Mobile Applications

Lecture 3

Navigation and user flow – React Navigation in React Native

Mateusz Pawełkiewicz

October 2026

Expo SDK 58 · React Native 0.88 · React 19.3 · TypeScript

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

React Navigation – types of navigation

React Navigation is a library for defining different kinds of navigation in a React Native app. The basic navigator types are: the Stack Navigator (a stack of screens), the Bottom Tab Navigator (navigation with tabs) and the Drawer Navigator (navigation with a side drawer). Each of them corresponds to a different pattern of moving around the app:

- **Stack Navigator:** Lets you move to the next screens by placing them on a stack – a new screen is put on top of the previous one, and going back returns to the earlier screens. It is analogous to navigation on iOS/Android: screens come in from the right or in the default system style. By default React Navigation provides transition animations that match the platform (a horizontal slide on iOS, a standard fade/slide on Android). The stack comes in two variants: `@react-navigation/stack` (in JavaScript) and `@react-navigation/native-stack` (using the native navigation APIs). The JavaScript version is more configurable but may be a little less performant, so for complex animations it is worth considering the native variant for better smoothness. The Stack Navigator works well for sequential flows of screens – e.g. a list of articles → the article details screen.
- **Bottom Tab Navigator:** Provides navigation with a tab bar, most often placed at the bottom of the screen. The user can switch between tabs that represent independent sections of the app (e.g. Home, Search, Profile). Each tab has its own screen (or a nested stack of screens). Tabs are initialized lazily (a tab screen loads only on the first visit) and kept in memory, which allows a quick return without losing the state. We can customize the tab icons and labels as well as the style of the bar. This navigator is ideal when the app has a few main modules available in parallel (e.g. the home page and the settings screen as separate tabs).
- **Drawer Navigator:** Offers navigation with a menu that slides out from the side of the screen (the so-called hamburger menu). On Android it is usually opened with a swipe from the edge, and on iOS often also with a hamburger icon in the header. The Drawer Navigator is useful for holding many navigation options or global navigation, e.g. a side panel of the app with links to different screens (profile, settings, FAQ, etc.). In React Navigation the drawer is based on the `react-native-drawer-layout` component under the hood and, to work properly, it needs dependencies such as `react-native-gesture-handler` and `react-native-reanimated` (installed with npx expo install). When configuring the Drawer Navigator we define the screens available in the menu; we can also customize the position of the drawer (left/right) or the content of the header. The drawer works well when we want to hide the navigation behind a gesture, leaving more space on the main screen.

Nesting navigators: In a real app we often use several types of navigation at once, nesting them in each other. For example, we can have a main Drawer Navigator in which each option opens a separate Stack or Tab navigator. Or a popular case: a Tab Navigator as the main navigation of the app, where the individual tabs have inner Stack Navigators (e.g. the Home tab has a stack with the main screen and a details screen). React Navigation lets you treat a navigator like a screen – e.g. we can define a screen in a Stack Navigator whose component is MyTabs (our Tab Navigator). This way the navigators form a hierarchy. Remember that each navigator has its own namespace for routes – i.e. the screens in a nested navigator have names independent of the names of the screens in the parent. To navigate between nested navigators, we usually navigate relative to the common NavigationContainer or use full paths (e.g. navigation.navigate('NavigatorName', { screen:

'ScreenName', params: {...} })). Nesting navigators lets you combine different navigation patterns – e.g. bottom tabs plus options in a drawer, or a sign-in stack separate from the main app stack.

Passing parameters between screens: When we go to the next screen, we often want to pass it some data (e.g. the ID of the object whose details it should show). React Navigation allows this through the second argument of the navigation function. For example, with 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 parameters 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 read the parameters:

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

We read `itemId` and `otherParam` from the parameters object passed with the navigation. Note that parameters are best passed as simple JSON (primitive types, objects, arrays) – this way they are serializable, which makes it easier e.g. to save the navigation state or handle deep links.

We can also define the initial parameters of a screen independently of navigation. If, for example, we want the Details screen to have `itemId`: 42 by default, we can set `initialParams` in the screen definition:

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

The parameters passed with navigate override the initial values, and if we pass none, the screen uses `initialParams`. A screen can also update the parameters of its own route while it is running by calling `navigation.setParams({ ... })` – e.g. to change parameters that affect the header UI, etc. (useful for dynamic titles).

A few good practices for passing parameters:

- Pass only the necessary data (e.g. an ID, a key), and fetch larger objects again on the target screen or use global state. This way you avoid serialization problems and separate responsibilities.
- Make sure the screen that receives the parameters handles a missing parameter (e.g. if the navigation happened without it). You can define default values: route.params?.`userName` ?? 'Anonymous', etc.
- In TypeScript it is worth specifying the parameter types for each route – the editor then catches missing required parameters already at compile time. We will discuss typing navigation later.

navigate, push, popTo and replace in React Navigation 7

`navigation.navigate(name, params)` – goes to a screen. If the screen with this name is on top, it updates its parameters; otherwise it adds a new screen to the stack (it does not go back to a screen lower in the stack – `popTo` is for that).

`navigation.popTo(name, params, { merge })` – goes back to an existing screen on the stack (removing the screens above it) and optionally merges new parameters – ideal for "returning a result" (e.g. filters → list).

`navigation.push(name, params)` – always adds a new screen, even the same one (e.g. one vet → another, similar vet). With the screen option `getId: ({ params }) => params.id`, a different identifier always means a new screen.

`navigation.replace(name, params)` – replaces the current screen (e.g. after finishing onboarding or a booking – "Back" does not return to the form).

Navigation hooks and navigation in practice

React Navigation provides special hooks that make using navigation in function components easier. The main ones are `useNavigation` and `useRoute`. We will discuss how they work and show how to use them with TypeScript for full type safety. We will also look at the methods for going back and resetting the stack – essential for managing the history of screens.

`useNavigation` – access to the navigation object

```
const navigation = useNavigation(); // the navigation object of the current screen
```

`useNavigation()` returns the navigation object (the navigation prop) of 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: let us assume we want to create a "Back" button component that works anywhere in the app:

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

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

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

Here the `useNavigation()` hook gives us the current navigation object, and we call `navigation.goBack()` on press. Pressing the button goes back to the previous screen on the stack (the same as using the system back gesture/button).

Use with classes: If we have to use `useNavigation` in a class component (which does not support hooks), we can wrap the class component in a function that uses the hook and pass navigation as a prop:

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

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

In general, however, new apps stick to function components and hooks.

Typing `useNavigation`: By default, without extra configuration, the object returned by `useNavigation` has a general type (it contains no information about the available routes and parameters). To get better hints and control, we can use a generic type. For example, if we have defined the parameter type of the main navigator, RootStackParamList (defining the types is discussed shortly), we can write:

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

Then e.g. `navigation.navigate('Home', ...)` checks that the route name and the parameter types match RootStackParamList. Another approach is to declare a global param list type – then `useNavigation()` uses it automatically without passing a generic. We will discuss this approach in the TypeScript section. Note: `useNavigation` must be called in the context of a screen connected to a navigator (i.e. inside the NavigationContainer tree). If we use it outside a navigator, we get an error about the missing context. In practice this means that, for example, we cannot call `useNavigation` in the code that renders NavigationContainer – only inside components that are screens or their children. If we need to navigate globally from outside components (e.g. from a module or a service), React Navigation offers a reference to the container created with `createNavigationContainerRef()` (or the `useNavigationContainerRef()` hook), passed as the ref of NavigationContainer – e.g. `navigationRef.navigate('Appointment', { id })` when handling a push notification.

`useRoute` – information about the current route

```
const route = useRoute(); // the current route object: name, params, key
```

`useRoute()` gives us the object of the current route in a screen. Normally a screen component receives route in its props ({ route, navigation }), but if we nest the logic deeper or want to use a hook instead of props, `useRoute` solves the problem.

A typical use case is reading the parameters passed to the screen or the route name. 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 the parameters as text. Of course, we are usually interested in a specific field of route.params – e.g. route.params.`userId`. `useRoute` is especially useful in a component that does not receive the route prop (e.g. a grandchild of a screen) but needs this data.

Typing `useRoute`: As with `useNavigation`, it is worth giving the returned object a proper type. 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>();
```

TypeScript then knows that route.params has the structure { `itemId`: number } – so it immediately catches typos in field names or wrong types.

To make full use of TypeScript in navigation, we should define the parameter types for all screens and use them when creating navigators and hooks. The process looks like this:

1. Defining the parameter list – we create an object type whose keys are the route names and whose values are the parameter types (or undefined if there is no parameter). An example for a simple stack:

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

and for the main part of the app with tabs:

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

Each key is a screen name, and its value is the type of the screen's parameters.

Here we assume that the Home screen needs no parameter, while the Profile screen requires `userId` (e.g. the identifier of the user whose profile we display).

1. Creating a navigator with the type above – when calling the navigator factory, we pass the param list type as a generic parameter. For example, for a stack:

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

and for tabs:

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

Now the `<Stack.Screen>` and `<Tab.Screen>` components expect a name and a component that match the routes defined in the type.

1. **Typing screen props:** We can get the navigation and route types for a specific route with the provided utility types. For example, for the Profile screen in the 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 to combine the stack and tab types. For simplicity, you 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;

...
```

}

This approach gives full safety: if we try navigation.navigate('Home',

{userId: '123'}) but Home takes no parameter, TS reports an error. Or the other way round – calling navigation.navigate('Profile') without the required userId is an error.

1. **A global param list type:** React Navigation lets you declare a default param list type in the global namespace (namespace ReactNavigation). It is enough to write something like this in a definition file (e.g. types.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 is applied. This makes using hooks simpler – we do not have to pass generics every time. Just make sure the declaration is within the project scope (included by tsconfig).

Summary: Setting up TypeScript with navigation takes a little work at the start, but it pays off. We get autocompletion of screen names, certainty about the parameters and their types, and fewer runtime errors. In large apps it is almost a necessity. React Navigation 7 also offers the so-called Static API, which simplifies the navigator configuration and improves type inference (we define the screens as objects with keys, which makes it easier to derive the param list types). Whatever the approach, it is always worth defining param lists and using the provided helper types.

Going back to the previous screen and resetting the stack

Navigation is not only moving “forward” (push/navigate), but also going back and managing the history of screens. These are the most important operations:

- Going back (goBack): Every navigation object has the navigation.goBack() method, which takes us one screen back (just like pressing the system back button on Android or the back gesture on iOS). If we are on the first screen of the stack, calling goBack() by default closes the whole app (or moves it to the background) on Android – keep this in mind when handling the hardware back button (this behavior can be overridden, more on this shortly). In the MyBackButton shown earlier we used navigation.goBack(). There is also the navigation.canGoBack() method, which checks whether there is anything to go back to (it returns true/false).
- **Popping the stack:** An alternative to goBack is navigation.pop(n), which goes back n screens (1 by default). In addition, navigation.popToTop() takes us straight to the beginning of the stack (the first screen). These methods are useful e.g. to leave a multi-step form straight to the main screen, etc.
- **Resetting the navigation (reset stack):** Sometimes we want to change the navigation state completely – e.g. clear the history and insert a new set of screens. An example is the sign-in flow: after a successful sign-in we do not want the user to return to the sign-in screen with the “back” button. In such a case we can reset the stack. React Navigation provides the CommonActions.reset action for defining a new navigation state. It is used more or less like this:

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

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

```
 })
);
```

The code above clears the whole history and sets a stack containing a single Home route. We can of course give more routes in the array, even specifying the parameters of each of them. For example:

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

Here we set the new stack to have two screens: Profile (at index 0) and Home (index 1), and we start right away at index 1, i.e. Home. The effect: the user sees Home, back goes to Profile, and the earlier screens have been removed.

Note: When using reset, take care of the consistency of the state. The reset action replaces the whole navigator state with a new state object. If we leave out some keys or give two screens the same key, we can make the state inconsistent. We usually limit ourselves to setting index and a new list of routes containing name and optionally params. Avoid manipulating the internal structures of the navigation state – the CommonActions.reset API is enough in 99% of cases. When reacting to changes in the app state (e.g. signing out), however, we often do not need to reset manually and instead render different navigators conditionally (discussed in the section on protected routes). Such a change of the navigation configuration automatically removes the previous screens from view.

- Blocking going back (preventing `goBack`): Sometimes we want to prevent going back from a specific screen (e.g. a "confirm the booking" screen – the user should not go back to the form). React Navigation provides the `usePreventRemove()` hook to block leaving a screen. You can also handle the `beforeRemove` event on the navigator. It is worth knowing that you can take control of the physical back button/gesture, show e.g. a modal "Are you sure you want to leave?" and block the navigation conditionally:

usePreventRemove (React Navigation 7)

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

function BookVisitScreen({ navigation }) {
  const [selectedSlots, setSelectedSlots] = useState<string[]>([]);

  // the block works for the back button, the gesture, goBack() and navigation actions
  usePreventRemove(selectedSlots.length > 0, ({ data }) => {
    // our own replace after booking passes without asking
    if (data.action.type === 'REPLACE') return navigation.dispatch(data.action);
    Alert.alert('Discard your selection?', 'The selected time slots will not be booked.', [
      { text: 'Stay', style: 'cancel' },
      { text: 'Discard', style: 'destructive', onPress: () => navigation.dispatch(data.action)
},
    ]);
  });
  // …
}
```

- **Android specifics:** On Android the physical back button is integrated with the navigator by default – it works like `navigation.goBack()` for the topmost navigator on the stack (unless we have overridden this behavior). When the user is on the initial screen of the main navigation, by default this button closes the app. We can change this behavior with BackHandler from RN, but if the navigation logic is built correctly (and e.g. we block going back where it makes no sense), the default behavior is usually fine.

Section summary: The `useNavigation` and `useRoute` hooks simplify access to the navigation and route information in function components. Combined with a well-configured TypeScript setup, they provide convenience and safety. Remember the methods for going back and resetting – when building more complex user flows (e.g. sign-in -> the main app), they are essential for correct behavior (e.g. not being able to go back to the sign-in screen). In the next sections we will see how to apply these mechanisms in practice, among others when implementing deep links and protected routes that require authentication.

Deep linking – handling external links

Deep linking is a mechanism that lets you open a mobile app at a specific place in its navigation with a link/URL. For example, tapping the link myapp://profile/42 in a browser can directly open our RN app and take the user to the profile screen of the user with ID 42. Thanks to deep linking we can integrate the app with emails, a website or other apps (e.g. opening a link from Facebook launches our app).

To handle deep links, a few things have to be configured in the app and in the external sources (the system and possibly a web server):

Defining the URL scheme

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

- **Expo:** For an Expo app it is easiest to define the scheme in the configuration file app.json / app.config.ts. In the expo section we add the field "scheme": "myapp" (of course with any unique name instead of myapp). When building the app, Expo sets this scheme in the right places on Android/iOS automatically.
- **An RN app without Expo configuration (bare workflow):** You have to register the URL types manually:
    - **iOS:** add CFBundleURLTypes with an entry for myapp in Info.plist.
    - **Android:** in AndroidManifest.xml add an <intent-filter> with <data scheme="myapp" ...> that allows opening the main activity with this scheme.
    - **(Bare workflow:** you can use the npx uri-scheme add myapp command, which automates these steps.)

After setting the scheme, e.g. myapp://, any external app that refers to such a URL launches/wakes up our app.

It is worth choosing a unique scheme so that it does not collide with others. A reversed domain name is often used, e.g. com.company.app://, but it is not required.

Configuring linking in React Navigation

Having a scheme is only the first step. Next we have to tell React Navigation how to map incoming links to routes in our navigator. The linking configuration in `<NavigationContainer>` is used for that.

React Navigation can integrate with the RN Linking module to listen for links. We can use the linking prop, passing a configuration object. An example of a minimal configuration (Expo):

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

```
const prefix = Linking.createURL('/'); // sets the prefix automatically, taking the Expo mode
into account (dev/build)

const linking = {
 prefixes: [prefix], // the list of allowed URL prefixes
 config: {
  screens: {

    Home: "home",
    Profile: "profile/:userId",
    // ...mapping more screens to paths
  }
 }
};

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

A few explanations of the above:

- prefixes – an array of URL prefixes that should be intercepted. Most often we put our scheme here (e.g. "myapp://") and possibly the URLs 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 right prefix both for the dev mode (exp://IP:PORT/--/) and for production (myapp://). It is also worth adding the https:// prefix of our website if we plan a web -> app integration. For example:

```
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" (i.e. myapp://home opens Home), and Profile under "profile/:`userId`". The colon denotes path parameters – in this case every path myapp://profile/XYZ goes to the Profile screen with route.params = { `userId`: "XYZ" }. We can also map nested screens using nested screens objects. For example, if Profile is in a nested stack, the config can look like this:

Configuring nested screens

```
config: {
  screens: {
    Home: 'home',
    ProfileStack: {
      // the name of the stack as a screen in the main navigator
      screens: {
        Profile: 'profile/:userId',
        EditProfile: 'profile/edit',
      },
    },
  },
},
```

Such a config may look complicated, but it comes down to mirroring the structure of the app's navigation with nested objects.

The fallback component – NavigationContainer accepts a fallback UI for the time it processes the link and opens the right screen. Above we simply set the text "Loading...". You can show a splash screen or nothing, as long as the user is not left in uncertainty.

After this configuration React Navigation automatically:

- At app start-up checks whether the app was launched from a link (the so-called initial URL). If so, it parses the URL against prefixes and config and sets the right initial state of the navigator. This means, for example, that it immediately loads the stack with the Profile screen and the `userId` parameter instead of the default screen.
- When the app is already running and a new link arrives (e.g. through Linking.`addEventListener` in RN), it navigates to the given place (updating the navigator state). We do not have to call navigate ourselves – the linking mechanism does it for us.
- Handles the web specifics correctly (if we build RN for the web, links use URL paths and the browser history).

Expo and deep linking: In development (Expo Go) the client uses a different scheme (exp://127.0.0.1:8081/--/ with a path), and we cannot easily test a custom scheme without a build. Fortunately `Linking.createURL('/')` does this for us – in development the prefix is the exp:// address with our manifest, and in a built app the prefix is myapp://. Thanks to this we can test deep links during development, e.g. by running the command:

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

(analogously --ios for iOS) – the link is passed to the app. Alternatively, in Expo Go you can copy the exp:// link with the path after --/ (which in practice means the path in the app). If we have a built app, on a physical device tapping a myapp://... link from any app (e.g. from a note or a browser) should open our app.

Universal Links / App Links: Apart from a custom scheme, you can also register the app to handle `HTTP(S)` links from your own domain. For example, tapping https://my-domain.com/invite?code=abc can open the app. On iOS this is called Universal Links, on Android App Links – they require meeting additional security conditions (confirming the ownership of the domain with an apple-app-site-association file on the server, plus entries in Xcode, and a Digital Asset Links JSON on Android). Expo supports this too: in app.json / app.config we add ios.`associatedDomains` (e.g. "applinks:my-domain.com") and android.`intentFilters` with `autoVerify`: true and the domain host – Expo generates the entries in Info.plist and AndroidManifest.xml. The details of this process are somewhat beyond the scope of this lecture (it is a topic for a separate session). Just be aware that it is possible – so that our web links open the native app, which improves UX (so-called app/website integration).

Handling outgoing links: Let us briefly mention that RN Linking also lets the app open external links – e.g. `Linking.openURL('tel:123456789')` starts a phone call, and `Linking.openURL('https://google.com')` opens the page in the default browser. It is the opposite of deep links, but it is useful (e.g. a link to the privacy policy opens the browser). In the context of React Navigation, be careful not to confuse navigation.navigate (for internal navigation) with Linking.openURL (for external navigation). If opening a URL is handled by ourselves (e.g. our own scheme), it is better to use navigation.navigate with the right parameter than to emulate it with openURL.

Debugging deep links:

- In dev mode watch the Metro logs – when a link arrives, there should be information about the attempt to parse the URL.
- If a link does not work, make sure prefixes contains the right scheme/protocol and config exactly matches the path. Typos in screen names in the config can make the link ignored.
- On Android, if a myapp://... link does not open the app, another app may have registered the same scheme – change the scheme to a more unique one.
- For testing use npx uri-scheme (quick and convenient) and e.g. Xcode Devices -> Open URL for the iOS Simulator. On the Android emulator adb shell am start -W -a android.intent.action.VIEW -d "myapp://..." com.yourpackage fires the intent.

- **Also check edge cases:** e.g. what if the user had the app open on another screen and taps a link? (It should go to the new screen within the already open app – RN handles this automatically by updating the navigation state.)

Deep linking opens up nice integration possibilities, but it adds a lot of complexity. For order, here is a short list of the steps needed to fully implement deep links:

1. **Registering the app's URL scheme (Expo:** app.json, iOS: Info.plist, Android: AndroidManifest.xml).
2. **Configuring React Navigation:** setting the linking prop of NavigationContainer with prefixes and a mapping of paths to screens.
3. **(Optionally) Universal/App Links:** configuring the domain association so that HTTPS links also reach the app.
4. **Testing:** using uri-scheme or tapping links physically, checking that the app opens correctly.
5. **Fallback:** providing a sensible fallback UI if the link is invalid (you can handle the `Linking.addEventListener('url', ...)` event manually if you want custom behavior in some cases).

For the purposes of this lecture, it is enough to be aware that deep linking exists and to know the basics of its configuration.

Protected routes and conditional flows

Many apps require the user to authenticate – i.e. some screens are available only after signing in. So we have to protect routes against unauthorized access. React Navigation has no ready "Route Guards" mechanism like e.g. Angular, but we can achieve the same by rendering the right screens/navigators conditionally depending on the app state (the sign-in state, completed onboarding, etc.).

Route guards – the concept

A route guard is a piece of logic that decides whether to let the user into a given route or redirect them elsewhere. In the web React Router it is e.g. a wrapper component that checks the sign-in state:

```
<Route path="/profile" element={<PrivateRoute><ProfilePage /></PrivateRoute>} />
```

In React Navigation the approach is a bit different: most often we hide whole sections of the navigation when the condition is not met, instead of redirecting at the moment of entering.

It can be done in two ways:

- **Statically in the navigator configuration:** In the Static API of React Navigation you can add the option `if: someCondition` to a screen definition. When the condition (a function/hook) returns false, the screen is not available in the navigation at all. For example:

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

The conditions themselves are ordinary hooks:

```
// the conditions are ordinary hooks – e.g. based on the authentication context
function useIsSignedIn() {
```

```
  return useAuth().isSignedIn;
}
function useIsSignedOut() {
  return !useAuth().isSignedIn;
}
```

In the pseudocode above `useIsSignedIn` and `useIsSignedOut` are hooks that return booleans depending on the auth state. React Navigation then automatically shows only the screens whose condition is met – never both at once. When the state changes (the user signs in/out), the navigator updates the set of screens: one disappears and the other appears. It is an elegant solution available in the Static API.

- **Dynamically in JSX code:** This approach is always available. It means rendering `<Screen>` components or whole navigators conditionally. For example:

```
<NavigationContainer>
<Stack.Navigator>
        {isSignedIn ? (

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

If the user is signed in (`isSignedIn` === true), only the Home screen is added to the stack. If not – only the SignIn screen. So an unauthorized user cannot even try to enter Home, because the navigator does not know it. When `isSignedIn` changes, the component re-renders with the new set of screens – the old screen is removed (unmounted) and the new one appears.

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

Varying the flow based on the user state

A typical scenario is an app that distinguishes between:

- A signed-out user – sees the sign-in/registration screens (the so-called AuthStack).
- A signed-in user – sees the main part of the app (the so-called AppStack or AppTabs).
- Onboarding (optional) – a new user who has signed in but has to e.g. go through a tutorial or complete the profile before getting full access.

So we can distinguish several states: `signedOut`, `signedIn`, possibly `signedInButNotOnboarded`. Depending on it, we show different navigation.

The implementation can look like this (pseudocode using the `useContext` hook to keep the sign-in state):

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

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

 if (isLoading) {
  // e.g. a splash/loading screen while the token in storage is being checked
  return <SplashScreen />;
 }
 return (
```

```
  <NavigationContainer>
    {user == null ? (

<AuthStackNavigator /> // the user is signed out
    ) : user.onboardComplete === false ? (
<OnboardingStackNavigator /> // signed in, but has not completed onboarding
    ) : (
<MainAppNavigator /> // signed in and ready – the main app
    )}
  </NavigationContainer>
 );
}
```

In the pseudocode above:

- `isLoading` is a state saying that we are still e.g. checking in SecureStore whether a token is saved (we show a Splash screen meanwhile instead of flashing the sign-in screen after the app starts).
- user is the user object or null, kept somewhere in global state (e.g. a Context or Zustand, more on this in a moment).
- **We check in order:** if there is no user -> we render the navigator with the sign-in screens (AuthStackNavigator); if there is a user but onboarding is not complete -> we render OnboardingStackNavigator; otherwise (fully signed in) -> MainAppNavigator with the actual screens of the app.

Each of these navigators (AuthStack, OnboardingStack, MainApp) is e.g. a separately defined Stack or Tab with its own screens. Instead of three separate navigators we could also use one and add the screens conditionally – it is a matter of code organization. They are often separated for readability.

What does this approach give us? The user never sees a screen they should not have access to:

- When signed out – the navigation tree contains no route from the main app (neither Home nor Profile, etc.). Even if they knew the screen name, they cannot go there (`navigation.navigate('Home')` reports that there is no such route).
- When signed in – we remove the sign-in screens from the stack, so they cannot go back to Login. In the example with a dynamic configuration change, the previous screens are unmounted, which means that e.g. pressing the hardware back button does not go back to Login, because the login screen no longer exists in the navigator. This way we meet the requirement of cutting off the history.
- During onboarding – until they finish it, we do not let them into MainApp (because the condition does not allow rendering MainAppNavigator).

This approach is preferred over trying to redirect individual screens in `useEffect`. Sometimes people implement something like this in components: if user = null then `navigation.replace('Login')`. It works, but it is less clear – it is better to decide what is available at the level of the navigation configuration. Implementing the user (auth) state: We usually need global state that stores whether we have a token/credentials. We can use:

- The Context API (AuthContext) – as in the example above, a simple context holding the user object and possibly the login/logout functions.
- Zustand or MobX/Redux – any central state storage. Zustand is a lightweight store where we can keep `isLoggedIn` and e.g. the user profile. Its advantage is simplicity and no Redux boilerplate. 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 that switches the navigators.

- AsyncStorage/SecureStore – used for persistent storage, i.e. e.g. saving a JWT token or a first-launch flag. When the app starts, we do something like this:

```
useEffect(() => {
SecureStore.getItemAsync('token').then(storedToken => {
        if(storedToken) {

         // we have a token, we can try to refresh the user profile from the API, etc.
         setUserToken(storedToken);
        }
        setLoading(false);
});
}, []);
```

When we have loaded the state from storage, we set `isLoading` to false and the conditional rendering shows the right screens. It is important that this loading is done before we show any screens (hence the Splash screen meanwhile).

Guards and going back: With a dynamic navigator change (Auth -> App) it is worth also securing the scenario where the user may have had some stack of screens during sign-in (e.g. a registration screen, etc.). When they sign in and we render MainApp instead of Auth, the old screens disappear. If, however, the user quickly presses “back” (e.g. right after signing in on Android), it may close the app, because the new navigator has no history. This is generally fine, but if we wanted e.g. to block closing, we would have to handle it manually (e.g. BackHandler and ignoring it in the first seconds after signing in). In most cases, however, pressing back immediately after signing in is unlikely, and even if it happens – leaving the app on the Home screen is in line with expectations (the user thinks they go back to sign-in, but the app closes because sign-in no longer exists – next time the app opens already signed in). It is a small UX detail to consider in a real project. Other uses of protected routes: Not only auth. We can show some screens conditionally e.g. if the user has permissions (role-based access). Or e.g. a PremiumContent screen only when user.`isPremium`. Then, analogously, the condition decides whether the screen is added to the navigator. If the condition can change while the app is running, adding/removing screens dynamically is fine. React Navigation 7 reacts quite elegantly to changing conditions (thanks to the if option in the static API) and e.g. removes unavailable screens from the navigation without errors.

Summary: Protecting routes in RN comes down to separating the public part of the app from the private one. The best practice is to prepare separate navigators for different states and switch between them depending on the state.

The sign-in flow (Auth Flow) in React Navigation

The sign-in flow is a special case of managing navigation in an app. It consists of the sign-in/registration screens, possibly a welcome screen (onboarding), and the main part of the app for the signed-in user. A well-designed flow should meet the following requirements:

- After the app starts we check the authentication state (e.g. whether there is a valid session token). While we check – we show a start screen (Splash).
- If the user is not signed in, we show the sign-in/registration screens (AuthStack).
- If the user is signed in, we take them straight to the main app (AppStack / AppTabs).
- After a successful sign-in/registration we take the user to the main app and remove the sign-in screens from the history so that they cannot go back.

- When the user signs out while the app is running, we clear the session data and take them back to the sign-in screens (preferably clearing the history of the main navigation).

In addition, if we have onboarding for new users (e.g. a short tutorial or a consent screen), we have to fit it into the scheme above. This is often done with a flag in the user profile or in storage (e.g. `hasSeenTutorial`). An example approach:

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

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

- **AuthStack – contains the screens:** Sign in, Register, Forgot password, a Welcome screen (e.g. with the logo), etc. None of these screens require authentication. It is often a `createNativeStackNavigator` with the header turned off or a custom header (e.g. the app logo at the top).
- AppStack (or AppTabs if we use tabs) – contains the screens available only after signing in, e.g. Home, Dashboard, Profile, Settings, etc. It can be a stack or tabs depending on the app design.

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

When implementing the sign-in flow in React Navigation, it is worth basing it on the examples from the documentation. The React Navigation authors suggest using a Context to keep the auth status and show an example that uses Expo SecureStore to keep the token. The most important thing is to separate the navigation trees: separate ones for “signed in” and “signed out”, and switching between them when the auth state changes.

Onboarding – the first launch

Onboarding can take different forms:

- A welcome screen with a "Get started" button (e.g. presenting the main value of the app).
- A few swipeable screens (a carousel) with instructions, questions about user preferences, etc.
- A screen for choosing the language/theme, etc.

From the navigation point of view, onboarding is also a mini-stack of screens that we show only once to a new user. It is most often implemented with:

- A persistent flag, e.g. saving `hasOnboarded`=true in AsyncStorage after completing it.
- **Conditionally adding a screen to the AuthStack:** e.g. the first screen of the AuthStack is Welcome and only then Login. If we find that the user has already seen the welcome screen (we check AsyncStorage), we take them straight to Login, skipping Welcome. This can be done at app start-up (e.g. in Splash, deciding which route to go to).

Or:

- A separate OnboardingStack, as mentioned earlier, which we render before the AppStack. When finished, this stack sets the flag and we switch to the AppStack.

Storing the sign-in information

We have partly covered this topic with route guards. In the context of the sign-in flow it comes down to:

- Storing the token/authorization – if we use an API, after signing in we probably get a token (a JWT or a session id). It has to be stored securely. It is recommended to use SecureStore (Expo) or the Keychain/Keystore on the devices for this. SecureStore (Expo) saves the data encrypted in the secure storage of the device. AsyncStorage does not encrypt, so a token there is in plain text – fine for less sensitive things, but keep tokens secure. As a last resort, when testing, AsyncStorage can be used too.
- Keeping the sign-in state in the app – so that the whole app knows the user is signed in and e.g. shows their data in different places. The Context API or a global store (Zustand/Redux) is great for this. A Context is enough: we create an AuthContext with the value { user, `login()`, `logout()` }. After a successful sign-in (e.g. when the API returns 200 OK) we call `login(userData)` from the context object. This changes user and re-renders the dependent components (including our conditional navigator). As a result we move to the AppStack.
- Zustand – has the advantage that the state stays even when the context components are unmounted (but keeping it in `useState` in a component above NavigationContainer gives the same). For beginners a Context is easier.
- **Signing out – should clear all the data:** remove the token from SecureStore, reset user in the state, possibly reset the navigation. If we render navigators conditionally, resetting user automatically re-renders the NavigationContainer with the AuthStack, removing all the app screens from view. It is worth making sure, however, that we do not leave e.g. a modal open, etc. Usually `logout()` in the context is enough.

An example of using a Context (pseudocode):

```
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 app in `<AuthProvider>` (e.g. in App.tsx). In components (e.g. sign-in screens) we call const { login } = `useContext(AuthContext)` to sign in, and on the profile screen const { logout } = `useContext(AuthContext)` to sign out.

A good practice: When signing in and moving to the main app, we can reset the stack (if we used a plain navigate) or, as we have shown, remove the sign-in screens with conditional rendering. Many ready-made templates use `navigation.reset({ routes: [{ name: 'MainApp' }] })` after sign-in, where MainApp is e.g. a tab navigator.

To sum up: separating the AuthStack and the AppStack is the basis of an orderly sign-in flow. Keep the sign-in state global so that you can easily react to its changes in the navigation tree. Remember to store any tokens securely and to “clean up” after signing out (clear the state and the navigation).

Demo: a mini app with navigation (AuthStack + AppTabs)

Now let us move to a practical example that combines all the concepts covered. We will build a simplified React Native app that has two main navigation “modes”:

- AuthStack – a stack with a sign-in screen (LoginScreen).
- **AppTabs – a tab navigator with two screens:** HomeScreen and ProfileScreen.

A signed-out user sees the sign-in screen. After "signing in" (simulated in the app), they are taken to the app tabs: Home and Profile. Home shows a greeting and lets you go to the Profile screen (e.g. with a button) – passing the user ID as a parameter along the way. Profile shows the identifier of the signed-in user and offers signing out (going back to the sign-in screen).

In the code we will use a React Context to keep the information about the signed-in user (`userId`), which allows rendering the right navigator conditionally. We will also show the use of the `useNavigation` and `useRoute` hooks inside the screens.

Here is the code of the demo app:

```
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. Parameter types for the navigators (TypeScript)
type AuthStackParamList = {
 Login: undefined;
};
type AppTabsParamList = {
 Home: undefined;
 Profile: { userId: string };
};

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

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

// 4. The sign-in screen
function LoginScreen() {
 const auth = useContext(AuthContext);
 return (

  <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
    <Text> Sign-in screen</Text>
    <Button title="Sign me in as user #123"

          onPress={() => auth?.login('123')} />
  </View>
 );
}
```

```
// 5. The Home screen (the 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>
    {/* An example button that navigates to Profile with a parameter */}
    <Button title="Go to Profile (param: userId)"

          onPress={() => navigation.navigate('Profile', { userId })} />
  </View>
 );
}

// 6. The Profile screen (the profile tab)
function ProfileScreen({ route }: { route: any }) {
 const auth = useContext(AuthContext);
 // We take userId from the parameter or from the context:
 const routeUserId = route.params?.userId;
 const userId = routeUserId || auth?.userId;
 return (

  <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
    <Text> Profile screen of user #{userId}</Text>
    <Button title="Sign out" onPress={() => auth?.logout()} />
  </View>
 );
}

// 7. The main navigation component based on the 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 ? (
// No signed-in user -> we show the AuthStack
<AuthStack.Navigator>
        <AuthStack.Screen

         name="Login"
         component={LoginScreen}
         options={{ headerShown: false }}
        />
</AuthStack.Navigator>
) : (
// Signed in -> we show the app 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>
 );
}
```

(The code is written for readability for beginners – in practice the navigation typing would be refined, but here we focus on the idea.)

Explanations of the code above:

- First we define the navigator types and create the 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 provided a `userId` parameter (of type string).
- We create the AuthContext, which will keep the `userId` state (or null) and the login and logout functions. The whole authorization logic is simplified here: we assume that tapping the "Sign in" button always succeeds and signs us in as user #123.
- **LoginScreen:** Takes login from the context (auth?.login) and, on press, signs in the user with ID "123". In a real app we would instead call e.g. a sign-in API and, on success, save the token and the user id. Here we immediately call auth?.`login('123')`, which sets `userId` in the context state.
- **HomeScreen:** Shows a greeting with the user number (we take it from the context). It also has a "Go to Profile (param: `userId`)" button. On press we call `navigation.navigate('Profile', { userId })`. This demonstrates passing a parameter to the Profile tab – although in this case Profile could just as well use the context, we pass the param for illustration. Because Home and Profile are in the same AppTabs navigator, the navigate call switches the tab to Profile, passing it the parameter.
- **ProfileScreen:** Receives the parameters through { route } (it gets them as a prop because it is a navigation screen). It takes the `userId` value from route.params. For safety, if the param was not passed (e.g. the user switched to the Profile tab without using the button with the parameter), route.params is undefined – then we take `userId` from the context. In our flow, when the user first taps "Go to Profile" on Home – the param is present. If, however, while on Home they use the bottom navigation (tap the Profile icon in the tab bar), the tab switches without a parameter (React Navigation does not pass a param on a plain tab tap, because it is not navigation.navigate but an internal switch). Thanks to our logic, ProfileScreen still determines `userId` from the context. ProfileScreen displays the identifier and has a "Sign out" button. On press we call auth?.`logout()`, which sets `userId` = null in our context. This re-renders the whole navigation tree – because in `App()` the condition `userId` == null becomes true again, the AuthStack is shown instead of AppTabs, i.e. we return to the sign-in screen. It is immediate and effective – the tabs disappear and the sign-in stack appears.
- In the main `<App>` component we place the NavigationContainer and choose conditionally between `<AuthStack.Navigator>` and `<AppTabs.Navigator>`. Note: AuthStack.Navigator and AppTabs.Navigator are two completely separate navigators – they have no common screens. The switch happens purely through React (the ternary operator), and React Navigation copes without problems with unmounting one navigator and replacing it with another.

- We wrap NavigationContainer in AuthContext.Provider so that the screens have access to the auth value.

Testing the scenario:

- **After start-up:** `userId` is null, so the AuthStack with LoginScreen is rendered. LoginScreen shows a button. When we press it:
- `auth.login('123')` sets `userId` = '123'. Now `userId` is not null, so App re-renders: instead of the AuthStack, AppTabs appear in the NavigationContainer. The user sees the Home screen (by default the first tab, Home).
- HomeScreen shows "Welcome, user #123". The user has two ways: either tap the "Go to Profile" button (which calls navigation.navigate with the param), or simply choose the Profile tab at the bottom.
    - **If they tapped the button:** they are switched to the Profile screen, which receives route.params { `userId`: '123' }. It shows "Profile of user #123".
    - If instead the user tapped the Profile tab in the tab bar: they are switched to ProfileScreen, but without a parameter. Our code in ProfileScreen sees that route.params is undefined and takes `userId` from the context, also getting '123'. The end result for the user is identical – they see profile #123.
- The Profile screen has a "Sign out" button. On tap: `auth.logout()` sets `userId` = null. AppTabs are unmounted and the AuthStack is mounted again (the NavigationContainer removes the tabs and inserts the sign-in screen). The user sees the sign-in screen again. If they pressed back on Android at this moment – the app would close (because we are on the only screen of the main stack). We can treat this as the expected behavior (they signed out, so leaving the app is logical, or they can sign in again).

Notes on debugging and extending:

- If we did not use a context, an alternative would be e.g. keeping `userId` in state higher up and passing it to the screens as props (via `initialParams` or custom props in the navigator). A context is, however, more convenient.
- In a real app we would probably want to call navigation.reset instead of a plain navigate after signing in – but here we did it “architecturally” by removing the AuthStack completely.
- During development you can easily check that the sign-in screen does not stay in memory after signing in – e.g. by checking the component tree in React DevTools or by logging unmount in a `useEffect` in LoginScreen. It should be unmounted during the switch.
- Our `userId` param is a string – in a real API it could be a JWT token or a user object. Then we would rather not pass it through navigation params (because it is sensitive data), but keep it in the context/state. A navigation parameter is more useful for data about a specific sub-page, e.g. `postId` for a PostDetails screen. For a global `userId` it is better to use a context, as we have shown, which we did anyway.
- If we had more screens in the AuthStack (e.g. Register, ForgotPassword), we could add them to AuthStack.Navigator. Then `navigation.navigate('Register')` from LoginScreen would work normally. After signing in, regardless, the whole AuthStack navigation is replaced.
- Similarly in AppTabs – we could easily add e.g. a third Settings tab without affecting the auth logic.

More React Navigation 7 features in practice

The mechanisms below are used in the practical assignments (the PawCare app). It is worth knowing them all – they let you reproduce typical patterns from system apps without extra libraries.

Ways of presenting screens (Native Stack)

Option	Effect	Example use
`presentation: "card"` (default)	the screen slides in from the side	vet details
`presentation: "modal"`	the screen slides up from the bottom; iOS: a card with a swipe-to-close gesture	booking a visit, a form
`presentation: "fullScreenModal"`	a full-screen modal without a close gesture	a QR code scanner (camera)
`presentation: "transparentModal"`	transparent – the previous screen is visible	a video player, a confirmation dialog
`presentation: "formSheet"` + `sheetAllowedDetents: [0.5, 1]`	a bottom sheet with detent heights and a grabber (`sheetGrabberVisible`)	list filters

Screen and header options

```
<Stack.Navigator screenOptions={{ headerBackButtonDisplayMode: 'minimal' }}>
  <Stack.Screen
    name="Vets"
    component={VetsScreen}
    options={{ headerLargeTitle: true, title: 'Vets' }}
  />
  <Stack.Screen
    name="VetDetails"
    component={VetDetailsScreen}
    getId={({ params }) => params.vetId} // another vet = a new screen
    options={({ route }) => ({
      title: '',
      headerTransparent: true,
      headerBlurEffect: 'systemChromeMaterial', // iOS
      headerRight: () => <VetHeaderActions vetId={route.params.vetId} />,
    })}
  />
  <Stack.Screen
    name="Filters"
    component={FiltersScreen}
    options={{ presentation: 'formSheet', sheetAllowedDetents: [0.5, 1], sheetGrabberVisible:
true }}
  />
  <Stack.Group screenOptions={{ presentation: 'modal' }}>
    <Stack.Screen name="BookVisit" component={BookVisitScreen} options={{ gestureEnabled: false
}} />
  </Stack.Group>
</Stack.Navigator>
```

We set up a search bar in the header from the screen: `navigation.setOptions({ headerSearchBarOptions: { placeholder: 'Search', onChangeText: (e) => setQuery(e.nativeEvent.text) } })` (in `useLayoutEffect`).

Tabs: events, badges, history

Bottom Tabs

```
<Tab.Navigator backBehavior="history" screenOptions={{ animation: 'shift', tabBarHideOnKeyboard:
true }}>
  <Tab.Screen name="VetsTab" component={VetsStack}
    options={{ headerShown: false, popToTopOnBlur: true }} />
  <Tab.Screen name="AppointmentsTab" component={AppointmentsScreen}
    options={{ tabBarBadge: count > 0 ? count : undefined }}
    listeners={{
```

```
      tabPress: (e) => {
        if (isGuest) {
          e.preventDefault(); // the tab does not open
          Alert.alert('Appointments', 'Sign in to book visits.');
        }
      },
    }} />
</Tab.Navigator>
```

Material Top Tabs (`@react-navigation/material-top-tabs` + `react-native-pager-view`) are tabs at the top of the screen switched with a swipe gesture – e.g. the days of a vet's schedule. One screen component can serve several tabs thanks to `initialParams`.

More React Navigation 7 features worth knowing

- **`navigation.preload(name, params)`** – renders a screen in the background (e.g. on `onPressIn` of a card), so the transition is instant. Note: a screen loaded in the background should not set its header with `setOptions` – header buttons go in the navigator's `options`.
- **`screenLayout`** – a common "wrapper" for every screen of a navigator, e.g. `Suspense` + `ErrorBoundary` (lecture 6).
- **`useFocusEffect`** and **`useIsFocused`** – starting/stopping effects only on the visible screen (the camera, sensors, refreshing data).
- **`useScrollToTop(ref)`** – tapping the active tab again scrolls the list to the top.
- **The Static API** (`createNativeStackNavigator({ screens: { … } })`) with the `if` option for protected screens and automatic typing (`StaticParamList`) – an alternative to the dynamic API used in this lecture.
- **Deep links from notifications:** the `getInitialURL` and `subscribe` options in the `linking` configuration let you treat a notification tap like opening a link (lecture 7).

> **Expo Router and React Navigation**
>
> New Expo projects (`npx create-expo-app` with the default template) use **Expo Router** – file-based navigation (`app/(tabs)/index.tsx` → a screen). Expo Router is built on React Navigation: the same navigators, options and concepts (stack, tabs, modals; deep links are created automatically from the file names). In class we configure React Navigation directly to see all the mechanisms that Expo Router hides.
>
> In React 19 we provide a context directly: `<AuthContext value={…}>` (without `.Provider`).

Summary and good practices

In this lecture we went through all the key aspects of navigation in React Native with React Navigation:

- We learned the different types of navigators and their uses (Stack for sequences of screens, Tabs for parallel sections, Drawer for a hidden menu).
- We learned how to pass parameters between screens and receive them, and how to type them correctly in TypeScript.
- We used the `useNavigation` and `useRoute` navigation hooks to conveniently trigger navigation and read route data in function components.
- We discussed how to go back in the navigation, both one step (`goBack`) and in bulk (`popToTop`), and how to reset the stack of screens – which turned out to be important in the sign-in scenario.

- We dealt with deep links, configuring URL schemes and the Expo integration, so that our app can react to external links and open the right screens.
- We introduced the concept of protected routes – limiting access to parts of the app by rendering navigators conditionally depending on the state (e.g. signing in). Thanks to this we implemented the authentication flow, separating the sign-in screens from the main app.
- We finished with a practical demo that showed step by step the implementation of a mini app with an authorization context and two navigators. The demo shows how to tie together React Navigation, a Context and RN components in real code to get a user-friendly flow (sign-in -> app -> sign-out -> back to sign-in).

References

All sources checked in October 2026.

1. https://reactnavigation.org/docs/getting-started/ – The official React Navigation documentation (home page).
2. https://reactnavigation.org/docs/typescript/ – The official guide to integrating React Navigation with TypeScript.
3. https://reactnavigation.org/docs/auth-flow/ – The key documentation describing the recommended authentication flow pattern.
4. https://reactnavigation.org/docs/deep-linking/ – The official guide to configuring deep links.
5. https://reactnavigation.org/docs/navigating/ – Documentation of the basic operations (navigate, push, `goBack`).
6. https://reactnavigation.org/docs/params/ – Documentation on passing and receiving parameters (route.params).
7. https://reactnavigation.org/docs/hooks/ – Documentation of the `useNavigation` and `useRoute` hooks.
8. https://reactnavigation.org/docs/native-stack-navigator/ – Documentation of the Native Stack Navigator (recommended for performance).
9. https://reactnavigation.org/docs/bottom-tab-navigator/ – Documentation of the Bottom Tab Navigator.
10. https://docs.expo.dev/routing/linking/ – The Expo guide to configuring linking (including `expo-linking`).
11. https://reactnavigation.org/docs/navigation-actions – navigation actions (`navigate`, `popTo`, `replace`, `reset`).
12. https://reactnavigation.org/docs/native-stack-navigator – the `presentation` and `sheetAllowedDetents` options, headers.
13. https://reactnavigation.org/docs/preventing-going-back – `usePreventRemove`.
14. https://reactnavigation.org/docs/navigation-container#ref – `createNavigationContainerRef`.
15. https://docs.expo.dev/linking/android-app-links/ and https://docs.expo.dev/linking/ios-universal-links/ – App Links and Universal Links in Expo.
16. https://docs.expo.dev/router/introduction/ – Expo Router.