KIELCE UNIVERSITY OF TECHNOLOGY

Mobile Applications – lecture 3

Navigation and User Flow- React Navigation in React Native

Mateusz Pawełkiewicz

1.10.2025

Introduction: In React Native mobile applications, navigation between screens plays a crucial role in the so-called user flow. The de facto standard library for navigation is React Navigation, offering a convenient API for routing on iOS, Android, and even the web. In this lecture, we will discuss types of navigators (stack, tabs, drawer), their nesting, passing parameters between screens, as well as more advanced topics: navigation hooks, integration with TypeScript, deep linking mechanisms, protected routes in the context of authentication, and a complete login flow.

React Navigation – navigation types React Navigation is a library enabling the definition of various types of navigation in a React Native application. The basic types of navigators are: Stack Navigator (stack of screens), Bottom Tab Navigator (navigation with tabs), and Drawer Navigator (drawer navigation). Each of them corresponds to a different pattern of moving around the application:

- **Stack Navigator (stack):** Enables transitioning to subsequent screens by placing them on a stack – a new screen is placed on top of the previous one, and going back returns to earlier screens. This is analogous to navigation in iOS/Android: screens enter from the right side or in the default system style. By default, React Navigation provides platform-consistent transition animations (horizontal slide on iOS, standard fade/slide on Android). The stack implementation comes in two varieties: `@react-navigation/stack` (in JavaScript) and `@react-navigation/native-stack` (utilizing native navigation APIs). The JavaScript version is more configurable but may be slightly less efficient, so for complex animations, it is worth considering the native variant for better smoothness. Stack Navigator works well in sequential screen flows – e.g., article list → article details screen.
- **Bottom Tab Navigator (tabs):** Provides navigation via a tab bar usually placed at the bottom of the screen. The user can switch between tabs representing independent sections of the application (e.g., Home, Search, Profile). Each tab has its own screen assigned (or a nested stack of screens). Tabs are initialized lazily (the tab screen loads only upon the first visit) and kept in memory, which enables a quick return without losing state. We can customize tab icons and labels, as well as the bar style. This navigator is ideal when the application has several main modules available in parallel (e.g., home page and settings screen as separate tabs).
- **Drawer Navigator (drawer):** Offers navigation with a menu sliding from the side of the screen (so-called hamburger menu). Usually opened on Android by a swipe gesture from the edge, and on iOS additionally often by a hamburger icon in the header. Drawer Navigator is useful for housing many navigation options or global navigation, e.g., an application side panel with links to various screens (profile, settings, FAQ, etc.). In React Navigation, the drawer implementation relies underneath on the `react-native-drawer-layout` component, and for correct operation, it requires installing dependencies like `react-native-gesture-handler` and `react-native-reanimated` (Expo installs them automatically). When configuring the Drawer Navigator, we define the screens available in the menu; we can also adjust the drawer position (left/right) or header content. The Drawer works well when we want to hide navigation under a gesture, leaving more space on the main screen.

Nesting navigators: Often in a real application, we use several navigation types simultaneously, nesting them within each other. For example, we can have a main Drawer Navigator, and inside it, each option opens a separate Stack or Tab navigator. Or a popular case: Tab Navigator as the main application navigation, where individual tabs have internal Stack Navigators (e.g., the Home tab has a stack with the main screen and a details screen). React Navigation allows treating a navigator like a screen – e.g., we can define a screen in the Stack Navigator whose component is MyTabs (our Tab Navigator). In this way, navigators are hierarchical. It should be remembered that each navigator has its own namespace for routes – i.e., screens in a nested navigator have their names independent of screen names in the parent. To navigate between nested navigators, we usually call navigation relative to the common NavigationContainer or use full paths (e.g., navigation.navigate('NavigatorName', { screen: 'ScreenName', params: {...} })). Nesting navigators allows combining different navigation patterns – e.g., bottom tabs plus options in a drawer, or a login stack separated from the main application stack.

Passing parameters between screens: Often when moving to the next screen, we want to pass data to it (e.g., the ID of the object whose details it is supposed to display). React Navigation enables this through the second argument of the navigating function. For example, having a stack navigation, we can go to the "Details" screen with parameters:

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

In the call above, we passed a parameter object to the "Details" route. On the target screen, we access this data through the route.params property. For example, in the DetailsScreen component, we can retrieve parameters:

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

We retrieved itemId and otherParam from the parameters object passed via navigation. It is worth noting that parameters are best passed in the form of simple JSON (simple types, objects, arrays) – thanks to this they are serializable, which facilitates e.g. saving navigation state or handling deep links. We can also define initial screen parameters independently of navigation. If, for example, we want the Details screen to have itemId: 42 by default, we can set initialParams at the screen definition:

```
<Stack.Screen
  name="Details"
  component={DetailsScreen}
  initialParams={{itemId: 42 }}
/>
```

Parameters passed during Maps will overwrite initial values, and if we do not pass any – the screen will use initialParams. A screen can also update its own route parameters during

operation by calling navigation.setParams({ ... }) – e.g., to change parameters affecting the header UI etc. (useful for dynamic titles).

A few best practices when passing parameters:

- Pass only necessary data (e.g., ID, key), and fetch larger objects again on the target screen or use global state. This way you avoid serialization problems and separate responsibilities.
- Ensure that the screen receiving parameters handles the case of missing parameters (e.g., if navigation occurred without them). You can define default values: route.params?.userName ?? 'Anonymous' etc.
- In TypeScript, it is worth refining parameter types for each route – thanks to this, the editor will catch missing required parameters already at the compilation stage. We will discuss navigation typing in a later part.

Note: In Stack Navigator, navigation.navigate('RouteName', params) behaves slightly differently than navigation.push('RouteName', params). Maps will try to find an existing screen with the given name in the stack and refresh its parameters (or go to it if it is lower on the stack). If such a screen does not exist, only then will it insert a new one on top of the stack. Whereas push always adds a new screen to the top, even if one is already on the stack. Therefore, when we want to enter the same screen multiple times (e.g., view different details in a loop), we use push. When we want to go to a screen without duplicating (e.g., from a menu to an already existing stack), Maps is better. There is also navigation.replace('RouteName', params) – it replaces the current screen with a new one, which is sometimes useful e.g. after finishing onboarding (we remove the welcome screen from the stack and insert the main application screen).

Navigation hooks and navigation in practice React Navigation provides special hooks that facilitate using navigation in functional components. The main ones are useNavigation and useRoute. We will discuss their operation and show how to use them with TypeScript for full type safety. We will also look at back navigation and stack reset methods – essential in managing screen history.

useNavigation – access to the navigation object useNavigation() returns the navigation object (navigation prop) for the screen in whose context the hook was called. Thanks to this, we can call navigation.navigate, navigation.goBack() and other methods without passing the navigation object through props. This is useful e.g. in nested components that are not screens but want to navigate (e.g., a button in a custom header).

Example: Suppose we want to create a "Back" button component working anywhere in the application:

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

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

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

```
}
```

Here, the useNavigation() hook provides us with the current navigation object, and we call navigation.goBack() upon clicking. Pressing the button will cause a return to the previous screen on the stack (identical to using the gesture/system back).

Usage with classes: If we must use useNavigation in a class component (which does not support hooks), we can wrap the class component in a function using the hook and pass navigation as a prop:

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

// replace export with a version with navigation passed:
export default function(props) {
  const navigation = useNavigation();
  return <MyLegacyComponent {...props} navigation={navigation} />;
}
```

Generally, however, in new applications, we stick to functional components and hooks.

Typing useNavigation: By default, without additional configuration, the object returned by useNavigation has a general type (it does not contain information about available routes and parameters). To have better hints and control, we can use a generic type. For example, if we have defined the parameter type of the main navigator RootStackParamList (about defining types in a moment), we can do:

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

Then e.g. navigation.navigate('Home', ...) will check the compatibility of the route name and parameter types with RootStackParamList. Another approach (in React Navigation 6+) is to declare a global param list type – then useNavigation() will automatically use it without the need to pass a generic. We will discuss the details of this approach in the section on TypeScript.

Note: useNavigation must be called within the context of a screen attached to a navigator (i.e., inside the NavigationContainer tree). If we use it outside a navigator, we will receive a context missing error. In practice, this means that e.g. we cannot call useNavigation in the code that renders NavigationContainer – only inside components being screens or their children. If we need to navigate globally from outside components (e.g., from the level of a module, service) – React Navigation offers a navigation ref object (React.createRef()), which we can manually use for imperative navigation, but this is an advanced case.

useRoute – information about the current route useRoute() allows us to obtain the current route object (route) in a given screen. Standardly, a component-screen receives route in

props ({ route, navigation }), but if we nest logic deeper or want to use a hook instead of props, useRoute solves the problem.

A typical use-case is retrieving parameters passed to the screen or reading the route name. Usage example:

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

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

In the example above, we display the name of the current screen (route.name) and parameters in text form. Of course, usually we are interested in a specific field from route.params – e.g. route.params.userId. useRoute is particularly useful in a component that does not receive the route prop (e.g. a grandchild of the screen) but needs this data.

Typing useRoute: Just like with useNavigation, it is worth ensuring the appropriate type of the returned object. React Navigation provides the generic RouteProp<ParamList, RouteName>. If for example we have:

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

then we can use:

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

Then TypeScript will know that route.params has the structure { itemId: number } – thanks to which it will immediately catch typos in field names or incorrect types.

Navigation typing in TypeScript To fully utilize the power of TypeScript in navigation, we should define parameter types for all screens and use them when creating navigators and hooks. The process looks as follows:

1. **Definition of parameter list** – we create an object type where keys are route names and values are parameter types (or undefined if no parameter). Example for a simple stack:

   ```
   type AuthStackParamList = {
     Login: undefined;
     Register: undefined;
     ForgotPassword: { email?: string }; // e.g. password reset screen with optional email
   };
   ```

   and for the main part of the application with tabs:

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

```
};
```

Here we assume that the Home screen does not need a parameter, and the Profile screen requires `userId` (e.g. the identifier of the user whose profile we are displaying).

2. **Creating a navigator with the above type** – when calling the navigator factory, we pass the param list type as a generic parameter. E.g. for stack:

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

and for tabs:

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

Now `<Stack.Screen>` and `<Tab.Screen>` components will expect a name and component consistent with the routes defined in the type.

3. **Typing screen props:** We can obtain navigation and route types for a specific route using provided utility types. For example, for the Profile screen in a tab navigator:

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

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

In a more complex case, when a screen is nested (e.g. a screen in a stack inside a tab), `CompositeScreenProps` is used combining the stack and tab type. for simplification, one can also use hooks with generics instead of typing the whole component. In practice, it is often done like this:

```
function ProfileScreen() {
  const navigation = useNavigation<BottomTabNavigationProp<AppTabsParamList, 'Profile'>>();
  const route = useRoute<RouteProp<AppTabsParamList, 'Profile'>>();
  // now navigation and route are strictly typed
  const { userId } = route.params;
  ...
}
```

Such an approach gives full safety: if we try `navigation.navigate('Home', {userId: '123'})` but Home does not accept a parameter, TS will report an error. Or conversely – calling `navigation.navigate('Profile')` without the required `userId` will be erroneous.

4. **Global param list type:** React Navigation v6+ enables declaring a default param list type in the global space (`ReactNavigation` namespace). It is enough to do something like this in a definition file (e.g., `types.d.ts`):

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

After this declaration, wherever React Navigation uses RootParamList (e.g. in useNavigation() without a generic), our type will be applied. This simplifies using hooks – we don't have to provide generics every time. Let's just make sure the declaration is in the project scope (included by TSconfig).

Summary: TypeScript configuration with navigation requires a bit of work at the start, but it pays off. We gain autocomplete for screen names, certainty about parameters and their types, and fewer errors at runtime. In large applications, this is almost a necessity. React Navigation in version 7 additionally introduced the so-called Static API, which simplifies navigator configuration and improves type inference (we define screens as objects with keys, which facilitates deriving param list types). Regardless of the approach, it is always worth defining param lists and using the provided helper types.

Return to previous screen and stack resetting Navigation is not only moving "forward" (push/navigate) but also returning and managing screen history. Below are the most important operations:

- **Return (goBack):** Each navigation object possesses a navigation.goBack() method, which moves us back one screen (analogously to pressing system back on Android or the back gesture on iOS). If we are on the first screen of the stack, calling goBack() will by default close the entire application (or move it to the background) on Android – it is worth keeping this in mind when handling hardware back (this behavior can be overridden, about which in a moment). In our previously shown MyBackButton, we used navigation.goBack(). There is also the navigation.canGoBack() method, which can check if there is anything to go back to (returns true/false).
- **Pop stack:** An alternative to goBack is navigation.pop(n), which goes back by n screens (default 1). Furthermore, navigation.popToTop() will take us back immediately to the beginning of the stack (first screen). These methods are useful e.g. to exit a multi-step form straight to the main screen etc.
- **Reset navigation (reset stack):** Sometimes we want to completely change the navigation state – e.g. clear history and insert a new set of screens. An example is the login flow: after successful login, we do not want the user to return via the "back" button to the login screen. In such a case, a stack reset can be applied. React Navigation provides the CommonActions.reset action to define a new navigation state. It is used more or less like this:

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

navigation.dispatch(
  CommonActions.reset({
    index: 0,
    routes: [ { name: 'Home' } ]  // new stack with only one Home screen
  })
);
```

The code above will clear the entire history and set a stack containing a single Home route. We can of course provide more routes in the array, specifying even parameters for each of them. E.g.:

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

Here we set that the new stack has two screens: Profile (will be at index 0) and Home (index 1), and we immediately start at index 1 i.e., Home. Effect: the user will see Home, back will return to Profile, and earlier screens were removed.

Note: When applying reset, one must be careful about state consistency. The reset action replaces the entire navigator state with a new state object. If we omit some keys or give two screens the same key name, we can lead to inconsistency. Usually, we limit ourselves to setting index and a new list of routes containing name and optionally params. Avoid manipulating internal navigation state structures – the CommonActions.reset API suffices in 99% of cases. Reacting to application state changes (e.g. logout), often however we do not have to manually perform a reset, and instead conditionally render different navigators (which we will discuss in the section on protected routes). Such a change in navigation configuration automatically removes previous screens from the view.

- **Blocking return (preventing goBack):** Sometimes we want to prevent backing out from a specific screen (e.g. "confirm order" screen – the user shouldn't go back to the cart). React Navigation provides the usePreventRemove() hook to block leaving the screen. One can also handle the beforeRemove event on the navigator. Within this lecture, we only signal the existence of such an option – it is worth knowing that one can take control over the physical back button/gesture, display e.g. a modal "Are you sure you want to exit?" and conditionally block navigation.
- **Android Specifics:** On Android, the physical back button is by default integrated with the navigator – it acts like navigation.goBack() for the highest navigator on the stack (unless we overrode this behavior). When the user is on the initial screen of the main navigation, by default this button will close the application. We can change this behavior using BackHandler from RN, but if the navigation logic is built correctly (and e.g. we block backing out where it doesn't make sense), usually the default operation is okay.

Section summary: Hooks useNavigation and useRoute simplify access to navigation and route information in functional components. Combined with well-configured TypeScript, they ensure convenience and safety. Let's remember about back navigation and reset methods – creating more complex user flows (e.g. login -> main application), they are essential to ensure correct behavior (e.g. no possibility of going back to the login screen). In the following sections, we will see how to apply these mechanisms in practice, among others in implementing deep links and protected routes requiring authentication.

Deep linking – handling external links Deep linking is a mechanism allowing a mobile application to be opened in a specific place of navigation using a link/URL. For example, clicking the link myapp://profile/42 in a browser can directly open our RN application and move

the user to the user profile screen with ID 42. Thanks to deep linking, we can integrate the application with emails, a website, or other applications (e.g. opening a link from Facebook launches our application). To handle deep links, a few things need to be configured on the application side and external sources (system and potentially web server):

Defining URL scheme The basis of deep links is a unique URI scheme assigned to our application. The scheme is the character string before :// in the URL – e.g. in myapp://home the scheme is myapp. In mobile systems, we can register an application to handle a specific scheme.

- **Expo (Managed):** In the case of expo applications, it is easiest to define the scheme in the configuration file app.json / app.config.js. In the expo section, we add the field "scheme": "myapp" (of course replacing myapp with any unique name). During the application build, expo will set this scheme in the appropriate Android/iOS places automatically.
- **RN Application (bare workflow):** One must manually register URL types:
    - **iOS:** in Info.plist add CFBundleURLTypes with an entry for myapp.
    - **Android:** in AndroidManifest.xml add <intent-filter> with <data scheme="myapp" ...> allowing the main activity to be opened with this scheme.
    - (Expo bare: one can use the command npx uri-scheme add myapp which automates these activities).

After setting the scheme e.g. myapp://, any external application referring to such a URL will cause the launch/waking up of our application. It is worth choosing a unique scheme so it doesn't collide with others. Often a reversed domain address is used, e.g. com.company.app://, but it is not necessary.

Linking configuration in React Navigation Having the scheme itself is just the first step. Next, we must tell React Navigation how to map incoming links to routes in our navigator. The linking configuration in <NavigationContainer> serves this purpose. React Navigation can integrate with the RN Linking module to listen for links. We can use the linking prop passing a configuration object. Example of minimal configuration (Expo):

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

const prefix = Linking.createURL('/');  // automatically sets prefix, considering Expo mode (dev/standalone)

const linking = {
  prefixes: [prefix],  // list of allowed URL prefixes
  config: {
    screens: {
      Home: "home",
      Profile: "profile/:userId",
      // ...mapping subsequent screens to paths
    }
  }
};

return (
  <NavigationContainer linking={linking} fallback={<Text>Loading...</Text>}>
    {/* navigators */}
```

```
  </NavigationContainer>
);
```

A few explanations for the above:

- prefixes – is an array of URL prefixes that are to be captured. Most often we place our scheme here (e.g. "myapp://") and potentially URL addresses of our domain if we want to handle so-called universal links/app links. The code uses Linking.createURL('/') from expo-linking, which generates the appropriate prefix both for dev mode (exp://IP:PORT/--/) and for production (myapp://). It is worth adding a prefix with https:// of our page as well, if we plan web -> app integration. E.g.: prefixes: [Linking.createURL('/'), 'https://my-domain.com']
- config – here we define the mapping of screen names to URL paths. In the example above, we defined the Home screen under the path "home" (so myapp://home fires Home), and Profile under "profile/:userId". The colon indicates parameters in the path – in this case, every path myapp://profile/XYZ will cause a transition to the Profile screen with route.params = { userId: "XYZ" }. We can also map nested screens using nested screens objects. E.g. if Profile is in a nested stack, config might look like:

  JavaScript

  ```
  config: {
   screens: {
     Home: "home",
     ProfileStack: {   // stack name as screen in main nav
       screens: {
         Profile: "profile/:userId",
         EditProfile: "profile/edit"
       }
     }
   }
  }
  ```

  Such config might seem complicated, but it boils down to reflecting the application navigation structure using nested objects.

- fallback component – NavigationContainer accepts a fallback UI for the time when it processes the link and fires the proper screen. In the above, we simply set the text "Loading...". One can display a splashscreen or nothing, just don't leave the user in uncertainty.

After this configuration, React Navigation automatically:

1. Checks at application start if it was launched from a link (so-called initial URL). If so, it parses the URL relative to prefixes and config and sets the appropriate initial state of the navigator. That is, e.g., it immediately loads the stack with the Profile screen and userId parameter instead of the default screen.
2. When the application is already running and a new link comes (e.g. via Linking.addEventListener in RN), navigation to the indicated place will occur (navigator

state update). We don't have to call Maps ourselves – the linking mechanism will do it for us.
3. Handles web specifics correctly (if we build RN for web, links will use URL paths and browser history).

Expo vs deep linking: In Expo Dev client mode uses a different scheme (exp://127.0.0.1:19000/--/ with path) and we cannot easily test a custom scheme without a standalone build. Fortunately, Linking.createURL('/') does this for us – in developer mode, the prefix will be the exp:// address with our manifest, and in the built app, the prefix will be myapp://. Thanks to this, we can test deep links during development: e.g. firing the command: npx uri-scheme open "myapp://profile/42" --android (analogously --ios for iOS) – expo CLI will transform this to the appropriate link (exp://...) and pass it to the application. Alternatively, in Expo Go, one can copy the exp:// link with a parameter after --/ (which in practice means path in the application). If we have a standalone application, then on a physical device clicking the link myapp://... from any application (e.g. from a note or browser) should trigger the opening of our application.

Universal Links / App Links: Besides a custom scheme, one can also register the application to handle HTTP(S) links from one's own domain. E.g. clicking https://my-domain.com/invite?code=abc can open the application. In iOS this is called Universal Links, in Android App Links – they require meeting additional security conditions (confirmation of domain ownership via apple-app-site-association file on the server, and entries in Xcode and Digital Asset Links JSON on Android). Expo also supports this: in app.json we add associatedDomains for iOS, and in AndroidManifest analogous intent-filter with android:autoVerify="true" and the domain host. The details of this process are somewhat beyond the scope of this lecture (it's a topic for a separate session). Let's just know that such a possibility exists – so that our web links open the native application, which improves UX (so-called app/website integration).

Handling outgoing links: Let's briefly mention that RN Linking also allows opening external links from the application level – e.g. Linking.openURL('tel:123456789') will call the phone, and Linking.openURL('https://google.com') will open the page in the default browser. This is the reverse of deep links but is sometimes useful (e.g. privacy policy link opens browser). In the context of React Navigation, it is worth being careful not to confuse navigation.navigate (for internal navigation) with Linking.openURL (for external). If opening a URL is handled by ourselves (e.g. own scheme), it is better to use navigation.navigate with an appropriate parameter than to emulate it via openURL.

Debugging deep links:

- In dev mode observe Metro logs – when a link arrives, information about URL parsing attempt should appear.
- If the link doesn't work, make sure prefixes contains the proper scheme/protocol, and config accurately reflects the path. Typos in screen names in config can cause the link to be ignored.
- On Android, if the link myapp://... does not open the application, possibly another application registered the same scheme – change the scheme to a more unique one.

- For testing use npx uri-scheme (fast and convenient) and e.g. in XCode Devices -> Open URL for iOS Simulator. On Android emulator adb shell am start -W -a android.intent.action.VIEW -d "myapp://..." com.yourpackage will trigger the intent.
- Check edge cases too: e.g. what if the user had the application open on another screen and clicks a link? (It should redirect to the new screen within the already open app – RN will handle this automatically via navigator state update).

Deep linking opens cool integration possibilities but adds a lot of complexity. For order, below is a short summary of steps that need to be performed to fully implement deep links:

1. Registration of application URL scheme (Expo: app.json, iOS: Info.plist, Android: AndroidManifest.xml).
2. Configuration of React Navigation: setting NavigationContainer prop linking with prefixes and mapping of paths to screens.
3. (Optional) Universal/App Links: domain association configuration so HTTPS links also hit the application.
4. Tests: use of uri-scheme or physical link clicking, checking correct opening.
5. Fallback: providing sensible UI fallback if the link is invalid (one can handle the Linking.addEventListener('url', ...) event manually if we want custom behavior in some cases). For the needs of this lecture, it is worth simply being aware that deep linking exists and knowing the configuration basics.

Protected Routes and conditional flow Many applications require user authentication – i.e., certain screens are available only after logging in. We must therefore protect routes from unauthorized access. In React Navigation there is no ready-made "Route Guards" mechanism like e.g. in Angular, but we can achieve the same by conditionally rendering appropriate screens/navigators depending on the application state (login state, onboarding completion, etc.).

Route guards – concept A route guard is a piece of logic that decides whether to let a user onto a given route or redirect them elsewhere. In web React Router this is e.g. a <PrivateRoute> component checking auth. In React Navigation, the approach is slightly different: we most often hide entire navigation sections when the condition is not met, instead of redirecting at the moment of entry. This can be done in two ways:

1. **Statically at navigator configuration:** In the newest Static API RN7, one can add the option if: someCondition at screen definition. When the condition (function/hook) returns false, the screen is not available in navigation at all. E.g.:

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

   In the pseudocode above, useIsSignedIn and useIsSignedOut are hooks returning booleans depending on auth state. React Navigation will then automatically show only those screens whose condition is met – never both simultaneously. When the state

changes (user logs in/logs out), the navigator updates the set of screens: one will disappear, the second will appear. This is an elegant solution available in Static API.

2. **Dynamically in JSX code:** This approach is available always (also in RN6 and lower). It involves conditionally rendering <Screen> components or entire navigators. Example:

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

   If the user is logged in (isSignedIn === true), only the Home screen will be added to the stack. If not – only the SignIn screen. An unauthorized user has no way to even try entering Home because the navigator doesn't know it. When the isSignedIn state changes, the component will re-render with a new set of screens – the old screen will be removed (unmount) and the new one will appear.

Both methods boil down to one thing: differentiating navigation configuration based on application state. The second method (dynamic JSX) is easier to understand and good enough, so we will focus on it.

Differentiating flow based on user state A typical scenario is an application with a distinction between:

- **Not logged-in user** – sees login/registration screens (so-called AuthStack).
- **Logged-in user** – sees the main part of the application (so-called AppStack or AppTabs).
- **Onboarding (optional)** – a new user who has logged in but needs e.g. to go through a tutorial or fill out a profile before gaining full access.

We can thus distinguish several states: signedOut, signedIn, possibly signedInButNotOnboarded. Depending on this, we show different navigation. Implementation can look as follows (pseudo-code using useContext hook to hold login state):

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

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

 if (isLoading) {
  // e.g. splash/loading screen while checking token in storage
  return <SplashScreen />;
 }

 return (
  <NavigationContainer>
```

```
    {user == null ? (
      <AuthStackNavigator />    // user not logged in
    ) : user.onboardComplete === false ? (
      <OnboardingStackNavigator />  // logged in, but didn't pass onboarding
    ) : (
      <MainAppNavigator />     // logged in and ready - main app
    )}
   </NavigationContainer>
  );
}
```

In the pseudo-code above:

- isLoading is a state saying that we are still e.g. checking in AsyncStorage if a token is saved (we show Splash then instead of flickering the login screen after launching the app).
- user is the user object or null, held somewhere in global state (e.g. Context or Zustand, about that in a moment).
- We check sequentially: if no user -> we render navigator with login screens (AuthStackNavigator); if there is a user but unfinished onboarding -> we render OnboardingStackNavigator; otherwise (fully logged in) -> MainAppNavigator with proper application screens.

Each of these navigators (AuthStack, OnboardingStack, MainApp) is e.g. a separately defined Stack or Tab with its own screens. We could also, instead of three separate navigators, use one conditionally adding Screens – it's a matter of code organization. Often they are separated for readability.

What does such an approach give? The user will never see a screen to which they should not have access:

- When they are not logged in – there is no route from the main application in the navigation tree (neither Home, nor Profile, etc.). Even if they knew the screen name, they won't go to it (navigation.navigate('Home') will throw an error/missing such route).
- When they are logged in – we remove login screens from the stack, so they cannot return to Login. In the example with dynamic configuration change, previous screens are unmounted, which means that e.g. pressing hardware back won't go back to Login because login no longer exists in the navigator. We thereby met the requirement of cutting off history.
- When they are in the middle of onboarding – until they finish, we won't let them into MainApp (because the condition won't allow rendering MainAppNavigator).

This approach is preferred over trying to redirect in useEffect of individual screens. Sometimes people implement in components something like: if user = null then navigation.replace('Login'). Something like that works but is less transparent – it is better to decide what is available at the navigation configuration level.

Implementation of user state (auth): Usually we need global state storing information about whether we have a token/credentials. One can use for this:

- **Context API (AuthContext)** – as in the example above, a simple context holding the user object and possibly login/logout functions.
- **Zustand or MobX/Redux** – any central state storage. Zustand is a lightweight store where we can hold `isLoggedIn` and e.g. user profile. Its advantage is simplicity and lack of Redux boilerplate. Usage example:

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

  Then we use `user` in the condition to switch navigators.

- **AsyncStorage/SecureStore** – used for persistent storage, i.e., e.g. saving a JWT token or first launch flag. During application start we do something like this:

  ```
  useEffect(() => {
    SecureStore.getItemAsync('token').then(storedToken => {
      if(storedToken) {
        // we have token, can try to refresh user profile from API etc.
        setUserToken(storedToken);
      }
      setLoading(false);
    });
  }, []);
  ```

  When we load the state from memory, we set `isLoading` to `false` and conditional rendering will show appropriate screens. It is important that this loading is performed before we show any screens (hence Splash during this time).

Guards vs back navigation: With dynamic navigator change (Auth -> App), it is worth additionally securing the scenario where the user during login could have had some stack of screens (e.g. registration screen etc.). When they log in and we render `MainApp` instead of `Auth`, old screens will disappear. If however the user quickly presses "back" (e.g. right after logging in on Android), this can cause exiting the application, because in the new navigator there will be no history. This is generally OK, but if we wanted e.g. to block exit, we would have to handle this manually (e.g. `BackHandler` and ignoring in the first seconds after login). In most cases however immediate pressing of back after login is unlikely, and even if so – leaving the application on the Home screen is consistent with expectations (user thinks they are going back to login, but the app closes because login no longer exists – next time the app will open already logged in). This is a minor UX detail to consider in a real project.

Other uses of protected routes: Not just auth. We can conditionally show certain screens e.g. if the user has permissions (role-based access). Or e.g. `PremiumContent` screen only when `user.isPremium`. Then analogously – the condition decides about adding the screen to the navigator. If the condition can change during application operation, dynamic addition/removal of screens is fine. React Navigation 7 can react quite elegantly to condition

changes (thanks to the if option in static API) and e.g. remove unavailable screens from navigation without errors.

Summary: Route protection in RN boils down to separating the public part from the private application. The best practice is preparing separate navigators for different states and switching between them depending on that state.

Auth Flow in React Navigation Login flow is a special case of navigation management in an application. It consists of login/registration screens, possibly a welcome screen (onboarding), and the main part of the application for the logged-in user. A well-designed flow should meet the following assumptions:

- After launching the application, we check the authentication state (e.g. if there is a valid session token). While checking – we show a start screen (Splash).
- If the user is not logged in, we show login/registration screens (AuthStack).
- If the user is logged in, we direct them immediately to the main application (AppStack / AppTabs).
- After successful login/registration, we redirect the user to the main application and remove login screens from history so that one cannot go back.
- When the user logs out during application operation, we clear session data and direct them again to login screens (preferably clearing the main navigation history).

Additionally, if we have onboarding for new users (e.g. a short tutorial or consent screen), we must fit this into the scheme above. Often this is done via a flag in the user profile or in memory (e.g. hasSeenTutorial). Example approach:

- If a logged-in user has the flag onboarded = false, instead of AppStack immediately we give them OnboardingStack (e.g. a few guide screens). Only after completing them (when we set the flag to true) we switch to the main AppStack.

The difference between AuthStack and AppStack lies mainly in the set of screens:

- **AuthStack** – contains screens: Login, Registration, Forgot Password, Welcome Screen (e.g. with logo) etc. All these screens do not require authentication. Often this is a createNativeStackNavigator with header disabled or a custom header (e.g. app logo at the top).
- **AppStack** (or AppTabs if we use tabs) – contains screens available only after logging in, e.g. Home, Dashboard, Profile, Settings etc. This can be a stack or tab depending on application design.

Sometimes the concept of SplashStack is also used with only one loading screen (Splash), which is displayed by default before we decide what next. But Splash can equally well be rendered conditionally as in the previous chapter.

When implementing the login flow in React Navigation, it is worth relying on examples from documentation. The creators of RN suggest using Context to hold auth status and show an example using SecureStore (expo) to hold the token. The most important thing is separating

navigation trees: separate for "signed in" and "signed out", and switching between them after auth state change.

Onboarding – first launch Onboarding can take various forms:

- Welcome screen with a "Start" button (e.g. presentation of the main value of the application).
- Several scrollable screens (carousel) with user manual, user preference questions etc.
- Language/theme selection screen etc.

From the navigation perspective, onboarding is also a certain mini-stack of screens that we show only once for a new user. Most often this is implemented via:

- Persistent flag, e.g. saving in AsyncStorage `hasOnboarded=true` after passing.
- Conditional inclusion of screen in AuthStack: e.g. first screen of AuthStack is Welcome, and only then Login. And if we determine that user already saw welcome (checking AsyncStorage), we navigate them immediately to Login skipping welcome. This can be done at application start (e.g. in Splash, deciding which route to go to). Or:
- Separate `OnboardingStack` as mentioned earlier, which we render before `AppStack`. This stack after completion sets the flag and we switch to `AppStack`.

Storing login information We partially discussed this topic already with route guards. In the context of login flow, it boils down to:

- **Storing token/authorization** – if we use an API, then after login we probably receive a token (JWT or session id). It must be stored securely. It is recommended to use `SecureStore` (Expo) or Keychain/Keystore on devices. `SecureStore` (Expo) saves data encrypted in the device's secure memory. `AsyncStorage` does not encrypt, so the token there is visible – for less sensitive things ok, but rather keep tokens securely. Ultimately, if testing, `AsyncStorage` can be used too.
- **Storing login state in the application** – so that the whole application knows the user is logged in and e.g. displays their data in various places. Context API or a global store (Zustand/Redux) is great for this. Context suffices: create `AuthContext` with value `{ user, login(), logout() }`. After successful login (e.g. when API returns 200 OK) we call `login(userData)` from the context object. This changes `user` and causes re-render of dependent components (including our conditional navigator). Effectively, transition to `AppStack` occurs.
- **Zustand** – has the advantage that even when context components unmount, state remains (but holding in `useState` in a component higher than `NavigationContainer` gives the same). For beginners – Context is easier.
- **Logout** – should clear all data: remove token from `SecureStore`, zero out `user` in state, possibly reset navigation. If we use conditional rendering of navigators, zeroing out the user will automatically re-render NavContainer to `AuthStack`, thereby removing all application screens from view. It is worth ensuring however that e.g. we don't leave some modal open etc. Usually `logout()` in context is enough.

Context usage example (pseudo-code):

```
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>
 );
}
```

We wrap the whole application in <AuthProvider> (e.g. in App.js). In components (e.g. login screens) we call const { login } = useContext(AuthContext) to log in, and in profile screen const { logout } = useContext(AuthContext) to log out.

Best Practice: During login, when we go to the main application, we can reset the stack (if we used simple Maps) or, as we showed, remove login screens via conditional rendering. Many ready-made templates use navigation.reset({ routes: [{ name: 'MainApp' }] }) after login, where MainApp is e.g. tab navigator.

Summarizing: separating AuthStack and AppStack is the basis for organizing login flow. Let's keep login state globally to easily react to its changes in the navigation tree. Let's remember about safe storage of any tokens and about "cleaning up" after logout (clearing state and navigation).

Demo: Mini-application with navigation (AuthStack + AppTabs) Now let's move to a practical example that combines all discussed concepts. We will build a simplified React Native application that possesses two main navigation "modes":

- AuthStack – stack with login screen (LoginScreen).
- AppTabs – navigator with tabs containing two screens: HomeScreen and ProfileScreen.

An unlogged user will see the login screen. After "logging in" (we simulate it in the application) they will be moved to application tabs: Home and Profile. Home will display a welcome and enable transition to the Profile screen (e.g. via button) – incidentally we will pass user ID as a parameter. Profile will display the identifier of the logged-in user and give a logout option (return to login screen).

In the code, we will use React Context to store information about the logged-in user (userId), which will allow conditional rendering of the appropriate navigator. We will also show the usage of useNavigation and useRoute hooks inside screens.

Here is the code of the demonstration application:

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

// 1. Definition of parameter types for navigators (TypeScript)
type AuthStackParamList = {
  Login: undefined;
};
type AppTabsParamList = {
  Home: undefined;
  Profile: { userId: string };
};

// 2. Creation of navigators
const AuthStack = createNativeStackNavigator<AuthStackParamList>();
const AppTabs = createBottomTabNavigator<AppTabsParamList>();

// 3. Authentication context
type AuthContextType = { userId: string | null, login: (id: string) => void, logout: () => void };
const AuthContext = createContext<AuthContextType | undefined>(undefined);

// 4. Login Screen
function LoginScreen() {
  const auth = useContext(AuthContext);
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text> Login Screen</Text>
      <Button title="Log me in as user #123"
            onPress={() => auth?.login('123')} />
    </View>
  );
}

// 5. Home Screen (main tab)
function HomeScreen({ navigation }: { navigation: any }) {  // navigation typing omitted for readability
  const auth = useContext(AuthContext);
  const userId = auth?.userId;
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text> Home Screen - welcome user #{userId}!</Text>
      {/* Example button navigating to Profile with parameter */}
      <Button title="Go to Profile (param: userId)"
            onPress={() => navigation.navigate('Profile', { userId })} />
    </View>
  );
}
```

```
// 6. Profile Screen (profile tab)
function ProfileScreen({ route }: { route: any }) {
  const auth = useContext(AuthContext);
  // We retrieve userId from parameter or from context:
  const routeUserId = route.params?.userId;
  const userId = routeUserId || auth?.userId;
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>□ User Profile Screen #{userId}</Text>
      <Button title="Logout" onPress={() => auth?.logout()} />
    </View>
  );
}

// 7. Main navigation component based on 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 ? (
          // When no logged-in user -> we show AuthStack
          <AuthStack.Navigator>
            <AuthStack.Screen
              name="Login"
              component={LoginScreen}
              options={{ headerShown: false }}
            />
          </AuthStack.Navigator>
        ) : (
          // When logged in -> we show application tabs
          <AppTabs.Navigator screenOptions={{ headerShown: false }}>
            <AppTabs.Screen
              name="Home"
              component={HomeScreen}
              options={{ title: 'Home' }}
            />
            <AppTabs.Screen
              name="Profile"
              component={ProfileScreen}
              options={{ title: 'Profile' }}
            />
          </AppTabs.Navigator>
        )}
      </NavigationContainer>
    </AuthContext.Provider>
  );
}
```

(Code written with readability for beginners in mind – in practice navigation typing would be refined, but here we focus on the idea.)

Explanations for the code above:

1. At the beginning we define navigator types and create AuthStack and AppTabs navigators. AuthStack is a stack (NativeStack) with one Login screen. AppTabs are bottom tabs with two screens: Home and Profile. Note that for Profile we anticipated a userId parameter (of type string).
2. We create a context AuthContext, which will store the userId state (or null) and login and logout functions. The entire authorization logic is simplified here: we assume that clicking the "Log in" button always succeeds and logs us in as user #123. In a real app, instead of this, we would call e.g. a login API, and after success save the token and user id. Here we immediately do auth?.login('123'), which sets userId in the context state.
3. LoginScreen: Retrieves login from context (auth?.login) and after clicking logs in the user with ID "123".
4. HomeScreen: Displays a welcome with the user number (we retrieve it from context). It also has a button "Go to Profile (param: userId)". After clicking, we call navigation.navigate('Profile', { userId }). This demonstrates passing a parameter to the Profile tab – although in this case Profile could just as well benefit from the context, we show passing param for illustration. Since Home and Profile are in the same AppTabs navigator, calling Maps will cause switching the tab to Profile, passing it the parameter.
5. ProfileScreen: Receives parameters via { route } (receives them as a prop because it is a navigation screen). Extracts userId from route.params. For safety, if the param was not passed (e.g. user switched to Profile tab without using the button with parameter), then route.params will be undefined – then we take userId from context. In our flow, when the user first clicks "Go to Profile" on Home – param will be present. If, however, being on Home they use bottom navigation (click Profile icon in tab bar), they will switch the tab without parameter (React Navigation does not pass param upon simple tap on tab, because it is not navigation.navigate but internal switching). Thanks to our logic ProfileScreen will establish userId from context anyway. ProfileScreen displays the identifier and possesses a "Logout" button. After tapping: auth?.logout() sets userId = null. This will cause re-rendering of the entire navigation tree – because in App() the condition userId == null will become true again, AuthStack will show instead of AppTabs, meaning we will return to the login screen. This is immediate and effective – tabs disappear, login stack appears.
6. In the main <App> component we embed NavigationContainer and conditionally choose between <AuthStack.Navigator> and <AppTabs.Navigator>. Let's note: AuthStack.Navigator and AppTabs.Navigator are two completely separate navigators – they do not have common screens. Switching occurs purely via React (ternary operator) and React Navigation handles mounting/unmounting one navigator and replacing it with the second one without problem.
7. We surround NavigationContainer with AuthContext.Provider so that screens have access to auth values.

Testing the scenario:

1. After launch: userId is null, so AuthStack with LoginScreen renders. LoginScreen shows the button. When we press it:
2. auth.login('123') sets userId = '123'. Now userId is not null, so App re-render: instead of AuthStack, AppTabs appears in NavigationContainer. The user will see the Home screen (default first tab Home).
3. HomeScreen displays "Welcome user #123". The user has two paths: either they click the "Go to Profile" button (which will call navigation.navigate with param), or they simply select the Profile tab at the bottom.
    - If they clicked the button: they get switched to the Profile screen, and it receives route.param { userId: '123' }. It will display "User Profile #123".
    - If instead the user touched the Profile tab on the tab bar: they will get switched to ProfileScreen, but without parameter. Our code in ProfileScreen will see that route.params is undefined and will take userId from context, also getting '123'. Final result for the user identical – they see profile #123.
4. On the Profile screen there is a "Logout" button. After tapping: auth.logout() sets userId = null. AppTabs unmount and AuthStack mount back occurs (NavigationContainer takes down tabs and inserts login). The user sees the login screen again. If they pressed back on Android at this moment – it will close the application (because we are on the only screen in the main stack). We can treat this as expected behavior (they logged out, so exiting the app is logical, or they can log in again).

Notes regarding debugging and extending:

- If we didn't use context, an alternative would be e.g. holding userId in state higher up and passing it to screens as props (via initialParams or own props in navigator). Context is however more convenient.
- In a real app after logging in, we would probably want to execute navigation.reset instead of simple Maps – but here we did it "architecturally" removing AuthStack completely.
- During development, one can easily check if definitely after logging in the login screen does not remain in memory – e.g. in React DevTools checking the component tree or logging unmount in useEffect in LoginScreen. It should be unmounted upon transition.
- Our userId param is a string – in a real API this could be a JWT token or user object. Then we rather don't pass this via navigation params (because it's sensitive data), just hold in context/state. Parameter in navigation is more useful for data regarding a specific sub-page, e.g. postId for PostDetails screen. For global userId better use context as we showed, which we did anyway.
- If we had more screens in AuthStack (e.g. Register, ForgotPassword), we could add them to AuthStack.Navigator. Then from LoginScreen normally navigation.navigate('Register') would work. After login independently, the whole AuthStack navigation is swapped.
- Similarly in AppTabs – we could easily add e.g. a third tab Settings without impact on auth logic.

Summary and best practices During this lecture, we went through all key aspects of navigation in React Native using React Navigation:

- We got to know different types of navigators and their applications (Stack for screen sequences, Tabs for parallel sections, Drawer for hidden menu).
- We learned to pass parameters between screens and receive them, as well as how to take care of their correct typing in TypeScript.
- We utilized navigation hooks `useNavigation` and `useRoute` for convenient navigation invocation and route data retrieval in functional components.
- We discussed how to reverse navigation, both singly (`goBack`) and in bulk (`popToTop`), and how to reset the screen stack – which turned out to be important in the login scenario.
- We dealt with the topic of deep links, configuring URL schemes and integration with Expo so that our application can react to external links and open proper screens.
- We introduced the concept of protected routes – limiting access to parts of the application via conditional rendering of navigators depending on state (e.g. login). Thanks to this we realized the authentication flow, separating login screens from the main application.
- We culminated everything with a practical demo, which step by step showed the implementation of a mini-app with authorization context and two navigators. The demo illustrates how to bind React Navigation, Context and RN components together in real code to obtain a user-friendly flow (login -> application -> logout -> back to login).

Literature:

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