POLITECHNIKA ŚWIĘTOKRZYSKA

Aplikacje mobilne – wykład 4

Stan aplikacji (stan lokalny) i kontekst

Mateusz Pawełkiewicz

1.10.2025

W React mamy kilka sposobów przechowywania i zarządzania stanem komponentów. Dobór właściwego mechanizmu zależy od złożoności stanu oraz tego, czy musimy go udostępniać wielu komponentom. Hook **useState** najlepiej sprawdza się do **prostego stanu lokalnego** w pojedynczym komponencie – np. pojedyncze wartości lub nieskomplikowane aktualizacje stanu. Z kolei **useReducer** jest zalecany, gdy logika stanu staje się **bardziej złożona**, obejmuje wiele powiązanych wartości lub operacje zależne od poprzedniego stanu. Dokumentacja Reacta wskazuje, że useReducer warto użyć, gdy stan zawiera liczne pod-pola lub wymagane są złożone tranzycje (np. wiele typów akcji) zamiast prostych ustawień wartości.

Natomiast **useContext** nie służy do definiowania logiki stanu, ale do **udostępniania danych** w głąb drzewa komponentów bez przekazywania ich przez kolejne propsy. Jest to tzw. *"savior"* problemu prop drilling – pozwala **wyeliminować ręczne przekazywanie** tej samej informacji przez każdy poziom komponentów. Context dobrze się sprawdza dla **wartości globalnych lub pół-globalnych** (np. motyw aplikacji, zalogowany użytkownik, ustawienia aplikacji), czyli danych, z których korzysta wiele niezależnych części interfejsu. Należy jednak pamiętać, że częste zmiany wartości kontekstu mogą spowodować **przeładowanie wielu komponentów jednocześnie**, co potencjalnie wpływa negatywnie na wydajność – każdy konsument kontekstu zareaguje na zmianę poprzez ponowny render. Dlatego Context API nadaje się głównie do stosunkowo rzadko zmieniających się, globalnych informacji, a nie szybkiej, często mutowanej logiki stanu.

Kiedy czego użyć? Przy projektowaniu warto kierować się kilkoma zasadami:

- **Zacznij od useState** – dla większości lokalnych stanów pojedynczego komponentu jest on **najprostszy w użyciu**. Zapewnia bezpośrednią aktualizację stanu za pomocą funkcji setter i jest łatwy do zrozumienia. Gdy komponent potrzebuje np. przechować wartość inputu, zaznaczenie checkboxa czy stan modala, useState w zupełności wystarczy.
- **Przejdź na useReducer w razie potrzeby** – jeśli logika stanu zaczyna się rozrastać (np. stan jest obiektem z wieloma polami, aktualizacje muszą być wykonywane warunkowo lub w odpowiedzi na różne typy akcji), **lepiej zorganizować kod za pomocą reducera**. useReducer umożliwia skupienie logiki aktualizacji stanu w jednej funkcji redukującej, co poprawia czytelność przy bardziej skomplikowanych sekwencjach zmian. Jest to także przydatne, gdy kolejne wartości stanu zależą od poprzednich – unika się wtedy problemów z nieaktualnym stanem poprzednim, bo reducer przetwarza akcje sekwencyjnie.
- **Dodaj useContext do udostępniania stanu** – gdy pojawia się potrzeba przekazania danych do odległych komponentów (głębiej w hierarchii), rozważ użycie kontekstu zamiast przekazywania wielu propsów po drodze. Często łączy się **useReducer z useContext** w celu zbudowania **własnego lekkiego systemu globalnego stanu**: reducer zarządza logiką i aktualizacją, a kontekst zapewnia dostęp do tego stanu oraz funkcji dispatch w dowolnym miejscu aplikacji. Dzięki temu można uniknąć prop drillingu i łatwo udostępnić np. wynik reducer'a (stan globalny) oraz metodę do zmiany stanu, do której odwołają się komponenty potomne.

Podsumowując, **useState** utrzymuje prostotę i nadaje się do lokalnych, izolowanych stanów; **useReducer** radzi sobie z większą **złożonością** stanu i sekwencjami akcji; natomiast **useContext**

pozwala **dzielić się stanem** (lub funkcjami) między komponentami bez przekazywania go przez propsy. Często wszystkie te mechanizmy współpracują – np. używamy useReducer do zarządzania stanem na wysokim poziomie, a następnie useContext do dostarczenia tego stanu i dispatcherów niżej, podczas gdy poszczególne drobne komponenty mogą mieć własne useState do prywatnych drobnych stanów.

Architektura stanu: stan UI vs stan serwera; unikanie *prop drilling*

Rozważając architekturę aplikacji, warto **oddzielić stan UI od stanu danych z serwera**. Nie cały stan w aplikacji jest jednakowy – ma różne źródła i charakterystyki. **Stan UI (User Interface state)** to wszystkie dane, które są **nietrwałe i trzymane tylko po stronie klienta**. Innymi słowy, jest to stan lokalny, który zwykle **resetuje się przy odświeżeniu aplikacji** i nie jest przechowywany w bazie danych ani na serwerze. Cechuje go zazwyczaj **synchronizacja i natychmiastowość** – zmiana następuje od razu w interfejsie i nie wiąże się z opóźnieniami sieciowymi czy oczekiwaniem na odpowiedź. Przykłady stanu UI to np. aktualny motyw kolorystyczny (dark/light mode), zaznaczone filtry na liście, otwarcie/zamknięcie modala czy wewnętrzny stan walidacji formularza. Zmiany tego typu stanu są zwykle wyzwalane bezpośrednio przez **akcje użytkownika** (kliknięcia, focus, wpisywanie tekstu itp.) i od razu odzwierciedlane w widoku.

Stan serwerowy (Server State) natomiast obejmuje dane, które **pochodzą z zewnętrznego źródła** – najczęściej z serwera via API – i są tam trwale przechowywane. Aplikacja kliencka musi takie dane **pobrać asynchronicznie** (np. fetch do API) i ewentualnie wysłać z powrotem zmiany do serwera, aby je zaktualizować. To oznacza, że stan serwerowy wiąże się z opóźnieniami (czas odpowiedzi, oczekiwanie na sieć) oraz **niepewnością** – nie wiemy z góry, kiedy dane przyjdą, czy operacja się powiedzie ani jaka będzie dokładnie wartość (może się zmienić w międzyczasie). Co więcej, dane na serwerze mają **współdzieloną naturę** – mogą zostać zmodyfikowane przez inne osoby lub procesy niezależnie od naszej aplikacji. Dlatego przy pracy ze stanem z serwera trzeba brać pod uwagę kwestie takie jak odświeżanie danych (re-fetch), obsługa błędów sieci, spójność danych lokalnych z zewnętrznym źródłem itd. – to zupełnie inne wyzwania niż przy stanie czysto UI.

Kluczowe jest, aby **nie traktować stanu serwerowego tak samo jak lokalnego** – mieszanie tych pojęć może prowadzić do błędów. Tradycyjne biblioteki do zarządzania stanem (jak Redux, MobX czy nawet kontekst) zostały zaprojektowane głównie z myślą o stanie UI (są świetne w synchronizowaniu wielu komponentów z daną wartością), ale **nie rozwiązują automatycznie problemów stanu serwerowego**, takich jak cache'owanie danych, strategie ponawiania zapytań czy utrata połączenia. Dlatego obecnie popularnym podejściem jest korzystanie ze specjalizowanych narzędzi do zarządzania stanem danych z serwera. Przykładem są biblioteki takie jak **React Query (TanStack Query)**, **SWR** czy **RTK Query**, które zostały stworzone, aby ułatwić obsługę asynchronicznych danych z API (REST lub GraphQL). Takie narzędzia automatyzują wiele zadań: śledzenie stanu "loading/error/success", odświeżanie i inwalidację cache, pobieranie tylko unikalnych zapytań (*deduplication*), optymistyczne aktualizacje itp. – rzeczy, które w czystym React musielibyśmy implementować ręcznie. Według ekspertów, **React Query** urosło do rangi jednego z

najpopularniejszych i najpotężniejszych narzędzi do zarządzania stanem serwerowym w aplikacjach React. W skrócie: **stan serwerowy warto wydzielić i zarządzać nim innymi technikami** niż stanem UI – można myśleć o nim bardziej jak o *danych* niż *stanie* aplikacji w klasycznym sensie.

Drugim zagadnieniem architektonicznym jest unikanie tzw. **prop drillingu**, czyli przekopywania się z danymi przez kolejne warstwy komponentów. Prop drilling ma miejsce, gdy **musimy przekazać dane z komponentu A do daleko zagnieżdżonego komponentu Z**, a po drodze komponenty B, C, D... które tych danych bezpośrednio nie potrzebują, muszą przekazywać je dalej przez swoje propsy. Prowadzi to do wzrostu ilości kodu, rozluźnienia enkapsulacji (komponenty muszą wiedzieć o propsach dla prawnuków) i utrudnia refaktoryzację struktury komponentów. Idealnie, każdy komponent powinien wiedzieć tylko to, co jest mu potrzebne.

Jak unikać prop drillingu? Jeśli dane muszą być dostępne głęboko w drzewie komponentów, najlepszym wbudowanym rozwiązaniem jest właśnie **Context API**. Kontekst pozwala zdefiniować wartość globalną (np. aktualny użytkownik, ustawienia aplikacji, motyw) i **udostępnić ją wszystkim potomkom** danego Providera, bez potrzeby przekazywania ręcznie przez kolejne propsy. Komponenty mogą pobrać taką wartość w dowolnym miejscu za pomocą `useContext(...)`. Przykładowo, kontekst motywu aplikacji może dostarczać wszystkim przyciskom informację, czy mają renderować się w stylu **dark** czy **light**, niezależnie od tego jak głęboko w hierarchii drzewa te przyciski się znajdują. Innym rozwiązaniem (jeśli nie chcemy używać kontekstu) bywa **lifting state up** do wspólnego przodka komponentów potrzebujących danych – jednak to rozwiązuje problem tylko częściowo i na niewielkiej głębokości. Gdy potrzebujemy dzielić stan globalnie lub na dużej głębokości, lepiej sięgnąć po kontekst lub dedykowany store.

Context API vs dedykowany store: W małych aplikacjach kontekst często wystarcza jako prosta forma globalnego store'u. W połączeniu z `useReducer` możemy nim zastąpić nawet Redux (przy mniejszej skali). Trzeba jednak pamiętać, że **kontekst nie posiada zaawansowanych narzędzi** takich jak middleware, narzędzia deweloperskie (devtools) czy time-travel debugging. Przy bardzo rozbudowanym stanie globalnym lub wielu rodzajach danych globalnych warto rozważyć wprowadzenie biblioteki do zarządzania stanem (o czym w następnym punkcie). Natomiast **prop drilling** jako problem powinien być sygnałem: jeśli łapiemy się na przekazywaniu tych samych propsów przez więcej niż 2–3 poziomy, warto przełączyć się na kontekst dla tych danych.

Lekki store (np. Zustand) – porównanie z Redux Toolkit

Tradycyjnie do globalnego stanu w React używano **Reduxa**, który jednak wymagał sporo implementacji na początek (akcje, reducery, store, itp.). **Redux Toolkit (RTK)** to nowocześniejszy, oficjalnie zalecany sposób korzystania z Redux – upraszcza jego konfigurację, dodaje domyślne ustawienia i integruje narzędzia (np. Immer, RTK Query) ułatwiając zarządzanie stanem globalnym. Mimo to, Redux (nawet w formie RTK) bywa dla prostszych aplikacji rozwiązaniem zbyt ciężkim. W ostatnich latach zyskały popularność **"lekkie" biblioteki stanowe** o dużo mniejszym narzucie kodowym. Jedną z nich jest **Zustand** – bardzo lekki store oparty na hookach, oferujący globalny stan bez potrzeby deklaracji

reducerów czy akcj. Przyjrzyjmy się ogólnie, jak wypada **Zustand vs Redux Toolkit** pod kilkoma względami:

- **API i prostota: Zustand** oferuje **minimalistyczne API** – definicja stanu sprowadza się do wywołania funkcji `create()` z obiektem zawierającym pola stanu oraz funkcje do ich modyfikacji. Nie ma tu konceptu akcji, typów akcji ani reducerów – aktualizujemy stan bezpośrednio poprzez funkcje setterów. To oznacza **bardzo mało boilerplate** i szybkie wdrożenie nawet w małym projekcie. Dla porównania **Redux Toolkit** nadal wymaga pewnej struktury: definiujemy slice (np. za pomocą `createSlice`), akcje i reducer są generowane wewnątrz, a potem konfigurujemy store przy użyciu `configureStore`. RTK znacząco zmniejsza ilość kodu w porównaniu do "czystego" Reduxa, ale **wciąż jest bardziej rozbudowany** niż Zustand. Początkujący mogą uznać, że ma on większą krzywą uczenia się i narzuca pewien wzorzec organizacji kodu (co bywa plusem w dużych projektach, ale minusem w mikro-aplikacjach).
- **Struktura i skalowanie: Redux Toolkit** wygrywa, jeśli chodzi o **ustrukturyzowanie dużej aplikacji**. Gdy mamy rozbudowany zespół, wiele modułów stanu, potrzebę utrzymania konwencji – RTK narzuca spójne podejście (slices, duże wsparcie community, kompatybilność z middleware jak Redux Saga/Thunk). Dodatkowo Redux DevTools pozwalają debugować każdy krok zmian stanu, co jest cenne w złożonych przypadkach. **Zustand** z kolei daje większą **elastyczność**, ale mniej struktury – deweloper sam decyduje jak podzielić store (np. można tworzyć wiele małych store'ów zustandowych dla różnych niezależnych funkcji aplikacji) lub trzymać wszystko w jednym. Przy większej skali może to prowadzić do pewnej niespójności, jeśli nie narzucimy sobie konwencji. Jednak do ~**średniej wielkości projektów** Zustand jest często wystarczający i ceniony za **szybkość implementacji**.
- **Wydajność:** Obie biblioteki są wydajne, ale charakterystyka jest nieco inna. **Zustand jest bardzo lekki i szybki** – ma minimalny narzut i ogranicza się do subskrybowania wybranych fragmentów stanu poprzez hooki. Aktualizacje w Zustand powodują **przerender tylko komponentów korzystających z danego kawałka stanu**, co przy odpowiednim użyciu zapewnia świetną wydajność. W praktyce Zustand często okazuje się **szybszy** w prostych scenariuszach ze względu na brak dodatkowej warstwy (typu kontekst/connector). **Redux Toolkit** natomiast ma więcej „mechanizmów pod maską" – choć jest zoptymalizowany, to posiada nieco większy narzut przez obsługę middleware, immutability checków (w trybie dev) itp.. Mimo to, w dużych aplikacjach różnice wydajnościowe są zwykle pomijalne – ważniejsza staje się organizacja kodu. W Redux można też precyzyjnie zapobiegać nadmiarowym renderom poprzez selektory i `React.memo`, natomiast Zustand daje to niejako domyślnie, bo każdy hook `useStore` może pobierać tylko potrzebny fragment stanu.
- **Ekosystem i możliwości: Redux Toolkit** korzysta z całego ekosystemu Redux. To znaczy, mamy dostęp do bogatej gamy *middleware*, integracji z innymi narzędziami, a także wbudowane rozwiązanie do pobierania danych z serwera – **RTK Query**, które znacząco ułatwia pracę z API w stylu REST/GraphQL. Redux jest też dobrze znany – łatwiej znaleźć developerów z doświadczeniem Redux, więcej tutoriali, wsparcia społeczności. **Zustand** jest nowszy i ma mniejszą społeczność, ale zdobywa popularność. Posiada kilka rozszerzeń (np. middleware do persystencji stanu w localStorage, devtools plugin, czy integrację z Immer), jednak skala ekosystemu jest

mniejsza. W praktyce do mniejszych projektów nie potrzebujemy rozbudowanych middleware – prostszy kod Zustand w zupełności wystarcza.

Podsumowanie wyboru: Nie ma jednoznacznego zwycięzcy – wszystko zależy od kontekstu projektu. Jeżeli tworzysz **niewielką lub średnią aplikację** (kilkadziesiąt komponentów, nieskomplikowana logika globalna) albo prototyp, **Zustand** dostarczy szybkie i przyjemne rozwiązanie z minimalnym nakładem pracy. Natomiast w **dużych, enterprise'owych aplikacjach** z rozbudowanym stanem, podziałem na moduły i dużym zespołem, **Redux Toolkit** może okazać się lepszym wyborem dzięki swojej strukturze, narzędziom i przewidywalności. Często przyjmuje się podejście: *zacznij od prostego rozwiązania*, i dopiero gdy okaże się niewystarczające – przejdź na cięższe. Jak ujął to jeden z autorów: **Zustand pokrywa ~80% przypadków** aplikacji ("prosty, szybki, przyjemny"), a **Redux Toolkit jest najlepszy tam, gdzie aplikacja jest naprawdę złożona i wymaga rozbudowanego zestawu narzędzi**. Warto więc dobierać narzędzie do skali problemu, a nie z przyzwyczajenia – obie opcje są dojrzałe i mają swoje miejsce w 2025 roku.

Wydajność i optymalizacje: memo, useCallback, useMemo, batchowanie

Wraz ze wzrostem skomplikowania interfejsu musimy dbać o wydajność – React na szczęście oferuje kilka mechanizmów, które pomagają **minimalizować niepotrzebne renderowanie** komponentów i kosztowne obliczenia. Zanim je omówimy, warto podkreślić: React już od dawna wykonuje wiele optymalizacji pod maską, a od wersji 18 zyskał kolejną – **automatyczne batchowanie aktualizacji stanu**.

Batchowanie (grupowanie) aktualizacji: W starszych wersjach React każda zmiana stanu wywołana poza wbudowanym event handlerem powodowała osobny render. Od **React 18** aktualizacje stanu są **automatycznie grupowane** w jedną transakcję niezależnie od źródła (promisy, setTimeouty, wydarzenia niestandardowe itd.). Oznacza to, że jeśli w krótkim czasie wywołamy kilka razy setState, React poczeka i zrenderuje komponenty **tylko raz**, z uwzględnieniem wszystkich zmian naraz, zamiast renderować po każdej zmianie osobno. Przykładowo, dawniej wywołanie dwóch setState w callbacku asynchronicznym skutkowało dwoma renderami, a w React 18 skutkuje **jednym**. To usprawnienie znacząco redukuje liczbę renderów, co przekłada się na płynniejsze działanie interfejsu (mniej pracy dla wątku głównego przeglądarki). Oczywiście React dba przy tym, by nie zgrupować *niepowiązanych* zdarzeń użytkownika – np. dwa osobne kliknięcia przycisku nadal będą obsłużone osobno, aby nie zmieniać logiki aplikacji. W większości przypadków **batchowanie dzieje się automatycznie** i nie wpływa na kod aplikacji poza tym, że działa szybciej. Jako deweloperzy możemy go niemal nie zauważać, choć warto wiedzieć, że istnieje. Tylko w rzadkich sytuacjach zaawansowanych potrzebujemy go kontrolować – np. React udostępnia flushSync() do ręcznego wymuszenia synchronizacji (gdy chcemy, by dana zmiana stanu natychmiast była widoczna w DOM, np. przed pomiarem wysokości elementu), lub startTransition() do oznaczenia mniej istotnych aktualizacji, które mogą poczekać (np. aktualizacja listy wyników wyszukiwania). Podsumowując: **React 18 sam dba o grupowanie aktualizacji stanu**, dzięki czemu aplikacje odczuwalnie zyskują na wydajności bez dodatkowego wysiłku.

Unikanie zbędnych renderów – memoizacja komponentów: Domyślnie React renderuje komponent ponownie za każdym razem, gdy jego rodzic się zrenderuje (chyba że komponent

nie zależy od żadnych propsów ani stanu, wtedy może zostać pominięty). W dużych listach lub złożonych drzewach UI może to oznaczać wiele niepotrzebnych obliczeń, jeśli dane nie uległy zmianie. Tutaj z pomocą przychodzi **React.memo()** – wyższy komponent (HOC) lub mechanizm dla komponentów funkcyjnych, który **zapamiętuje wynik renderu** danego komponentu i **pomija ponowny render** jeśli **jego propsy się nie zmieniły**. Działa to przez płytkie porównanie (shallow compare) poprzednich i nowych propsów. React.memo warto stosować wokół komponentów, które: (a) **są kosztowne w renderowaniu** (np. zawierają skomplikowane listy, tabele, wykresy) lub (b) bardzo często się renderują, mimo że ich dane rzadko się zmieniają. Przykładowo, karta z wykresem lub złożoną logiką wyświetlania danych powinna renderować się ponownie tylko, gdy dane faktycznie się zmienią – opakowanie jej w React.memo to zapewni. Nie ma natomiast sensu memoizować prostych, tanich komponentów (jak pojedynczy przycisk czy ikonka), bo **koszt porównania propsów może przewyższyć zysk** z pominięcia renderu. Należy traktować memoizację komponentów jak **precyzyjne narzędzie (skalpel), a nie młotek** do wszystkiego. Innymi słowy – optymalizujemy te miejsca, które są wąskimi gardłami.

W kontekście memoizacji komponentów trzeba wspomnieć o **stabilności referencji** propsów. React.memo porównuje nowe i stare props **referencyjnie**, co oznacza, że np. dwie różne funkcje choćby z tą samą implementacją będą traktowane jako różne (bo mają inny adres w pamięci). Dlatego, jeśli nasz komponent w propsach dostaje funkcje lub obiekty, które **zmieniają się przy każdym renderze rodzica**, to React.memo i tak nie pomoże – bo props za każdym razem będzie „nowy". Aby temu zaradzić, stosujemy **useCallback** i **useMemo** dla stabilizacji tych wartości.

- **useCallback(fn, deps)** zwraca referencję do tej samej funkcji fn między renderami, o ile nie zmieniły się deklarowane zależności. Innymi słowy, **zapamiętuje funkcję** – dzięki temu np. możemy przekazywać dziecku callback (onClick, onChange etc.) który nie będzie się zmieniać przy każdym renderze rodzica, co zapobiegnie niepotrzebnemu re-renderowaniu dziecka (jeśli dziecko jest memo). Przykładowo, przy liście elementów gdzie każdy element ma przycisk „usuń" wywołujący onRemove(id): bez useCallback funkcja onRemove tworzona w rodzicu jest za każdym razem nowym obiektem-funkcją, przez co memoizowane dziecko i tak by się odświeżało; z useCallback rodzic podaje wciąż tę samą referencję funkcji między renderami (dopóki np. lista się nie zmieni), więc dziecko pozostaje skutecznie *czyste* i nie renderuje się ponownie.
- **useMemo(calcFn, deps)** z kolei **zapamiętuje wynik** wywołania funkcji calcFn dopóki zależności się nie zmienią. Używamy go, aby **unikać kosztownych obliczeń** przy każdym renderze. Jeśli np. musimy przefiltrować dużą tablicę danych albo wykonać skomplikowane przeliczenie na podstawie propsów, możemy opakować to w useMemo – wtedy **wynik poprzedniego obliczenia zostanie zwrócony z cache** tak długo, jak długo istotne dane wejściowe są identyczne, a funkcja calcFn nie będzie wywoływana ponownie bez potrzeby. To potrafi drastycznie poprawić wydajność w scenariuszach typu generowanie tabeli, filtrowanie, sortowanie, formatowanie dużych datasetów itp., zwłaszcza gdy komponent często się renderuje ale dane zmieniają rzadko.

Ogólnie rzecz biorąc, **useMemo i useCallback to narzędzia pomagające optymalizować ponowne renderowanie**. Można je podsumować tak: służą do **zmniejszenia pracy**

wykonywanej podczas renderu oraz **zmniejszenia liczby samych renderów** komponentów. Dzięki nim unikamy sytuacji, gdzie przy każdym odświeżeniu UI na nowo liczymy te same wartości lub tworzymy te same funkcje. Należy jednak stosować je z rozwagą – pamiętajmy, że samo użycie hooka memoizującego też ma pewien koszt (trzeba wykonać porównanie zależności, przechować wartość itp.). Dlatego **nie ma sensu prewencyjnie memoizować wszystkiego**; najlepiej namierzyć wąskie gardła (np. poprzez Profile w React DevTools, który pokaże które komponenty często się renderują i ile to trwa) i tam założyć optymalizacje.

Najlepsze praktyki wydajności w React: Podsumujmy kilka wskazówek:

- **Batch aktualizacji** – React 18 zrobi to za Nas automatycznie w większości przypadków. Jeśli mamy wiele powiązanych zmian stanu w jednym zdarzeniu, wykonuj je razem (np. wywołaj setState kilka razy w jednym handlerze zamiast w osobnych, a React zgrupuje je w jeden render). Unikaj jednak wymuszania synchronicznych renderów bez potrzeby (sporadycznie flushSync może się przydać, ale nadużywanie go neguje zyski z batchingu).
- **Minimalizuj głębokość zależności renderów** – jeśli jakiś komponent wysyła do dziecka funkcję lub obiekt, który zmienia się co render, rozważ useCallback/useMemo, aby dziecko nie widziało ciągle zmieniającego się propsa. Im mniej "propów dynamicznych" tym lepiej dla możliwości memoizacji.
- **Stosuj React.memo do ciężkich komponentów** – komponenty prezentacyjne wyświetlające duże zbiory danych, listy, tabele, wykresy, itp. warto opakować w React.memo, aby nie przerysowywać ich jeśli nie zmieniły się propsy. Pamiętaj jednak o stabilności propsów (jw.).
- **Unikaj kosztownych obliczeń w trakcie renderu** – jeżeli w funkcji komponentu widzisz drogie operacje (pętle po dużych tablicach, sortowania, parsowanie danych), rozważ przeniesienie ich do useMemo. Dzięki temu ta praca wykona się tylko wtedy, gdy dane źródłowe faktycznie się zmienią, a nie przy każdym renderze z osobna.
- **Używaj jednego stanu dla grup zależnych** – jeśli masz kilka zmiennych stanu, które zawsze zmieniają się razem, rozważ przechowywanie ich w jednym obiekcie stanu lub użycie reducer, żeby uniknąć lawiny wielu setState (choć w React 18 i tak by się zbatchowały, ale logicznie bywa to prostsze).
- **Testuj i profiluj** – narzędzia takie jak React DevTools Profiler pomogą zrozumieć, gdzie faktycznie aplikacja spędza czas. Czasem optymalizacje w złym miejscu mogą wręcz pogorszyć wydajność (np. niepotrzebne użycie useMemo, które samo w sobie ma koszt, może wydłużyć czas renderu jeśli obliczenie było trywialne).

Na koniec warto wspomnieć o **najnowszych trendach**: zespół React pracuje nad automatyzacją wielu z tych optymalizacji. Pojawił się tzw. **React Compiler** (w fazie beta dla React 19), który potrafi automatycznie przekształcać kod w czasie kompilacji tak, aby dołączał React.memo, useMemo i useCallback tam, gdzie to bezpieczne. W praktyce oznacza to, że w przyszłości możemy pisać kod bez ręcznego owijania wszystkiego w memo, a kompilator zrobi optymalizację za nas. Już teraz w środowisku produkcyjnym Meta (np. Instagram) testuje się to rozwiązanie i wygląda obiecująco – developerzy odnotowali wzrost produktywności (mniej czasu na myślenie o optymalizacji) oraz mniej błędów związanych z zależnościami, przy jednoczesnym zysku wydajnościowym, bo kompilator może memoizować *cały* kod domyślnie. **Na ten moment (2025)** jednak, większość projektów nadal korzysta z

manualnych technik optymalizacji, więc warto dobrze rozumieć memo, useMemo, useCallback. Pisząc nowy kod pamiętajmy: **czytelność przede wszystkim** – nie “premature optimize”, chyba że wiemy (lub przeczuwamy) iż dana część UI będzie wrażliwa wydajnościowo.

Demo: globalny AuthContext + przełącznik motywu (light/dark)

Na koniec zobaczmy praktyczny przykład wykorzystania kontekstu do zarządzania stanem globalnym. Wiele aplikacji potrzebuje np. przechowywać informację o **aktualnie zalogowanym użytkowniku** (auth) oraz umożliwiać **globalną zmianę motywu** (jasny/ciemny tryb). Obie te rzeczy są dobrymi kandydatami na kontekst: powinny być dostępne w wielu miejscach aplikacji (np. stan zalogowania wpływa na wyświetlanie całego menu, a motyw wpływa na styl wszystkich komponentów), a jednocześnie zmieniają się stosunkowo rzadko (logowanie/wylogowanie to rzadkie zdarzenie, zmiana motywu też). Stworzymy dwa konteksty: **AuthContext** i **ThemeContext**, wraz z odpowiadającymi im dostawcami (Providerami). AuthContext będzie przechowywał np. obiekt użytkownika lub informację czy user jest zalogowany, oraz metody login i logout do zmiany tego stanu. ThemeContext będzie przechowywał aktualny motyw ('light' lub 'dark') oraz funkcję toggleTheme do jego przełączania. Następnie pokażemy, jak użyć tych kontekstów w komponentach, np. w nawigacji.

Kroki implementacji:

1. **Utworzenie kontekstów i providerów:** Za pomocą createContext tworzymy obiekty kontekstu. Definiujemy komponenty Provider, które będą opakowywać całą aplikację. W nich używamy useState (lub useReducer przy bardziej rozbudowanej logice) do zarządzania stanem i przekazujemy w value kontekstu zarówno aktualny stan, jak i funkcje do jego zmiany.
2. **Opakowanie aplikacji w Provider’y:** W pliku głównym (np. App.js lub index.js) otaczamy główny komponent aplikacji naszymi providerami: <AuthProvider> i <ThemeProvider>. Dzięki temu dowolny komponent wewnątrz będzie miał dostęp do wartości kontekstu poprzez hook useContext.
3. **Korzystanie z kontekstu w komponentach:** Wewnątrz dowolnego komponentu, który potrzebuje np. informacji o zalogowanym użytkowniku lub motywie, wywołujemy const { user, logout } = useContext(AuthContext) lub const { theme, toggleTheme } = useContext(ThemeContext) by wydobyć potrzebne wartości/funkcje. Możemy następnie użyć tych danych – np. wyświetlić powitanie z nazwą użytkownika, warunkowo renderować przycisk "Zaloguj/Wyloguj", a także zmieniać klasę CSS czy style w zależności od motywu.
4. **Przykładowy komponent (Navbar):** Załóżmy, że mamy komponent nawigacji, który chce pokazać inne opcje w menu w zależności od tego czy użytkownik jest zalogowany, i umieścić przycisk do zmiany motywu. Korzysta on z obu kontekstów równocześnie.

Poniżej znajduje się przykładowa implementacja w kodzie JSX ilustrująca powyższe kroki (dla prostoty pomijamy szczegóły typu faktyczne logowanie – zakładamy, że login ustawia fikcyjny obiekt użytkownika):

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

// 1. Tworzymy konteksty
const AuthContext = createContext(null);
const ThemeContext = createContext('light');

// 1. Definiujemy Provider dla 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. Definiujemy Provider dla 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. Opakowujemy aplikację w providery
function App() {
  return (
    <AuthProvider>
      <ThemeProvider>
        <MainApplicationContent /> {/* reszta aplikacji */}
      </ThemeProvider>
    </AuthProvider>
  );
}

// 3. Przykład konsumpcji kontekstu w komponencie
function Navbar() {
  const { user, logout } = useContext(AuthContext);
  const { theme, toggleTheme } = useContext(ThemeContext);
  return (
    <header className={`header-${theme}`}>
      <nav>
        {user ? (
          <>
            <span>Witaj, {user.name}!</span>
            <button onClick={logout}>Wyloguj</button>
          </>
        ) : (
          <button onClick={() => alert('Przejdź do logowania...')}>Zaloguj</button>
```

```
      )}
    </nav>
    <button onClick={toggleTheme}>
      {theme === 'light' ? 'Dark mode' : 'Light mode'}
    </button>
  </header>
 );
}
```

W powyższym kodzie komponent <AuthProvider> zarządza stanem user (przechowuje np. obiekt z danymi zalogowanego użytkownika lub null, gdy niezalogowany) i udostępnia go wraz z funkcjami login/logout poprzez kontekst. <ThemeProvider> udostępnia aktualny motyw i funkcję do jego zmiany. Cała aplikacja jest opakowana w te providery, więc np. nasz <Navbar> czy inny komponent **nie musi otrzymywać tych danych przez propsy** – korzysta bezpośrednio z useContext.

Takie podejście eliminuje prop drilling – np. nie musimy przekazywać informacji o użytkowniku do każdego komponentu w drzewie; wystarczy raz pobrać z kontekstu tam, gdzie potrzebujemy (np. Navbar, Panel użytkownika, itp.). Podobnie z motywem – dowolny komponent (przycisk, tło strony) może sprawdzić aktualny motyw przez useContext(ThemeContext) i dostosować styl. Co ważne, zmiana kontekstu (np. wywołanie toggleTheme zmieniające theme na 'dark') **automatycznie re-renderuje wszystkie komponenty korzystające z ThemeContext** – w naszym przypadku natychmiast **cała aplikacja przełączy się na nowy motyw** bez ręcznego przekazywania tej zmiany.

Jednocześnie, oddzielenie na dwa osobne konteksty **Auth** i **Theme** to świadomy zabieg architektoniczny: zmiana motywu nie powoduje np. ponownego renderu komponentów korzystających tylko z AuthContext i vice versa. Gdybyśmy wrzucili wszystko do jednego kontekstu globalnego, nawet niepowiązane zmiany mogłyby odświeżać niepotrzebnie wiele części UI. Dlatego **warto tworzyć odrębne konteksty dla niezależnych domen stanu**. React pozwala bez problemu zagnieżdżać wielu providerów, co widzimy na przykładzie – kolejność zagnieżdżania nie ma większego znaczenia, ważne by objąć tymi providerami odpowiedni zakres komponentów (tu: całą aplikację).

Na koniec dodajmy, że **Context API** jest świetne do tego typu globalnych ustawień i prostego stanu aplikacji. Gdy aplikacja rośnie, możemy na bazie podobnego wzorca rozbudować go np. używając useReducer w AuthProvider (by obsłużyć więcej akcji, np. odświeżenie tokenu, update profilu, itp.) lub sięgnąć po omówione wcześniej rozwiązania jak Redux/Zustand gdyby stan globalny stał się bardzo rozbudowany. Niemniej, dla potrzeb tego dema widać, że z użyciem najnowszych React API możemy zaimplementować **globalny stan autoryzacji oraz motyw aplikacji w prosty i czytelny sposób**, zgodny z dzisiejszymi najlepszymi praktykami.

Literatura:

1. https://react.dev/learn/managing-state (Data dostępu: 1.10.2025) – Oficjalny przewodnik React dotyczący zarządzania stanem, wyjaśniający podstawy strukturyzacji danych w aplikacji.
2. https://react.dev/reference/react/useContext (Data dostępu: 1.10.2025) – Dokumentacja techniczna hooka useContext, opisująca udostępnianie danych bez "prop drillingu".
3. https://react.dev/reference/react/useReducer (Data dostępu: 1.10.2025) – Dokumentacja hooka useReducer, zalecanego do obsługi złożonej logiki stanu i wielu powiązanych wartości.
4. https://tanstack.com/query/latest/docs/framework/react/overview (Data dostępu: 1.10.2025) – Oficjalna dokumentacja React Query, kluczowego narzędzia do zarządzania stanem serwerowym i asynchronicznymi danymi.
5. https://zustand-demo.pmnd.rs/ (Data dostępu: 1.10.2025) – Dokumentacja i demo biblioteki Zustand, przedstawiające nowoczesne i lekkie podejście do globalnego stanu aplikacji.