Version: 6.x

Authentication flows

Most apps require that a user authenticates in some way to have access to data associated with a user or other private content. Typically the flow will look like this:

- The user opens the app.
- The app loads some authentication state from encrypted persistent storage (for example, `SecureStore`).
- When the state has loaded, the user is presented with either authentication screens or the main app, depending on whether valid authentication state was loaded.
- When the user signs out, we clear the authentication state and send them back to authentication screens.

> Note: We say "authentication screens" because usually there is more than one. You may have a main screen with a username and password field, another for "forgot password", and another set for sign up.

What we need

This is the behavior that we want from the authentication flow: when users sign in, we want to throw away the state of the authentication flow and unmount all of the screens related to authentication, and when we press the hardware back button, we expect to not be able to go back to the authentication flow.

How it will work

We can define different screens based on some condition. For example, if the user is signed in, we can define `Home`, `Profile`, `Settings` etc. If the user is not signed in, we can define `SignIn` and `SignUp` screens.

For example:

```
isSignedIn ? (
  <>
    <Stack.Screen name="Home" component={HomeScreen} />
    <Stack.Screen name="Profile" component={ProfileScreen} />
    <Stack.Screen name="Settings" component={SettingsScreen} />
  </>
) : (
  <>
    <Stack.Screen name="SignIn" component={SignInScreen} />
    <Stack.Screen name="SignUp" component={SignUpScreen} />
  </>
);
```

Try this example on Snack

When we define screens like this, when `isSignedIn` is `true`, React Navigation will only see the `Home`, `Profile` and `Settings` screens, and when it's `false`, React Navigation will see the `SignIn` and `SignUp` screens. This makes it impossible to navigate to the `Home`, `Profile` and `Settings` screens when the user is not signed in, and to `SignIn` and `SignUp` screens when the user is signed in.

This pattern has been in use by other routing libraries such as React Router for a long time, and is commonly known as "Protected routes". Here, our screens which need the user to be signed in are "protected" and cannot be navigated to by other means if the user is not signed in.

The magic happens when the value of the `isSignedIn` variable changes. Let's say, initially `isSignedIn` is `false`. This means, either `SignIn` or `SignUp` screens are shown. After the user signs in, the value of `isSignedIn` will change to `true`. React Navigation will see that the `SignIn` and `SignUp` screens are no longer defined and so it will remove them. Then it'll show the `Home` screen automatically because that's the first screen defined when `isSignedIn` is `true`.

The example shows stack navigator, but you can use the same approach with any navigator.

By conditionally defining different screens based on a variable, we can implement auth flow in a simple way that doesn't require additional logic to make sure that the correct screen is shown.

Don't manually navigate when conditionally rendering screens

It's important to note that when using such a setup, you **don't manually navigate** to the `Home` screen by calling `navigation.navigate('Home')` or any other method. **React Navigation will automatically navigate to the correct screen** when `isSignedIn` changes - `Home` screen when `isSignedIn` becomes `true`, and to `SignIn` screen when `isSignedIn` becomes `false`. You'll get an error if you attempt to navigate manually.

Define our screens

In our navigator, we can conditionally define appropriate screens. For our case, let's say we have 3 screens:

- `SplashScreen` - This will show a splash or loading screen when we're restoring the token.
- `SignInScreen` - This is the screen we show if the user isn't signed in already (we couldn't find a token).
- `HomeScreen` - This is the screen we show if the user is already signed in.

So our navigator will look like:

```
if (state.isLoading) {
  // We haven't finished checking for the token yet
  return <SplashScreen />;
}

return (
  <Stack.Navigator>
    {state.userToken == null ? (
      // No token found, user isn't signed in
      <Stack.Screen
        name="SignIn"
        component={SignInScreen}
        options={{
          title: 'Sign in',
          // When logging out, a pop animation feels intuitive
          // You can remove this if you want the default 'push' animation
          animationTypeForReplace: state.isSignout ? 'pop' : 'push',
        }}
      />
    ) : (
      // User is signed in
      <Stack.Screen name="Home" component={HomeScreen} />
    )}
  </Stack.Navigator>
);
```

Try this example on Snack

In the above snippet, `isLoading` means that we're still checking if we have a token. This can usually be done by checking if we have a token in `SecureStore` and validating the token. After we get the token and if it's valid, we need to set the `userToken`. We also have another state called `isSignout` to have a different animation on sign out.

The main thing to notice is that we're conditionally defining screens based on these state variables:

- `SignIn` screen is only defined if `userToken` is `null` (user is not signed in)
- `Home` screen is only defined if `userToken` is non-null (user is signed in)

Here, we're conditionally defining one screen for each case. But you could also define multiple screens. For example, you probably want to define password reset, signup, etc screens as well when the user isn't signed in. Similarly, for the screens accessible after signing in, you probably have more than one screen. We can use `React.Fragment` to define multiple screens:

```
state.userToken == null ? (
  <>
    <Stack.Screen name="SignIn" component={SignInScreen} />
    <Stack.Screen name="SignUp" component={SignUpScreen} />
    <Stack.Screen name="ResetPassword" component={ResetPassword} />
  </>
) : (
  <>
    <Stack.Screen name="Home" component={HomeScreen} />
    <Stack.Screen name="Profile" component={ProfileScreen} />
  </>
);
```

> If you have both your login-related screens and rest of the screens in two different Stack navigators, we recommend to use a single Stack navigator and place the conditional inside instead of using 2 different navigators. This makes it possible to have a proper transition animation during login/logout.

Implement the logic for restoring the token

> Note: The following is just an example of how you might implement the logic for authentication in your app. You don't need to follow it as is.

From the previous snippet, we can see that we need 3 state variables:

- `isLoading` - We set this to `true` when we're trying to check if we already have a token saved in `SecureStore`
- `isSignout` - We set this to `true` when user is signing out, otherwise set it to `false`
- `userToken` - The token for the user. If it's non-null, we assume the user is logged in, otherwise not.

So we need to:

- Add some logic for restoring token, signing in and signing out
- Expose methods for signing in and signing out to other components

We'll use `React.useReducer` and `React.useContext` in this guide. But if you're using a state management library such as Redux or Mobx, you can use them for this functionality instead. In fact, in bigger apps, a global state management library is more suitable for storing authentication tokens. You can adapt the same approach to your state management library.

First we'll need to create a context for auth where we can expose necessary methods:

```
import * as React from 'react';

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

So our component will look like this:

```
import * as React from 'react';
import * as SecureStore from 'expo-secure-store';

export default function App({ navigation }) {
  const [state, dispatch] = React.useReducer(
    (prevState, action) => {
      switch (action.type) {
        case 'RESTORE_TOKEN':
          return {
            ...prevState,
            userToken: action.token,
            isLoading: false,
          };
        case 'SIGN_IN':
          return {
            ...prevState,
            isSignout: false,
            userToken: action.token,
          };
        case 'SIGN_OUT':
          return {
            ...prevState,
            isSignout: true,
            userToken: null,
          };
      }
    },
    {
      isLoading: true,
      isSignout: false,
      userToken: null,
    }
  );

  React.useEffect(() => {
    // Fetch the token from storage then navigate to our appropriate place
    const bootstrapAsync = async () => {
      let userToken;

      try {
        userToken = await SecureStore.getItemAsync('userToken');
      } catch (e) {
        // Restoring token failed
      }

      // After restoring token, we may need to validate it in production apps

      // This will switch to the App screen or Auth screen and this loading
      // screen will be unmounted and thrown away.
      dispatch({ type: 'RESTORE_TOKEN', token: userToken });
    };
    bootstrapAsync();
  }, []);

  const authContext = React.useMemo(
    () => ({
      signIn: async (data) => {
        // In a production app, we need to send some data (usually username,
password) to server and get a token
        // We will also need to handle errors if sign in failed
        // After getting token, we need to persist the token using `SecureStore`
        // In the example, we'll use a dummy token

        dispatch({ type: 'SIGN_IN', token: 'dummy-auth-token' });
      },
      signOut: () => dispatch({ type: 'SIGN_OUT' }),
      signUp: async (data) => {
        // In a production app, we need to send user data to server and get a
token
        // We will also need to handle errors if sign up failed
        // After getting token, we need to persist the token using `SecureStore`
        // In the example, we'll use a dummy token

        dispatch({ type: 'SIGN_IN', token: 'dummy-auth-token' });
      },
    }),
    []
  );

  return (
    <AuthContext.Provider value={authContext}>
      <Stack.Navigator>
        {state.userToken == null ? (
          <Stack.Screen name="SignIn" component={SignInScreen} />
        ) : (
          <Stack.Screen name="Home" component={HomeScreen} />
        )}
      </Stack.Navigator>
    </AuthContext.Provider>
  );
}
```

Try this example on Snack

Fill in other components

We won't talk about how to implement the text inputs and buttons for the authentication screen, that is outside of the scope of navigation. We'll just fill in some placeholder content.

```
function SignInScreen() {
  const [username, setUsername] = React.useState('');
  const [password, setPassword] = React.useState('');

  const { signIn } = React.useContext(AuthContext);

  return (
    <View>
      <TextInput
        placeholder="Username"
        value={username}
        onChangeText={setUsername}
      />
      <TextInput
        placeholder="Password"
        value={password}
        onChangeText={setPassword}
        secureTextEntry
      />
      <Button title="Sign in" onPress={() => signIn({ username, password })} />
    </View>
  );
}
```

Removing shared screens when auth state changes

Consider the following example:

```
isSignedIn ? (
  <>
    <Stack.Screen name="Home" component={HomeScreen} />
    <Stack.Screen name="Profile" component={ProfileScreen} />
    <Stack.Screen name="Help" component={HelpScreen} />
  </>
) : (
  <>
    <Stack.Screen name="SignIn" component={SignInScreen} />
    <Stack.Screen name="SignUp" component={SignUpScreen} />
    <Stack.Screen name="Help" component={HelpScreen} />
  </>
);
```

Here we have specific screens such as `SignIn`, `Home` etc. which are only shown depending on the sign in state. But we also have the `Help` screen which can be shown in both cases. This also means that if the signin state changes when the user is in the `Help` screen, they'll stay on the `Help` screen.

This can be a problem, we probably want the user to be taken to the `SignIn` screen or `Home` screen instead of keeping them on the `Help` screen. To make this work, we can use the `navigationKey` prop. When the `navigationKey` changes, React Navigation will remove all the screen.

So our updated code will look like following:

```
<>
  {isSignedIn ? (
    <>
      <Stack.Screen name="Home" component={HomeScreen} />
      <Stack.Screen name="Profile" component={ProfileScreen} />
    </>
  ) : (
    <>
      <Stack.Screen name="SignIn" component={SignInScreen} />
      <Stack.Screen name="SignUp" component={SignUpScreen} />
    </>
  )}
  <Stack.Screen navigationKey={isSignedIn ? 'user' : 'guest'} name="Help"
component={HelpScreen} />
</>
```

If you have a bunch of shared screens, you can also use `navigationKey` with a `Group` to remove all of the screens in the group. For example:

```
<>
  {isSignedIn ? (
    <>
```

```
      <Stack.Screen name="Home" component={HomeScreen} />
      <Stack.Screen name="Profile" component={ProfileScreen} />
    </>
  ) : (
    <>
      <Stack.Screen name="SignIn" component={SignInScreen} />
      <Stack.Screen name="SignUp" component={SignUpScreen} />
    </>
  )}
  <Stack.Group navigationKey={isSignedIn ? 'user' : 'guest'}>
    <Stack.Screen name="Help" component={HelpScreen} />
    <Stack.Screen name="About" component={AboutScreen} />
  </Stack.Group>
</>
```

Edit this page