Published in Level Up Coding

Lucas Garcez Follow

Apr 5, 2021 · 3 min read · Listen

Save

React Native Authentication Flow, the Simplest and Most Efficient Way

Connect with API, persists data, and recover them in future sessions

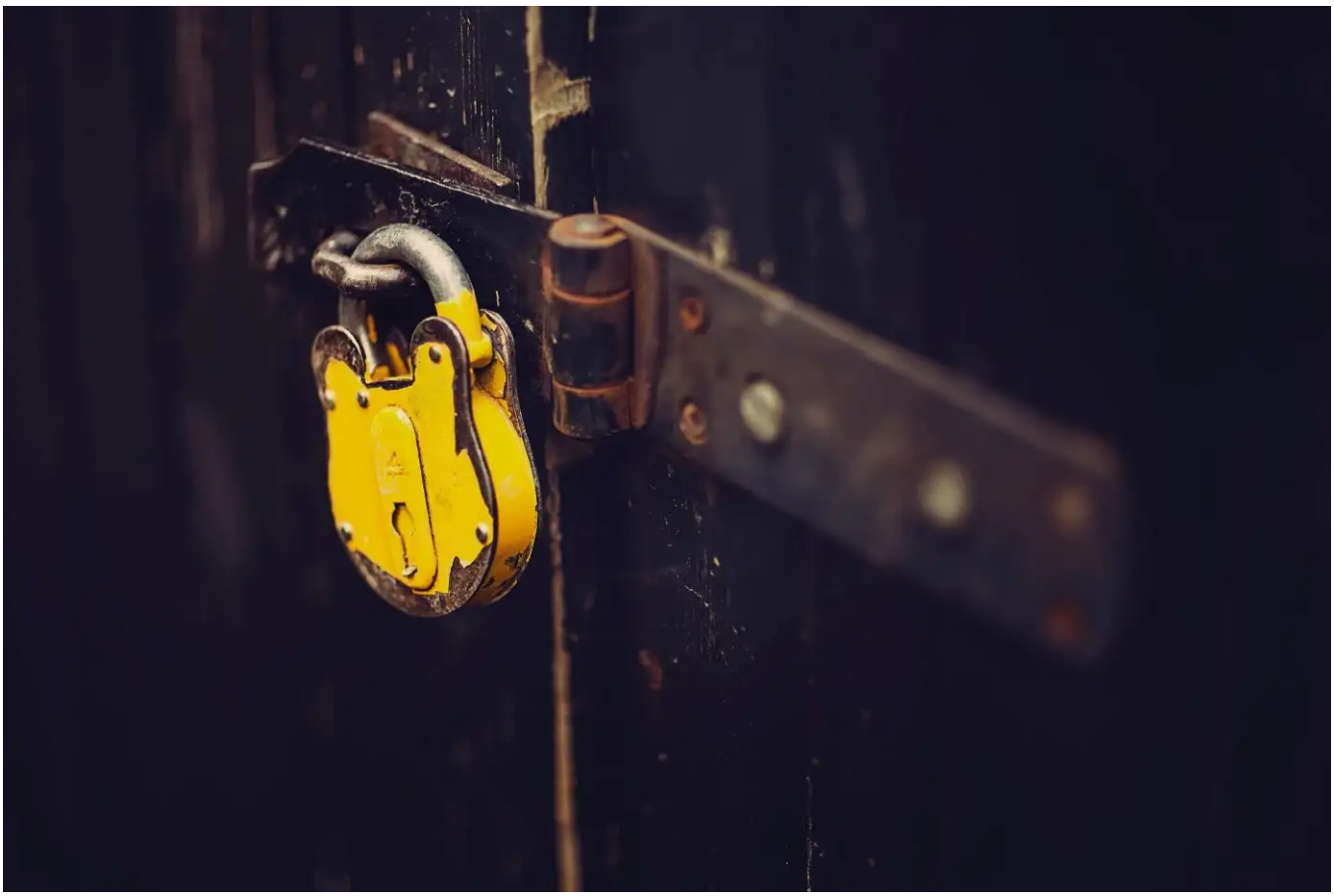

Photo by Chris Panas on Unsplash

Almost all apps need authentication flow because they contain contents that authorized users should only access.

This article will explain how created a React Native authentication flow that connects to APIs, persists data to be recovered in future sessions, and provides an efficient way for the whole application to subscribe to the auth state changes.

The focus here is to build the entire structure for the flow described above, not create the App from scratch. But you can access the complete code **repository here**.

Authenticated and unauthenticated users

In React Native, a common way to separate these users is to create different "groups" screens. One for unauthenticated users, which contains screens like sign in and sign up, and the other for the authenticated users, containing screens like home, settings, and all others related to the user content.

The react-navigation library will help create these groups, more specifically, the **createStackNavigator** function. **AppStack**, the first group, will contain the home screen, and the **AuthStack** will include the sign screen.

```
import {createStackNavigator} from '@react-navigation/stack';
const Stack = createStackNavigator();

const AppStack = () => {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Home Screen" component={HomeScreen} />
    </Stack.Navigator>
  );
};

const AuthStack = () => {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Sign In Screen" component={SignInScreen} />
    </Stack.Navigator>
  );
};
```

rnAuth-Stack.tsx hosted with ❤️ by GitHub view raw

Create a **Router** component responsible for choose which stack show depending on whether the user is authenticated or not. The stacks need to be inside the **NavigationContainer**, a component from react-navigation that manages the navigation tree and navigation state.

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

import {AppStack} from './AppStack';
import {AuthStack} from './AuthStack';


export const Router = () => {
//More explanations about "authData" below
  return (
    <NavigationContainer>
      {authData ? <AppStack /> : <AuthStack />}
    </NavigationContainer>
  );
};
```

rnAuth-Router-init.tsx hosted with ❤️ by GitHub view raw

The **authData** represents the user authentication state, in other words, if there is an authentication user or not. The **authData** cannot be a simple property from the **Router** because others components can update it from different places in the App, so it's necessary another solution. An excellent alternative to resolve that is to use the React Context.

> *Context provides a way to pass data through the component tree without having to pass props down manually at every level, and any component in the tree can "listener" Context changes.*

Use React Context to provide the authData

The Context needs to provide the **authData** itself, a function to sign in, a function to sign out, and a loading state. The sign-in function will connect with the API and update the **authData** based on the response. The sign-out function will clean the data, and in some cases, connect with API to invalidate a token or something like that. The loading state is necessary because the authentication process is an async task, connected with API as the local persistent. So the data shape can be like that:

```tsx
type AuthContextData = {
  authData?: AuthData;
  loading: boolean;
  signIn(): Promise<void>;
  signOut(): void;
};

type AuthData = {
```

```
  token: string;
  email: string;
  name: string;
};
```

rnAuth-AuthContextData.ts hosted with ❤️ by GitHub view raw

With these **types** to orientate, we can create the Context, called by **AuthContext,** and your relative provider, called by **AuthProvide.**

The code below is a slightly longer but we can read from top to bottom. Every part has a comment that explains what is and what the intention is.

```
import React, {createContext, useState, useContext} from 'react';
import {AuthData, authService} from '../services/authService';

const AuthContext = createContext<AuthContextData>({} as AuthContextData);

const AuthProvider: React.FC = ({children}) => {
  const [authData, setAuthData] = useState<AuthData>();

  //The loading part will be explained in the persist step session
  const [loading, setLoading] = useState(true);

  const signIn = async () => {
    //call the service passing credential (email and password).
    //In a real App this data will be provided by the user from some InputText componer
    const _authData = await authService.signIn(
      'lucasgarcez@email.com',
      '123456',
    );

    //Set the data in the context, so the App can be notified
    //and send the user to the AuthStack
    setAuthData(_authData);
  };

  const signOut = async () => {
    //Remove data from context, so the App can be notified
    //and send the user to the AuthStack
    setAuthData(undefined);
  };

  return (
    //This component will be used to encapsulate the whole App,
    //so all components will have access to the Context
    <AuthContext.Provider value={{authData, loading, signIn, signOut}}>
```

```
      <AuthContext.Provider value={{authData, loading, signIn, signOut}}>
        {children}
      </AuthContext.Provider>
    );
  };
```

rnAuth-Auth-initial.tsx hosted with ❤️ by GitHub view raw

To any component listener authentication status changes and call the sing-in or sing-out functions, the **App** root component should encapsulate the **Router** with the **AuthProvider**.

```
import React from 'react';
import {Router} from './src/routes/Router';
import {AuthProvider} from './src/contexts/Auth';

const App = () => {
  return (
    <AuthProvider>
      <Router />
    </AuthProvider>
  );
};

export default App;
```

rnAuth-App.tsx hosted with ❤️ by GitHub view raw

To facilitate de access to the **AuthContext** we can create a simple React Hooks that abstracts the context connections logic.

```
function useAuth(): AuthContextData {
  const context = useContext(AuthContext);

  if (!context) {
    throw new Error('useAuth must be used within an AuthProvider');
  }

  return context;
}
```

rnAuth-useAuth.tsx hosted with ❤️ by GitHub view raw

The **Router** component will use the **useAuth** hooks to decide which correct stack to display and show a **Loading** component if data is not ready.

```tsx
import {useAuth} from '../contexts/Auth';
import {Loading} from '../components/Loading';

export const Router = () => {
  const {authData, loading} = useAuth();

  if (loading) {
    //You can see the component implementation at the repository
    return <Loading />;
  }
  return (
    <NavigationContainer>
      {authData?.token ? <AppStack /> : <AuthStack />}
    </NavigationContainer>
  );
};
```

rnAuth-Router.tsx hosted with ❤️ by GitHub view raw

The screens, also can use the hooks to sign-in and sign-out.

```tsx
import React from 'react';
import {Button, Text, View} from 'react-native';

import {styles} from './styles';
import {useAuth} from '../contexts/Auth';

//SIGN IN SCREEN
export const SignInScreen = () => {
  const auth = useAuth();

  return (
    <View style={styles.container}>
      <Text>Sign In Screen</Text>
      <Button title="Sign In" onPress={auth.signIn} />
    </View>
  );
};

//HOME SCREEN
export const HomeScreen = () => {
  const auth = useAuth();

  return (
    <View style={styles.container}>
      <Text>HOME SCREEN</Text>
      <Button title="Sign Out" onPress={auth.signOut} />
```

```
    </View>
  );
};
```

rnAuth-Screens-hooks.tsx hosted with ❤️ by GitHub view raw

Persist data

The last problem to be resolved is persistence. Up to now, when the App is closed and opened, all data is lost, because the Context only exists in the App's memory. The @react-native-async-storage/async-storage library will handle this and persist the data. As **AppProvider** concentrate the auth state changes, then this is the most suitable location to put the persistence logic.

- When the App starts: Check the storage; if there are data, update the **authData** state.
- When the user sign in: Saves the **authData** from API at the storage.
- When the user sign out: Remove the **authData** from the storage.

```
//...other imports
import AsyncStorage from '@react-native-community/async-storage';

```



```

  useEffect(() => {
    //Every time the App is opened, this provider is rendered
    //and call de loadStorageData function.
    loadStorageData();
  }, []);

  async function loadStorageData(): Promise<void> {
    try {
      //Try get the data from Async Storage
      const authDataSerialized = await AsyncStorage.getItem('@AuthData');
      if (authDataSerialized) {
        //If there are data, it's converted to an Object and the state is updated.
        const _authData: AuthData = JSON.parse(authDataSerialized);
        setAuthData(_authData);
      }
    } catch (error) {
```

```
    } finally {
      //loading finished
      setLoading(false);
    }
  }

  const signIn = async () => {
    //...call service and setAuthData

    //Persist the data in the Async Storage
    //to be recovered in the next user session.
    AsyncStorage.setItem('@AuthData', JSON.stringify(_authData));
  };

  const signOut = async () => {
    //... setAuthData

    //Remove the data from Async Storage
    //to NOT be recovered in next session.
    await AsyncStorage.removeItem('@AuthData');
  };

 //... return AuthContext.Provider
};
```


That's all, folks! Now you have a complete authentication flow, with well separated responsibilities, persistence, and API connection. Again, you can access the entire code from this project at the GitHub repository.

Thanks for getting here; if you have any questions, leave a comment, and I will be happy to help.