Modulo 4 — Gestione dello stato (useState • Context+useReducer • Redux • Zustand)

Obiettivo: scegliere lo strumento giusto e saperlo spiegare (con debug)

Obiettivi del modulo

Cosa saprai fare

- Capire dove deve vivere lo stato: locale, condiviso, globale.
- Usare useState per casi semplici e prevedere i re-render.
- Creare uno stato condiviso con Context + useReducer (pattern completo).
- Impostare Redux Toolkit passo-passo, senza ‘magia’.
- Usare Zustand come store minimale e confrontarlo con Redux.
- Aggiungere strumenti di debug per ‘vedere’ lo stato mentre l’app gira.

Che cos’è lo stato

Definizione pratica

- È l’insieme dei dati che cambiano mentre l’app è aperta.
- Quando lo stato cambia, React ricalcola la UI e aggiorna lo schermo.
- Esempi: testo di un input, lista di task, utente loggato, tema.

Problemi che risolviamo

Perché serve un metodo

- Prop drilling: passare props attraverso tanti componenti.
- Dati duplicati: la stessa informazione in più posti.
- Async: loading/error e richieste ripetute.
- Debug: capire chi ha cambiato cosa.

Mappa: dove mettere lo stato

Regole pratiche

- Locale: riguarda una singola schermata/pezzo di UI.
- Condiviso: serve a più componenti nella stessa area.
- Globale: serve in molte schermate/feature.

useState

Quando è la scelta migliore

- Stato che riguarda un solo componente (o pochi figli diretti).
- UI semplice: toggle, input, contatore, modal, loading locale.
- Ottimo punto di partenza: prima di introdurre store globali.

useState — Esempio

Contatore minimale

Counter.tsx

```
import { useState } from "react";
import { Button, Text, View } from "react-native";

export function Counter() {
  const [count, setCount] = useState(0);

  return (
    <View>
      <Text>Count: {count}</Text>
      <Button title="+" onPress={() => setCount(count + 1)}
      />
    </View>
  );
}
```

- Premi → cambia state → render → il testo si aggiorna.

useState

Cosa succede quando aggiorni

- setState pianifica un nuovo render (non cambia ‘subito’ la variabile).
- React ricalcola il componente e aggiorna solo ciò che serve.
- In RN, il bridge aggiorna componenti nativi necessari.

useState — Forma ‘prev’

Evitare stale state

Pattern

```
setCount((prev) => prev + 1);
setCount((prev) => prev + 1); // +2 sicuro
```

- Usalo quando l’update dipende dal valore precedente.

useState: errori comuni

Checklist

- Mettere nello stato dati derivabili (duplicazione).
- Aggiornare oggetti/array senza copia (mutazione).
- setState in loop o senza condizioni (render infiniti).

useState — array/oggetti

Update immutabile

Pattern

```
// ✅ array
setTodos((prev) => [...prev, newTodo]);

// ✅ oggetto
setUser((prev) => ({ ...prev, name: "Matteo" }));
```

- Non mutare prev: crea una nuova struttura.

Context + useReducer

Perché insieme (pattern completo)

- Context evita prop drilling (condivide dati).
- useReducer centralizza la logica (state + action → newState).
- Insieme creano un mini-store condiviso senza librerie esterne.

Pattern (passi)

Context + useReducer

- 1) Definisci state iniziale e reducer.
- 2) Crea due context: StateContext e DispatchContext.
- 3) Provider: useReducer e passa state/dispatch.
- 4) Hooks: useAppState() e useAppDispatch().

Step 1 — State + reducer

Definizione

appState.ts

```
// appState.ts
export const initialState = {
  user: null,
  theme: "light",
};

export function appReducer(state, action) {
  switch (action.type) {
    case "login":
      return { ...state, user: action.payload };
    case "logout":
      return { ...state, user: null };
    case "toggleTheme":
      return { ...state, theme: state.theme === "light" ?
      "dark" : "light" };
    default:
      return state;
  }
}
```

- Reducer = funzione pura: nessun fetch qui.

Step 2 — Context

State + Dispatch separati

contexts.ts

```
import { createContext } from "react";
import { initialState } from "./appState";

export const StateContext = createContext(initialState);
export const DispatchContext = createContext(() => {});
```

- Separare State/Dispatch riduce re-render inutili nei consumer.

Step 3 — Provider

Condividere state e dispatch

AppProvider.tsx

```
import { useReducer } from "react";
import { StateContext, DispatchContext } from "./contexts";
import { appReducer, initialState } from "./appState";

export function AppProvider({ children }) {
  const [state, dispatch] = useReducer(appReducer,
  initialState);

  return (
    <StateContext.Provider value={state}>
      <DispatchContext.Provider value={dispatch}>
        {children}
      </DispatchContext.Provider>
    </StateContext.Provider>
  );
}
```

- Avvolgi la root dell’app con <AppProvider>.

Step 4 — Hooks

useAppState / useAppDispatch

hooks.ts

```
import { useContext } from "react";
import { StateContext, DispatchContext } from "./contexts";

export const useAppState = () => useContext(StateContext);
export const useAppDispatch = () =>
useContext(DispatchContext);
```

- Nei componenti importi solo i hooks.

Uso in UI

Leggere e dispatchare

Esempio

```
import { useAppDispatch, useAppState } from "./hooks";

const { user, theme } = useAppState();
const dispatch = useAppDispatch();

dispatch({ type: "toggleTheme" });
dispatch({ type: "login", payload: { id: "1", name: "Matteo"
 } });
```

- Questo è un mini-store chiaro e didattico.

Quando basta Context+useReducer

Regole pratiche

- Tema/lingua/auth semplice e poche azioni.
- App piccola/media dove vuoi evitare dipendenze.
- Se lo stato cresce troppo, valuta Zustand o Redux.

Pausa

15 minuti

Riprendiamo tra 15 minuti

Redux

Che problema risolve

- Uno store globale: un solo posto dove vive lo stato condiviso.
- Regole standard: utile in team e app grandi.
- Debug: puoi vedere azioni e stato nel tempo.

Flusso Redux

Step-by-step

- 1) UI fa dispatch(Action).
- 2) Store riceve l’action.
- 3) Reducer calcolano il nuovo stato.
- 4) Store salva il nuovo stato.
- 5) UI si aggiorna.

Redux Toolkit

Perché lo usiamo

- Meno boilerplate rispetto a Redux classico.
- createSlice: state + reducers + actions nello stesso file.
- configureStore: setup standard e middleware di default.

Pacchetti (install)

Cosa serve davvero

Install

```
npm install @reduxjs/toolkit react-redux
```

- Toolkit + bindings React.

Checklist

Redux Toolkit

- A) Crea una slice (todosSlice).
- B) Crea lo store e registra la slice.
- C) Avvolgi l’app con Provider.
- D) Usa useSelector/useDispatch in UI.

A1 — Slice: struttura

createSlice + initialState

todosSlice.ts

```
// todosSlice.ts
import { createSlice, nanoid } from "@reduxjs/toolkit";

const initialState = {
  items: [], // { id, title, done }
};

export const todosSlice = createSlice({
  name: "todos",
  initialState,
  reducers: {
    // reducers qui sotto...
  },
});
```

- createSlice genera actions e reducer.

A2 — Slice: addTodo

Action + reducer

reducers

```
addTodo: {
  reducer: (state, action) => {
    state.items.push(action.payload);
  },
  prepare: (title) => ({
    payload: { id: nanoid(), title, done: false },
  }),
},
```

- Immer: puoi ‘mutare’ state ma resta immutabile sotto.

A3 — Slice: toggle/remove

Altre azioni

reducers

```
toggleTodo: (state, action) => {
  const t = state.items.find((x) => x.id ===
  action.payload);
  if (t) t.done = !t.done;
},
removeTodo: (state, action) => {
  state.items = state.items.filter((x) => x.id !==
  action.payload);
},
```

- Funzioni piccole e prevedibili.

A4 — Export

Azioni + reducer

export

```
export const { addTodo, toggleTodo, removeTodo } =
todosSlice.actions;
export default todosSlice.reducer;
```

- In UI importerai actions.

B — Store

configureStore

store.ts

```
// store.ts
import { configureStore } from "@reduxjs/toolkit";
import todosReducer from "./todosSlice";

export const store = configureStore({
  reducer: {
    todos: todosReducer,
  },
});

export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
```

- Qui registri tutte le slice.

C — Provider

Collegare Redux alla root

App.tsx

```
// App.tsx
import { Provider } from "react-redux";
import { store } from "./store";
import Root from "./Root";

export default function App() {
  return (
    <Provider store={store}>
      <Root />
    </Provider>
  );
}
```

- Senza Provider i hooks non funzionano.

D1 — useSelector

Leggere lo stato

UI

```
import { useSelector } from "react-redux";

const items = useSelector((state) => state.todos.items);
```

- Leggi solo il necessario.

D2 — useDispatch

Inviare azioni

UI

```
import { useDispatch } from "react-redux";
import { addTodo, toggleTodo } from "./todosSlice";

const dispatch = useDispatch();

dispatch(addTodo("Studiare Redux"));
dispatch(toggleTodo("some-id"));
```

- Dispatch invia un’azione allo store.

Async in Redux

Thunk (spiegazione semplice)

- Le API sono lente: serve loading/error.
- createAsyncThunk crea un’azione async con 3 stati.
- extraReducers aggiorna lo stato in base al risultato.

Thunk 1 — Definizione

createAsyncThunk

fetchUsers.ts

```
import { createAsyncThunk } from "@reduxjs/toolkit";

export const fetchUsers = createAsyncThunk("users/fetch",
async () => {
  const res = await fetch("https://example.com/api/users");
  return res.json();
});
```

- Genera: pending / fulfilled / rejected.

Thunk 2 — extraReducers

Gestire i 3 stati

usersSlice

```
extraReducers: (builder) => {
  builder
    .addCase(fetchUsers.pending, (state) => {
      state.loading = true;
      state.error = null;
    })
    .addCase(fetchUsers.fulfilled, (state, action) => {
      state.loading = false;
      state.items = action.payload;
    })
    .addCase(fetchUsers.rejected, (state) => {
      state.loading = false;
      state.error = "Errore di rete";
    });
}
```

- La UI può mostrare spinner/errore.

Thunk 3 — Dispatch

Avviare il fetch

UI

```
import { useDispatch } from "react-redux";
import { fetchUsers } from "./fetchUsers";

const dispatch = useDispatch();
dispatch(fetchUsers());
```

- Tipico: in useEffect quando la schermata si monta.

Debug dello stato

Perché serve DevTools

- DevTools mostra: azioni, payload, stato prima/dopo.
- Aiuta a capire velocemente perché la UI è in uno stato strano.
- Utile soprattutto quando più schermate modificano lo stesso dato.

Redux DevTools in RN

Opzioni

- React Native Debugger (desktop): console + network + Redux DevTools.
- Expo DevTools Plugins: plugin di debug che comunicano con l’app in dev.
- Altre soluzioni: Reactotron o strumenti equivalenti.

Expo DevTools plugin (Redux)

Install (se usi plugin)

Install

```
npm install @rozenite/redux-devtools-plugin
```

- Plugin per ispezionare lo stato Redux in sviluppo.

Debug: regole pratiche

Per non complicarsi

- Abilita DevTools solo in __DEV__.
- Nomina bene actions e slice (per leggere la timeline).
- Selector piccoli: evita re-render inutili.

Zustand

Perché piace

- Store minimale: stato + azioni in un file.
- Nessun Provider.
- Selettori: aggiorni solo chi usa quel pezzo di stato.

Install

Zustand

Install

```
npm install zustand
```

- Minimale e rapido.

Store (1/2)

State + add

todo.store.ts

```
import { create } from "zustand";

export const useTodoStore = create((set) => ({
  items: [],
  add: (title) =>
    set((s) => ({
      items: [...s.items, { id: Date.now().toString(),
      title, done: false }],
    })),
}));
```

- Definisci stato e azione add.

Store (2/2)

toggle/remove

todo.store.ts

```
import { create } from "zustand";

export const useTodoStore = create((set) => ({
  // ... items, add
  toggle: (id) =>
    set((s) => ({
      items: s.items.map((t) => (t.id === id ? { ...t, done:
       !t.done } : t)),
    })),
  remove: (id) =>
    set((s) => ({ items: s.items.filter((t) => t.id !== id)
    })),
}));
```

- Spezzato per leggibilità e zero overflow.

Uso in UI

Selettori

UI

```
const items = useTodoStore((s) => s.items);
const add = useTodoStore((s) => s.add);

add("Studiare Zustand");
```

- Seleziona solo ciò che ti serve.

Async in Zustand

Action async

users.store.ts

```
export const useUsersStore = create((set) => ({
  items: [],
  loading: false,
  error: null,
  fetchUsers: async () => {
    set({ loading: true, error: null });
    try {
      const res = await
      fetch("https://example.com/api/users");
      const data = await res.json();
      set({ items: data, loading: false });
    } catch (e) {
      set({ error: "Errore di rete", loading: false });
    }
  },
}));
```

- Stesso concetto di thunk, ma nello store.

Redux vs Zustand

Scelta rapida

- Zustand: meno regole, più velocità di sviluppo.
- Redux: più struttura, più standard, più strumenti.
- Team grande o logiche complesse: Redux spesso vince.
- App media/rapida: Zustand spesso basta.

Esercizio guidato

To-do list (2 approcci)

- Stessa UI, 2 implementazioni: Redux e Zustand.
- Funzioni: add, toggle, remove.
- Extra: filtro (tutte/attive/completate).

Esercizio — checklist UI

Cosa costruire

- Input + bottone Aggiungi.
- Lista (FlatList) con riga: titolo + toggle + elimina.
- Contatore: task totali e completati.

Best practice (organizzazione)

Struttura file

- Redux: features/todos/todosSlice.ts + store.ts.
- Zustand: stores/todo.store.ts (per feature).
- Evita store monolitici: separa per dominio.

Best practice (performance)

Re-render

- Selector piccoli e specifici.
- Evita di selezionare oggetti enormi se ti serve un campo.
- Usa memoizzazione solo se c’è un problema reale.

Riepilogo

Messaggio chiave

- Parti da useState (semplice).
- Context+useReducer è un ottimo step intermedio.
- Zustand = minimal; Redux = standard + debug.

Domande di ripasso

Domanda 1 di 5

1. Quando useState è la scelta migliore?

- ☐ Quando lo stato serve solo in un componente o pochi figli
- ☐ Quando vuoi sempre uno store globale
- ☐ Quando hai molte API async
- ☐ Quando vuoi evitare re-render

Domande di ripasso

Domanda 2 di 5

2. Perché usare Context insieme a useReducer?

- ☐ Per evitare prop drilling e centralizzare la logica di update
- ☐ Per rendere l’app più lenta
- ☐ Per sostituire FlatList
- ☐ Per compilare Android

Domande di ripasso

Domanda 3 di 5

3. Quale coppia di pacchetti è tipica per Redux Toolkit in RN?

- ☐ @reduxjs/toolkit + react-redux
- ☐ zustand + axios
- ☐ react-navigation + expo-router
- ☐ react-native-svg + expo-image

Domande di ripasso

Domanda 4 di 5

4. Qual è un vantaggio tipico di Zustand?

- ☐ Non richiede Provider e ha poco boilerplate
- ☐ Funziona solo su iOS
- ☐ Non supporta async
- ☐ Obbliga a usare class components

Domande di ripasso

Domanda 5 di 5

5. A cosa servono i DevTools Redux?

- ☐ A vedere azioni e stato nel tempo per fare debug
- ☐ A cambiare la UI senza re-render
- ☐ A importare immagini
- ☐ A creare componenti nativi