POLITECHNIKA ŚWIĘTOKRZYSKA

Aplikacje mobilne – wykład 6

Sieć, API i React Query w React Native

Mateusz Pawełkiewicz

1.10.2025

1. Podstawy pracy z API w React Native

W aplikacjach **React Native** komunikacja z zewnętrznymi API odbywa się podobnie jak w aplikacjach webowych. Dostępne są standardowe metody takie jak **Fetch API** oraz popularne biblioteki typu **Axios**. W tej części omówimy oba podejścia, w tym ich możliwości konfiguracyjne (nagłówki, interceptory, timeouty), anulowanie zapytań oraz porównamy ich zalety i wady.

1.1 Fetch API – wbudowane zapytania sieciowe

Fetch API jest wbudowaną w JavaScript metodą do wykonywania zapytań HTTP. W środowisku React Native `fetch` jest dostępny globalnie (polifill). Jego składnia jest podobna jak w przeglądarce:

```
// Przykładowe zapytanie GET za pomocą fetch:
try {
  const response = await fetch('https://api.example.com/data');
  if (!response.ok) {
    // Sprawdzenie kodu statusu (fetch nie rzuca błędu dla błędów HTTP)
    throw new Error(`Błąd HTTP: ${response.status}`);
  }
  const data = await response.json();  // Ręczne przetworzenie JSON
  console.log('Otrzymane dane:', data);
} catch (error) {
  console.error('Błąd sieci lub zapytania:', error);
}
```

Charakterystyka Fetch API:

- **Brak konfiguracji globalnej** – Nie ma wbudowanego mechanizmu ustawiania bazowego URL czy domyślnych nagłówków dla wszystkich zapytań. Konieczne jest każdorazowe podanie opcji lub opakowanie fetch w własną funkcję pomocniczą.
- **Obsługa odpowiedzi** – Fetch **nie rzuca wyjątku dla odpowiedzi HTTP z błędem** (np. 404, 500). Trzeba samodzielnie sprawdzać `response.ok` lub `response.status`. W razie błędu sieci (brak połączenia, DNS itp.) dopiero wtedy `fetch` rzuci wyjątek, ale błędny kod HTTP nie jest automatycznie traktowany jako wyjątek.
- **Parsowanie JSON** – Fetch nie przetwarza automatycznie treści odpowiedzi. Aby uzyskać dane, zwykle wywołujemy metodę `response.json()` (lub `response.text()`, `response.blob()` w zależności od formatu).
- **Brak wbudowanych interceptorów** – Fetch nie oferuje natywnego mechanizmu przechwytującego żądania/odpowiedzi. Można jednak osiągnąć podobny efekt, pisząc własne funkcje-wrappers lub korzystając z innych bibliotek do logowania czy modyfikacji zapytań
- **Timeouty i anulowanie** – Fetch nie posiada opcji `timeout` wprost. Aby ograniczyć czas oczekiwania lub móc anulować zapytanie, stosujemy **AbortController** (omówiony poniżej).

Przykład ustawienia timeoutu i anulowania zapytania fetch:

```
const controller = new AbortController();
const id = setTimeout(() => controller.abort(), 5000); // przerwij po 5s

try {
  const response = await fetch(url, { signal: controller.signal });
  clearTimeout(id);
  // ... przetwarzanie odpowiedzi
} catch (err) {
  if (err.name === 'AbortError') {
    console.log('Zapytanie zostało przerwane (timeout lub anulowanie)');
  } else {
    console.log('Inny błąd zapytania:', err);
  }
}
```

W powyższym kodzie używamy AbortController do wygenerowania sygnału przerwania. Metoda controller.abort() powoduje odrzucenie promisy zwracanej przez fetch z błędem o nazwie "AbortError". Jest to uniwersalny sposób na **anulowanie zapytań** (np. przy wychodzeniu z ekranu, aby nie próbować zaktualizować nieistniejącego komponentu).

Debugowanie: W React Native warto użyć narzędzia **Flipper** do monitorowania zapytań sieciowych. Flipper posiada plugin **Network**, który wyświetla listę wszystkich zapytań HTTP wykonywanych przez aplikację (zarówno przez fetch, jak i XHR/axios). Upewnij się, że aplikacja jest uruchomiona w trybie deweloperskim i połączona z Flipperem – wtedy możesz podejrzeć szczegóły każdego requestu (URL, nagłówki, body, odpowiedź, czas trwania) co bardzo ułatwia diagnozowanie problemów z API. Dodatkowo, w kodzie możesz dodać własne logi (np. w .then().catch() dla fetch lub w interceptorach axios), aby logować requesty i odpowiedzi podczas developmentu.

1.2 Axios – biblioteka HTTP z dodatkowymi funkcjami

Axios to popularna biblioteka do wykonywania zapytań HTTP, używana zarówno w przeglądarkach, Node.js, jak i React Native. Wnosi ona sporo ułatwień w porównaniu z czystym Fetch API:

- **Prostsza składnia i automatyczne przetwarzanie** – Axios automatycznie dokonuje **serializacji danych do JSON** przy wysyłaniu (obiekt JavaScript przekazany jako data zostanie wysłany jako JSON) oraz **deserializacji odpowiedzi JSON** do obiektu JavaScript (pole response.data zawiera już sparsowane dane). Nie musimy ręcznie wywoływać .json() na odpowiedzi.
- **Globalna konfiguracja i instancje** – Możemy utworzyć instancję axios z zdefiniowanym baseURL, domyślnymi nagłówkami czy parametrami. Ułatwia to zarządzanie wspólnymi elementami żądań w całej aplikacji:

```
import axios from 'axios';
const api = axios.create({
  baseURL: 'https://api.example.com/v1',
  timeout: 10000, // maksymalny czas oczekiwania 10s
  headers: { 'Content-Type': 'application/json' }
});
```

Taka instancja api będzie używana do wszystkich zapytań, co eliminuje powtarzanie adresu bazowego i nagłówków.

- **Interceptory** – Jedną z największych zalet axios są **interceptory żądań i odpowiedzi**. Pozwalają one przechwycić każde zapytanie lub odpowiedź i zmodyfikować je lub obsłużyć błędy globalnie:

```
// Przechwycenie każdego żądania – np. dołączenie tokena auth:
api.interceptors.request.use(config => {
  const token = authStorage.getToken();  // pseudokod pobierający token
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});
// Przechwycenie odpowiedzi – np. obsługa błędów 401 globalnie:
api.interceptors.response.use(
  response => response,
  error => {
    if (error.response?.status === 401) {
      // np. automatyczna próba odświeżenia tokena lub wylogowanie
      console.log('Błąd 401 - nieautoryzowany, odświeżanie tokena...');
      // ... (implementacja logiki odświeżania lub przekierowania do logowania)
    }
    return Promise.reject(error);
  }
);
```

Interceptory ułatwiają dodawanie wspólnej logiki – np. logowanie wszystkich requestów w trybie debug, ustawianie nagłówków **Authorization**, obsługę błędów uwierzytelnienia itp. W przypadku fetch trzeba by ręcznie opakowywać funkcje, podczas gdy axios oferuje to gotowe.

- **Obsługa błędów HTTP** – W przeciwieństwie do fetch, **axios automatycznie odrzuca Promise dla kodów odpowiedzi spoza zakresu 2xx**. Oznacza to, że odpowiedź z kodem 404 lub 500 trafi od razu do bloku .catch() z wygodnym obiektem błędu (zawierającym m.in. error.response z danymi odpowiedzi). Dzięki temu łatwiej jest pisać jednolitą obsługę błędów w jednym miejscu.
- **Timeouty i anulowanie** – Axios posiada w konfiguracji opcję timeout (jak pokazano w axios.create powyżej). Jeśli odpowiedź nie nadejdzie w podanym czasie, request zostanie przerwany automatycznie z błędem. Ponadto, axios wspiera mechanizm **anulowania zapytań** poprzez AbortController (od wersji 0.22) oraz historyczny (już *deprecated*) CancelToken. Współczesne podejście to użycie signal z AbortController – bardzo podobnie jak w fetch:

```
const controller = new AbortController();
try {
  await api.get('/tasks', { signal: controller.signal });
} catch(e) {
  if (axios.isCancel(e)) {
    // Sprawdzenie czy błąd wynika z anulowania
    console.log('Zapytanie anulowane.');
```

```
  }
}
controller.abort(); // można przerwać zapytanie w dowolnym momencie
```

Od wersji v0.22 axios zaleca używanie AbortController zamiast CancelToken. Mechanizm jest taki sam jak w fetch – przerwanie sygnału spowoduje odrzucenie obietnicy. W axios metoda axios.isCancel(error) pozwala rozpoznać, czy błąd wynika z anulowania.

- **Dodatkowe funkcje** – Axios umożliwia np. śledzenie postępu wysyłania/odbierania danych (onUploadProgress, onDownloadProgress), co bywa przydatne przy wysyłaniu plików lub pobieraniu większych zasobów (w RN jednak do dużych plików często używa się dedykowanych bibliotek natywnych). Pozwala też łatwo wykonywać zapytania równoległe (metoda axios.all lub użycie Promise.all z instancją axios).

Przykład zapytania POST z axios (dla porównania z fetch):

```
const newTask = { title: 'Nowe zadanie', completed: false };
try {
  const response = await api.post('/tasks', newTask);
  // Axios sam zserializuje newTask do JSON i ustawi nagłówek Content-Type
  console.log('Utworzono zadanie, dane:', response.data);
} catch (error) {
  if (error.response) {
    console.log('Błąd zapytania:', error.response.status, error.response.data);
  } else {
    console.log('Błąd sieci lub inny:', error.message);
  }
}
```

W powyższym kodzie widzimy, że nie ma potrzeby robić JSON.stringify na danych ani response.json() na odpowiedzi – axios robi to automatycznie. Obsługa błędu też jest prostsza: obiekt error.response istnieje tylko gdy serwer zwrócił odpowiedź z błędem (kod 4xx/5xx).

Porada: W środowisku React Native **nie trzeba osobno instalować polyfill** do fetch – jest on dostępny globalnie. Jeśli jednak decydujemy się na axios, należy zainstalować go przez npm/yarn (npm install axios) i pamiętać, że opiera się on na natywnym module XHR, który RN wspiera. Można używać jednocześnie fetch i axios w projekcie, ale zazwyczaj lepiej wybrać jedno podejście dla spójności.

1.3 Interceptory, nagłówki i konfiguracja globalna

Jak wspomniano, **axios** wyróżnia się możliwością globalnej konfiguracji i interceptorami:

- **Nagłówki globalne:** Możemy definiować nagłówki domyślne na poziomie instancji axios. Np. api.defaults.headers.common['Accept-Language'] = 'pl-PL'; ustawi domyślny język. Podobnie api.defaults.headers.common.Authorization = 'Bearer TOKEN' może globalnie dodać token (choć lepiej zrobić to dynamicznie w interceptorze, by zawsze pobierać aktualny token).
- **Interceptory żądań:** Idealne miejsce na **dołączanie tokena uwierzytelnienia** przed wysłaniem. W interceptorze mamy dostęp do obiektu config – możemy tam dodać

lub zmodyfikować nagłówki, zmienić adres itp. Warto również obsługiwać przypadki odświeżania tokena (więcej o tym w sekcji autoryzacji).

- **Interceptory odpowiedzi:** Pozwalają przechwycić błąd globalnie. Np. możemy sprawdzić kod 500 i zalogować zdarzenie do zewnętrznej usługi, albo dla 401 wykonać automatyczny **refresh token**. Trzeba jednak uważać, by nie spowodować zapętlenia (np. jeśli odświeżanie tokena też zwróci 401). Często używa się mechanizmu kolejki – wstrzymuje się nowe żądania w trakcie odświeżania tokena i po odświeżeniu kontynuuje.

Fetch nie ma takiego mechanizmu – jak to rozwiązać? Możemy pisać nasze funkcje, np. customFetch(url, options) które przed wywołaniem uzupełnią nagłówki (np. doda token z pamięci) i po otrzymaniu odpowiedzi sprawdzą status (np. jeśli 401 – odświeżą token i ponowią zapytanie). To jednak więcej pracy i większe pole na błędy. Dlatego **wiele aplikacji RN korzysta z axios** dla wygody, choć czysty fetch również jest w porządku dla prostych potrzeb.

Porównanie fetch vs axios: Fetch jest natywny i nie wymaga zależności, ma prostą składnię i jest **wspierany przez przeglądarki oraz RN out-of-the-box**. Axios to dodatkowa biblioteka, ale oferuje bogatsze API: interceptory, automatyczne przetwarzanie JSON, lepszą obsługę błędów i anulowania. **Z punktu widzenia wydajności** różnice są pomijalne przy typowych użyciach (axios pod spodem używa podobnego mechanizmu co fetch/XHR). Jeśli aplikacja wymaga wielu zapytań i rozbudowanej obsługi sieci (np. autoryzacja, globalne logowanie, specjalne nagłówki), axios zapewnia większą wygodę. Jeśli potrzebujesz minimalizmu lub chcesz ograniczyć zależności, fetch w zupełności wystarczy – większość funkcjonalności axios można zaimplementować ręcznie przy odrobinie wysiłku. Wybór więc zależy od preferencji i potrzeb projektu.

Podsumowując: Dla React Native oba rozwiązania są dobre. W kolejnych sekcjach (np. przy React Query) zobaczymy, że narzędzia wyższego poziomu mogą częściowo przejąć zadania związane z zapytaniami (np. mechanizmy retry, cache, itp.), co zmniejsza różnicę między fetch a axios, bo pewne rzeczy będą zarządzane przez bibliotekę.

2. React Query – zaawansowane zarządzanie stanem danych z API

Wraz ze wzrostem złożoności aplikacji, ręczne zarządzanie stanem danych pochodzących z API (np. listy elementów, szczegóły obiektów, stany ładowania, błędy, odświeżanie danych) staje się trudne. **React Query** (część TanStack Query) to biblioteka, która upraszcza pracę z danymi asynchronicznymi: **pobieraniem, cache'owaniem, odświeżaniem i mutacjami**. W kontekście React Native działa tak samo jak w React na webie – z pewnymi dodatkami dla środowiska mobilnego. Obecnie dostępna jest **React Query v5** (TanStack Query v5), która wprowadza usprawnienia względem poprzednich wersji (m.in. mniejszy rozmiar, ujednolicone API). Skupimy się na aktualnej wersji i nowoczesnych hookach.

2.1 Podstawy i konfiguracja React Query

Aby używać React Query, należy zainstalować paczkę @tanstack/react-query (nowsza nazwa biblioteki – dawniej react-query). W projekcie Expo/RN instalacja przebiega standardowo przez npm/yarn.

Następnie **konfigurujemy klient zapytań** (QueryClient) i **Provider** na najwyższym poziomie aplikacji, zwykle w pliku App.tsx:

```
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      retry: 2,          // domyślnie ponów nieudane zapytanie 2 razy
      staleTime: 5 * 60 * 1000, // czas (ms) po którym dane uznajemy za "nieświeże"
    },
  },
});

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <NavigationContainer>{/* reszta aplikacji */}</NavigationContainer>
    </QueryClientProvider>
  );
}
```

Powyżej tworzymy instancję QueryClient z domyślnymi opcjami. Możemy tam określić globalne ustawienia, np. **ile razy retry** ma nastąpić, jaki jest domyślny staleTime, czy query ma się odświeżać przy ponownym wejściu na ekran itp. Umieszczenie całej aplikacji wewnątrz QueryClientProvider zapewnia dostęp do funkcjonalności React Query w komponentach.

Uwaga: React Query v5 uprościł API – wszystkie konfiguracje przekazujemy jako **pojedynczy obiekt** zamiast wielu pozycyjnych argumentów, jak bywało wcześniej. Dla przykładu, dawniej useQuery('tasks', fetchTasks) teraz zapisujemy jako useQuery({ queryKey: ['tasks'], queryFn: fetchTasks }). Dzięki temu TypeScript lepiej wspiera to wywołanie i cała konfiguracja znajduje się w jednym obiekcie.

2.2 Zapytania – hook useQuery

Hook **useQuery** służy do pobierania danych (zapytania typu GET). Podstawowe użycie składa się z **unikalnego klucza zapytania** oraz funkcji która zwraca obietnicę (Promise) z danymi:

```
import { useQuery } from '@tanstack/react-query';
import api from './api'; // załóżmy, że to instancja axios

function TasksList() {
  const {
    data: tasks,
    error,
    isLoading,
    isFetching,
    refetch
```

```
 } = useQuery({
  queryKey: ['tasks'],          // unikalny klucz cache
  queryFn: async () => {
   const res = await api.get('/tasks');
   return res.data;           // zwracamy dane (lista tasków)
  },
  staleTime: 1000 * 30,        // dane ważne przez 30s (opcjonalnie)
  refetchOnMount: false        // opcjonalnie: nie odświeżaj na montażu jeśli dane są w cache
 });

 if (isLoading) {
  return <Text>Ładowanie zadań...</Text>;
 }
 if (error) {
  return <Text>Błąd: nie udało się pobrać listy zadań.</Text>;
 }

 return (
  <FlatList
   data={tasks}
   /* ... renderowanie listy zadań ... */
   refreshing={isFetching}      // można wykorzystać isFetching do wskaźnika odświeżania
   onRefresh={refetch}          // pociągnięcie listy w dół do odświeżenia
  />
 );
}
```

Kilka rzeczy dzieje się automatycznie powyżej:

- **Pierwsze wykonanie zapytania** następuje natychmiast po montowaniu komponentu (chyba że ustawimy opcję `enabled: false` – wtedy uruchomienie jest ręczne przez `refetch`).
- Hook `useQuery` zwraca obiekt z wieloma przydatnymi właściwościami:
  - `data` – wynik zapytania (zcache'owany). W naszym przykładzie `tasks` to zapewne tablica zadań lub obiekt pobrany z API.
  - `error` – obiekt błędu (jeśli zapytanie się nie powiodło). Gdy nie było błędu, będzie `undefined`.
  - **Stan ładowania:** `isLoading` jest **prawdziwy przy pierwszym uruchomieniu zapytania** (dopóki nie nadejdzie odpowiedź). `isFetching` jest true za każdym razem, gdy trwa **jakiś** fetch danych dla tego zapytania – np. również podczas odświeżania (refetch). W React Query v5 wprowadzono nową nazwę `isPending` zamiast `isLoading` dla ujednolicenia stanu w zapytaniach i mutacjach. W kodzie można spotkać obie nazwy zależnie od wersji – tu dla czytelności używamy `isLoading`.
  - `refetch` – funkcja, której wywołanie spowoduje ponowne wykonanie zapytania (ignorując ewentualny cache). Przydaje się np. do obsługi gestu **pull-to-refresh** listy w RN.
  - Inne: `status` (string `'loading' | 'error' | 'success'`), `isSuccess`, `isError` itp. dla wygody.
- **Cache i staleTime:** Pod kluczem `['tasks']` React Query przechowa w cache wynik. Póki dane są uznawane za świeże (przez `staleTime` lub domyślnie 0 ms, co oznacza *natychmiast nieświeże*), wejście ponownie na ekran nie spowoduje ponownego

pobierania – od razu pokażą się dane z cache. Po upływie staleTime kolejny mount spowoduje odświeżenie w tle. Możemy dostosować to zachowanie:

- Ustawienie staleTime większego (np. kilka minut) oznacza, że w tym czasie dane będą uznawane za aktualne i nie będą automatycznie refetchowane przy ponownym wejściu.
- Ustawienie refetchOnMount (domyślnie true gdy dane nieświeże) pozwala np. całkowicie wyłączyć odświeżanie na montowanie (refetchOnMount: false), lub wymusić zawsze odświeżanie (refetchOnMount: 'always').
- Istnieje też opcja refetchInterval do odpytywania co jakiś czas (polling).

Współpraca z TypeScript: useQuery świetnie współpracuje z TS. Można określić typ danych wynikowych: useQuery<MyDataType>({ queryKey: [...], queryFn: ... }), ale jeśli używamy funkcji queryFn która sama ma dobrze zdefiniowany zwracany typ (np. funkcja wywołująca axios z typowanym wynikiem), TS zwykle sam wydedukuje typ data. W przypadku błędu, domyślnie typ to unknown, ale można korzystać z własnych klas błędów lub rzutować w razie potrzeby.

Retry (ponawianie): Domyślnie React Query **ponawia nieudane zapytanie 3 razy** (z krótkim opóźnieniem) zanim oznaczy je jako błędne. Można to zmienić globalnie lub per zapytanie (retry: liczba lub retry: false żeby wyłączyć). Ponawiane są tylko błędy sieciowe lub błędy odpowiedzi (np. kod 5xx). To zachowanie bywa przydatne przy niestabilnym połączeniu – użytkownik nie zobaczy błędu od razu jeśli np. pierwsze żądanie nie dotarło. Oczywiście należy uważać, by nie powtarzać operacji, które nie powinny być idempotentne (choć domyślnie React Query nie ponawia mutacji, tylko zapytania typu get).

Przykład: Jeśli endpoint /tasks czasem losowo zwraca błąd 500, React Query automatycznie spróbuje do 3 razy go pobrać zanim przekaże błąd do interfejsu.

2.3 Mutacje – hook useMutation

Drugim ważnym filarem jest **useMutation**, służący do operacji zmieniających dane (**POST/PUT/DELETE** itp.). Mutacje mogą powodować zmiany w stanie serwera, więc często po ich wykonaniu chcemy **zaktualizować cache** lub odświeżyć zapytania GET.

Podstawowe użycie useMutation:

```
import { useMutation, useQueryClient } from '@tanstack/react-query';

function NewTaskForm() {
 const queryClient = useQueryClient();

 const { mutate: addTask, isLoading: isAdding } = useMutation({
  mutationFn: async (newTaskData) => {
   const res = await api.post('/tasks', newTaskData);
   return res.data; // zakładamy, że API zwraca utworzony obiekt zadania
  },
  onSuccess: (createdTask) => {
   // Po pomyślnym dodaniu zadania, odśwież listę tasków:
   queryClient.invalidateQueries({ queryKey: ['tasks'] });
   // Alternatywnie: można było od razu zaktualizować cache zamiast refetch (omówione poniżej)
  },
```

```
  onError: (error) => {
    console.error('Nie udało się dodać zadania:', error);
  }
 });

 // ... UI formularza z przyciskiem:
 const handleSubmit = () => {
  addTask({ title: 'Nowe zadanie', completed: false });
 };

 return (
  <Button onPress={handleSubmit} disabled={isAdding} title="Dodaj zadanie" />
 );
}
```

W powyższym kodzie definiujemy mutację do dodawania zadania:

- mutationFn – funkcja wykonująca zapytanie (tu POST /tasks z danymi nowego zadania). Może zwracać wynik (np. utworzony obiekt), który potem jest dostępny w onSuccess.
- onSuccess – opcjonalny callback wywoływany, gdy mutacja się powiedzie. Idealne miejsce by **inwalidować cache** zapytań, których dane mogły się zmienić. Używamy do tego queryClient.invalidateQueries z kluczem – w tym wypadku lista ['tasks'] powinna zostać ponownie pobrana, aby uwzględnić nowe zadanie. Zamiast invalid, można też **zaktualizować dane w cache ręcznie**
- onError – obsługa błędu (np. pokazanie komunikatu). Można też wykorzystać obiekt błędu w komponencie poprzez mutation.error, podobnie jak przy query.

Hook useMutation zwraca m.in.:

- mutate (lub mutateAsync) – funkcję do wywołania mutacji z przekazaniem danych.
- Stany: isLoading (w v5 nazywany też isPending), isSuccess, isError – analogicznie jak dla zapytań, informują o stanie danego żądania modyfikującego.
- Dodatkowo data – wynik zwrócony z mutacji (np. createdTask w onSuccess), error – obiekt błędu jeśli wystąpił.

Optimistic updates – optymistyczne aktualizacje UI

Optymistyczna aktualizacja polega na tym, że **zakładamy sukces operacji** i natychmiast modyfikujemy UI, zanim otrzymamy odpowiedź z serwera. Jeśli później okaże się, że operacja się nie powiodła, musimy wycofać te zmiany lub zasygnalizować błąd użytkownikowi.

React Query wspiera optymistyczne aktualizacje na dwa sposoby:

1. **Poprzez UI i stan mutacji (prostsza metoda w React Query v5):** Możemy wykorzystać fakt, że useMutation zwraca nam parametry ostatniej mutacji (variables) oraz stan isPending. Dzięki temu możemy np. tymczasowo dodać element do listy zadań bez grzebania w cache:

```
const addTaskMutation = useMutation({
  mutationFn: addTaskApiCall,
  onSettled: () => queryClient.invalidateQueries({ queryKey: ['tasks'] })
});
const { mutate: addTask, variables: newTaskVars, isPending: isAdding } = addTaskMutation;

// ... w komponencie listy zadań:
return (
  <View>
    {tasks.map(task => <TaskItem key={task.id} {...task} />)}
    {isAdding && (
      <TaskItem task={{ ...newTaskVars, id: 'temp-id' }} style={{ opacity: 0.5 }} />
    )}
  </View>
);
```

W powyższym schemacie, gdy wywołamy addTask(newTaskData), stan isPending będzie true, a variables przechowa newTaskData. Możemy więc do listy wyrenderować tymczasowy element z danymi nowego zadania (np. z półprzezroczystym stylem, by odróżnić). Gdy mutacja się powiedzie, isPending wróci do false, a refetch w onSettled odświeży listę już z faktycznym nowym elementem z bazy. Jeśli mutacja się nie powiedzie, tymczasowy element również zniknie (choć możemy go zostawić i np. dodać przycisk "Spróbuj ponownie" – wykorzystując nadal variables nawet po błędzie, bo React Query nie usuwa ich automatycznie przy errorze).

Ten sposób jest **prostszy**, bo nie dotykamy ręcznie cache – po prostu warunkowo renderujemy dodatkowy element w oparciu o stan mutacji. W React Query v5 dodano także hook **useMutationState**, który pozwala śledzić zmiany nawet w innej części aplikacji, jeśli mutacja i wyświetlanie tymczasowych danych nie są w jednym komponencie.

2. **Poprzez modyfikację cache (tradycyjna metoda):** Bardziej zaawansowane podejście polega na tym, że w onMutate mutacji dokonujemy bezpośredniej zmiany w cache, a w onError w razie niepowodzenia przywracamy poprzedni stan. Przykład dla dodawania zadania:

```
const queryClient = useQueryClient();
useMutation({
  mutationFn: addTaskApiCall,
  // przed wykonaniem mutacji:
  onMutate: async (newTask) => {
    await queryClient.cancelQueries({ queryKey: ['tasks'] });
    const previousTasks = queryClient.getQueryData(['tasks']);
    // Optymistycznie ustaw nowe zadanie w cache:
    queryClient.setQueryData(['tasks'], old => [...(old || []), { id: '__temp__', ...newTask }]);
    return { previousTasks }; // zwracamy snapshota
  },
  onError: (err, newTask, context) => {
    // przy błędzie odtwarzamy poprzednią listę zadań
    queryClient.setQueryData(['tasks'], context.previousTasks);
  },
  onSettled: () => {
    // niezależnie od wyniku, pobierz aktualne dane z serwera
```

```
    queryClient.invalidateQueries({ queryKey: ['tasks'] });
  }
});
```

- onMutate zatrzymuje ewentualne bieżące odświeżenia (cancelQueries), pobiera bieżącą listę zadań z cache (previousTasks), a następnie **dodaje nowy obiekt** do listy za pomocą setQueryData. Używamy tymczasowego id: '__temp__' lub czegoś unikalnego, by odróżnić optymistyczny obiekt.
- Zwracamy z onMutate wartość (tutaj obiekt z poprzednią listą), która będzie przekazana do onError jeśli mutacja się nie uda.
- W onError przy błędzie przywracamy poprzedni stan listy zadań z context.previousTasks.
- onSettled (wywoływane zarówno po sukcesie jak i błędzie) służy do upewnienia się, że finalnie i tak mamy zgodność z serwerem – tu poprzez invalidację i refetch listy.

To podejście zapewnia pełną kontrolę i natychmiastową reakcję w UI. Jest nieco bardziej skomplikowane (trzeba pamiętać o przywracaniu stanu), ale bywa konieczne w niektórych scenariuszach, np. gdy zmiana jest trudniejsza do odzwierciedlenia warunkowym renderowaniem.

Wybór metody zależy od przypadku. Dla prostego dodawania elementu do listy, metoda pierwsza (UI) jest wystarczająca i mniej podatna na błędy – w razie niepowodzenia element po prostu zniknie lub możemy zaoferować ponawianie. Metoda druga daje pełną elastyczność (można np. zaktualizować wiele różnych query w cache optymistycznie). **React Query v5** stara się ułatwić ten proces, dlatego wprowadzono uproszczenia i hook useMutationState.

Mutacje równoległe: Warto wspomnieć, że React Query pozwala wykonywać kilka mutacji naraz i zarządzać ich stanem globalnie. Dzięki useIsMutating() można sprawdzić, czy *jakakolwiek* mutacja jest w toku (np. by zablokować przyciski globalnie), a useMutationState pozwala monitorować aktywne mutacje po kluczu. Np. można w jednym komponencie wywoływać useMutation(mutationKey: ['deleteTask'], …) dla kasowania zadania, a w innym nasłuchiwać czy jest jakaś mutacja deleteTask pending i wyświetlać np. spinner obok elementu.

2.4 Cache, refetching i invalidacja danych

Jedną z największych zalet React Query jest **cache danych** oparty o klucze. Zrozumienie kilku pojęć pomoże efektywnie z tego korzystać:

- **Klucz zapytania (queryKey):** tablica lub string unikalnie identyfikująca dane. Np. ['tasks'] dla listy wszystkich zadań, ale ['task', id] dla szczegółów pojedynczego zadania. Struktura tablicy może zawierać parametry zapytania (np. ['tasks', { page: 2 }] jeśli kluczem ma być strona 2). Klucz decyduje o tym, co jest traktowane jako ta sama dane w cache.
- **Cache time vs stale time:**

    - *Stale Time* – czas świeżości danych (konfigurujemy, np. 0 domyślnie oznacza dane od razu stają się „stale” po pobraniu). Dopóki dane są świeże, React Query nie będzie inicjował ponownego pobrania przy okazji re-renderów czy focusu aplikacji.
    - *Cache Time* – czas przechowywania *starych* danych w pamięci po ich „wygaśnięciu”. Domyślnie 5 minut. Po tym czasie, jeśli nie ma żadnego komponentu używającego danego query, dane zostaną usunięte z pamięci cache (tzw. garbage collect). W **v5** nazwano to lepiej `gcTime` (garbage collection time). Jeżeli w tym czasie ponownie odwołamy się do tego query, to nawet nieświeże dane mogą zostać pokazane natychmiast (z flagą że są stale) i równolegle nastąpi refetch.
- **Invalidacja (invalidateQueries):** programowe oznaczenie danych jako nieaktualne. Używamy tego np. po mutacjach w `onSuccess`, aby powiadomić React Query: „hej, dane pod kluczem X zmieniły się, pobierz je ponownie przy najbliższej okazji”. Invalidacja ustawia stan *stale* i jeśli dany query jest aktualnie używany w UI, to spowoduje automatyczny refetch. Jeśli nie jest w użyciu, to kolejny raz gdy się pojawi (mount), to pobierze nowsze dane.
- **Refetching na fokus/połączenie:** Domyślnie (w przeglądarce) React Query odświeża zapytania przy powrocie do okna (window focus) oraz po odzyskaniu połączenia (network reconnect). W React Native te mechanizmy też możemy mieć, ale musimy ręcznie zintegrować:
    - *Fokus aplikacji:* za pomocą **focusManager** i modułu AppState (React Query udostępnia focusManager).
    - *Online status:* za pomocą **onlineManager** i np. biblioteki NetInfo do wykrywania braku/odzyskania sieci. Po konfiguracji React Query będzie wiedział, że np. był offline i teraz jest online – może odświeżyć zaległe zapytania.
- **Ręczne odświeżanie (refetch):** jak pokazano, hook `useQuery` daje metodę `refetch`. Można też globalnie odświeżyć pewne grupy zapytań: `queryClient.refetchQueries({ queryKey: ['tasks'] })` by odświeżyć wszystkie zapytania zaczynające się na `tasks` (co przydaje się, gdy np. wiele różnych wariantów danych zadań ma być odświeżonych, choć najczęściej invalidate jest wystarczające).

Integracja z React Native – focus i online: W czystym RN **React Query nie ma dostępu do zdarzeń fokusu okna** (bo nie ma okna przeglądarki) ani do statusu sieci, ale można to łatwo dodać:

```
// Gdzieś w kodzie inicjalizacyjnym aplikacji (np. zaraz po utworzeniu QueryClient):
import NetInfo from '@react-native-community/netinfo';
import { onlineManager, focusManager } from '@tanstack/react-query';
import { AppState } from 'react-native';

// Połączenie: nasłuchuj zmian statusu sieci i informuj React Query
onlineManager.setEventListener(setOnline => {
  return NetInfo.addEventListener(state => {
    setOnline(!!state.isConnected);
  });
});

// Fokus aplikacji: nasłuchuj czy aplikacja jest aktywna (foreground)
```

```
AppState.addEventListener('change', state => {
  focusManager.setFocused(state === 'active');
});
```

Powyższy kod sprawi, że np. jeśli urządzenie utraci internet, React Query *wstrzyma automatyczne zapytania* i oznaczy tryb offline (zapytania mogą wejść w stan error, który można obsłużyć odpowiednio). Gdy sieć wróci, onlineManager powiadomi bibliotekę – domyślnie spowoduje to **refetch wszystkich wcześniej nieudanych zapytań** od razu (co jest pożądane zachowanie w większości przypadków). Podobnie z fokusem: gdy użytkownik zminimalizuje i przywróci aplikację, wywołanie focusManager.setFocused(true) może spowodować odświeżenie danych (o ile jakieś query było stale). Możemy też dostosować to zachowanie parametrami refetchOnReconnect, refetchOnFocus (tak jak w web, tylko tutaj to zależy od tej integracji).

Obsługa błędów w React Query: Domyślnie błędy zapytań nie są rzucane jako wyjątki do komponentu, tylko przekazywane w error i zmieniają isError. Można jednak włączyć tryb, gdzie przy błędzie zostanie rzucony wyjątek i złapany przez najbliższy **Error Boundary** w drzewie. Służy do tego opcja useErrorBoundary: true (globalnie lub przy poszczególnym query). Jest to przydatne w większych aplikacjach, gdzie chcemy np. jeden globalny **komponent graniczny błędów** zamiast w każdym miejscu sprawdzać error.

2.5 Obsługa stanów ładowania i błędów (UI)

W interfejsie musimy przewidzieć trzy główne stany dla zapytań: **ładowanie**, **błąd** i **dane załadowane**. Powyżej w kodzie TasksList widzieliśmy prostą obsługę przez if (isLoading) ... else if (error) ... else .... W praktyce można to rozbudować:

- **Loading placeholders:** Zamiast zwykłego tekstu "Ładowanie..." często stosuje się **komponenty placeholder** lub **spinnery**. Np. w React Native ActivityIndicator jako wskaźnik ładowania, lub przygotowane "wydmuszki" UI (tzw. *skeleton screens* – np. szare belki imitujące listę). Przykład:

```
{isLoading && <ActivityIndicator size="large" color="#0000ff" />}
```

  Alternatywnie, biblioteki jak **react-native-paper** czy **NativeBase** oferują gotowe komponenty placeholderów.

- **Stan błędu:** W razie błędu warto wyświetlić użytkownikowi przyjazny komunikat. Sam obiekt error od React Query może być różny (dla błędów HTTP może to być błąd axios z informacją o statusie, dla błędów sieci – może to być TypeError z fetch). Dlatego często przygotowujemy funkcję pomocniczą, która mapuje różne błędy na komunikat:

```
function getErrorMessage(error) {
  if (!error?.response) {
    return 'Brak połączenia z internetem.';
  }
  const status = error.response.status;
  if (status === 404) return 'Nie znaleziono zasobu.';
```

```
  if (status === 500) return 'Błąd serwera, spróbuj ponownie później.';
  return 'Wystąpił błąd. Kod: ' + status;
}
// ...
{error && <Text style={{color: 'red'}}>{getErrorMessage(error)}</Text>}
```

Taka spójna obsługa błędów sprawia, że użytkownik dostaje czytelny komunikat zamiast np. surowego `TypeError: Network request failed`. Można też rozważyć wysyłanie logów błędów do zewnętrznego systemu (Sentry, LogRocket itp.), ale to już wykracza poza nasz wykład.

- **Stan odświeżania:** Często chcemy sygnalizować kiedy dane są odświeżane w tle (np. po zeskrolowaniu w dół listy, lub automatycznie co pewien czas). W React Query służy do tego `isFetching` (dla query) lub `isFetchingNextPage` (dla infinite query, o tym za chwilę). Możemy np. pokazać mały wskaźnik *"Odświeżanie..."* na górze listy albo zmienić tytuł pull-to-refresh. W RN `FlatList` ma prop `refreshing` i `onRefresh` – podpięcie tam `isFetching` i `refetch` jak wyżej automatycznie pokaże spinner przy gestach pull-to-refresh.

React Query dba też o **zachowanie poprzednich danych** podczas refetchu (tzw. *keepPreviousData* w v4, obecnie w v5 łączy się to z placeholderData) – dzięki temu można np. przy stronicowaniu pokazywać starą stronę dopóki nowa się nie załaduje, zamiast pustego ekranu. Domyślnie jednak przy odświeżaniu danych `data` zostaje nadpisane. Jeśli chcemy, możemy ustawić `keepPreviousData: true` by w trakcie fetchu trzymać stare dane (w v5 ta opcja została scalona z `placeholderData` i możliwe, że trzeba by to inaczej ustawić, ale idea jest podobna).

Podsumowując: **stan ładowania i błędów powinien być czytelnie obsłużony w UI**. Nie zostawiajmy użytkownika z zawieszoną aplikacją – lepiej pokazać spinner. I nie ukrywajmy błędów w konsoli – wyświetlmy komunikat lub spróbujmy automatycznie ponowić (jeśli ma to sens). React Query dużo ułatwia, bo daje te stany od razu bez konieczności własnoręcznego tworzenia zmiennych typu `isLoading`.

3. Paginacja i zapytania „nieskończone” (infinite queries)

Wiele API udostępnia paginację wyników – np. listy elementów są podzielone na strony po 10, 20 elementów albo udostępniają mechanizm stronicowania za pomocą kursora. **React Query** obsługuje te scenariusze za pomocą dwóch podejść:

- **useQuery z parametrem strony** – tradycyjne podejście: każda strona jest osobnym zapytaniem, np. `useQuery(['tasks', page], queryFn)`. Można wtedy w UI mieć przyciski "Następna strona" / "Poprzednia strona", lub przy przewijaniu dynamicznie zwiększać numer strony i wykonywać kolejne zapytania.
- **useInfiniteQuery** – specjalny hook React Query, który wspiera **dociąganie kolejnych „paczek” danych** i łączenie ich automatycznie. Idealny do list z **infinite scroll** (przewijanie w dół powoduje pobranie kolejnej partii wyników).

3.1 Paginacja offset/limit vs kursory

Zanim przejdziemy do kodu, wyjaśnijmy dwa sposoby stronicowania, jakie możemy spotkać w API:

- **Offset/Limit (lub page/size):** Najczęstsze podejście – np. mamy endpoint /tasks?page=2&limit=10 lub /tasks?offset=20&limit=10. Serwer zwraca wtedy np. tasks oraz informację o łącznej liczbie elementów albo adresy do kolejnej/poprzedniej strony. Offset page wymaga znajomości ile elementów pominąć. Wadą bywa, że przy dynamicznie zmieniających się danych stronicowanie offsetem może pomijać lub dublować elementy jeśli w międzyczasie doszły/ubyły (np. wstawienie nowego elementu na początku listy zmieni offsety).
- **Cursor (lub tokeny kontynuacji):** Coraz popularniejsze podejście – serwer zwraca znacznik do następnej strony, np. nextCursor albo nextPageToken. Zapytanie wtedy wygląda np. /tasks?cursor=abc123 gdzie abc123 wskazuje, skąd kontynuować. Najczęściej API zwraca też informację czy jest kolejna strona (np. hasNextPage: true/false). Kursory są bardziej odporne na zmiany danych – wskazują konkretny punkt w czasie/porządku.

W React Query obsłużymy oba scenariusze za pomocą useInfiniteQuery, ale logika getNextPageParam będzie nieco inna.

3.2 Hook **useInfiniteQuery**

useInfiniteQuery działa podobnie do useQuery, ale zakłada, że będziemy pobierać wiele stron danych. Jego **queryFn** powinien akceptować parametr pageParam, a w opcjach musimy podać funkcję getNextPageParam, która powie bibliotece, **jaki pageParam użyć dla kolejnej strony**.

Przykład użycia (dla API offset/page):

```
const {
  data,
  fetchNextPage,
  hasNextPage,
  isFetchingNextPage,
  isLoading
} = useInfiniteQuery({
  queryKey: ['tasks'],
  queryFn: ({ pageParam = 1 }) => api.get(`/tasks?page=${pageParam}`).then(res => res.data),
  getNextPageParam: (lastPage, pages) => {
    // Zakładamy, że odpowiedź lastPage zawiera pole nextPage (numer kolejnej strony) lub null jeśli koniec
    return lastPage.nextPage ?? false;
    // jeśli zwrócimy false/undefined, React Query uzna że nie ma kolejnej strony
  }
});
```

Kilka wyjaśnień:

- pageParam – parametry strony; przy pierwszym wywołaniu jest równy domyślnej wartości (tu 1, bo tak ustawiliśmy). Później React Query będzie wstawiać kolejne wartości zwrócone z getNextPageParam.
- getNextPageParam(lastPage, allPages) – funkcja dostaje ostatnią pobraną stronę (dane zwrócone przez queryFn) oraz tablicę wszystkich dotychczasowych stron. Musi zwrócić wartość pageParam dla **następnej** strony albo false/undefined jeśli to już koniec. W powyższym pseudo-kodzie zakładamy, że lastPage (dane z serwera) ma pole nextPage. Alternatywnie, jeśli API zwraca np. currentPage i totalPages, moglibyśmy napisać:

```
getNextPageParam: (lastPage) => {
  return lastPage.currentPage < lastPage.totalPages
    ? lastPage.currentPage + 1
    : undefined;
}
```

- hasNextPage – wartość boolean, którą React Query ustala automatycznie na podstawie getNextPageParam. Jeśli getNextPageParam zwróci wartość (np. numer strony lub cursor), to hasNextPage będzie true. Jeśli zwróci undefined lub false, hasNextPage będzie false (koniec danych).
- fetchNextPage – funkcja do pobrania kolejnej strony. Wywoła wewnętrznie queryFn z następnym pageParam (tym, co getNextPageParam zwrócił poprzednio).
- data – tu struktura jest inna niż przy useQuery. data zawiera obiekt z polami:
    - data.pages – tablica, gdzie każdy element to wynik jednej strony (dokładnie to, co zwraca queryFn).
    - data.pageParams – tablica z parametrami użytymi dla tych stron (można zwykle ignorować, chyba że debugujemy).

Aby przedstawić dane w komponencie, zazwyczaj **łączymy wszystkie strony** w jedną listę. Np.:

```
<FlatList
  data={ data ? data.pages.flatMap(page => page.results) : [] }
  renderItem={...}
  onEndReached={() => { if (hasNextPage) fetchNextPage(); }}
  ListFooterComponent={ isFetchingNextPage ? <ActivityIndicator /> : null }
/>
```

W powyższym przykładzie:

- Zakładamy, że każda strona (page) ma pole results z tablicą elementów (jak np. API zwraca listę zadań). Używamy flatMap (lub map(...).flat()) by stworzyć jedną *spłaszczoną* tablicę wszystkich wyników. Dzięki temu FlatList traktuje to jak jedną ciągłą listę.
- onEndReached – event FlatList wywoływany gdy użytkownik doskroluje blisko dołu listy. Wewnątrz wywołujemy fetchNextPage() jeśli jest kolejna strona. Dodatkowo warto dać onEndReachedThreshold, np. 0.3 (30%), by wywołać dociąganie trochę przed samym końcem scrolla, co zapewni płynniejsze ładowanie.
- ListFooterComponent – komponent wyświetlany na końcu listy. Używamy tego, by pokazać spinner (ActivityIndicator) w momencie, gdy isFetchingNextPage jest true (czyli

kolejna strona się pobiera). Dzięki temu użytkownik widzi kręcące się kółko u dołu listy podczas ładowania kolejnych danych.

Uwaga: FlatList wymaga także propów keyExtractor i renderItem – to implementujemy standardowo. Pamiętajmy, by klucze elementów listy były unikalne globalnie. Jeśli dołączamy elementy z kolejnych stron, najlepszym kluczem będzie np. unikalne id każdego zadania. Jeżeli używamy indeksów listy jako key, przy doładowywaniu stron może to powodować problemy z odświeżaniem elementów – lepiej unikać indeksu jako key.

Alternatywa – paginacja z przyciskiem „Załaduj więcej”: Niekiedy, zamiast infinite scroll, stosuje się klasyczny przycisk *"Load more"*. W React Query można to zaimplementować na bazie useInfiniteQuery lub zwykłych useQuery. Np. z useInfiniteQuery można w ogóle zrezygnować z onEndReached, a poniżej listy dać:

```
{ hasNextPage && !isFetchingNextPage && (
   <Button title="Pokaż więcej" onPress={() => fetchNextPage()} />
  )
}
```

To wyświetli przycisk dopóki jest kolejna strona, a po kliknięciu pobierze następną. Po pobraniu (gdy hasNextPage zmieni się na false, np. osiągnięto koniec listy) przycisk zniknie. To podejście bywa czytelniejsze dla użytkownika gdy strony są wyraźne.

Porównanie useInfiniteQuery vs useQuery dla paginacji: Można oczywiście zrobić paginację używając wielu useQuery i stanu strony w komponencie (np. trzymać const [page, setPage] = useState(1), potem przycisk zwiększa page, a useQuery reaguje na page jako część queryKey). Jednak useInfiniteQuery upraszcza to o tyle, że:

- Sam zarządza tablicą stron i ich połączeniem.
- Zachowuje informacje czy jest kolejna strona (hasNextPage),
- Zapewnia unikalny isFetchingNextPage odróżniający dociąganie od początkowego isLoading.

Zatem w przypadku **infinite scroll** React Query wyraźnie zaleca użycie useInfiniteQuery, bo kod jest czystszy i mniej podatny na błędy.

3.3 Przewijanie listy i dynamiczne dociąganie danych (Infinite Scroll)

Łącząc powyższe, implementacja **infinite scroll w RN** wygląda następująco:

1. Używamy useInfiniteQuery z odpowiednim getNextPageParam.
2. Wykorzystujemy komponent **FlatList** do wyświetlania danych:
    - Ustawiamy data jako spłaszczoną listę wszystkich elementów ze stron.
    - Prop onEndReached wywołuje fetchNextPage (z zabezpieczeniem na hasNextPage).
    - Opcjonalnie onEndReachedThreshold np. 0.5 (lub inna wartość) by kontrolować moment dociągania. Domyślnie jest 0.5, co znaczy że gdy przewiniemy do połowy przed końcem listy, wywoła się onEndReached.
    - Prop ListFooterComponent wyświetla spinner, gdy isFetchingNextPage jest true.

- Można też dodać refreshing i onRefresh aby obsłużyć pull-to-refresh, np.:

```
refreshing={ isLoading && data?.pages?.length > 0 }
onRefresh={ refetch }
```

Powyższe zakłada, że jeśli mamy już jakieś dane i isLoading jest true, to znaczy wykonujemy refresh (np. ręczny).

3. **Obsługa unmount/wychodzenia z ekranu:** React Query automatycznie może anulować zapytania, które są w toku gdy komponent się unmountuje (np. użytkownik wyjdzie z listy zanim doczytała się kolejna strona). Mechanizm ten opiera się o AbortController pod spodem – React Query przekazuje sygnał do queryFn. Jednak queryFn musi używać fetch lub axios z obsługą sygnału. W praktyce, korzystając z axios w RN, też możemy przekazać signal (React Query w argumentach queryFn udostępnia je w context.signal w v5, lub abort sam w v4). Trzeba się upewnić, że nasza funkcja fetchująca to uwzględnia, jeśli chcemy czerpać korzyści z automatycznego anulowania niepotrzebnych zapytań (np. scroll szybko do dołu i w górę, by nie trafiały w międzyczasie stare zapytania). Szczegóły tego mechanizmu są opisane w dokumentacji TanStack Query, ale w wielu przypadkach nie musimy nic specjalnego robić – wystarczy użyć axios (≥0.22) lub fetch i React Query samo anuluje sygnał przy dezaktywacji zapytania.

Porada wydajnościowa: Przy bardzo długich listach warto ograniczyć liczbę zbuforowanych stron. React Query v5 umożliwia ustawienie maksymalnej liczby przechowywanych stron w pamięci (opcja maxPages w konfiguracji infiniteQuery). Jeżeli lista może mieć setki stron, to trzymanie ich wszystkich może zużywać pamięć – można np. zawsze wyrzucać najstarsze strony z cache. Jednak typowe listy (kilkadziesiąt elementów) nie wymagają takiej optymalizacji na starcie. Również FlatList ma mechanizmy optymalizacyjne (recycling elementów, removeClippedSubviews itp.), które zapobiegają problemom z wydajnością przy długich listach.

4. Error Boundaries i jednolity fallback UI

Mimo najlepszych starań, błędy w aplikacjach są nieuniknione – może to być błąd programistyczny (np. wyjątek w renderze komponentu) lub błąd środowiska (np. brak sieci). **Error Boundaries** to mechanizm Reacta, który pozwala przechwycić błędy JavaScript w czasie renderowania komponentów i wyświetlić zamiast nich zapasowy interfejs, zamiast rozbicia całej aplikacji.

W React (od wersji 16) można korzystać z gotowych implementacji. W kontekście React Native często sięga się po biblioteki takie jak **react-native-error-boundary**, które dostarczają łatwy w użyciu komponent <ErrorBoundary>.

Przykład użycia ErrorBoundary:

```
import ErrorBoundary from "react-native-error-boundary";

const FallbackUI = ({ error, resetError }) => (
```

```
  <View style={{ padding: 20 }}>
    <Text>Wystąpił nieoczekiwany błąd.</Text>
    <Text>{error.toString()}</Text>
    <Button title="Spróbuj ponownie" onPress={resetError} />
  </View>
);

export default function App() {
  return (
    <ErrorBoundary FallbackComponent={FallbackUI}>
      {/* Cała reszta aplikacji / nawigacji */}
      <MainNavigator />
    </ErrorBoundary>
  );
}
```

W powyższym kodzie obwijamy główną część aplikacji w ErrorBoundary. Jeśli **którykolwiek z komponentów podrzędnych rzuci błąd podczas renderowania**, zostanie on złapany, a zamiast niego wyświetli się FallbackUI. Pozwala to zapobiec ekranowi białej śmierci (white screen of death) i dać użytkownikowi opcję np. ponownej próby (przycisk resetError resetuje stan w ErrorBoundary, co pozwala ponownie wyrenderować komponenty jak gdyby nic – można to wykorzystać np. by zresetować błąd po nawigacji lub odświeżeniu danych).

Granularność: Możemy mieć jedno globalne ErrorBoundary na całą aplikację (wyświetlające np. komunikat „Coś poszło nie tak. Zrestartuj aplikację.”) lub umieszczać je bardziej lokalnie, np. wokół pojedynczych wrażliwych ekranów, dzięki czemu awaria jednego ekranu nie wyłączy całej aplikacji. Przykładowo, jeśli mamy osobny ErrorBoundary wokół komponentu listy zadań, a ten komponent wpadnie w wyjątek, to pokaże fallback UI tylko zamiast listy, ale reszta aplikacji (np. header, menu) będą działać.

Warto podkreślić: **ErrorBoundary chroni przed błędami renderowania i metod życia komponentu**. Nie wyłapie błędów asynchronicznych w *event handlerach* czy wewnątrz promisa (tam trzeba użyć try/catch w kodzie lub .catch w Promise). Błędy zapytań sieciowych (np. error w React Query) nie są same w sobie wyjątkami runtime (chyba że włączymy tryb useErrorBoundary). Dlatego error boundary przyda się głównie na nieprzewidziane wyjątki (np. błąd typu TypeError przy odwołaniu do niezdefiniowanego pola).

4.1 Globalna obsługa błędów sieci i komunikaty offline

Poza ErrorBoundary, w aplikacji mobilnej warto pomyśleć o **spójnej obsłudze braku połączenia i błędów API** na poziomie UX:

- **Brak sieci:** Używając wspomnianego modułu **NetInfo**, możemy nasłuchiwać zmian stanu połączenia. Gdy isConnected zmieni się na false, możemy np. wyświetlić globalny baner na górze aplikacji z informacją "Brak połączenia z internetem". Można to zrobić poprzez jakiś globalny komponent (np. w NavigationContainer w headerze warunkowo coś wstawić), albo nawet Popup/Toast. Upewnijmy się też, że akcje wymagające internetu są zablokowane lub kolejkują się, by użytkownik nie klikał przycisków na darmo.

- **Jednolite komunikaty błędów:** Jak już omówiono, dobrze jest przetłumaczyć błędy na ludzki język. Można to scentralizować. Np. mieć *middleware* w axios interceptorze, który dla każdego błędu wywoła funkcję showErrorToast(message). Albo w React Query możemy wykorzystać onError globalnie:

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      onError: error => {
        showErrorToast(getErrorMessage(error));
      }
    },
    mutations: {
      onError: error => {
        showErrorToast(getErrorMessage(error));
      }
    }
  }
});
```

  Wtedy każdy błąd zapytania/mutacji wywoła naszą funkcję pokazującą np. Toast z komunikatem. Biblioteki jak **react-native-toast-message** czy **react-native-flash-message** umożliwiają łatwe wyświetlenie ładnie wyglądającego powiadomienia o błędzie na ekranie. Taka globalna strategia zapewnia, że błędy nie zostaną przeoczone – użytkownik zawsze dostanie jakiś komunikat.

- **Retry / ponawianie akcji przez użytkownika:** Jeśli operacja się nie powiodła z powodu sieci, można rozważyć mechanizm ponawiania. Na poziomie UI – np. po błędzie listy zadań, można pokazać przycisk "Spróbuj ponownie" (który wywoła refetch()). Dla mutacji (np. dodawanie zadania) w przykładzie optymistycznym pokazywaliśmy nawet przycisk "Retry" obok elementu jeśli się nie udało. Ważne, by użytkownik nie czuł się zablokowany – jeśli coś nie zadziałało, dajmy mu możliwość reakcji.

Przechwytywanie błędów niezłapanych: W JavaScript można zarejestrować globalny handler dla nieobsłużonych obietnic (UnhandledPromiseRejection) oraz dla błędów nieobsłużonych (ErrorUtils w RN lub globalThis.onerror). Jednak w praktyce lepiej użyć do tego dedykowanych narzędzi jak Sentry – tutaj wykracza to poza zakres wykładu, ale w dużych aplikacjach integracja np. z Sentry umożliwia automatyczne raportowanie każdego wyjątku JS wraz ze stack trace.

Podsumowując, **spójna obsługa błędów** to:

- reagowanie na brak internetu (np. informacja "Jesteś offline"),
- pokazywanie zrozumiałych komunikatów zamiast pozostawiania użytkownika bez informacji,
- logowanie błędów dla deweloperów (by móc je naprawić),
- oraz nie rozbijanie całej aplikacji w razie wyjątku (Error Boundary).

5. Autoryzacja i uwierzytelnianie w zapytaniach

W aplikacjach mobilnych często musimy komunikować się z API, które wymaga **tokena uwierzytelniającego** (np. token JWT przekazywany w nagłówku **Authorization** jako Bearer). W tej sekcji omówimy:

- Przechowywanie tokenów po zalogowaniu,
- Automatyczne dołączanie ich do zapytań (np. w interceptorze axios),
- Odświeżanie tokenów (refresh token flow),
- Bezpieczne przechowywanie poświadczeń (SecureStore, EncryptedStorage).

5.1 Przechowywanie tokena – gdzie i jak?

Najprostsze podejście to trzymanie tokena w **pamięci stanu** (np. w kontekście React lub w jakimś globalnym store) w trakcie działania aplikacji, a także zapisanie go do pamięci trwałej, by po ponownym uruchomieniu aplikacji nie wymagać logowania od nowa.

AsyncStorage vs Secure Storage: Choć można użyć AsyncStorage (asynchroniczna pamięć klucz-wartość w RN) do przechowania tokena, **nie jest to zalecane dla wrażliwych danych**, ponieważ AsyncStorage przechowuje dane jawnie na dysku. Lepszym rozwiązaniem są **bezpieczne magazyny**:

- **Expo SecureStore:** jeśli używamy Expo, mamy łatwy dostęp do modułu SecureStore. Pozwala on zapisać dane **zaszyfrowane** w Keychain (iOS) lub Keystore (Android). Przykład:

```
import * as SecureStore from 'expo-secure-store';
await SecureStore.setItemAsync('token', accessToken);
// ... potem aby pobrać:
const token = await SecureStore.getItemAsync('token');
```

  SecureStore jest częścią Expo SDK i służy do szyfrowanego przechowywania par klucz-wartość. Jest idealny do tokenów, haseł itp. (max rozmiar wartości to ok. ~2048 bajtów, co w zupełności starcza na tokeny).

- **EncryptedStorage (react-native-encrypted-storage):** w aplikacjach RN poza Expo można użyć biblioteki react-native-encrypted-storage. Działa podobnie – pod spodem na iOS używa Keychain, na Androidzie EncryptedSharedPreferences – i abstrakcyjne daje prosty API JS:

```
import EncryptedStorage from 'react-native-encrypted-storage';
await EncryptedStorage.setItem('token', accessToken);
const token = await EncryptedStorage.getItem('token');
```

  EncryptedStorage jest zabezpieczoną alternatywą dla AsyncStorage – używa systemowych mechanizmów bezpiecznego składowania, więc dane są szyfrowane i powiązane z aplikacją (nikt bezpośrednio nie odczyta ich z plików). Wymaga dodania natywnego modułu do projektu (podlinkowanie lub autolink).

- **Keychain na iOS / Keystore na Androidzie bezpośrednio:** Istnieją też biblioteki pozwalające bezpośrednio korzystać z iOS Keychain, np. react-native-keychain. SecureStore czy EncryptedStorage to w zasadzie ułatwiają, ale można też bez Expo użyć react-native-keychain do zapisu haseł/tokenów z pewnymi dodatkowymi opcjami (np. kontrola dostępu typu „tylko po odblokowaniu urządzenia").

Ważne jest, by **nie przechowywać tokenów w zwykłych, niezabezpieczonych miejscach** (AsyncStorage, plik, redut,plaintext, itp.), bo jeśli ktoś uzyska dostęp do pamięci telefonu (np. przez złośliwą aplikację na zrootowanym/jailbreakowanym urządzeniu), łatwiej mu wyciągnąć takie dane. SecureStore/Keychain utrudnia atak – dane są zaszyfrowane per aplikacja i system dba o ich ochronę.

Informacja: SecureStore i EncryptedStorage przechowują dane permanentnie – tj. zostają nawet po wyłączeniu aplikacji, a *nawet po odinstalowaniu* (w przypadku iOS Keychain). Jeśli nie chcemy, by tokeny przeżywały odinstalowanie, w dokumentacji EncryptedStorage opisano trik czyszczenia Keychain przy pierwszym uruchomieniu po reinstalacji. Zazwyczaj jednak nie jest to problemem – token i tak wygaśnie do tego czasu.

5.2 Dołączanie tokena do zapytań (Bearer)

Najlepsze miejsce na to to nasza warstwa komunikacji, np. **interceptor axios**. Jak wcześniej pokazano:

```
api.interceptors.request.use(config => {
  const token = authToken; // np. zmienna globalna lub z jakiegoś modułu auth
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});
```

Takie rozwiązanie wymaga, by w momencie ustawiania interceptora mieć dostęp do tokena. Można:

- Wczytać token z SecureStore przy starcie aplikacji do zmiennej (np. w stanie kontekstu AuthContext). Gdy użytkownik się loguje, również zapisać zmienną.
- Ewentualnie, jeśli token może wygasać i być odświeżany, to interceptor mógłby używać zawsze aktualnej wartości z jakiegoś centralnego miejsca (kontekst, moduł).
- Pamiętajmy, że SecureStore.getItemAsync jest asynchroniczne. Interceptor axios może zwracać promisa – czyli można w nim użyć async/await. Alternatywnie prościej jest trzymać token w pamięci (bo i tak po logowaniu go mamy) i zsynchronizować go z secure storage.

W przypadku używania Fetch: nie ma interceptorów, więc trzeba każdorazowo dodawać nagłówek:

```
const token = await SecureStore.getItemAsync('token');
fetch(url, {
  headers: { Authorization: `Bearer ${token}`, ... }
```

```
});
```

Co oczywiście bywa uciążliwe. Można więc owinąć fetch:

```
async function authorizedFetch(endpoint, options = {}) {
  const token = await SecureStore.getItemAsync('token');
  const authHeaders = token ? { Authorization: `Bearer ${token}` } : {};
  const finalOptions = { ...options, headers: { ...options.headers, ...authHeaders } };
  return fetch(baseURL + endpoint, finalOptions);
}
```

I używać authorizedFetch w całej aplikacji zamiast gołego fetch. Taki pattern zapewnia centralizację logiki uwierzytelniania.

5.3 Odświeżanie tokena (Refresh Token flow)

Wiele systemów uwierzytelniania wydaje dwa tokeny: **Access Token** (krótkiego życia, np. 15 min) do autoryzacji requestów oraz **Refresh Token** (dłuższego życia, np. 7 dni lub więcej), którym można uzyskać nowy Access Token gdy stary wygaśnie. Scenariusz jest taki:

1. Użytkownik się loguje – dostajemy AccessToken + RefreshToken.
2. AccessToken używamy w nagłówkach do każdej akcji chronionej.
3. Gdy serwer odpowie, że AccessToken jest nieważny/expired (np. kod 401 z informacją "Token expired"), **musimy wywołać endpoint odświeżenia** z RefreshTokenem.
4. Po otrzymaniu nowego AccessToken (i czasem nowego RefreshTokena), zapisujemy je i ponawiamy oryginalne zapytanie, które się nie powiodło.
5. Jeśli odświeżenie się nie uda (np. RefreshToken także nieważny), to musimy zalogować ponownie użytkownika (np. wyrzucić go na ekran logowania, wyczyścić stan).

Implementacja w axios interceptorze:

```
let isRefreshing = false;
let pendingRequests = []; // kolejka zapytań oczekujących na odświeżenie

api.interceptors.response.use(
  res => res,
  async error => {
    const { response } = error;
    if (response?.status === 401) {
      // Jeżeli błąd 401 dotyczy wygaśnięcia tokena:
      if (response.data?.message === 'TokenExpired') {
        if (!isRefreshing) {
          isRefreshing = true;
          try {
            const refreshToken = await SecureStore.getItemAsync('refreshToken');
            const refreshRes = await api.post('/auth/refresh', { token: refreshToken });
            const newAccessToken = refreshRes.data.accessToken;
            // Zapisz nowy token
            await SecureStore.setItemAsync('token', newAccessToken);
            authToken = newAccessToken; // zaktualizuj zmienną w pamięci
```

```
          isRefreshing = false;
          // Odrób zaległe zapytania z nowym tokenem
          pendingRequests.forEach(cb => cb(newAccessToken));
          pendingRequests = [];
        } catch (err) {
          isRefreshing = false;
          pendingRequests = [];
          // Refresh nie powiódł się – wyloguj użytkownika
          navigateToLogin();
          return Promise.reject(err);
        }
      }
      // Zwracamy nowy Promise, który spowoduje opóźnienie wykonania oryginalnego requesta do czasu
odświeżenia:
      return new Promise((resolve, reject) => {
        pendingRequests.push((token) => {
          // Retry oryginalnego requesta z nowym tokenem
          error.config.headers.Authorization = 'Bearer ' + token;
          resolve( axios(error.config) );
        });
      });
    }
  }
  return Promise.reject(error);
 }
);
```

Powyższy kod jest nieco złożony, ale kluczowe jest:

- **isRefreshing** – flaga, by jednocześnie tylko jedno odświeżenie się wykonywało, nawet jeśli wiele requestów naraz dostało 401.
- **pendingRequests** – kolejka funkcji, które zostaną wykonane po odświeżeniu (czyli ponowią oryginalne requesty).
- Gdy pierwszy 401 z powodem "TokenExpired" przyjdzie, wchodzimy do if. Jeśli akurat nikt nie odświeża (!isRefreshing), to rozpoczynamy proces odświeżania:
    - Wyciągamy refreshToken ze storage,
    - Wołamy /auth/refresh (zakładamy, że zwraca nowy accessToken, ewentualnie też nowy refreshToken – to też należałoby obsłużyć).
    - Po sukcesie zapisujemy nowy token (w pamięci i secure storage), ustawiamy isRefreshing = false i wykonujemy wszystkie oczekujące zapytania z nowym tokenem (wywołując callbacki z kolejki).
    - Jeśli odświeżenie się nie uda, to znaczy sesja stracona – czyścimy tokeny, kierujemy na logowanie.
- Jeśli kolejne zapytanie trafia 401 podczas gdy isRefreshing jest true (czyli refresh już w toku), to nie odświeżamy ponownie, tylko zwracamy nowy Promise i **dodajemy jego resolver do kolejki**. Ten Promise jest tym co interceptor będzie zwracał dla tego drugiego requesta – więc oryginalny kod czekający na odpowiedź zostanie wstrzymany. Gdy odświeżenie dobiegnie końca, wywołamy callback z kolejki, który:
    - Ustawi nowy nagłówek Authorization,
    - Wywoła oryginalne zapytanie ponownie (axios(error.config)),
    - Zewnętrzny Promise się rozwiąże, a więc oryginalne zapytanie otrzyma swoją odpowiedź (drugi raz nie wpadnie do interceptora 401, bo token jest nowy).

Taki wzorzec zapewnia, że przy wygaśnięciu tokena nie zalogujemy użytkownika od razu tylko płynnie odnowimy token w tle. Użytkownik może nawet nie zauważyć, poza może minimalnym opóźnieniem. Trzeba oczywiście dopracować szczegóły (np. co jeśli refresh endpoint sam zwróci 401 – wtedy też logout).

Jeśli korzystamy z React Query, większość tego dzieje się na poziomie axios i jest przezroczyste dla React Query – po prostu zapytanie trwa dłużej bo czeka na refresh, ale finalnie dostaje dane. Można ewentualnie zareagować na globalny error 401 inaczej, ale powyższe jest dość kompleksowym rozwiązaniem.

Uwaga do Refresh Token: Przechowujemy **dwa** tokeny – access i refresh. Access trzymamy np. w pamięci/kontekście (i secure store dla ponownego otwarcia aplikacji), refresh w secure store. Refresh token jest jeszcze bardziej wrażliwy – jeśli ktoś go zdobędzie, może wyłudzić kolejne access tokeny. Dlatego:

- Czas życia refresh tokena powinien być ograniczony (np. wymuszenie logowania co 2 tygodnie).
- Można rozważyć trzymanie refresh tokena *tylko* w SecureStorage i *nie* trzymanie go w pamięci nigdy jawnie (tylko jak trzeba to go pobrać do odświeżenia).
- Po wylogowaniu pamiętaj usunąć oba tokeny ze storage.

5.4 Bezpieczne przechowywanie – podsumowanie

- **expo-secure-store** – prosty sposób na bezpieczne przechowanie tokenów w Expo. **Korzysta z mechanizmów systemowych (Keychain/Keystore)**, dlatego jest rekomendowany do poufnych danych.
- **react-native-encrypted-storage** – analogiczne rozwiązanie poza Expo, też używa Keychain/EncryptedSharedPrefs. Prosty interfejs czterech metod: setItem, getItem, removeItem, clear.
- **Nie używaj AsyncStorage na tokeny** – brak szyfrowania, dane w pliku JS. Ostatecznie, jeśli aplikacja nie przechowuje nic krytycznego i nie obawiamy się o to, można użyć AsyncStorage

6. Demo: Aplikacja "Tasks" – praktyczne połączenie wszystkich elementów

Na koniec połączmy teorię w mini przykładzie. Wyobraźmy sobie aplikację **Tasks** (lista zadań do zrobienia) z backendem REST. Mamy następujące funkcjonalności:

- **Logowanie użytkownika** (uzyskanie tokena – załóżmy, że już jest zalogowany i mamy token).
- **Pobieranie listy zadań** – endpoint GET /tasks, zwracający listę zadań (paginowaną).
- **Dodawanie nowego zadania** – endpoint POST /tasks (zwraca utworzone zadanie).
- **Edycja zadania** – endpoint PUT /tasks/:id (zwraca zaktualizowane zadanie).
- **Paginacja** – zakładamy, że lista zadań może być długa, więc API paginuje wyniki (np. parametry ?page=).
- **Autoryzacja** – wszystkie powyższe wymagają tokena Bearer w nagłówku.

Spróbujmy zbudować uproszczoną architekturę:

Krok 1: Konfiguracja API i React Query

```
// api.ts - konfiguracja Axios i tokenów
import axios from 'axios';
import * as SecureStore from 'expo-secure-store';

const API_URL = 'https://example.com/api';

export const api = axios.create({
  baseURL: API_URL,
  timeout: 5000,
});

// Wstaw token do każdego żądania (jeśli jest)
api.interceptors.request.use(async config => {
  const token = await SecureStore.getItemAsync('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

// (Opcjonalnie) Interceptor odpowiedzi do obsługi odświeżania tokena/błędów 401:
api.interceptors.response.use(
  response => response,
  async error => {
    const originalRequest = error.config;
    if (error.response?.status === 401 && !originalRequest._retry) {
      originalRequest._retry = true;
      // Zakładamy istnienie funkcji refreshToken() która pobierze i zapisze nowy token
      try {
        const newToken = await refreshToken();
        originalRequest.headers.Authorization = `Bearer ${newToken}`;
        return api(originalRequest); // ponów oryginalne żądanie z nowym tokenem
      } catch (e) {
        // Refresh nie powiódł się - można wylogować użytkownika
        // navigateToLoginScreen();
        return Promise.reject(e);
      }
    }
    return Promise.reject(error);
  }
);
```

Powyżej:

- Tworzymy axios z baseURL.
- Request interceptor dołącza token (pobierany z SecureStore za każdym żądaniem – to może być nieco nieefektywne, lepiej byłoby trzymać token w zmiennej, ale dla pewności używamy storage).
- Response interceptor sprawdza 401. Używamy tu uproszczonej logiki: sprawdzamy flagę `_retry` by nie wpaść w pętlę. Wołamy `refreshToken()` – to powinna być funkcja

wykonująca np. api.post('/auth/refresh') (musielibyśmy uważać by nie wpaść ponownie w interceptor – można użyć innej instancji axios bez interceptorów do tego). Po uzyskaniu nowego tokena, zapisujemy go (tu zakładamy, że refreshToken() już to robi i zwraca accessToken) i ponawiamy oryginalne żądanie. Jeśli się nie uda, odrzucamy błąd i pewnie w aplikacji przechwycimy to by wylogować.

Krok 2: Hooki korzystające z React Query do zapytań:

Aby uporządkować, zrobimy własne hooki do obsługi zadań:

```
// tasksApi.ts - funkcje API do tasks
export const fetchTasksPage = async ({ pageParam = 1 }) => {
  const res = await api.get(`/tasks?page=${pageParam}`);
  return res.data;
};
export const addTaskRequest = async (newTask) => {
  const res = await api.post('/tasks', newTask);
  return res.data;
};
export const updateTaskRequest = async ({ id, updates }) => {
  const res = await api.put(`/tasks/${id}`, updates);
  return res.data;
};

// tasksHooks.ts - hooki używające powyższych funkcji z React Query
import { useInfiniteQuery, useMutation, useQueryClient } from '@tanstack/react-query';
import { fetchTasksPage, addTaskRequest, updateTaskRequest } from './tasksApi';

export function useTasksList() {
  return useInfiniteQuery({
    queryKey: ['tasks'],
    queryFn: fetchTasksPage,
    getNextPageParam: (lastPage) => lastPage.nextPage ?? undefined
  });
}

export function useAddTask() {
  const queryClient = useQueryClient();
  return useMutation({
    mutationFn: addTaskRequest,
    onMutate: async (newTask) => {
      // Optymistycznie dodaj nową task do cache:
      await queryClient.cancelQueries({ queryKey: ['tasks'] });
      const prevData = queryClient.getQueryData(['tasks']);
      if (prevData) {
        queryClient.setQueryData(['tasks'], (oldData) => {
          const newTaskObj = { ...newTask, id: Math.random().toString(36) }; // tymczasowe ID
          // Wstaw nowy task na początek pierwszej strony (załóżmy)
          return {
            ...oldData,
            pages: [
              [ newTaskObj, ...oldData.pages[0] ],
              ...oldData.pages.slice(1)
            ]
          };
```

```
      });
    }
    return { prevData };
  },
  onError: (err, newTask, context) => {
    // cofnij zmiany przy błędzie
    if (context?.prevData) {
      queryClient.setQueryData(['tasks'], context.prevData);
    }
  },
  onSettled: () => {
    // niezależnie od wyniku - odśwież listę zadań z serwera
    queryClient.invalidateQueries({ queryKey: ['tasks'] });
  }
 });
}

export function useUpdateTask() {
 const queryClient = useQueryClient();
 return useMutation({
  mutationFn: updateTaskRequest,
  onMutate: async ({ id, updates }) => {
    await queryClient.cancelQueries({ queryKey: ['tasks'] });
    const prevData = queryClient.getQueryData(['tasks']);
    if (prevData) {
      queryClient.setQueryData(['tasks'], (oldData) => {
        // Zaktualizuj task lokalnie
        return {
          ...oldData,
          pages: oldData.pages.map(page =>
            page.map(task => task.id === id ? { ...task, ...updates } : task)
          )
        };
      });
    }
    return { prevData };
  },
  onError: (err, vars, context) => {
    if (context?.prevData) {
      queryClient.setQueryData(['tasks'], context.prevData);
    }
  },
  onSettled: () => {
    queryClient.invalidateQueries({ queryKey: ['tasks'] });
  }
 });
}
```

Co tu się dzieje:

- Mamy `useTasksList()` korzystający z `useInfiniteQuery` dla listy. Używa `fetchTasksPage` (który robi GET /tasks?page=). Zakładamy, że serwer zwraca np. `{ tasks: [...], nextPage: 2 }` albo `nextPage: null`. Funkcja `getNextPageParam` pobiera `lastPage.nextPage`.
- Mutacja `useAddTask`:

    - **onMutate:** Anulujemy bieżące zapytania tasks, pobieramy poprzednie dane. Następnie, jeśli były jakieś dane (cache istnieje), wstawiamy optymistycznie nowy task. Tutaj dla uproszczenia generujemy tymczasowe id (losowy string) i dodajemy nowy obiekt na **początek** pierwszej strony listy (przyjęliśmy, że chcemy go na początku listy). Struktura cache w infiniteQuery to obiekt z pages, więc wstawiamy do oldData.pages[0]. Zwracamy previousData, by móc odrollować.
    - **onError:** Przy błędzie przywracamy prevData jeśli było.
    - **onSettled:** Po wszystkim (sukces lub błąd), invalidujemy tasks – serwerowy stan ma być prawdą ostateczną.
- Mutacja useUpdateTask:
    - onMutate: podobnie anulujemy, pobieramy prevData. Następnie iterujemy po wszystkich stronach (oldData.pages.map), i każdą listę task w page mapujemy – jeśli task.id === updatedId, to tworzymy nowy obiekt z zastosowanymi updates (np. { completed: true }). To natychmiast zmieni UI – użytkownik zobaczy np. że zadanie oznaczyło się jako ukończone. Zwracamy prevData.
    - onError: Przy błędzie przywracamy.
    - onSettled: invalidacja.

To podejście powoduje minimalne odczuwalne opóźnienie dla użytkownika – większość akcji wydaje się natychmiastowa.

Krok 3: Komponenty UI korzystające z tych hooków:

Zakładamy, że mechanizm logowania jest już za nami i mamy token (inaczej należałoby dodać ekran logowania i logikę do zapisu tokena w SecureStore po pomyślnym logowaniu itp.).

Dla listy zadań z infinite scroll:

```
// TasksListScreen.tsx
import { useTasksList } from './tasksHooks';

export function TasksListScreen({ navigation }) {
  const {
    data,
    isLoading,
    isError,
    error,
    fetchNextPage,
    hasNextPage,
    isFetchingNextPage,
    refetch
  } = useTasksList();

  if (isLoading) {
    return <ActivityIndicator size="large" color="#000" style={{ flex: 1, justifyContent: 'center' }} />;
  }
  if (isError) {
    return (
      <View style={{ padding: 20 }}>
```

```
      <Text style={{ color: 'red' }}>Błąd ładowania listy zadań: {error.message}</Text>
      <Button title="Spróbuj ponownie" onPress={refetch} />
    </View>
   );
 }

 const tasks = data.pages.flatMap(pageData => pageData.tasks); // załóżmy, że API zwraca {tasks: [], nextPage:
...}

 return (
  <FlatList
    data={tasks}
    keyExtractor={task => task.id.toString()}
    renderItem={({ item }) => (
     <TaskListItem task={item} onPress={() => navigation.navigate('TaskDetails', { id: item.id })} />
    )}
    onEndReached={() => { if (hasNextPage) fetchNextPage(); }}
    onEndReachedThreshold={0.5}
    ListFooterComponent={ isFetchingNextPage ? <ActivityIndicator /> : null }
    refreshing={ isLoading }    {/* ewentualnie obsługa pull-to-refresh */}
    onRefresh={ refetch }
  />
 );
}
```

Tutaj:

- Korzystamy z useTasksList(). Gdy ładuje pierwszą stronę (isLoading true) pokazujemy duży spinner centrycznie.
- Jeśli błąd (isError), pokazujemy komunikat i przycisk, który wywoła refetch (ponowne pobranie).
- Gdy dane są, łączymy wszystkie strony w jedną tablicę tasks.
- FlatList wyświetla każdy task poprzez komponent TaskListItem (to może być np. element listy z checkboxem ukończenia, itd. – szczegóły stylizacji pomijamy).
- onEndReached logic jak omawialiśmy.
- refreshing/onRefresh umożliwia także odświeżenie gestem – użyliśmy isLoading co tutaj przy powrocie na ekran mogłoby być false jeśli dane w cache. Ewentualnie lepiej: refreshing={isFetching && data} – czyli jeśli trwa fetch po pierwszym załadowaniu.
- (Założyliśmy, że API zwraca obiekt z kluczem tasks i nextPage. W naszym tasksApi funkcjach powinniśmy to ewentualnie dostosować, ale to szczegół implementacji.)

Komponent dodawania/edycji zadania:

Załóżmy, że mamy ekran dodawania nowego zadania oraz edycji istniejącego zadania (np. zmiana tytułu lub oznaczenie ukończenia). Możemy to rozwiązać różnie – np. ekran "TaskDetails" z opcją edycji. Dla uproszczenia pokażemy tylko jak by wyglądało użycie hooków useAddTask i useUpdateTask:

```
// NewTaskScreen.tsx
export function NewTaskScreen({ navigation }) {
 const { mutate: addTask, isLoading: isAdding } = useAddTask();
 const [title, setTitle] = useState('');
```

```
  const handleSave = () => {
    addTask(
      { title, completed: false },
      {
        onSuccess: () => {
          navigation.goBack(); // po dodaniu wróć do listy
        }
      }
    );
  };

  return (
    <View style={{ padding: 20 }}>
      <Text>Nowe zadanie:</Text>
      <TextInput value={title} onChangeText={setTitle} placeholder="Tytuł zadania" />
      {isAdding && <ActivityIndicator />}
      <Button title="Dodaj" onPress={handleSave} disabled={isAdding || !title} />
    </View>
  );
}
```

Tutaj `useAddTask` jest użyty. Gdy użytkownik kliknie "Dodaj":

- Wywołujemy `addTask({ title, completed: false }, { onSuccess: ... })`. Można w `onSuccess` przekazać callback bezpośrednio w wywołaniu `mutate` jeśli chcemy wykonać akcję po sukcesie (alternatywa do definiowania onSuccess w samym hooku, co i tak zrobiliśmy dla invalidacji). Tutaj po sukcesie wracamy do poprzedniego ekranu.
- `isAdding` pozwala pokazać spinner lub zablokować przycisk by nie dodać wiele razy.
- Dzięki optymistycznej aktualizacji w onMutate listy, użytkownik wracając do listy od razu zobaczy nowy wpis (półprzezroczysty by sygnalizować, że jeszcze się zapisuje – to by trzeba dodać warunkowo styl w TaskListItem np. jeśli id jest tymczasowe albo można bazować na `useAddTask` state tak jak w przykładzie UI optimistic wcześniej). W naszym kodzie powyżej daliśmy ActivityIndicator dla `isAdding` elementu – to uproszczony przykład, w praktyce wolelibyśmy np. przekazać jakiś flagę lub użyć `variables` z useMutation.

Ekran edycji (np. oznacz jako ukończone):

```
// TaskDetailsScreen.tsx
export function TaskDetailsScreen({ route, navigation }) {
  const { id } = route.params;
  const { mutate: updateTask, isLoading: isUpdating } = useUpdateTask();
  const task = // ... załóżmy że przekazaliśmy obiekt task w nawigacji lub mamy useQuery do pojedynczego taska

  const toggleComplete = () => {
    updateTask({ id, updates: { completed: !task.completed } }, {
      onSuccess: () => {
        // można nawiguować z powrotem lub powiadomić
      }
    });
  };
```

```
  return (
    <View>
      <Text>{task.title}</Text>
      <Text>Status: {task.completed ? 'ukończone' : 'do zrobienia'}</Text>
      {isUpdating && <ActivityIndicator />}
      <Button title={task.completed ? "Oznacz jako nieukończone" : "Oznacz jako ukończone"}
            onPress={toggleComplete} disabled={isUpdating} />
    </View>
  );
}
```

Tutaj useUpdateTask zadziała optymistycznie – natychmiast zmieni status na liście (jeśli lista jest wyświetlana gdzieś, np. poprzedni ekran), bo onMutate zaktualizuje cache. Gdy wrócimy na listę (jeśli używamy react-navigation stack, pewnie lista już została zaktualizowana w tle dzięki temu), użytkownik widzi od razu zmianę.

Obsługa tokenów: dzięki interceptorowi, nasze żądania api.get, api.post automatycznie dołączają token. Gdyby token był nieaktualny, interceptor odświeżania spróbuje go naprawić.

Globalny błąd: jeśli odświeżenie się nie powiedzie i interceptor wyrzuci błąd 401, to możemy przechwycić to np. globalnym error boundary lub w nawigacji wykryć i przenieść użytkownika do ekranu logowania. Można też sprawdzić w onError mutacji czy błąd ma status 401 i np. wyświetlić modal "Sesja wygasła".

Brak sieci: Warto dodać nasłuchiwanie NetInfo i np. gdy isConnected === false, to:

- Zablokować przyciski dodawania/edycji (lub dać komunikat "Offline - nie można zsynchronizować zmian").
- Pokazać baner offline jak wcześniej wspomniano.

React Query wraz z onlineManager może też spowodować, że kiedy sieć wróci, automatycznie odświeży dane lub wykona zaległe mutacje (choć mutacje offline domyślnie od razu zwracają błąd – TanStack Query ma osobne pluginy do *persistowania mutacji offline*, ale to zaawansowane zagadnienie).

Debugowanie: Podczas tworzenia takiej aplikacji:

- Korzystaj z logów (console.log) wewnątrz onError, by sprawdzić co poszło nie tak.
- Sprawdzaj w Flipperze czy zapytania faktycznie wychodzą, jakie URL i statusy wracają.
- Użyj trybu debug w React Query – np. enableDevTools() w RN nie jest automatyczne, ale można skorzystać z **Flipper pluginu do React Query** lub Reactotron plugin. Wspomniany plugin Flipper (react-query-native-devtools) pozwala podejrzeć aktywne query, cache, etc., co może być ogromnie pomocne w zrozumieniu co się dzieje w React Query. W razie problemów z cache, można też wywołać queryClient.getQueryData(['tasks']) w konsoli debuggera, by zobaczyć co jest w pamięci.

Podsumowanie: Wykorzystując **React Query** w React Native możemy znacznie uprościć pracę z siecią. Dostajemy automatyczną obsługę stanów zapytań, cache danych, odświeżanie, a także mechanizmy zaawansowane jak optymistyczne aktualizacje czy infinite scroll, które w czystym podejściu wymagałyby dużo kodu. Połączenie tego z biblioteką **axios** (lub natywnym fetch) pozwala pokryć większość potrzeb: globalne nagłówki, obsługa tokenów przez interceptory i bezpieczne przechowywanie poświadczeń. Pamiętajmy o obsłudze błędów – zarówno tych od API (komunikaty dla użytkownika), jak i wyjątków JS (Error Boundaries). Dzięki temu aplikacja będzie reagować w czytelny sposób nawet w sytuacjach problemowych, co poprawi doświadczenie użytkownika.

Literatura:

1. https://tanstack.com/query/latest/docs/framework/react/overview (Data dostępu: 1.10.2025) – Oficjalna dokumentacja TanStack Query (React Query), opisująca zasady zarządzania stanem danych asynchronicznych.
2. https://axios-http.com/docs/intro (Data dostępu: 1.10.2025) – Dokumentacja biblioteki Axios, omawiająca konfigurację instancji, interceptory oraz automatyczne przetwarzanie JSON.
3. https://reactnative.dev/docs/network (Data dostępu: 1.10.2025) – Oficjalny przewodnik React Native dotyczący wykonywania zapytań sieciowych za pomocą Fetch API.
4. https://docs.expo.dev/versions/latest/sdk/securestore/ (Data dostępu: 1.10.2025) – Dokumentacja modułu Expo SecureStore, służącego do bezpiecznego przechowywania tokenów uwierzytelniających.
5. https://github.com/react-native-netinfo/react-native-netinfo (Data dostępu: 1.10.2025) – Dokumentacja biblioteki NetInfo, niezbędnej do monitorowania stanu połączenia sieciowego w aplikacjach mobilnych.