**Seminarul 11 — Gestionarea stării cu Redux Toolkit  
Partea 1 — Teorie extinsă (narativ + exemple)**

HOOK (situație realistă). „ClubHub+” crește: asociația studențească are dintr‑odată mii de membri, mai multe filiale, coordonatori (lead), evenimente recurente și roluri diferite (admin, treasurer, content, logistics). Datele sosesc din multe direcții: formulare de înscriere, endpoint‑uri HTTP (REST), CSV‑uri importate, iar UI‑ul afişează filtre, paginare, dialoguri de confirmare, meniuri. La început, am folosit state local (useState) şi context pentru câteva setări. După câteva săptămâni, apar simptome clasice: „prop drilling”, duplicare de „surse ale adevărului” (aceeaşi persoană apare diferit în două locuri), efecte asincrone presărate prin componente, teste fragile, re‑randări aleatoare, bug‑uri greu de reprodus. Echipa cere un „control room” pentru starea aplicaţiei: un loc previzibil, urmărit în timp, cu reguli clare pentru cum intră, se schimbă şi iese datele. Aici intră Redux Toolkit (RTK): recomandarea oficială pentru a scrie logică Redux modernă, cu API‑uri ce reduc boilerplate‑ul, standardizează structura şi oferă instrumente de performanţă şi testare.

1. De ce Redux (şi de ce Redux Toolkit) „astăzi”  
În aplicaţii reale, nu toate stările sunt egale. O parte este „UI state” (toggle‑uri, meniuri deschise, input‑uri temporare), o parte este „app state” (date partajate între ecrane: utilizatorul curent, setări, colecţii de entităţi precum clubs/members/events), iar o parte este „server cache” (date provenite din reţea, cu invalidări şi politici de re‑fetch). Încercarea de a le trata pe toate la fel, doar cu state local, produce rapid inconsistenţe. Redux propune un model simplu şi riguros: o „single source of truth”, actualizări prin „actions” şi „reducers” pure (fără efecte colaterale), flux unidirecţional şi posibilitatea de „time‑travel” (DevTools). În practică însă, Redux „clasic” cerea multe fişiere, tipare repetitive şi configurări migăloase. Redux Toolkit (RTK) a apărut exact pentru a face din bunele practici reguli implicite: configureStore, createSlice, createAsyncThunk, createEntityAdapter, plus RTK Query pentru „server state”. Împreună, acestea reduc zgomotul, previn greşelile tipice şi fac codul mai lizibil, mai uşor de testat şi extins.

Analogia din viaţa de zi cu zi: gândiţi‑vă la un registru central al unei asociaţii. Dacă fiecare departament ţine un Excel propriu (state local), la final nu mai ştiţi care este „adevărul”. Un registru unic (store) şi reguli clare despre „cine scrie ce” (actions/reducers) şi „cine citeşte ce” (selectors) reduc ambiguitatea. RTK este secretariatul digital care impune format unic, validări şi proceduri, astfel încât orice actualizare e înregistrată, verificabilă şi reversibilă.

2. Fundamente Redux (revizitare în stil practic)  
Redux porneşte de la ideea că starea (state) este un arbore imutabil. Schimbările se fac prin „actions” (descrieri ale intenţiilor) care sunt procesate de „reducers” pure. Reducerul primeşte (state, action) şi returnează un nou state. „Store”‑ul menţine starea, oferă „dispatch(action)” şi „subscribe(listener)”, iar „React‑Redux” pune la dispoziţie hooks pentru a citi şi a dispeceriza: useSelector şi useDispatch. Beneficiile: predictibilitate, debuggabilitate, testare simplă a reducerelor (sunt funcţii pure), şi „time‑travel” în DevTools. Costul: fără RTK, mult boilerplate, actualizări imutabile scrise manual, configurări diferite per proiect.

3. De ce „Redux = Redux Toolkit” în practică  
RTK este „bateriile incluse”: configureStore (store + middleware implicit + DevTools), createSlice (reducer + acţiuni generate), createAsyncThunk (model asincron standard), createEntityAdapter (normalizare + CRUD + selectori), createListenerMiddleware (efecte orchestrate în funcţie de acţiuni / schimbări de state), autoBatchEnhancer (reduce notificările către abonaţi pentru acţiuni „low‑priority”). În plus, RTK Query adresează „server state” cu cache, invalidări şi hooks auto‑generate. Într‑un proiect modern, aproape întotdeauna e mai simplu să porneşti cu RTK decât cu „Redux de bază”.

4. Tipuri de stare şi responsabilităţi (UI vs App vs Server)  
• UI state: local şi efemer (ex.: „modal open”, „active tab”). Se rezolvă cu useState/useReducer în componentă sau context de mică amploare.   
• App state: entităţi partajate (users, clubs, events), permisiuni, filtre globale. Necesită o „single source of truth”, normalizare, selectori memoizaţi şi politici de actualizare previzibile → Redux Toolkit (slices + entityAdapter).   
• Server state: date sincronizate cu backend, cu revalidare, cache şi invalidări (tag‑uri). Aici, RTK Query este de regulă soluţia nativă, cu un plus net la ergonomie.

5. Imutabilitate şi „mutating immutable updates”  
Imutabilitatea ajută la reasoning şi performanţă (comparare structurală, memoizare). Scrierea manuală (spread, concat) e obositoare şi predispusă la erori. RTK foloseşte Immer sub capotă, astfel încât reducerul poate scrie „ca şi cum ar muta” (state.value++) dar produce, de fapt, un nou arbore imutabil. Aceasta reduce masiv „accidental complexity”.

6. createSlice: standardizarea logicii pe „feature”  
createSlice acceptă un nume, starea iniţială şi un obiect de „case reducers”. Generează acţiuni (action creators) şi tipuri de acţiune din oficiu. Reduce fragmentarea fişierelor şi asigură o cartografiere clară între intenţii şi modificarea stării. Combinat cu configureStore, duce la o iniţializare aproape trivială a întregului flux.

7. Selectori şi Reselect: citire eficientă, derivări pure  
Selectori = funcţii care extrag (sau derivă) date din store. Reselect (createSelector) oferă memoizare: dacă intrările nu s‑au schimbat (===), rezultatul e returnat din cache. Evită recalculări costisitoare şi re‑randări. „Small is beautiful”: scrie selectori mici şi compune‑i. Evită selectori care creează obiecte noi la fiecare apel (rup memoizarea).

8. Entităţi normalizate cu createEntityAdapter  
Listele mari (clubs, members, events) se gestionează mai simplu când sunt normalizate (byId + allIds). createEntityAdapter oferă reduceri CRUD predefinite (addOne, upsertMany etc.) şi selectori (selectAll, selectById, selectIds), plus posibilitatea de a defini „sortComparer” pentru ordonare. Beneficiu: operaţii O(1) pentru lookup, actualizări predictibile, UI performant pe liste mari.

9. Async cu createAsyncThunk şi orchestration cu listenerMiddleware  
createAsyncThunk defineşte un „thunk” cu ciclul de viaţă pending/fulfilled/rejected şi acţiuni aferente, integrabile în extraReducers. Simplifică modelarea I/O (fetch, POST, PUT, DELETE) şi tratarea erorilor. Pentru scenarii care cer reacţii la acţiuni (debounce, throttle, cancelări, „închide dialogul la succes”), listenerMiddleware oferă un mod declarativ de a porni efecte când se dispecerizează o acţiune sau când anumite segmente de state se schimbă.

10. RTK Query: „server state” ca prim cetăţean  
createApi + fetchBaseQuery (sau un baseQuery personalizat) generează reducer, middleware, selectori şi (opţional) hooks React pentru fiecare endpoint. Beneficii: cache automat, „stale‑while‑revalidate”, invalidări pe tag‑uri, re‑fetch la focus/reconnect, „optimistic updates” şi funcţii de lifecycle. E de preferat să separăm clar „server cache” (RTK Query) de „app state” (slices), pentru a evita dublarea datelor şi conflictele.

11. TypeScript: tipuri puternice, ergonomie în IDE  
Tiparea în RTK presupune definirea RootState, AppDispatch şi a „typed hooks” (useAppDispatch, useAppSelector). createSlice creează „action creators” tipate; createAsyncThunk poate avea tipare stricte pentru argument, rezultat şi error payload; entityAdapter oferă tipuri pentru selectori. Beneficiu: „autocomplete” coerent şi erori descoperite din timp.

12. Arhitecturi orientate „feature” şi code‑splitting (combineSlices)  
Structura recomandată: /app (store, typed hooks), /features/<featureName> (slice, selectors, components), /entities (adapters pentru entităţi), /services (RTK Query). Pentru aplicaţii mari, RTK 2.x introduse utilitare precum combineSlices ce simplifică „lazy loading” de reduceri şi tipurile aferente. E util când rutele se încarcă la cerere sau când anumite zone sunt opţionale (ex.: „admin”).

13. Testare: reduceri, selectori, thunks, UI  
• Reduceri: teste pure, uşor de citit.   
• Selectori: teste pentru corectitudinea derivărilor şi pentru memoizare (număr de recomputări).   
• Thunks: teste cu MSW (Mock Service Worker) sau cu mock‑uri ale reţelei; aserţii pe acţiunile dispecerizate.   
• UI: teste RTL peste provider + store real sau unul configurat special pentru test.   
Scopul: încredere în refactorizări, detectarea regresiilor, design orientat spre testabilitate.

14. Performanţă: autoBatchEnhancer, selectori şi granularitate  
UI‑ul este scump; reducerea frecvenţei de notificare şi a ariei de re‑randare este esenţială. autoBatchEnhancer amână notificările pentru rafale de acţiuni „low‑priority” → mai puţine re‑randări. Selectori bine scrişi limitează recalculările; „feature slicing” reduce fragmentarea. Lista virtualizată şi chei stabile (id‑uri) ajută la scări mari.

15. Persistenţă şi serializabilitate  
Stocarea (parţială) a stării în localStorage se poate face prin middleware dedicat (cu debounce, migraţii, „rehydration”). RTK oferă „serializableCheck” — atenţie la tipuri nerecomandate (Date, Map, class instances); dacă e nevoie, serializaţi explicit. La „logout”, efectuaţi reset de store (pattern cu „root reducer” care tratează acţiunea de reset).

16. AI‑assist (VSL) pentru viteză şi calitate  
Cu GitHub Copilot (sub VSL) ori LLM‑uri (ChatGPT/Mistral/Claude), acceleraţi: schelete de slice, thunks, adapteri, selectori şi teste. Recomandare: prompts scurte („Create a createSlice for clubs with entityAdapter and add basic CRUD reducers”) şi bucle rapide (rulat teste imediat, citit diff‑uri mici, corectat). Evitaţi generarea masivă fără înţelegere; validaţi manual tipurile şi respectaţi convenţiile proiectului.

17. Antipattern‑uri frecvente  
• Duplicare „server cache” în store (aceleaşi date şi în RTK Query, şi în slice).   
• Stocare de valori nedefinite ca „derived state” (ex.: număr de rezultate) — calculaţi prin selectori.   
• Reduceri impure (I/O, date random) — mutaţi în thunks/listeners.   
• Selectori care creează obiecte noi la fiecare apel — rupe memoizarea; preferaţi compoziţia de selectori.   
• Non‑serializabile în store (Date/Map/classes) — preferaţi forme serializabile.   
• Chei din index de listă în UI — pot duce la re‑randări incorecte; folosiţi id‑uri.

18. Checklist rapid (decizii informate)  
• UI state local, app state în RTK slices, server cache în RTK Query.   
• Entităţi mari → entityAdapter (normalizare + selectori).   
• I/O simplu → createAsyncThunk; reacţii la acţiuni → listenerMiddleware.   
• TS „typed hooks”, RootState, AppDispatch; selectori memoizaţi.   
• autoBatchEnhancer activ (implicit în configureStore moderne).   
• Teste: reduceri, selectori, thunks, UI.   
• Persistenţă strictă şi reset pe logout.   
• DevTools în toate mediile non‑prod.

19. Concluzie  
RTK a devenit „limbajul comun” pentru gestionarea stării în aplicaţiile React serioase: scade complexitatea accidentală, converteşte bunele practici în implicituri şi păstrează uşor separarea dintre „app state” şi „server state”. În laboratoarele următoare, vom proiecta „ClubHub+ State” cu entităţi normalizate (clubs/members/events), selectori compuşi, fluxuri asincrone tipizate şi testare sistematică. Cu această bază teoretică, echipa poate raţiona clar despre „ce este unde”, „cine actualizează ce” şi „cum demonstrăm că funcţionează”.

—  
Exemple scurte (orientative).

[Ex. 1] createSlice minimal (TS)  
(Observaţie: reducerul pare „mutant”, dar Immer produce stare imutabilă nouă.)

import { createSlice, PayloadAction } from '@reduxjs/toolkit'  
  
type CounterState = { value: number }  
const initialState: CounterState = { value: 0 }  
  
export const counterSlice = createSlice({  
 name: 'counter',  
 initialState,  
 reducers: {  
 increment(state) {  
 state.value++  
 },  
 addBy(state, action: PayloadAction<number>) {  
 state.value += action.payload  
 },  
 },  
})  
  
export const { increment, addBy } = counterSlice.actions  
export default counterSlice.reducer

[Ex. 2] Entity adapter (normalizare implicită + selectori)

import { createSlice, createEntityAdapter } from '@reduxjs/toolkit'  
  
type Club = { id: string; name: string; category?: string }  
const clubsAdapter = createEntityAdapter<Club>({ sortComparer: (a, b) => a.name.localeCompare(b.name) })  
  
const clubsSlice = createSlice({  
 name: 'clubs',  
 initialState: clubsAdapter.getInitialState(),  
 reducers: {  
 upsertMany: clubsAdapter.upsertMany,  
 removeOne: clubsAdapter.removeOne,  
 },  
})  
export const clubsSelectors = clubsAdapter.getSelectors<RootState>(s => s.clubs)

[Ex. 3] Async cu createAsyncThunk + extraReducers

import { createAsyncThunk, createSlice } from '@reduxjs/toolkit'  
  
export const fetchClubs = createAsyncThunk('clubs/fetchAll', async () => {  
 const res = await fetch('/api/clubs')  
 if (!res.ok) throw new Error('Network error')  
 return (await res.json()) as { id: string; name: string }[]  
})  
  
const slice = createSlice({  
 name: 'clubs',  
 initialState: { items: [] as any[], status: 'idle' as 'idle'|'loading'|'succeeded'|'failed', error: null as string|null },  
 reducers: {},  
 extraReducers: builder => {  
 builder  
 .addCase(fetchClubs.pending, state => { state.status = 'loading' })  
 .addCase(fetchClubs.fulfilled, (state, action) => { state.status = 'succeeded'; state.items = action.payload })  
 .addCase(fetchClubs.rejected, (state, action) => { state.status = 'failed'; state.error = String(action.error.message || 'error') })  
 }  
})

[Ex. 4] RTK Query — createApi + fetchBaseQuery (server cache)

import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'  
  
export const clubsApi = createApi({  
 reducerPath: 'clubsApi',  
 baseQuery: fetchBaseQuery({ baseUrl: '/api' }),  
 tagTypes: ['Clubs'],  
 endpoints: builder => ({  
 getClubs: builder.query<any[], void>({  
 query: () => '/clubs',  
 providesTags: (result) => result ? [...result.map(({ id }) => ({ type: 'Clubs' as const, id })), { type: 'Clubs', id: 'LIST' }] : [{ type: 'Clubs', id: 'LIST' }],  
 }),  
 addClub: builder.mutation<any, Partial<any>>({  
 query: (body) => ({ url: '/clubs', method: 'POST', body }),  
 invalidatesTags: [{ type: 'Clubs', id: 'LIST' }],  
 })  
 })  
})  
  
export const { useGetClubsQuery, useAddClubMutation } = clubsApi

[Ex. 5] Typed hooks (TS) pentru store

import { configureStore } from '@reduxjs/toolkit'  
import { useDispatch, useSelector, TypedUseSelectorHook } from 'react-redux'  
import counter from './counterSlice'  
  
export const store = configureStore({ reducer: { counter } })  
export type RootState = ReturnType<typeof store.getState>  
export type AppDispatch = typeof store.dispatch  
export const useAppDispatch: () => AppDispatch = useDispatch  
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector

Bibliografie selecționată (APA, ediția a 7‑a)  
  
Surse oficiale (web):  
Redux maintainers. (2024–2025). Why Redux Toolkit is how to use Redux Today. https://redux.js.org/introduction/why-rtk-is-redux-today  
Redux Toolkit docs. (2024–2025). API: createSlice; createAsyncThunk; createEntityAdapter; createApi; fetchBaseQuery; autoBatchEnhancer; listenerMiddleware; Usage with TypeScript; RTK Query Overview. https://redux-toolkit.js.org/  
  
Referințe academice (cu DOI):  
Bainomugisha, E., Lombide Carreton, A., Van Cutsem, T., Mostinckx, S., & De Meuter, W. (2013). A survey on reactive programming. ACM Computing Surveys, 45(4), 52. https://doi.org/10.1145/2501654.2501666  
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  
Michie, D. (1968). “Memo” functions and machine learning. Nature, 218, 19–22. https://doi.org/10.1038/218019a0  
Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), 1053–1058. https://doi.org/10.1145/361598.361623  
Brooks, F. P. (1987). No Silver Bullet—Essence and Accident in Software Engineering. IEEE Computer, 20(4), 10–19. https://doi.org/10.1109/MC.1987.1663532