreze identitatea\*\* elementelor între re‑randări (important pentru starea internă a copiilor). Documentaţia modernă explică „\*\*Rendering Lists\*\*” şi „\*\*Preserving and Resetting State\*\*” – cheile nu sunt doar pentru liste; pot controla şi \*\*resetarea\*\* stării. citeturn1search0turn1search10

Exemplu:

jsx  
{clubs.map(club => <ClubCard key={club.id} club={club} />)}

> \*\*Observaţie:\*\* folosirea \*\*indexului\*\* din `map` drept key e o \*\*ultimă soluţie\*\* (doar când lista este statică şi nu se reordonează).

# 8. Condiţionale: ramuri lizibile fără „if” imperative în DOM

Poţi \*\*condiţiona\*\* markup‑ul folosind `if`, `&&` sau `? :`. Documentaţia „\*\*Conditional Rendering\*\*” arată idiomurile uzuale şi când merită sparte componentele pentru claritate. citeturn1search2

Exemplu:

jsx  
{results.length === 0 ? <EmptyState /> : <Results items={results} />}

# 9. Controlled vs. uncontrolled (form‑uri) şi „lifting state up”

Pentru formular, „controlled” înseamnă că \*\*valoarea\*\* inputului vine din \*\*state\*\* şi se schimbă prin \*\*onChange\*\*; „uncontrolled” foloseşte \*\*refs\*\* şi lasă browserul să gestioneze starea nativă. Cu „controlled” ai \*\*o singură sursă a adevărului\*\* (mai uşor de sincronizat cu restul UI‑ului).

Analogic: „controlled” e ca un \*\*caiet unic\*\* de prezenţă în care toată lumea semnează; „uncontrolled” e ca \*\*bileţele\*\* adunate de fiecare şi centralizate abia la final.

# 10. Compoziţie şi `children`: mai puţină moştenire, mai multă compunere

În React favorizăm \*\*compoziţia\*\*: construieşti \*primitive\* UI (ex. `Card`, `Stack`, `Grid`) şi le combini prin `children` şi \*props\*. Moştenirea clasică (OOP) apare rar în designul componentelor; compoziţia e mai \*\*flexibilă\*\* şi \*\*testabilă\*\*.

Exemplu:

jsx  
function Card({ children }) {  
 return <div className="card">{children}</div>;  
}

jsx  
<Card>  
 <ClubCard club={c} />  
</Card>

# 11. Separarea „event‑urilor” de „efecte”

React recomandă să diferenţiem \*\*codul determinist\*\* de \*\*efecte\*\* (I/O, setări globale). „\*\*Separating events from effects\*\*” clarifică faptul că \*event handlers\* rulează ca răspuns la utilizator, în timp ce \*effects\* sunt reacţii la \*render state\* (nu abuzăm de ele pentru orice). citeturn1search9

# 12. Testarea componentelor (principii pentru Partea 2)

În testare, mentalitatea este „\*\*cum vede utilizatorul\*\*”: \*\*React Testing Library\*\* (RTL) încurajează căutarea după \*\*roluri\*\* şi \*\*text\*\*. Noi vom rula \*\*Vitest\*\* şi \*\*Jest\*\* \*\*în paralel\*\* (aceeaşi suită logică), pentru a discuta micile diferenţe de \*runner\*.

# 13. Iniţializarea proiectelor moderne (Vite/Parcel/Rsbuild)

Documentaţia oficială recomandă să porneşti de la un \*\*build tool\*\* modern ca \*\*Vite\*\*, \*\*Parcel\*\* sau \*\*Rsbuild\*\* atunci când \*\*construieşti de la zero\*\*. Noi vom standardiza pe \*\*Vite\*\* (rapid, ESM, integrare simplă cu TS). citeturn0search3turn0search8turn0search20

Exemplu iniţializare (în Partea 2 vom detalia):

bash  
npm create vite@latest clubhub -- --template react  
cd clubhub && npm i && npm run dev

# 14. Cum folosim AI‑assist eficient (Copilot/LLM) — VSL (Very Short Loop)

\*\*VSL = paşi mici + feedback rapid.\*\* În loc să ceri „scrie toată aplicaţia”, formulezi \*\*prompts scurte\*\* pentru exact bucata la care lucrezi (max. 10–20 linii), rulezi testele, ajustezi.

\*\*Exemple de prompts utile:\*\*  
- „Write a React functional component `ClubCard` with props `{ name, category }` and a simple test using React Testing Library (Vitest).”  
- „Turn this uncontrolled input into a controlled one (`value` + `onChange`) and explain the difference in one paragraph.”  
- „Refactor: extract a `useFilteredClubs(clubs, query)` custom hook. Provide usage example.”  
- „Show a small example of conditional rendering with `? :` and `&&`, and discuss when to split into separate components.”

\*\*Regulă de igienă:\*\* chiar dacă AI te ajută cu \*boilerplate\*, \*\*verifică\*\* în documentaţia oficială React când apar \*\*API‑uri noi\*\* (ex. React 19, \*actions\*/\*transitions\*) şi învaţă \*\*de ce\*\* funcţionează, nu doar \*\*ce\*\* tastezi. (Blogul React şi secţiunile \*Learn\*/\*Reference\* sunt surse canonic validate.) citeturn0search0turn0search11

# 15. Capcane frecvente şi „hărţi mentale”

- \*\*Key instabilă\*\* → re‑montări surpriză, pierdere de stare. Foloseşte ID‑uri reale. citeturn1search0   
- \*\*State duplicat\*\* în părinte + copil → rămân „pe dinafară”. Fă \*\*lifting state up\*\*. citeturn0search23   
- \*\*Event handler\*\* care conţine logică de „efect” persistent → mută în \*effect\* sau în serviciu. citeturn1search9   
- \*\*JSX ca HTML\*\* → atenţie la `className`, `htmlFor`, obiectul `style`, `onClick` camelCase. citeturn0search6   
- \*\*Condiţionale haotice\*\* → preferă \*guarded returns\* sau sparge în componente. citeturn1search2

# 16. Mini‑glosar (React 19 în câteva cuvinte)

- \*\*Componentă\*\*: funcţie care primeşte \*\*props\*\* şi descrie UI.   
- \*\*JSX\*\*: sintaxă „HTML‑like” în JS; transpune în apeluri de funcţii. citeturn0search6   
- \*\*State\*\*: memorie locală; actualizat declarativ prin `setState`.   
- \*\*Props\*\*: date externe, imuabile pentru componenta copil.   
- \*\*Keys\*\*: identitate stabilă în liste/structuri. citeturn1search0   
- \*\*Transition\*\*: marchează actualizări \*\*ne‑critice\*\*; în React 19, funcţiile \*\*asincrone\*\* în transitions pot gestiona pending/erori/optimistic \*\*automat\*\*. citeturn0search0   
- \*\*Build tool\*\*: Vite/Parcel/Rsbuild pentru proiecte „from scratch”. citeturn0search3

# 17. Ce urmează în Partea 2 (laborator)

Vom construi „\*\*ClubHub\*\*” incremental:  
1) componente + JSX; 2) events + state; 3) lists & keys; 4) controlled forms; 5) composition; 6) un mic \*transition\* pentru filtrare grea; 7) refactor + a11y. Vom livra \*\*worksheet\*\* (cerinţă + checklist), \*\*starter code\*\* şi \*\*unit tests\*\* (Vitest/Jest).

# Bibliografie (APA 7) — surse oficiale web (fără DOI)

- React Team. (2024, Dec 5). \*React v19\*. react.dev. https://react.dev/blog/2024/12/05/react-19 citeturn0search0   
- React Team. (2024, Apr 25). \*React 19 Upgrade Guide\*. react.dev. https://react.dev/blog/2024/04/25/react-19-upgrade-guide citeturn0search4   
- React Team. (n.d.). \*Writing markup with JSX\*. react.dev. https://react.dev/learn/writing-markup-with-jsx citeturn0search6   
- React Team. (n.d.). \*Thinking in React\*. react.dev. https://react.dev/learn/thinking-in-react citeturn0search2   
- React Team. (n.d.). \*Build a React app from scratch\*. react.dev. https://react.dev/learn/build-a-react-app-from-scratch citeturn0search3   
- React Team. (n.d.). \*Rendering Lists\*. react.dev. https://react.dev/learn/rendering-lists citeturn1search0   
- React Team. (n.d.). \*Responding to Events\*. react.dev. https://react.dev/learn/responding-to-events citeturn1search1   
- React Team. (n.d.). \*Conditional Rendering\*. react.dev. https://react.dev/learn/conditional-rendering citeturn1search2   
- React Team. (n.d.). \*Preserving and Resetting State\*. react.dev. https://react.dev/learn/preserving-and-resetting-state citeturn1search10   
- React Team. (n.d.). \*React Versions\*. react.dev. https://react.dev/versions citeturn0search12   
- Vite Team. (n.d.). \*Getting Started\*. vite.dev. https://vite.dev/guide/ citeturn0search20

# Referinţe academice (APA 7, cu DOI)

- Bainomugisha, E., Carreton, A. L., Van Cutsem, T., Mostinckx, S., & De Meuter, W. (2013). \*A survey on reactive programming\*. ACM Computing Surveys, 45(4), Article 52. https://doi.org/10.1145/2501654.2501666 citeturn2search0   
- Elliott, C., & Hudak, P. (1997). \*Functional reactive animation\*. In \*Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming\* (pp. 263–273). https://doi.org/10.1145/258948.258973 citeturn2search1   
- Crnkovic, I. (2005). \*Towards component-based software engineering\*. In \*Component-Based Software Engineering\* (pp. 1–20). Springer. https://doi.org/10.1007/11560647\_5 citeturn2search22   
- Nielsen, J. (1990). \*Heuristic evaluation of user interfaces\*. In \*Proceedings of the SIGCHI Conference on Human Factors in Computing Systems\* (pp. 249–256). https://doi.org/10.1145/97243.97281 citeturn2search11