KIELCE UNIVERSITY OF TECHNOLOGY

Mobile Applications – lecture 4

Application State (Local State) and Context

Mateusz Pawełkiewicz

1.10.2025

In React, we have several ways of storing and managing component state. Choosing the right mechanism depends on the complexity of the state and whether we need to share it with many components. The useState hook works best for simple local state in a single component – e.g., single values or uncomplicated state updates. On the other hand, useReducer is recommended when state logic becomes more complex, involves multiple related values, or operations dependent on the previous state. React documentation indicates that useReducer is worth using when the state contains numerous sub-fields or complex transitions are required (e.g., multiple action types) instead of simple value settings.

However, useContext is not used for defining state logic, but for sharing data deep into the component tree without passing it through subsequent props. It is the so-called "savior" of the prop drilling problem – it allows eliminating the manual passing of the same information through every component level. Context works well for global or semi-global values (e.g., application theme, logged-in user, application settings), i.e., data used by many independent parts of the interface. However, it should be remembered that frequent changes in context values can cause multiple components to reload simultaneously, which potentially negatively affects performance – every consumer of the context will react to the change by re-rendering. Therefore, Context API is suitable mainly for relatively rarely changing, global information, and not for fast, frequently mutated state logic.

When to use what? When designing, it is worth following a few rules:

- **Start with useState** – for most local states of a single component, it is the simplest to use. It provides direct state update using a setter function and is easy to understand. When a component needs, for example, to store an input value, a checkbox selection, or modal state, useState is completely sufficient.
- **Switch to useReducer if necessary** – if state logic begins to grow (e.g., state is an object with many fields, updates must be performed conditionally or in response to different action types), it is better to organize the code using a reducer. useReducer enables focusing state update logic in a single reducing function, which improves readability with more complicated sequences of changes. It is also useful when subsequent state values depend on previous ones – avoiding problems with stale previous state, because the reducer processes actions sequentially.
- **Add useContext for sharing state** – when the need arises to pass data to distant components (deeper in the hierarchy), consider using context instead of passing many props along the way. Often useReducer is combined with useContext to build a custom lightweight global state system: the reducer manages logic and updates, and the context provides access to this state and the dispatch function anywhere in the application. Thanks to this, one can avoid prop drilling and easily share, for example, the result of the reducer (global state) and the method to change the state, to which descendant components will refer.

Summarizing, useState maintains simplicity and is suitable for local, isolated states; useReducer handles greater state complexity and sequences of actions; whereas useContext allows sharing state (or functions) between components without passing it through props. Often all these mechanisms cooperate – e.g., we use useReducer to manage state at a high level, and then

useContext to deliver this state and dispatchers lower down, while individual small components may have their own useState for private minor states.

State architecture: UI state vs. server state; avoiding prop drilling

Considering application architecture, it is worth separating **UI state** from **server data state**. Not all state in an application is identical – it has different sources and characteristics. **UI State** is all data that is impermanent and kept only on the client side. In other words, it is **local state**, which usually resets upon refreshing the application and is not stored in a database or on a server. It is usually characterized by synchronization and immediacy – the change occurs immediately in the interface and does not involve network delays or waiting for a response. Examples of UI state are, for example, the current color theme (dark/light mode), selected filters on a list, opening/closing a modal, or internal form validation state. Changes to this type of state are usually triggered directly by user actions (clicks, focus, typing text, etc.) and immediately reflected in the view.

Server State, on the other hand, includes data that comes from an external source – most often from a server via API – and is permanently stored there. The client application must fetch such data asynchronously (e.g., fetch to API) and possibly send changes back to the server to update them. This means that **server state involves delays** (response time, waiting for the network) and uncertainty – we do not know in advance when data will arrive, whether the operation will succeed, or what exactly the value will be (it may change in the meantime). Moreover, data on the server has a **shared nature** – it can be modified by other people or processes independently of our application. Therefore, when working with server state, one must take into account issues such as data refreshing (re-fetch), handling network errors, consistency of local data with the external source, etc. – these are completely different challenges than with purely UI state.

It is key not to treat server state the same as local state – mixing these concepts can lead to errors. Traditional state management libraries (like Redux, MobX, or even context) were designed mainly with UI state in mind (they are great at synchronizing multiple components with a given value), but they do not automatically solve server state problems, such as data caching, retry strategies, or connection loss. Therefore, a currently popular approach is using **specialized tools for managing server data state**. Examples are libraries like **React Query (TanStack Query), SWR, or RTK Query**, which were created to facilitate handling asynchronous data from APIs (REST or GraphQL). Such tools automate many tasks: tracking "loading/error/success" state, refreshing and invalidating cache, fetching only unique requests (deduplication), optimistic updates, etc. – things that in pure React we would have to implement manually. According to experts, React Query has grown to the rank of one of the most popular and powerful tools for managing server state in React applications. **In short: server state is worth separating and managing with different techniques than UI state – one can think of it more as data than application state in the classical sense.**

The second architectural issue is avoiding so-called **prop drilling**, i.e., digging through with data through subsequent layers of components. Prop drilling occurs when we have to pass data from component A to a deeply nested component Z, and along the way components B, C, D... which do not directly need this data, must pass it further through their props. This

leads to an increase in code volume, loosening of encapsulation (components must know about props for great-grandchildren), and hinders refactoring of the component structure. Ideally, every component should know only what is needed for it.

How to avoid prop drilling? If data must be available deep in the component tree, the best built-in solution is precisely the **Context API**. Context allows defining a **global value** (e.g., current user, application settings, theme) and making it available to all descendants of a given Provider, without the need to pass it manually through subsequent props. Components can retrieve such a value anywhere using useContext(…). For example, an application theme context can provide all buttons with information whether they should render in dark or light style, regardless of how deep in the tree hierarchy these buttons are located. Another solution (if we do not want to use context) is sometimes **lifting state up** to a common ancestor of components needing data – however, this solves the problem only partially and at shallow depth. When we need to share state globally or at great depth, it is better to reach for context or a dedicated store.

Context API vs dedicated store: In small applications, context often suffices as a simple form of global store. Combined with useReducer, we can even replace Redux with it (at a smaller scale). However, it must be remembered that context does not possess advanced tools such as middleware, developer tools (devtools), or time-travel debugging. With very extensive global state or many types of global data, it is worth considering introducing a state management library (about which in the next point). However, prop drilling as a problem should be a signal: if we catch ourselves passing the same props through more than 2–3 levels, it is worth switching to context for this data.

Lightweight store (e.g., Zustand) – comparison with Redux Toolkit

Traditionally, Redux was used for global state in React, which however required a lot of implementation at the start (actions, reducers, store, etc.). **Redux Toolkit (RTK)** is a more modern, officially recommended way of using Redux – it simplifies its configuration, adds default settings, and integrates tools (e.g., Immer, RTK Query) facilitating global state management. Despite this, Redux (even in RTK form) can be too heavy a solution for simpler applications. In recent years, "lightweight" state libraries with much lower code overhead have gained popularity. One of them is **Zustand** – a very light store based on hooks, offering global state without the need to declare reducers or actions. Let's look generally at how **Zustand vs Redux Toolkit** compares in several respects:

- **API and simplicity: Zustand** offers a minimalist API – state definition comes down to calling the create() function with an object containing state fields and functions for modifying them. There is no concept of actions, action types, or reducers here – we update state **directly** through setter functions. This means very little boilerplate and fast implementation even in a small project. For comparison, **Redux Toolkit** still requires a certain structure: we define a slice (e.g., using createSlice), actions and reducer are generated internally, and then we configure the store using configureStore. RTK significantly reduces the amount of code compared to "pure" Redux, but it is still more extensive than Zustand. Beginners may find that it has a steeper learning curve

and imposes a certain code organization pattern (which can be a plus in large projects, but a minus in micro-applications).

- **Structure and scaling: Redux Toolkit** wins when it comes to structuring a large application. When we have an extensive team, many state modules, a need to maintain conventions – RTK imposes a consistent approach (slices, large community support, compatibility with middleware like Redux Saga/Thunk). Additionally, Redux DevTools allow debugging every step of state changes, which is valuable in complex cases. **Zustand**, in turn, gives greater flexibility but less structure – the developer decides how to divide the store (e.g., one can create many small Zustand stores for various independent application functions) or keep everything in one. At a larger scale, this can lead to some inconsistency if we do not impose a convention on ourselves. However, for ~medium-sized projects, Zustand is often sufficient and valued for speed of implementation.
- **Performance:** Both libraries are performant, but the characteristic is slightly different. **Zustand** is very light and fast – it has minimal overhead and limits itself to subscribing to selected fragments of state via hooks. Updates in Zustand cause re-rendering only of components using a given piece of state, which with appropriate use ensures great performance. In practice, Zustand often turns out faster in simple scenarios due to the lack of an additional layer (like context/connector). **Redux Toolkit**, on the other hand, has more "mechanisms under the hood" – although it is optimized, it has slightly higher overhead due to handling middleware, immutability checks (in dev mode), etc. Despite this, in large applications performance differences are usually negligible – code organization becomes more important. In Redux one can also precisely prevent redundant renders via selectors and `React.memo`, whereas Zustand gives this somewhat by default, because every `useStore` hook can retrieve only the needed fragment of state.
- **Ecosystem and possibilities: Redux Toolkit** benefits from the entire Redux ecosystem. That is, we have access to a rich range of middleware, integration with other tools, as well as a built-in solution for fetching data from the server – **RTK Query**, which significantly facilitates work with APIs in REST/GraphQL style. Redux is also well known – it is easier to find developers with Redux experience, more tutorials, community support. **Zustand** is newer and has a smaller community, but is gaining popularity. It possesses several extensions (e.g., middleware for state persistence in localStorage, devtools plugin, or integration with Immer), however, the scale of the ecosystem is smaller. In practice, for smaller projects we do not need extensive middleware – simple Zustand code is completely sufficient.

Summary of choice: There is no clear winner – everything depends on the project context. If you are creating a small or medium application (several dozen components, uncomplicated global logic) or a prototype, **Zustand** will provide a fast and pleasant solution with minimal effort. However, in large, enterprise applications with extensive state, division into modules, and a large team, **Redux Toolkit** may prove to be a better choice thanks to its structure, tools, and predictability. An approach is often adopted: start with a simple solution, and only when it turns out insufficient – switch to a heavier one. As one of the authors put it: Zustand covers ~80% of application cases ("simple, fast, pleasant"), and Redux Toolkit is best where the application is truly complex and requires an extensive set of tools. It is therefore worth

selecting the tool for the scale of the problem, and not out of habit – both options are mature and have their place in 2025.

Performance and optimizations: memo, useCallback, useMemo, batching

Along with the increase in interface complexity, we must take care of performance – React fortunately offers several mechanisms that help minimize unnecessary component rendering and costly calculations. Before we discuss them, it is worth emphasizing: React has long been performing many optimizations under the hood, and from version 18 it gained another one – automatic batching of state updates.

Batching (grouping) of updates: In older React versions, every state change triggered outside a built-in event handler caused a separate render. From **React 18**, state updates are **automatically grouped** into one transaction regardless of the source (promises, timeouts, custom events, etc.). This means that if we call setState several times in a short time, React will wait and render components only once, taking into account all changes at once, instead of rendering after each change separately. For example, previously calling two setStates in an asynchronous callback resulted in two renders, and in React 18 it results in one. This improvement significantly reduces the number of renders, which translates into smoother interface operation (less work for the browser's main thread). Of course, React takes care not to group unrelated user events – e.g., two separate button clicks will still be handled separately so as not to change application logic. In most cases, batching happens automatically and does not affect application code except that it works faster. As developers, we can almost not notice it, though it is worth knowing that it exists. Only in rare advanced situations do we need to control it – e.g., React provides flushSync() to manually force synchronization (when we want a given state change to be immediately visible in the DOM, e.g., before measuring element height), or startTransition() to mark less important updates that can wait (e.g., updating a search result list). **Summarizing: React 18 itself takes care of grouping state updates, thanks to which applications noticeably gain in performance without additional effort.**

Avoiding unnecessary renders – component memoization: By default, React renders a component again every time its parent renders (unless the component depends on no props or state, then it might be skipped). In large lists or complex UI trees, this can mean many unnecessary calculations if data has not changed. Here React.memo() comes to aid – a higher-order component (HOC) or mechanism for functional components, which **memorizes the result of a given component's render** and skips re-rendering if its props have not changed. This works by **shallow comparison** of previous and new props. React.memo is worth using around components that: (a) are costly to render (e.g., contain complicated lists, tables, charts) or (b) render again very often, even though their data rarely changes. For example, a card with a chart or complex data display logic should render again only when data actually changes – wrapping it in React.memo will ensure this. However, it makes no sense to memoize simple, cheap components (like a single button or icon), because the cost of comparing props may exceed the gain from skipping the render. One should treat component memoization like a precise tool (scalpel), and not a hammer for everything. In other words – we optimize those places that are bottlenecks.

In the context of component memoization, one must mention **prop reference stability**. React.memo compares new and old props referentially, which means that e.g., two different functions even with the same implementation will be treated as different (because they have a different address in memory). Therefore, if our component receives functions or objects in props that change with every render of the parent, then React.memo won't help anyway – because the prop will be "new" every time. To remedy this, we use useCallback and useMemo to stabilize these values.

- useCallback(fn, deps) returns a reference to the same function fn between renders, as long as declared dependencies have not changed. In other words, it **memorizes the function** – thanks to this, e.g., we can pass a callback (onClick, onChange etc.) to a child which will not change with every parent render, which will prevent unnecessary re-rendering of the child (if the child is memoized). For example, with a list of elements where each element has a "remove" button calling onRemove(id): without useCallback, the onRemove function created in the parent is a new function-object every time, causing the memoized child to refresh anyway; with useCallback, the parent provides still the same function reference between renders (as long as e.g. the list does not change), so the child remains effectively pure and does not render again.
- useMemo(calcFn, deps) in turn **memorizes the result of the calcFn function call** as long as dependencies do not change. We use it to avoid expensive calculations with every render. If e.g. we must filter a large array of data or perform a complicated calculation based on props, we can wrap this in useMemo – then the result of the previous calculation will be returned from cache as long as the relevant input data is identical, and the calcFn function will not be called again without need. This can drastically improve performance in scenarios like table generation, filtering, sorting, formatting large datasets etc., especially when the component renders often but data changes rarely.

Generally speaking, useMemo and useCallback are tools helping to optimize re-rendering. They can be summarized like this: they serve to **reduce work performed during a render** and **reduce the number of component renders themselves**. Thanks to them we avoid a situation where with every UI refresh we calculate the same values anew or create the same functions. However, they should be used with caution – let's remember that using a memoizing hook itself also has a certain cost (one must perform dependency comparison, store the value etc.). Therefore, it makes no sense to preventively memoize everything; it is best to target bottlenecks (e.g. via Profile in React DevTools, which will show which components render often and how long it takes) and apply optimizations there.

Best performance practices in React: Let's summarize a few tips:

- **Update batch** – React 18 will do this for Us automatically in most cases. If you have many related state changes in one event, execute them together (e.g. call setState several times in one handler instead of in separate ones, and React will group them into one render). Avoid however forcing synchronous renders without need (occasionally flushSync may be useful, but abusing it negates gains from batching).
- **Minimize render dependency depth** – if some component sends a function or object to a child that changes every render, consider useCallback/useMemo so the child does

not see a constantly changing prop. The fewer "dynamic props", the better for memoization possibilities.

- **Apply React.memo to heavy components** – presentation components displaying large data sets, lists, tables, charts, etc. are worth wrapping in React.memo not to redraw them if props have not changed. Remember however about prop stability (as above).
- **Avoid costly calculations during render** – if in a component function you see expensive operations (loops over large arrays, sorts, data parsing), consider moving them to useMemo. Thanks to this, this work will be performed only when source data actually changes, and not with every render separately.
- **Use one state for dependent groups** – if you have several state variables that always change together, consider storing them in one state object or using a reducer to avoid an avalanche of many setStates (although in React 18 they would be batched anyway, but logically it is sometimes simpler).
- **Test and profile** – tools such as React DevTools Profiler will help understand where the application actually spends time. Sometimes optimizations in the wrong place can even worsen performance (e.g. unnecessary use of useMemo, which itself has a cost, can lengthen render time if calculation was trivial).

Finally, it is worth mentioning the latest trends: the React team is working on automating many of these optimizations. The so-called **React Compiler** has appeared (in beta phase for React 19), which can automatically transform code at compile time so that it includes React.memo, useMemo, and useCallback where it is safe. In practice, this means that in the future we can write code without manually wrapping everything in memo, and the compiler will do the optimization for us. Already now in Meta's production environment (e.g. Instagram) this solution is being tested and looks promising – developers noted an increase in productivity (less time thinking about optimization) and fewer errors related to dependencies, with a simultaneous performance gain because the compiler can memoize all code by default. At this moment (2025) however, most projects still use manual optimization techniques, so it is worth understanding memo, useMemo, useCallback well. Writing new code, let's remember: **readability above all** – do not "premature optimize", unless we know (or suspect) that a given part of the UI will be performance-sensitive.

Demo: global AuthContext + theme switch (light/dark)

Finally, let's see a practical example of using context for global state management. Many applications need e.g. to store information about the currently logged-in user (auth) and enable global theme change (light/dark mode). Both these things are good candidates for context: they should be available in many places of the application (e.g. login state affects the display of the entire menu, and the theme affects the style of all components), and at the same time they change relatively rarely (login/logout is a rare event, theme change too). We will create two contexts: AuthContext and ThemeContext, together with corresponding providers. AuthContext will store e.g. a user object or information whether the user is logged in, and login and logout methods to change this state. ThemeContext will store the current theme ('light' or 'dark') and the toggleTheme function to switch it. Next, we will show how to use these contexts in components, e.g. in navigation.

Implementation steps:

1. **Creation of contexts and providers:** Using createContext we create context objects. We define Provider components that will wrap the entire application. In them, we use useState (or useReducer for more extensive logic) to manage state and pass in the context value both the current state and functions to change it.
2. **Wrapping the application in Providers:** In the main file (e.g. App.js or index.js) we surround the main application component with our providers: <AuthProvider> and <ThemeProvider>. Thanks to this, any component inside will have access to context values via the useContext hook.
3. **Using context in components:** Inside any component that needs e.g. information about the logged-in user or theme, we call const { user, logout } = useContext(AuthContext) or const { theme, toggleTheme } = useContext(ThemeContext) to extract needed values/functions. We can then use this data – e.g. display a welcome with the user name, conditionally render a "Log in/Log out" button, and also change the CSS class or styles depending on the theme.
4. **Example component (Navbar):** Let's assume we have a navigation component that wants to show different options in the menu depending on whether the user is logged in, and place a button to change the theme. It uses both contexts simultaneously.

Below is an example implementation in JSX code illustrating the steps above (for simplicity we omit details like actual logging in – we assume that login sets a fictitious user object):

```
import React, { createContext, useState, useContext } from 'react';

// 1. Create contexts
const AuthContext = createContext(null);
const ThemeContext = createContext('light');

// 1. Define Provider for Auth
function AuthProvider({ children }) {
  const [user, setUser] = useState(null);
  const login = (userData) => setUser(userData);
  const logout = () => setUser(null);
  const authValue = { user, login, logout };
  return (
    <AuthContext.Provider value={authValue}>
      {children}
    </AuthContext.Provider>
  );
}

// 1. Define Provider for Theme
function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');
  const toggleTheme = () =>
    setTheme((prev) => (prev === 'light' ? 'dark' : 'light'));
  const themeValue = { theme, toggleTheme };
  return (
    <ThemeContext.Provider value={themeValue}>
      {children}
```

```
    </ThemeContext.Provider>
  );
}

// 2. Wrap application in providers
function App() {
  return (
    <AuthProvider>
      <ThemeProvider>
        <MainApplicationContent /> {/* rest of the application */}
      </ThemeProvider>
    </AuthProvider>
  );
}

// 3. Example of context consumption in a component
function Navbar() {
  const { user, logout } = useContext(AuthContext);
  const { theme, toggleTheme } = useContext(ThemeContext);
  return (
    <header className={`header-${theme}`}>
      <nav>
        {user ? (
          <>
            <span>Welcome, {user.name}!</span>
            <button onClick={logout}>Log out</button>
          </>
        ) : (
          <button onClick={() => alert('Go to login...')}>Log in</button>
        )}
      </nav>
      <button onClick={toggleTheme}>
        {theme === 'light' ? 'Dark mode' : 'Light mode'}
      </button>
    </header>
  );
}
```

In the code above, the `<AuthProvider>` component manages the `user` state (stores e.g. an object with logged-in user data or null when not logged in) and makes it available along with `login`/`logout` functions via context. `<ThemeProvider>` makes available the current theme and a function to change it. The entire application is wrapped in these providers, so e.g. our `<Navbar>` or another component does not have to receive this data via props – it uses `useContext` directly.

Such an approach eliminates prop drilling – e.g. we don't have to pass user information to every component in the tree; it suffices to retrieve it from context once where we need it (e.g. Navbar, User Panel, etc.). Similarly with the theme – any component (button, page background) can check the current theme via `useContext(ThemeContext)` and adjust the style. Importantly, a context change (e.g. calling `toggleTheme` changing `theme` to `'dark'`) automatically re-renders all components using `ThemeContext` – in our case immediately the entire application will switch to the new theme without manually passing this change.

At the same time, separating into two separate Auth and Theme contexts is a conscious architectural measure: changing the theme does not cause e.g. a re-render of components using only AuthContext and vice versa. If we threw everything into one global context, even unrelated changes could unnecessarily refresh many parts of the UI. Therefore, it is worth creating separate contexts for independent state domains. React allows nesting multiple providers without problems, which we see in the example – the nesting order does not matter much, it is important to cover the appropriate range of components with these providers (here: the entire application).

Finally, let's add that the Context API is great for this type of global settings and simple application state. When the application grows, we can expand it based on a similar pattern e.g. using useReducer in AuthProvider (to handle more actions, e.g. token refresh, profile update, etc.) or reach for previously discussed solutions like Redux/Zustand if global state became very extensive. Nevertheless, for the needs of this demo, it is visible that using the latest React APIs we can implement global authorization state and application theme in a simple and readable way, consistent with today's best practices.

Literature:

1. https://reactnavigation.org/docs/getting-started/ (Access Date: 1.10.2025) - Official React Navigation documentation (main page).
2. https://reactnavigation.org/docs/typescript/ (Access Date: 1.10.2025) - Official guide for integrating React Navigation with TypeScript.
3. https://reactnavigation.org/docs/auth-flow/ (Access Date: 1.10.2025) - Key documentation describing the recommended authentication flow pattern.
4. https://reactnavigation.org/docs/deep-linking/ (Access Date: 1.10.2025) - Official guide for configuring Deep Links.
5. https://reactnavigation.org/docs/navigating/ (Access Date: 1.10.2025) - Documentation for basic operations (navigate, push, goBack).
6. https://reactnavigation.org/docs/params/ (Access Date: 1.10.2025) - Documentation on passing and receiving parameters (route.params).
7. https://reactnavigation.org/docs/hooks/ (Access Date: 1.10.2025) - Documentation for useNavigation and useRoute hooks.
8. https://reactnavigation.org/docs/native-stack-navigator/ (Access Date: 1.10.2025) - Documentation for Native Stack Navigator (recommended for performance).
9. https://reactnavigation.org/docs/bottom-tab-navigator/ (Access Date: 1.10.2025) - Documentation for Bottom Tab Navigator.
10. https://docs.expo.dev/routing/linking/ (Access Date: 1.10.2025) - Expo guide regarding linking configuration (including expo-linking).