POLITECHNIKA ŚWIĘTOKRZYSKA

Aplikacje mobilne – wykład 3

Nawigacja i przepływ użytkownika – React Navigation w React Native

Mateusz Pawełkiewicz

1.10.2025

Wprowadzenie: W aplikacjach mobilnych React Native nawigacja między ekranami odgrywa kluczową rolę w tzw. *user flow* (przepływie użytkownika). Biblioteką de facto standardową do nawigacji jest **React Navigation**, oferująca wygodne API do tras (routingu) na iOS, Androidzie, a nawet na web. W tym wykładzie omówimy **typy nawigatorów** (stos, zakładki, szuflada), ich zagnieżdżanie, przekazywanie parametrów między ekranami, a także bardziej zaawansowane zagadnienia: **hooki nawigacyjne**, integrację z **TypeScript**, mechanizmy **deep linków**, ochronę tras (protected routes) w kontekście uwierzytelniania oraz kompletny **przepływ logowania**.

React Navigation – typy nawigacji

React Navigation to biblioteka umożliwiająca definiowanie różnych rodzajów nawigacji w aplikacji React Native. Podstawowe typy nawigatorów to: **Stack Navigator (stos ekranów)**, **Bottom Tab Navigator (nawigacja z zakładkami)** oraz **Drawer Navigator (nawigacja szufladkowa)**. Każdy z nich odpowiada innemu wzorcowi poruszania się po aplikacji:

- **Stack Navigator (stos):** Umożliwia przechodzenie do kolejnych ekranów układając je na stosie – nowy ekran jest nakładany na poprzedni, a powrót cofa do wcześniejszych ekranów. Jest to analogia do nawigacji w iOS/Android: ekrany wchodzą z prawej strony lub w domyślnym stylu systemowym. Domyślnie React Navigation zapewnia animacje przejścia zgodne z platformą (przesunięcie w poziomie na iOS, standardowy fade/slide na Androidzie). Implementacja stosu występuje w dwóch odmianach: **@react-navigation/stack** (w JavaScript) oraz **@react-navigation/native-stack** (wykorzystująca natywne API nawigacji). Wersja JavaScript jest bardziej konfigurowalna, ale może być nieco mniej wydajna, dlatego w przypadku złożonych animacji warto rozważyć wariant natywny dla lepszej płynności. Stack Navigator sprawdza się w sekwencyjnych przepływach ekranów – np. lista artykułów → ekran szczegółów artykułu.
- **Bottom Tab Navigator (zakładki):** Zapewnia nawigację za pomocą paska kart (tabs) najczęściej umieszczonego na dole ekranu. Użytkownik może przełączać się między zakładkami reprezentującymi niezależne sekcje aplikacji (np. *Home*, *Wyszukaj*, *Profil*). Każda zakładka ma przypisany własny ekran (lub **zagnieżdżony stos** ekranów). **Taby** są inicjalizowane leniwie (ekran zakładki ładuje się dopiero przy pierwszym przejściu) i utrzymywane w pamięci, co umożliwia szybki powrót bez utraty stanu. Możemy dostosować ikonki i etykiety zakładek, a także styl paska. Ten nawigator jest idealny, gdy aplikacja ma kilka głównych modułów dostępnych równolegle (np. strona główna i ekran ustawień jako osobne zakładki).
- **Drawer Navigator (szuflada):** Oferuje nawigację z menu wysuwanym z boku ekranu (tzw. hamburger menu). Zazwyczaj na Androidzie otwierane gestem przesunięcia od krawędzi, a na iOS dodatkowo często ikoną hamburgera w nagłówku. Drawer Navigator jest użyteczny do pomieszczenia wielu opcji nawigacyjnych lub nawigacji globalnej, np. panel boczny aplikacji z linkami do różnych ekranów (profil, ustawienia, FAQ itp.). W React Navigation implementacja szuflady opiera się pod spodem na komponencie react-native-drawer-layout, a do poprawnego działania wymaga zainstalowania zależności jak **react-native-gesture-handler** i **react-native-reanimated** (Expo instaluje je automatycznie). Konfigurując Drawer Navigator definiujemy ekrany dostępne w menu; możemy także dostosować pozycję szuflady (lewa/prawa) czy

zawartość nagłówka. Drawer sprawdza się, gdy chcemy ukryć nawigację pod gestem, zostawiając więcej miejsca na ekranie głównym.

Zagnieżdżanie nawigatorów (nesting): Często w realnej aplikacji używamy kilku typów nawigacji jednocześnie, zagnieżdżając je w sobie. Przykładowo, możemy mieć główny **Drawer Navigator**, a w nim każda opcja otwiera osobny **Stack** lub **Tab** navigator. Albo popularny przypadek: **Tab Navigator** jako główna nawigacja aplikacji, gdzie poszczególne zakładki mają wewnętrzne Stack Navigatory (np. zakładka *Home* ma stos z ekranem głównym i ekranem szczegółów). React Navigation pozwala traktować navigator jak ekran – np. możemy zdefiniować w Stack Navigatorze ekran, którego komponentem jest MyTabs (nasz Tab Navigator). W ten sposób nawigatory są hierarchiczne. Należy pamiętać, że **każdy nawigator ma własną przestrzeń nazw dla routów** – tzn. ekrany w zagnieżdżonym navigatorze mają swoje nazwy niezależne od nazw ekranów w rodzicu. Aby nawigować pomiędzy zagnieżdżonymi navigatorami, zazwyczaj wywołujemy nawigację względem wspólnego **NavigationContainer** lub używamy pełnych ścieżek (np. navigation.navigate('NazwaNavigatora', { screen: 'NazwaEkranu', params: {...} })). Zagnieżdżanie nawigatorów pozwala łączyć różne wzorce nawigacji – np. zakładki dolne plus opcje w szufladzie, albo stos logowania oddzielony od głównego stosu aplikacji.

Przekazywanie parametrów między ekranami: Często przy przechodzeniu na kolejny ekran chcemy przekazać mu dane (np. ID obiektu, który ma wyświetlić szczegóły). React Navigation umożliwia to poprzez drugi argument funkcji nawigującej. Przykładowo, mając nawigację stosu, możemy przejść do ekranu "Details" z parametrami:

```
navigation.navigate('Details', {
  itemId: 86,
  otherParam: 'dowolna wartość'
});
```

W powyższym wywołaniu przekazaliśmy obiekt parametrów do routa "Details". Na ekranie docelowym dostęp do tych danych uzyskamy przez właściwość route.params. Dla przykładu, w komponencie *DetailsScreen* możemy pobrać parametry:

```
function DetailsScreen({ route }) {
  const { itemId, otherParam } = route.params;
  // użycie itemId i otherParam...
}
```

Pobraliśmy itemId i otherParam z obiektu parametrów przekazanych nawigacją. Warto zauważyć, że parametry najlepiej przekazywać w formie prostego JSON (typy proste, obiekty, tablice) – dzięki temu są **serializowalne**, co ułatwia np. zapisywanie stanu nawigacji czy obsługę deep linków.

Możemy także definiować **parametry początkowe** ekranu niezależnie od nawigacji. Jeśli np. chcemy, by ekran *Details* domyślnie miał itemId: 42, możemy ustawić initialParams przy definicji ekranu:

```
<Stack.Screen
  name="Details"
```

```
  component={DetailsScreen}
  initialParams={{ itemId: 42 }}
/>
```

Parametry przekazane podczas navigate nadpiszą wartości początkowe, a jeśli nie przekażemy żadnych – ekran użyje initialParams. Ekran może również **aktualizować parametry** własnej routy w trakcie działania, wywołując navigation.setParams({ … }) – np. aby zmienić parametry wpływające na UI nagłówka itp. (przydatne przy dynamicznych tytułach).

Kilka dobrych praktyk przy przekazywaniu parametrów:

- Przekazuj tylko niezbędne dane (np. ID, klucz), a większe obiekty pobieraj ponownie na ekranie docelowym lub użyj globalnego stanu. Unikniesz w ten sposób problemów z serializacją i wydzielisz odpowiedzialności.
- Upewnij się, że ekran odbierający parametry obsłuży przypadek braku parametru (np. gdyby nawigacja nastąpiła bez niego). Możesz zdefiniować wartości domyślne: route.params?.userName ?? 'Anonim' itp.
- W TypeScript warto **uściślić typy parametrów** dla każdej trasy – dzięki temu edytor wychwyci brak wymaganych parametrów już na etapie kompilacji. Typowanie nawigacji omówimy w dalszej części.

Uwaga: W Stack Navigatorze navigation.navigate('RouteName', params) zachowuje się nieco inaczej niż navigation.push('RouteName', params). navigate spróbuje **znaleźć istniejący** ekran o danej nazwie w stosie i odświeżyć jego parametry (lub przejść do niego, jeśli jest niżej na stosie). Jeśli taki ekran nie istnieje, dopiero wtedy wstawi nowy na górę stosu. Natomiast push **zawsze dodaje nowy ekran** na wierzch, nawet jeśli już taki znajduje się na stosie. Dlatego gdy chcemy wejść na ten sam ekran wielokrotnie (np. oglądać różne szczegóły w pętli), używamy push. Gdy chcemy przejść do ekranu bez duplikowania (np. z menu do istniejącego już stosu), lepsze jest navigate. Istnieje też navigation.replace('RouteName', params) – zastępuje bieżący ekran nowym, co bywa użyteczne np. po zakończeniu onboardingu (usuwamy ekran powitalny ze stosu i wstawiamy główny ekran aplikacji).

Hooki nawigacyjne i nawigacja w praktyce

React Navigation dostarcza specjalne **hooki**, które ułatwiają korzystanie z nawigacji w komponentach funkcyjnych. Główne to **useNavigation** oraz **useRoute**. Omówimy ich działanie, a także pokażemy, jak używać ich z TypeScriptem dla pełnego bezpieczeństwa typów. Przyjrzymy się też metodom nawigacji wstecz i resetowania stosu – niezbędnym w zarządzaniu historią ekranów.

useNavigation – dostęp do obiektu nawigacji

useNavigation() zwraca obiekt nawigacji (*navigation prop*) dla ekranu, w kontekście którego hook został wywołany. Dzięki temu możemy wołać navigation.navigate, navigation.goBack() i inne metody bez przekazywania obiektu nawigacji przez propsy. Jest to przydatne np. w komponentach zagnieżdżonych, które nie są ekranami, ale chcą nawigować (np. przycisk w customowym headerze).

Przykład: Załóżmy, że chcemy stworzyć komponent przycisku “Wstecz” działający w dowolnym miejscu aplikacji:

```
import { useNavigation } from '@react-navigation/native';

function MyBackButton() {
  const navigation = useNavigation();

  return (
    <Button title="Powrót" onPress={() => navigation.goBack()} />
  );
}
```

Tutaj hook useNavigation() dostarcza nam aktualny obiekt nawigacji, a my wywołujemy navigation.goBack() po kliknięciu. Naciśnięcie przycisku spowoduje cofnięcie do poprzedniego ekranu na stosie (tożsame z użyciem gestu/cofnięcia systemowego).

Użycie z klasami: Jeśli musimy skorzystać z useNavigation w komponencie klasowym (który nie obsługuje hooków), można owinąć komponent klasowy w funkcję korzystającą z hooka i przekazać navigation jako prop:

```
class MyLegacyComponent extends React.Component {
  render() {
    const { navigation } = this.props;
    // ...
  }
}

// eksport zastępujemy wersją z przekazanym navigation:
export default function(props) {
  const navigation = useNavigation();
  return <MyLegacyComponent {...props} navigation={navigation} />;
}
```

Generalnie jednak w nowych aplikacjach trzymamy się komponentów funkcyjnych i hooków.

Typowanie useNavigation: Domyślnie, bez dodatkowej konfiguracji, obiekt zwracany przez useNavigation ma ogólny typ (nie zawiera informacji o dostępnych routach i parametrach). Aby mieć lepsze podpowiedzi i kontrolę, możemy skorzystać z typu generycznego. Przykładowo, jeśli mamy zdefiniowany typ parametrów nawigatora głównego RootStackParamList (o definiowaniu typów za chwilę), możemy zrobić:

```
type NavProp = NativeStackNavigationProp<RootStackParamList, 'Profile'>;
const navigation = useNavigation<NavProp>();
```

Wtedy np. navigation.navigate('Home', …) będzie sprawdzać zgodność nazwy routy i typów parametrów z RootStackParamList. Innym podejściem (w React Navigation 6+) jest **zadeklarowanie globalnego typu param list** – wtedy useNavigation() automatycznie użyje go bez konieczności przekazywania generyka. Szczegóły tego podejścia omówimy w sekcji o TypeScript.

Uwaga: useNavigation **musi być** wywoływany w kontekście ekranu podpiętego do nawigatora (tj. wewnątrz drzewa NavigationContainer). Jeśli użyjemy go poza nawigatorem, otrzymamy błąd o braku kontekstu. W praktyce oznacza to, że np. nie możemy wywołać useNavigation w kodzie, który renderuje NavigationContainer – tylko wewnątrz komponentów będących ekranami lub ich dziećmi. Jeśli potrzebujemy nawigować globalnie spoza komponentów (np. z poziomu modułu, serwisu) – React Navigation oferuje obiekt **navigation ref** (React.createRef()), który możemy ręcznie wykorzystywać do nawigacji imperatywnej, ale to zaawansowany przypadek.

useRoute – informacje o bieżącej trasie

useRoute() pozwala nam uzyskać obiekt aktualnej trasy (route) w danym ekranie. Standardowo komponent-ekran otrzymuje route w propsach ({ route, navigation }), ale jeśli zagnieżdżamy logikę głębiej lub chcemy skorzystać z hooka zamiast props, useRoute rozwiązuje problem.

Typowy use-case to pobranie parametrów przekazanych do ekranu lub odczytanie nazwy trasy. Przykład użycia:

```
import { useRoute } from '@react-navigation/native';

function MyText() {
  const route = useRoute();
  return <Text>{route.name}: {JSON.stringify(route.params)}</Text>;
}
```

W powyższym przykładzie wyświetlamy nazwę aktualnego ekranu (route.name) oraz parametry w formie tekstowej. Oczywiście zazwyczaj interesuje nas konkretne pole z route.params – np. route.params.userId. *useRoute* jest szczególnie przydatny w komponencie, który nie otrzymuje props route (np. wnuk ekranu), a potrzebuje tych danych.

Typowanie useRoute: Podobnie jak z useNavigation, warto zapewnić odpowiedni typ zwracanego obiektu. React Navigation udostępnia generyk RouteProp<ParamList, RouteName>. Jeśli np. mamy:

```
type RootStackParamList = { Details: { itemId: number } };
type DetailsRouteProp = RouteProp<RootStackParamList, 'Details'>;
```

to możemy użyć:

```
const route = useRoute<DetailsRouteProp>();
```

Wówczas TypeScript będzie wiedział, że route.params ma strukturę { itemId: number } – dzięki czemu od razu złapie literówki w nazwach pól czy niewłaściwe typy.

Typowanie nawigacji w TypeScript

Aby w pełni wykorzystać moc TypeScript w nawigacji, powinniśmy **zdefiniować typy parametrów dla wszystkich ekranów** oraz używać ich przy tworzeniu navigatorów i hooków. Proces wygląda następująco:

1. **Definicja listy parametrów** – tworzymy typ obiektu, w którym kluczami są nazwy tras, a wartościami typy parametrów (lub undefined, jeśli brak parametru). Przykład dla prostego stosu:

```
type AuthStackParamList = {
  Login: undefined;
  Register: undefined;
  ForgotPassword: { email?: string }; // przykładowo ekran resetu hasła z opcjonalnym e-mailem
};
```

   oraz dla głównej części aplikacji z zakładkami:

```
type AppTabsParamList = {
  Home: undefined;
  Profile: { userId: string };
};
```

   Tutaj zakładamy, że ekran Home nie potrzebuje parametru, a ekran Profile wymaga userId (np. identyfikatora użytkownika, którego profil wyświetlamy).

2. **Tworzenie navigatora z powyższym typem** – przy wywołaniu fabryki navigatora przekazujemy typ param list jako parametr generyczny. Np. dla stack:

```
const Stack = createNativeStackNavigator<AuthStackParamList>();
```

   a dla tabs:

```
const Tab = createBottomTabNavigator<AppTabsParamList>();
```

   Teraz komponenty <Stack.Screen> i <Tab.Screen> będą oczekiwać nazwy oraz komponentu zgodnych z zdefiniowanymi w typie trasami.

3. **Typowanie propsów ekranów:** Możemy uzyskać typy nawigacji i routy dla konkretnej trasy używając dostarczonych typów utility. Przykładowo, dla ekranu Profil w tab navigatorze:

```
import type { CompositeScreenProps } from '@react-navigation/native';
import type { BottomTabScreenProps } from '@react-navigation/bottom-tabs';

type ProfileScreenProps = BottomTabScreenProps<AppTabsParamList, 'Profile'>;
```

   W przypadku bardziej złożonym, gdy ekran jest zagnieżdżony (np. ekran w stacku wewnątrz taba), używa się CompositeScreenProps łącząc typ stacku i taba. Dla

uproszczenia, można też skorzystać z hooków z generykami zamiast typować cały komponent.

W praktyce często robi się to tak:

```
function ProfileScreen() {
  const navigation = useNavigation<BottomTabNavigationProp<AppTabsParamList, 'Profile'>>();
  const route = useRoute<RouteProp<AppTabsParamList, 'Profile'>>();
  // teraz navigation i route są ściśle typowane
  const { userId } = route.params;
  ...
}
```

Takie podejście daje pełne bezpieczeństwo: jeśli spróbujemy navigation.navigate('Home', {userId: '123'}) ale Home nie przyjmuje parametru, TS zgłosi błąd. Lub odwrotnie – wywołanie navigation.navigate('Profile') bez wymaganego userId będzie błędne.

4. **Globalny typ param list:** React Navigation v6+ umożliwia zadeklarowanie w przestrzeni globalnej (namespace ReactNavigation) domyślnego typu param list. Wystarczy zrobić coś takiego w pliku definicji (np. types.d.ts):

```
declare global {
  namespace ReactNavigation {
    interface RootParamList extends AppTabsParamList {}
  }
}
```

Po tej deklaracji, wszędzie tam gdzie React Navigation używa RootParamList (np. w useNavigation() bez generyka), zostanie zastosowany nasz typ. To upraszcza korzystanie z hooków – nie musimy podawać generyków za każdym razem. Upewnijmy się tylko, że deklaracja jest w zasięgu projektu (włączona przez TSconfig).

Podsumowanie: Konfiguracja TypeScript z nawigacją wymaga odrobiny pracy na starcie, ale opłaca się. Zyskujemy autouzupełnianie nazw ekranów, pewność co do parametrów i ich typów, mniej błędów w czasie wykonania. W dużych aplikacjach to niemal konieczność. React Navigation w wersji 7 dodatkowo wprowadził tzw. *Static API*, które upraszcza konfigurację navigatora i poprawia inferencję typów (definiujemy ekrany jako obiekty z kluczami, co ułatwia wyprowadzenie typów param list). Niezależnie od podejścia, warto zawsze definiować param listy i korzystać z dostarczonych typów pomocniczych.

Powrót do poprzedniego ekranu i resetowanie stosu

Nawigacja to nie tylko przechodzenie “do przodu” (push/navigate), ale też wracanie i zarządzanie historią ekranów. Poniżej najważniejsze operacje:

- **Powrót (goBack):** Każdy obiekt navigation posiada metodę navigation.goBack(), która cofa nas o jeden ekran w tył (analogicznie do naciśnięcia systemowego back na Androidzie czy gestu cofania na iOS). Jeśli jesteśmy na pierwszym ekranie stosu, wywołanie goBack() domyślnie zamknie całą aplikację (lub przeniesie ją w tło) na Androidzie – warto to mieć na uwadze przy obsłudze hardware back (można ten behavior

nadpisać, o czym za chwilę). W naszym wcześniej pokazanym *MyBackButton* użyliśmy navigation.goBack(). Istnieje też metoda navigation.canGoBack(), którą można sprawdzić czy jest do czego wracać (zwraca true/false).

- **Pop stack:** Alternatywą do goBack jest navigation.pop(n), która cofa o *n* ekranów (domywnie 1). Ponadto navigation.popToTop() cofnie nas od razu na początek stosu (pierwszy ekran). Te metody są przydatne np. aby wyjść z wieloetapowego formularza od razu na ekran główny itp.
- **Reset nawigacji (reset stack):** Czasem chcemy całkowicie zmienić stan nawigacji – np. wyczyścić historię i wstawić nowy zestaw ekranów. Przykładem jest przepływ logowania: po pomyślnym zalogowaniu nie chcemy, by użytkownik wrócił przyciskiem "wstecz" do ekranu logowania. W takim wypadku można zastosować **reset stosu**. React Navigation udostępnia akcję CommonActions.reset do zdefiniowania nowego stanu nawigacji. Używa się jej mniej więcej tak:

```
import { CommonActions } from '@react-navigation/native';

navigation.dispatch(
  CommonActions.reset({
    index: 0,
    routes: [ { name: 'Home' } ]  // nowy stos z tylko jednym ekranem Home
  })
);
```

Powyższy kod wyczyści całą historię i ustawi stos zawierający pojedynczą trasę Home. Możemy oczywiście podać więcej routes w tablicy, określając nawet parametry każdej z nich. Np.:

```
CommonActions.reset({
  index: 1,
  routes: [
    { name: 'Profile', params: { user: 'janek' } },
    { name: 'Home' }
  ]
})
```

Tutaj ustawiamy, że nowy stos ma dwa ekrany: *Profile* (będzie pod indeksem 0) i *Home* (indeks 1), i od razu startujemy na indeksie 1 czyli Home. Efekt: użytkownik zobaczy Home, wstecz cofnie do Profile, a wcześniejsze ekrany zostały usunięte.

Uwaga: Stosując reset trzeba uważać na spójność stanu. Akcja reset zastępuje cały stan nawigatora nowym obiektem stanu. Jeśli pominiemy jakieś klucze lub nadamy dwa ekrany z tą samą nazwą klucza, możemy doprowadzić do niespójności. Zazwyczaj ograniczamy się do ustawienia index oraz nowej listy routes zawierającej name i opcjonalnie params. Unikajmy manipulowania wewnętrznymi strukturami stanu nawigacji – API *CommonActions.reset* wystarcza w 99% przypadków. Reagując na zmiany stanu aplikacji (np. wylogowanie), często jednak nie musimy ręcznie wykonywać reset, a zamiast tego **warunkowo renderujemy różne nawigatory** (co omówimy w sekcji o protected routes). Taka zmiana konfiguracji nawigacji automatycznie usuwa poprzednie ekrany z widoku.

- **Blokowanie powrotu (preventing goBack):** Czasem chcemy zapobiec cofnięciu się z konkretnego ekranu (np. ekran "potwierdź zamówienie" – użytkownik nie powinien cofnąć do koszyka). React Navigation udostępnia hook usePreventRemove() do zablokowania opuszczania ekranu. Można też obsłużyć zdarzenie beforeRemove na nawigatorze. W ramach tego wykładu tylko sygnalizujemy istnienie takiej opcji – warto wiedzieć, że można przejąć kontrolę nad fizycznym przyciskiem wstecz/gestem, wyświetlić np. modal "Czy na pewno chcesz wyjść?" i warunkowo zablokować nawigację.
- **Specyfika Androida:** Na Androidzie fizyczny przycisk cofania jest domyślnie zintegrowany z nawigatorem – działa jak navigation.goBack() dla najwyższego nawigatora na stosie (o ile nie nadpisaliśmy tego zachowania). Gdy użytkownik jest na ekranie początkowym nawigacji głównej, domyślnie przycisk ten zamknie aplikację. Możemy to zachowanie zmienić używając BackHandler z RN, ale jeśli logika nawigacji jest poprawnie zbudowana (i np. blokujemy cofnięcie tam gdzie nie ma sensu), zwykle domyślne działanie jest okej.

Podsumowanie sekcji: Hooki useNavigation i useRoute upraszczają dostęp do nawigacji i informacji o trasie w komponentach funkcyjnych. W połączeniu z dobrze skonfigurowanym TypeScriptem zapewniają wygodę i bezpieczeństwo. Pamiętajmy o metodach nawigacji wstecz i reset – tworząc bardziej złożone flow użytkownika (np. logowanie -> główna aplikacja) są one niezbędne do zapewnienia poprawnego zachowania (np. brak możliwości cofnięcia do ekranu logowania). W kolejnych sekcjach zobaczymy, jak zastosować te mechanizmy w praktyce, m.in. przy implementacji **deep linków** oraz **chronionych tras wymagających uwierzytelnienia**.

Deep linking – obsługa linków zewnętrznych

Deep linking to mechanizm pozwalający otworzyć aplikację mobilną w określonym miejscu nawigacji za pomocą linku/URL. Przykładowo, kliknięcie w przeglądarce linku myapp://profile/42 może bezpośrednio otworzyć naszą aplikację RN i przenieść użytkownika do ekranu profilu użytkownika o ID 42. Dzięki deep linking możemy integrować aplikację z e-mailami, stroną WWW, czy innymi aplikacjami (np. otwieranie odnośnika z Facebooka uruchamia naszą aplikację).

Aby obsłużyć deep linki, trzeba skonfigurować kilka rzeczy po stronie aplikacji **oraz** zewnętrznych źródeł (systemu i ewentualnie serwera WWW):

Definiowanie schematu URL

Podstawą deep linków jest unikalny **schemat URI** przypisany do naszej aplikacji. Schemat to ciąg znaków przed :// w URL – np. w myapp://home schematem jest myapp. W systemach mobilnych możemy zarejestrować aplikację do obsługi konkretnego schematu.

- **Expo (Managed):** W przypadku aplikacji expo najłatwiej zdefiniować schemat w pliku konfiguracyjnym **app.json / app.config.js**. W sekcji expo dodajemy pole "scheme": "myapp" (oczywiście zamiast myapp dowolna unikalna nazwa). Podczas budowy

aplikacji expo ustawi ten schemat w odpowiednich miejscach Android/iOS automatycznie.

- **Aplikacja RN (bare workflow):** Trzeba ręcznie zarejestrować URL types:
  - iOS: w Info.plist dodaj CFBundleURLTypes z wpisem dla myapp.
  - Android: w AndroidManifest.xml dodać <intent-filter> z <data scheme="myapp" ...> pozwalający otwierać aktywność główną tym schematem.
  - (Expo bare: można użyć komendy npx uri-scheme add myapp która automatyzuje te czynności).

Po ustawieniu schematu np. myapp://, każda zewnętrzna aplikacja odwołująca się do takiego URL spowoduje uruchomienie/wybudzenie naszej aplikacji.

Warto wybrać unikalny schemat, by nie kolidował z innymi. Często używa się odwróconego adresu domeny, np. com.firma.apka://, ale nie jest to konieczne.

Konfiguracja linkingu w React Navigation

Samo posiadanie schematu to tylko pierwszy krok. Następnie musimy powiedzieć React Navigation, jak mapować przychodzące linki na trasy w naszym nawigatorze. Służy do tego konfiguracja **linking** w <NavigationContainer>.

React Navigation może integrować się z modułem RN **Linking** w celu nasłuchiwania linków. Możemy skorzystać z *prop* linking przekazując obiekt konfiguracji. Przykład minimalnej konfiguracji (Expo):

```
import * as Linking from 'expo-linking';

const prefix = Linking.createURL('/');  // automatycznie ustali prefix, uwzględniając tryb Expo (dev/standalone)

const linking = {
  prefixes: [prefix],  // lista dozwolonych prefixów URLi
  config: {
    screens: {
      Home: "home",
      Profile: "profile/:userId",
      // ...mapowanie kolejnych ekranów na ścieżki
    }
  }
};

return (
  <NavigationContainer linking={linking} fallback={<Text>Ładowanie...</Text>}>
    {/* nawigatory */}
  </NavigationContainer>
);
```

Kilka wyjaśnień do powyższego:

- prefixes – to tablica prefiksów URL, które mają być przechwytywane. Najczęściej umieszczamy tu nasz schemat (np. "myapp://") **oraz** ewentualne adresy URL naszej domeny, jeśli chcemy obsłużyć tzw. **universal links/app links**. W kodzie użyto

Linking.createURL('/') z expo-linking, które generuje odpowiedni prefix zarówno dla trybu dev (exp://IP:PORT/--/) jak i dla produkcji (myapp://). Warto dodać również prefix z https:// naszej strony, jeśli planujemy integrację web -> app. Np.:

```
prefixes: [Linking.createURL('/'), 'https://moja-domena.pl']
```

config – tu definiujemy mapowanie nazw ekranów na ścieżki URL. W przykładzie powyżej ekran *Home* zdefiniowaliśmy pod ścieżką "home" (czyli myapp://home odpali Home), a *Profile* pod "profile/:userId". Dwukropek oznacza parametry w ścieżce – w tym wypadku każda ścieżka myapp://profile/XYZ spowoduje przejście do ekranu Profile z route.params = { userId: "XYZ" }. Możemy mapować również ekrany zagnieżdżone, używając zagnieżdżonych obiektów screens. Np. jeśli Profile jest w zagnieżdżonym stacku, config może wyglądać:

```
config: {
  screens: {
    Home: "home",
    ProfileStack: {   // nazwa stosu jako ekran w głównym nav
      screens: {
        Profile: "profile/:userId",
        EditProfile: "profile/edit"
      }
    }
  }
}
```

Taki config może robić wrażenie skomplikowanego, ale sprowadza się do odzwierciedlenia struktury nawigacji aplikacji za pomocą zagnieżdżonych obiektów.

fallback komponent – NavigationContainer przyjmuje fallback UI na czas, gdy przetwarza link i odpala właściwy ekran. W powyższym ustawiliśmy po prostu tekst "Ładowanie...". Można wyświetlić splashscreen lub nic, byle nie zostawić użytkownika w niepewności.

Po tej konfiguracji React Navigation automatycznie:

- **Przy starcie aplikacji** sprawdzi, czy została uruchomiona z linka (tzw. initial URL). Jeśli tak, sparsuje URL względem prefixes i config i ustawi odpowiedni **stan początkowy nawigatora**. To znaczy, np. od razu załaduje stos z ekranem Profile i parametrem userId zamiast ekranu domyślnego.
- **Gdy aplikacja już działa** i przyjdzie nowy link (np. poprzez Linking.addEventListener w RN), nastąpi nawigacja do wskazanego miejsca (update state nawigatora). Nie musimy sami wywoływać navigate – zrobi to za nas mechanizm linkingu.
- Obsłuży poprawnie specyfikę web (jeśli budujemy RN na web, linki będą używać ścieżek URL i historii przeglądarki).

Expo a deep linking: W trybie Expo Dev klient używa innego schematu (exp://127.0.0.1:19000/--/ z path) i nie możemy łatwo testować custom schematu bez budowy standalone. Na szczęście Linking.createURL('/') robi to za nas – w trybie deweloperskim prefixem będzie adres exp:// z naszym manifestem, a w zbudowanej apce prefixem będzie *myapp://*. Dzięki temu możemy testować deep linki w trakcie developmentu: np. odpalając komendę:

npx uri-scheme open "myapp://profile/42" --android

(analogicznie --ios dla iOS) – expo CLI przekształci to na odpowiedni link (exp://...) i poda do aplikacji. Alternatywnie, w Expo Go można skopiować link exp:// z parametrem za --/ (co w praktyce oznacza path w aplikacji). Jeśli mamy aplikację standalone, to na fizycznym urządzeniu kliknięcie w link myapp://… z dowolnej aplikacji (np. z notatki czy przeglądarki) powinno wywołać otwarcie naszej aplikacji.

Universal Links / App Links: Poza custom schematem, można też zarejestrować aplikację do obsługi linków HTTP(S) z własnej domeny. Np. kliknięcie w https://moja-domena.pl/invite?code=abc może otwierać aplikację. W iOS nazywa się to **Universal Links**, w Androidzie **App Links** – wymagają one spełnienia dodatkowych warunków bezpieczeństwa (potwierdzenie właścicielstwa domeny poprzez plik apple-app-site-association na serwerze, oraz wpisy w Xcode i Digital Asset Links JSON na Androidzie). Expo również to wspiera: w app.json dodajemy associatedDomains dla iOS, a w AndroidManifest analogiczne intent-filter z android:autoVerify="true" i hostem domeny. Szczegóły tego procesu są nieco poza zakresem tego wykładu (to temat na osobną sesję). Wiedzmy tylko, że jest taka możliwość – aby nasze linki webowe otwierały aplikację natywną, co poprawia UX (tzw. app/website integration).

Obsługa linków wychodzących: Wspomnijmy krótko, że RN Linking pozwala też otwierać z poziomu aplikacji linki zewnętrzne – np. Linking.openURL('tel:123456789') wywoła telefon, a Linking.openURL('https://google.com') otworzy stronę w domyślnej przeglądarce. To odwrotność deep linków, ale bywa przydatna (np. link do polityki prywatności otwiera przeglądarkę). W kontekście React Navigation warto uważać, by nie pomylić navigation.navigate (do wewnętrznej nawigacji) z Linking.openURL (do zewnętrznej). Jeśli otwarcie URL jest obsługiwane przez nas samych (np. własny schemat), lepiej użyć navigation.navigate z odpowiednim parametrem niż emulować to przez openURL.

Debugowanie deep linków:

- W trybie dev obserwuj logi Metro – gdy przyjdzie link, powinna pojawić się informacja o próbie parsowania URL.
- Jeśli link nie działa, upewnij się, że prefixes zawiera właściwy schemat/protokół, a config dokładnie odwzorowuje ścieżkę. Literówki w nazwach ekranów w configu mogą sprawić, że link zostanie zignorowany.
- Na Androidzie, jeśli link myapp://… nie otwiera aplikacji, możliwe że inna aplikacja zarejestrowała ten sam schemat – zmień schemat na bardziej unikalny.
- Do testowania używaj npx uri-scheme (szybkie i wygodne) oraz np. w XCode Devices -> Open URL dla iOS Simulator. Na Android emulatorze adb shell am start -W -a android.intent.action.VIEW -d "myapp://..." com.twojpakiet wywoła intencję.
- Sprawdzaj też wypadki graniczne: np. co jeśli użytkownik miał aplikację otwartą na innym ekranie i kliknie link? (Powinno przekierować na nowy ekran w ramach już otwartej apki – RN to obsłuży automatycznie poprzez update stanu nawigacji).

Deep linking otwiera fajne możliwości integracji, ale dodaje sporo złożoności. Dla porządku, poniżej krótkie zestawienie kroków, które należy wykonać, by w pełni zaimplementować deep linki:

1. **Rejestracja schematu URL** aplikacji (Expo: app.json, iOS: Info.plist, Android: AndroidManifest.xml).
2. **Konfiguracja React Navigation**: ustawienie NavigationContainer prop linking z prefixami i mapowaniem ścieżek na ekrany.
3. **(Opcjonalnie) Universal/App Links**: konfiguracja asocjacji domeny, by linki HTTPS też trafiały do aplikacji.
4. **Testy**: użycie uri-scheme lub fizyczne klikanie linków, sprawdzenie poprawnego otwierania.
5. **Fallback**: zapewnienie sensownego fallbacku UI, jeśli link jest nieprawidłowy (można obsłużyć zdarzenie Linking.addEventListener('url', ...) manualnie jeśli chcemy custom zachowanie w niektórych wypadkach).

Na potrzeby tego wykładu warto po prostu mieć świadomość, że deep linking istnieje i znać podstawy konfiguracji.

Ochrona tras (Protected Routes) i warunkowe flow

Wiele aplikacji wymaga uwierzytelnienia użytkownika – tzn. pewne ekrany są dostępne tylko po zalogowaniu. Musimy zatem chronić trasy przed nieautoryzowanym dostępem. W React Navigation nie ma gotowego mechanizmu "Route Guards" takiego jak np. w Angular, ale możemy osiągnąć to samo, **warunkowo renderując odpowiednie ekrany/nawigatory** w zależności od stanu aplikacji (stanu zalogowania, ukończenia onboardingu itp.).

Strażnicy tras – koncept

Route guard to kawałek logiki, który decyduje, czy użytkownika wpuścić na daną trasę, czy przekierować go gdzie indziej. W webowym React Router jest to np. komponent <PrivateRoute> sprawdzający auth. W React Navigation podejście jest nieco inne: najczęściej **ukrywamy całe sekcje nawigacji** gdy warunek nie jest spełniony, zamiast przekierowywać w momencie wejścia.

Można to zrobić na dwa sposoby:

- *Statycznie przy konfiguracji navigatora:* W najnowszym Static API RN7 można przy definicji ekranu dodać opcję if: someCondition. Gdy warunek (funkcja/hook) zwraca false, ekran nie jest w ogóle dostępny w nawigacji. Np.:

```
const RootStack = createNativeStackNavigator({
  screens: {
    Home: { if: useIsSignedIn, screen: HomeScreen },
    SignIn: { if: useIsSignedOut, screen: SignInScreen }
  }
});
```

W powyższym pseudokodzie useIsSignedIn i useIsSignedOut to hooki zwracające booleany zależnie od stanu auth. React Navigation wtedy **automatycznie pokaże tylko te ekrany, których warunek jest spełniony** – nigdy oba jednocześnie. Gdy stan się zmieni (użytkownik

zaloguje się/wyloguje), navigator zaktualizuje zestaw ekranów: jeden zniknie, pojawi się drugi. To eleganckie rozwiązanie dostępne w Static API.

- *Dynamicznie w kodzie JSX:* To podejście dostępne zawsze (także w RN6 i niżej). Polega na tym, że warunkowo renderujemy komponenty <Screen> lub całe navigatory. Przykład:

```
<NavigationContainer>
  <Stack.Navigator>
    {isSignedIn ? (
      <Stack.Screen name="Home" component={HomeScreen} />
    ) : (
      <Stack.Screen name="SignIn" component={SignInScreen} />
    )}
  </Stack.Navigator>
</NavigationContainer>
```

Jeżeli użytkownik jest zalogowany (isSignedIn === true), do stosu zostanie dodany tylko ekran *Home*. Jeśli nie jest – tylko ekran *SignIn*. Nieautoryzowany użytkownik nie ma więc jak nawet spróbować wejść na *Home*, bo nawigator go nie zna. Gdy stan isSignedIn zmieni się, komponent się przerenderuje z nowym zestawem ekranów – stary ekran zostanie usunięty (unmount) i pojawi się nowy.

Obie metody sprowadzają się do jednego: **różnicowanie konfiguracji nawigacji na podstawie stanu aplikacji**. Druga metoda (dynamiczna JSX) jest łatwiejsza do zrozumienia i wystarczająco dobra, więc skupimy się na niej.

Różnicowanie flow na podstawie stanu użytkownika

Typowy scenariusz to aplikacja z rozróżnieniem na:

- **Niezalogowany użytkownik** – widzi ekrany logowania/rejestracji (tzw. *AuthStack*).
- **Zalogowany użytkownik** – widzi główną część aplikacji (tzw. *AppStack* lub *AppTabs*).
- **Onboarding** (opcjonalnie) – nowy użytkownik, który się zalogował, ale musi np. przejść tutorial lub uzupełnić profil, zanim uzyska pełny dostęp.

Możemy wyróżnić więc kilka stanów: *signedOut*, *signedIn*, ewentualnie *signedInButNotOnboarded*. W zależności od tego, pokazujemy inną nawigację.

Implementacja może wyglądać następująco (pseudo-kod z wykorzystaniem hooka useContext do trzymania stanu zalogowania):

```
const AuthContext = React.createContext();

function AppNavigator() {
  const { user, isLoading } = useContext(AuthContext);

  if (isLoading) {
    // np. ekran splash/loading podczas sprawdzania tokenu w pamięci
    return <SplashScreen />;
  }
```

```
 return (
  <NavigationContainer>
   {user == null ? (
    <AuthStackNavigator />    // użytkownik niezalogowany
   ) : user.onboardComplete === false ? (
    <OnboardingStackNavigator />  // zalogowany, ale nie przeszedł onboardingu
   ) : (
    <MainAppNavigator />     // zalogowany i gotowy - główna aplikacja
   )}
  </NavigationContainer>
 );
}
```

W powyższym pseudokodzie:

- isLoading to stan mówiący, że jeszcze np. sprawdzamy w AsyncStorage czy jest zapisany token (pokazujemy wtedy Splash zamiast migotać ekranem logowania po odpaleniu apki).
- user to obiekt użytkownika lub null, trzymany gdzieś w globalnym stanie (np. Context lub Zustand, o tym za moment).
- Sprawdzamy kolejno: jeśli brak użytkownika -> renderujemy navigator z ekranami logowania (AuthStackNavigator); jeśli jest użytkownik, ale nieukończony onboarding -> renderujemy OnboardingStackNavigator; w przeciwnym razie (zalogowany w pełni) -> MainAppNavigator z właściwymi ekranami aplikacji.

Każdy z tych navigatorów (*AuthStack*, *OnboardingStack*, *MainApp*) to np. oddzielnie zdefiniowany Stack lub Tab ze swoimi ekranami. Moglibyśmy też zamiast trzech oddzielnych navigatorów użyć jednego warunkowo dodając Screeny – to kwestia organizacji kodu. Często dla czytelności rozdziela się je.

Co daje takie podejście? Użytkownik nigdy nie zobaczy ekranu, do którego nie powinien mieć dostępu:

- Gdy nie jest zalogowany – w drzewie nawigacji nie ma żadnej trasy z głównej aplikacji (ani Home, ani Profile, etc.). Nawet jeśli by znał nazwę ekranu, nie przejdzie do niego (navigation.navigate('Home') rzuci błąd/brak takiej trasy).
- Gdy jest zalogowany – usuwamy ekrany logowania ze stosu, więc nie może wrócić do Login. W przykładzie z dynamiczną zmianą konfiguracji, poprzednie ekrany są *unmountowane*, co oznacza że np. naciśnięcie hardware back nie cofnie do Loginu, bo login już nie istnieje w nawigatorze. Spełniliśmy tym samym wymóg odcięcia historii.
- Gdy jest w trakcie onboardingu – dopóki nie skończy, nie wpuścimy go do MainApp (bo warunek nie pozwoli wyrenderować MainAppNavigator).

To podejście jest preferowane względem prób przekierowywania w useEffect pojedynczych ekranów. Czasem ludzie implementują w komponentach coś w stylu: *jeśli user = null to navigation.replace('Login')*. Takie coś działa, ale jest mniej przejrzyste – lepiej na poziomie konfiguracji nawigacji decydować, co jest dostępne.

Implementacja stanu użytkownika (auth): Zazwyczaj potrzebujemy globalnego stanu przechowującego informację o tym, czy mamy token/credentials. Można do tego użyć:

- **Context API (AuthContext)** – jak w powyższym przykładzie, prosty kontekst trzymający obiekt użytkownika i ewentualnie funkcje login/logout.
- **Zustand** lub MobX/Redux – dowolne centralne przechowywanie stanu. Zustand jest lekkim storem, gdzie możemy trzymać `isLoggedIn` oraz np. profil użytkownika. Jego przewaga to prostota i brak boilerplate Reduxowego. Przykład użycia:

```
const useAuthStore = create((set) => ({
  user: null,
  login: (userData) => set({ user: userData }),
  logout: () => set({ user: null })
}));
// ...
const user = useAuthStore(state => state.user);
```

  Następnie `user` używamy w warunku do przełączania nawigatorów.

- **AsyncStorage/SecureStore** – służą do *persistent storage*, czyli np. zapisania tokenu JWT lub flagi pierwszego uruchomienia. Podczas startu aplikacji robimy coś takiego:

```
useEffect(() => {
  SecureStore.getItemAsync('token').then(storedToken => {
    if(storedToken) {
      // mamy token, można spróbować odświeżyć profil użytkownika z API etc.
      setUserToken(storedToken);
    }
    setLoading(false);
  });
}, []);
```

Kiedy załadujemy stan z pamięci, ustawiamy `isLoading` na false i warunkowe renderowanie pokaże odpowiednie ekrany. Ważne, by ten załadunek był wykonany zanim pokażemy jakiekolwiek ekrany (stąd Splash w tym czasie).

Guardy a nawigacja wstecz: Przy dynamicznej zmianie nawigatora (Auth -> App) warto dodatkowo zabezpieczyć scenariusz, gdy użytkownik **w trakcie** logowania mógł mieć jakiś stos ekranów (np. ekran rejestracji itp.). Gdy zaloguje się i wyrenderujemy MainApp zamiast Auth, stare ekrany znikną. Jeśli jednak użytkownik szybko naciśnie “back” (np. tuż po zalogowaniu na Androidzie), może to spowodować wyjście z aplikacji, bo w nowym nawigatorze nie będzie historii. To generalnie OK, ale jeśli chcielibyśmy np. zablokować wyjście, musielibyśmy obsłużyć to ręcznie (np. BackHandler i ignorowanie w pierwszych sekundach po zalogowaniu). W większości przypadków jednak natychmiastowe wciśnięcie back po zalogowaniu jest mało prawdopodobne, a nawet jeśli – opuszczenie aplikacji na ekranie Home jest zgodne z przewidywaniami (użytkownik myśli, że cofa do logowania, ale aplikacja się zamyka, bo logowanie już nie istnieje – następnym razem aplikacja otworzy się już zalogowana). To drobny szczegół UX do rozważenia w realnym projekcie.

Inne zastosowania protected routes: Nie tylko auth. Możemy warunkowo pokazywać pewne ekrany np. jeśli użytkownik ma uprawnienia (role-based access). Albo np. ekran *PremiumContent* tylko gdy `user.isPremium`. Wtedy analogicznie – warunek decyduje o dodaniu ekranu do navigatora. Jeżeli warunek może się zmieniać w trakcie działania aplikacji, dynamiczne dodawanie/usuwanie ekranów jest w porządku. React Navigation 7 potrafi dość elegancko reagować na zmiany warunków (dzięki opcji `if` we static API) i np. usuwać niedostępne ekrany z nawigacji bez błędów.

Podsumowanie: Ochrona tras w RN sprowadza się do *oddzielenia* części publicznej od prywatnej aplikacji. Najlepszą praktyką jest przygotowanie osobnych navigatorów dla różnych stanów i przełączanie między nimi w zależności od tego stanu.

Przepływ logowania (Auth Flow) w React Navigation

Przepływ logowania to szczególny przypadek zarządzania nawigacją w aplikacji. Składa się na niego ekrany logowania/rejestracji, ewentualnie ekran powitalny (onboarding), oraz główna część aplikacji dla zalogowanego użytkownika. Dobrze zaprojektowany flow powinien spełniać następujące założenia:

- **Po uruchomieniu aplikacji** sprawdzamy stan uwierzytelnienia (np. czy jest ważny token sesji). Dopóki sprawdzamy – pokazujemy ekran startowy (Splash).
- Jeśli użytkownik **nie jest zalogowany**, pokazujemy ekrany logowania/rejestracji (**AuthStack**).
- Jeśli użytkownik **jest zalogowany**, od razu kierujemy go do głównej aplikacji (**AppStack** / **AppTabs**).
- Po pomyślnym zalogowaniu/rejestracji przekierowujemy użytkownika do głównej aplikacji i **usuwamy ekrany logowania z historii**, aby nie można było wrócić wstecz.
- Gdy użytkownik **wyloguje się** w trakcie działania aplikacji, czyścimy dane sesji i kierujemy go ponownie na ekrany logowania (najlepiej czyszcząc historię głównej nawigacji).

Dodatkowo, jeśli mamy **onboarding dla nowych użytkowników** (np. krótki tutorial albo ekran zgód), musimy to wpasować w powyższy schemat. Często robi się to poprzez flagę w profilu użytkownika lub w pamięci (np. `hasSeenTutorial`). Przykładowe podejście:

- Jeśli zalogowany użytkownik ma flagę `onboarded = false`, zamiast od razu AppStacka dajemy mu OnboardingStack (np. kilka ekranów przewodnika). Dopiero po ich ukończeniu (gdy ustawiamy flagę na true) przestawiamy na główny AppStack.

Różnica między **AuthStack a AppStack** polega głównie na zestawie ekranów:

- **AuthStack** – zawiera ekrany: Logowanie, Rejestracja, Zapomniane hasło, Ekran powitalny (np. z logo) itd. Wszystkie te ekrany nie wymagają uwierzytelnienia. Często jest to `createNativeStackNavigator` z wyłączonym headerem lub custom headerem (np. logo apki na górze).

- **AppStack** (bądź **AppTabs** jeśli używamy tabów) – zawiera ekrany dostępne tylko po zalogowaniu, np. Home, Dashboard, Profil, Ustawienia itp. Może to być stack albo tab w zależności od designu aplikacji.

Czasem stosuje się też koncepcję *SplashStack* z tylko jednym ekranem ładowania (Splash), który jest domyślnie wyświetlany zanim zdecydujemy co dalej. Ale równie dobrze Splash można wyrenderować warunkowo jak w poprzednim rozdziale.

Przy implementacji przepływu logowania w React Navigation, warto oprzeć się na przykładach z dokumentacji. Twórcy RN sugerują użycie Context do trzymania statusu auth i pokazują przykład z użyciem SecureStore expo do trzymania tokenu. Najważniejsze jest rozdzielenie drzew nawigacji: osobne dla “signed in” i “signed out”, oraz przełączanie między nimi po zmianie stanu auth.

Onboarding – pierwsze uruchomienie

Onboarding może przybierać różne formy:

- **Ekran powitalny** z przyciskiem "Rozpocznij" (np. prezentacja głównej wartości aplikacji).
- **Kilka ekranów przewijanych** (carousel) z instrukcją obsługi, pytaniami preferencji użytkownika itp.
- **Ekran wyboru języka/tematu** itp.

Z perspektywy nawigacji, onboarding to też pewien mini-stos ekranów, który pokazujemy **tylko raz** dla nowego użytkownika. Najczęściej implementuje się to poprzez:

- *Flagę persistent*, np. zapis w AsyncStorage hasOnboarded=true po przejściu.
- Warunkowe dołączenie ekranu do AuthStack: np. pierwszy ekran AuthStack to Welcome, a dopiero potem Login. I jeśli stwierdzimy, że user już widział welcome (sprawdzamy AsyncStorage), to nawigujemy go od razu do Login pomijając welcome. Można to zrobić na starcie aplikacji (np. w Splash, decydując do której trasy iść).

Albo:

- Osobny OnboardingStack jak wcześniej wspomniano, który renderujemy przed AppStack. Ten stack po zakończeniu ustawia flagę i przełączamy na AppStack.

Przechowywanie informacji o logowaniu

Ten temat częściowo już omówiliśmy przy route guards. W kontekście przepływu logowania sprowadza się on do:

- **Przechowywanie tokenu/autoryzacji** – jeżeli korzystamy z API, to po zalogowaniu zapewne otrzymujemy token (JWT lub session id). Należy go bezpiecznie przechować. Zaleca się użyć do tego **SecureStore** (Expo) lub **Keychain/Keystore** na urządzeniach. SecureStore (Expo) zapisuje dane zaszyfrowane w bezpiecznej pamięci urządzenia. AsyncStorage nie szyfruje, więc token tam jest jawny – do mniej wrażliwych rzeczy

ok, ale tokeny raczej trzymajmy bezpiecznie. W ostateczności, jeśli testujemy, można i AsyncStorage użyć.

- **Przechowywanie stanu zalogowania w aplikacji** – aby cała aplikacja wiedziała, że user jest zalogowany i np. wyświetliła jego dane w różnych miejscach. Do tego świetnie nadaje się **Context API** lub globalny store (Zustand/Redux). Context wystarcza: tworzymy AuthContext z wartością { user, login(), logout() }. Po pomyślnym logowaniu (np. gdy API zwróci 200 OK) wywołujemy login(userData) z obiektu kontekstu. To zmienia user i powoduje re-render zależnych komponentów (w tym naszego warunkowego navigatora). W efekcie nastąpi przejście do AppStack.
- **Zustand** – ma przewagę, że nawet gdy komponenty contextowe się rozmontują, stan zostaje (ale to samo daje trzymanie w useState w komponencie wyżej niż NavigationContainer). Dla początkujących – Context jest łatwiejszy.
- **Wylogowanie** – powinno wyczyścić wszystkie dane: usunąć token z SecureStore, wyzerować user w state, ewentualnie zresetować nawigację. Jeśli korzystamy z warunkowego renderowania navigatorów, to wyzerowanie usera automatycznie przerenderuje NavContainer na AuthStack, usuwając tym samym wszystkie ekrany aplikacji z widoku. Warto jednak upewnić się, że np. nie pozostawimy jakiegoś modala otwartego itp. Zwykle wystarczy logout() w kontekście.

Przykład użycia Context (pseudo-kod):

```
const AuthContext = React.createContext();

function AuthProvider({ children }) {
  const [user, setUser] = useState(null);

  const login = async (credentials) => {
    const token = await API.login(credentials);
    await SecureStore.setItemAsync('token', token);
    const userData = await API.fetchProfile(token);
    setUser(userData);
  };

  const logout = async () => {
    await SecureStore.deleteItemAsync('token');
    setUser(null);
  };

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}
```

Całą aplikację opakowujemy w <AuthProvider> (np. w pliku App.js). W komponentach (np. ekranach logowania) wywołujemy const { login } = useContext(AuthContext) by zalogować, a w ekranie profilu const { logout } = useContext(AuthContext) by się wylogować.

Dobra praktyka: Podczas logowania, gdy przechodzimy do głównej aplikacji, możemy **zresetować stos** (jeśli używaliśmy zwykłego navigate) lub, jak pokazaliśmy, usunąć ekrany

logowania poprzez warunkowe renderowanie. Wiele gotowych template'ów używa navigation.reset({ routes: [{ name: 'MainApp' }] }) po loginie, gdzie *MainApp* to np. navigator zakładek.

Podsumowując: oddzielenie AuthStack i AppStack to podstawa uporządkowania flow logowania. Trzymajmy stan logowania globalnie, by łatwo reagować na jego zmiany w drzewie nawigacji. Pamiętajmy o bezpiecznym przechowaniu wszelkich tokenów oraz o tym, by po wylogowaniu "posprzątać" (wyczyścić stan i nawigację).

Demo: Mini-aplikacja z nawigacją (AuthStack + AppTabs)

Teraz przejdźmy do praktycznego przykładu, który łączy wszystkie poruszone koncepty. Zbudujemy uproszczoną aplikację React Native, która posiada dwa główne "tryby" nawigacji:

- **AuthStack** – stos z ekranem logowania (LoginScreen).
- **AppTabs** – nawigator z zakładkami zawierający dwa ekrany: HomeScreen i ProfileScreen.

Użytkownik niezalogowany zobaczy ekran logowania. Po "zalogowaniu" (symulujemy je w aplikacji) zostanie przeniesiony do zakładek aplikacji: Home i Profile. Home wyświetli powitanie i umożliwi przejście do ekranu Profil (np. przyciskiem) – przy okazji przekażemy ID użytkownika jako parametr. Profil wyświetli identyfikator zalogowanego użytkownika i da opcję wylogowania (powrót do ekranu logowania).

W kodzie wykorzystamy **React Context** do przechowywania informacji o zalogowanym użytkowniku (userId), co pozwoli na warunkowe renderowanie odpowiedniego navigatora. Pokażemy również użycie hooków useNavigation i useRoute wewnątrz ekranów.

Oto kod demonstracyjnej aplikacji:

```
import React, { useState, useContext, createContext } from 'react';
import { Text, View, Button } from 'react-native';
// Navigatory
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';

// 1. Definicje typów parametrów dla nawigatorów (TypeScript)
type AuthStackParamList = {
  Login: undefined;
};
type AppTabsParamList = {
  Home: undefined;
  Profile: { userId: string };
};

// 2. Utworzenie navigatorów
const AuthStack = createNativeStackNavigator<AuthStackParamList>();
const AppTabs = createBottomTabNavigator<AppTabsParamList>();

// 3. Kontekst uwierzytelnienia
```

```
type AuthContextType = { userId: string | null, login: (id: string) => void, logout: () => void };
const AuthContext = createContext<AuthContextType | undefined>(undefined);

// 4. Ekran logowania
function LoginScreen() {
  const auth = useContext(AuthContext);
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text> Ekran logowania</Text>
      <Button title="Zaloguj mnie jako user #123"
            onPress={() => auth?.login('123')} />
    </View>
  );
}

// 5. Ekran Home (zakładka główna)
function HomeScreen({ navigation }: { navigation: any }) {  // typowanie navigation pominięte dla czytelności
  const auth = useContext(AuthContext);
  const userId = auth?.userId;
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text> Ekran Home - witaj użytkowniku #{userId}!</Text>
      {/* Przykładowy przycisk nawigujący do Profilu z parametrem */}
      <Button title="Przejdź do Profilu (param: userId)"
            onPress={() => navigation.navigate('Profile', { userId })} />
    </View>
  );
}

// 6. Ekran Profil (zakładka profilowa)
function ProfileScreen({ route }: { route: any }) {
  const auth = useContext(AuthContext);
  // Pobieramy userId z parametru lub z kontekstu:
  const routeUserId = route.params?.userId;
  const userId = routeUserId || auth?.userId;
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text> Ekran Profil użytkownika #{userId}</Text>
      <Button title="Wyloguj" onPress={() => auth?.logout()} />
    </View>
  );
}

// 7. Główny komponent nawigacji oparty o context
export default function App() {
  const [userId, setUserId] = useState<string | null>(null);

  const authContext: AuthContextType = {
    userId,
    login: (id: string) => setUserId(id),
    logout: () => setUserId(null)
  };

  return (
    <AuthContext.Provider value={authContext}>
      <NavigationContainer>
        {userId == null ? (
```

```
        // Gdy brak zalogowanego użytkownika -> pokazujemy AuthStack
        <AuthStack.Navigator>
          <AuthStack.Screen
            name="Login"
            component={LoginScreen}
            options={{ headerShown: false }}
          />
        </AuthStack.Navigator>
      ) : (
        // Gdy jest zalogowany -> pokazujemy zakładki aplikacji
        <AppTabs.Navigator screenOptions={{ headerShown: false }}>
          <AppTabs.Screen
            name="Home"
            component={HomeScreen}
            options={{ title: 'Home' }}
          />
          <AppTabs.Screen
            name="Profile"
            component={ProfileScreen}
            options={{ title: 'Profil' }}
          />
        </AppTabs.Navigator>
      )}
    </NavigationContainer>
  </AuthContext.Provider>
 );
}
```

(Kod pisany z myślą o czytelności dla początkujących – w praktyce typowanie nawigacji byłoby dopracowane, ale tutaj skupiamy się na idei.)

Objaśnienia do powyższego kodu:

- Na początku definiujemy typy nawigatorów i tworzymy navigatory AuthStack i AppTabs. AuthStack to stos (NativeStack) z jednym ekranem *Login*. AppTabs to dolne zakładki z dwoma ekranami: *Home* i *Profile*. Zauważ, że dla Profile przewidzieliśmy parametr userId (typu string).
- Tworzymy kontekst AuthContext, który będzie przechowywać stan userId (lub null) oraz funkcje login i logout. Cała logika autoryzacji jest tu uproszczona: zakładamy, że kliknięcie przycisku "Zaloguj" zawsze się udaje i loguje nas jako user #123.
- **LoginScreen:** Pobiera login z kontekstu (auth?.login) i po kliknięciu loguje użytkownika o ID "123". W realnej aplikacji zamiast tego wywołalibyśmy np. API logowania, a po sukcesie zapisali token i user id. Tu robimy natychmiast auth?.login('123'), co ustawia userId w stanie kontekstu.
- **HomeScreen:** Wyświetla powitanie z numerem użytkownika (pobieramy go z kontekstu). Ma również przycisk *"Przejdź do Profilu (param: userId)"*. Po kliknięciu wywołujemy navigation.navigate('Profile', { userId }). To demonstruje przekazanie parametru do zakładki Profil – choć w tym wypadku Profil mógłby równie dobrze skorzystać z kontekstu, pokazujemy przekazanie param dla ilustracji. Ponieważ Home i Profile są w tym samym navigatorze AppTabs, wywołanie navigate spowoduje **przełączenie zakładki** na Profile, przekazując jej parametr.

- **ProfileScreen:** Odbiera parametry przez { route } (otrzymuje je jako prop, bo to ekran nawigacji). Wyciąga z route.params wartość userId. Dla bezpieczeństwa, jeśli param nie został przekazany (np. użytkownik przełączył się na zakładkę Profile bez użycia przycisku z parametrem), wtedy route.params będzie undefined – wówczas bierzemy userId z kontekstu. W naszym flow, gdy użytkownik najpierw kliknie na Home *"Przejdź do Profilu"* – param będzie obecny. Jeśli jednak będąc na Home użyje dolnej nawigacji (kliknie ikonkę Profil w tab barze), to przełączy zakładkę **bez parametru** (React Navigation nie przekazuje param przy zwykłym tapnięciu w tab, bo to nie navigation.navigate tylko wewnętrzne przełączenie). Dzięki naszej logice ProfileScreen i tak ustali userId z kontekstu.
  ProfileScreen wyświetla identyfikator i posiada przycisk "Wyloguj". Po naciśnięciu wywołujemy auth?.logout(), które w naszym kontekście ustawia userId = null. To spowoduje **przerenderowanie całego drzewa nawigacji** – ponieważ w App() warunek userId == null znowu stanie się true, pokaże się AuthStack zamiast AppTabs, czyli wrócimy do ekranu logowania. Jest to natychmiastowe i skuteczne – zakładki znikają, stos logowania się pojawia.
- W komponencie głównym <App> osadzamy NavigationContainer oraz warunkowo wybieramy między <AuthStack.Navigator> a <AppTabs.Navigator>. Zwróćmy uwagę: *AuthStack.Navigator* i *AppTabs.Navigator* to dwa zupełnie osobne nawigatory – nie mają wspólnych ekranów. Przełączenie następuje czysto przez React (ternary operator) i React Navigation bez problemu radzi sobie z montowaniem/odmontowaniem jednego navigatora i zastąpieniem go drugim.
- NavigationContainer otaczamy AuthContext.Provider, żeby ekrany miały dostęp do auth wartości.

Testowanie scenariusza:

- Po uruchomieniu: userId jest null, więc renderuje się AuthStack z LoginScreen. LoginScreen pokazuje przycisk. Gdy go naciśniemy:
- auth.login('123') ustawia userId = '123'. Teraz userId nie jest null, więc *App* re-render: zamiast AuthStack w NavigationContainer pojawia się AppTabs. Użytkownik zobaczy ekran Home (domyślnie pierwsza zakładka *Home*).
- HomeScreen wyświetla "Witaj użytkowniku #123". Użytkownik ma dwie drogi: albo kliknie przycisk *"Przejdź do Profilu"* (co wywoła navigation.navigate z param), albo po prostu wybierze zakładkę Profile na dole.
  - Jeśli kliknął przycisk: zostaje przełączony na ekran Profile, a ten otrzymuje route.param { userId: '123' }. Wyświetli "Profil użytkownika #123".
  - Jeśli zamiast tego użytkownik dotknął zakładki *Profile* na tab barze: zostanie przełączony na ProfileScreen, ale bez parametru. Nasz kod w ProfileScreen zobaczy, że route.params jest undefined i weźmie userId z kontekstu, również dostając '123'. Wynik końcowy dla użytkownika identyczny – widzi profil #123.
- Na ekranie Profil jest przycisk "Wyloguj". Po tapnięciu: auth.logout() ustawia userId = null. Następuje *unmount* AppTabs i *mount* AuthStack z powrotem (NavigationContainer zdejmuje zakładki i wstawia login). Użytkownik widzi znów ekran logowania. Gdyby wcisnął w tym momencie back na Androidzie – zamknie aplikację (bo jesteśmy na jedynym ekranie w stacku głównym). Możemy to traktować jako oczekiwane

zachowanie (wylogował się, więc wyjście z apki jest logiczne, albo może zalogować się ponownie).

Uwagi dot. debugowania i rozszerzania:

- Gdybyśmy nie używali kontekstu, alternatywą byłoby np. trzymanie `userId` w state wyżej i przekazywanie go do screens jako props (via initialParams lub własne propsy w nawigatorze). Context jest jednak wygodniejszy.
- W realnej apce po zalogowaniu pewnie chcielibyśmy wykonać `navigation.reset` zamiast zwykłego navigate – ale tutaj zrobiliśmy to "architektonicznie" usuwając AuthStack całkowicie.
- Podczas developmentu łatwo można sprawdzić, czy na pewno po zalogowaniu nie zostaje w pamięci ekran logowania – np. w React DevTools sprawdzając drzewo komponentów albo logując w useEffect unmount w LoginScreen. Powinien zostać odmontowany przy przejściu.
- Nasz param `userId` jest stringiem – w prawdziwym API to mógłby być token JWT albo obiekt użytkownika. Wtedy raczej nie przekazujemy tego przez paramy nawigacji (bo to wrażliwe dane), tylko trzymamy w kontekście/state. Parametr w nawigacji bardziej przydaje się do *danych dotyczących konkretnej pod-strony*, np. `postId` dla ekranu *PostDetails*. Dla globalnego `userId` lepiej użyć context jak pokazaliśmy, co i tak zrobiliśmy.
- Gdybyśmy mieli więcej ekranów w AuthStack (np. Register, ForgotPassword), moglibyśmy je dodać do AuthStack.Navigator. Wtedy z LoginScreen normalnie `navigation.navigate('Register')` by działało. Po zalogowaniu niezależnie, cała nawigacja AuthStack jest wymieniana.
- Podobnie w AppTabs – moglibyśmy łatwo dodać np. trzecią zakładkę Ustawienia bez wpływu na logikę auth.

Podsumowanie i dobre praktyki

W trakcie tego wykładu przeszliśmy przez wszystkie kluczowe aspekty nawigacji w React Native z użyciem React Navigation:

- Poznaliśmy różne **rodzaje nawigatorów** i ich zastosowania (Stack do sekwencji ekranów, Tabs do równoległych sekcji, Drawer do ukrytego menu).
- Nauczyliśmy się **przekazywać parametry** między ekranami i odbierać je, a także jak dbać o ich poprawne typowanie w TypeScript.
- Wykorzystaliśmy **hooki nawigacyjne** `useNavigation` i `useRoute` do wygodnego wywoływania nawigacji i pobierania danych routy w komponentach funkcyjnych.
- Omówiliśmy jak **cofać** nawigację, zarówno pojedynczo (`goBack`), jak i hurtowo (`popToTop`), oraz jak **resetować** stos ekranów – co okazało się ważne w scenariuszu logowania.
- Zajęliśmy się tematem **deep linków**, konfigurując schematy URL i integrację z Expo, aby nasza aplikacja mogła reagować na linki zewnętrzne i otwierać właściwe ekrany.

- Wprowadziliśmy koncepcję **ochrony tras** – ograniczania dostępu do części aplikacji poprzez warunkowe renderowanie nawigatorów zależnie od stanu (np. logowania). Dzięki temu zrealizowaliśmy **przepływ uwierzytelniania**, oddzielając ekrany logowania od głównej aplikacji.
- Zwieńczyliśmy wszystko **praktycznym demo**, które krok po kroku pokazało implementację mini-apki z kontekstem autoryzacji i dwoma nawigatorami. Demo ilustruje, jak w realnym kodzie spiąć razem React Navigation, Context oraz komponenty RN, by uzyskać przyjazny dla użytkownika flow (logowanie -> aplikacja -> wylogowanie -> z powrotem logowanie).

Literatura:

1. https://reactnavigation.org/docs/getting-started/ (Data dostępu: 1.10.2025) - Oficjalna dokumentacja React Navigation (strona główna).
2. https://reactnavigation.org/docs/typescript/ (Data dostępu: 1.10.2025) - Oficjalny przewodnik po integracji React Navigation z TypeScript.
3. https://reactnavigation.org/docs/auth-flow/ (Data dostępu: 1.10.2025) - Kluczowa dokumentacja opisująca rekomendowany wzorzec przepływu uwierzytelniania.
4. https://reactnavigation.org/docs/deep-linking/ (Data dostępu: 1.10.2025) - Oficjalny przewodnik po konfiguracji Deep Linków.
5. https://reactnavigation.org/docs/navigating/ (Data dostępu: 1.10.2025) - Dokumentacja podstawowych operacji (navigate, push, goBack).
6. https://reactnavigation.org/docs/params/ (Data dostępu: 1.10.2025) - Dokumentacja na temat przekazywania i odbierania parametrów (route.params).
7. https://reactnavigation.org/docs/hooks/ (Data dostępu: 1.10.2025) - Dokumentacja hooków useNavigation i useRoute.
8. https://reactnavigation.org/docs/native-stack-navigator/ (Data dostępu: 1.10.2025) - Dokumentacja Native Stack Navigator (rekomendowanego dla wydajności).
9. https://reactnavigation.org/docs/bottom-tab-navigator/ (Data dostępu: 1.10.2025) - Dokumentacja Bottom Tab Navigator.
10. https://docs.expo.dev/routing/linking/ (Data dostępu: 1.10.2025) - Przewodnik Expo dotyczący konfiguracji linkowania (w tym expo-linking).