React Native Auth, Perf, Deployment and Testing

CS571 – Mobile Application Development

Maharishi University of Management

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

Authentication Flows

Most apps require that a user authenticate in some way to have access to data associated with a user or other private content.

- The user opens the app.
- The app loads some authentication state from persistent storage (*AsyncStorage*).
- 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.

https://reactnavigation.org/docs/auth-flow/

Authentication for ReactNative apps

- Users send their credentials to the server which are verified against a database. If everything checks out, a JWT is sent back to them.
- The JWT is persisted in the user’s device by holding it in async storage.
- The app keeps a live copy of the token at a global context level.
- The presence of a JWT saved in the app state is used as an indicator that a user is currently logged in.
- Access to protected client-side screens are limited to only authenticated users.
- When the user makes requests to some protected backend APIs, the request must include the JWT as an Authorization header using the Bearer.
- Middleware on the server, which is configured with the app's secret key checks the incoming JWT for validity and expiration, if valid, sends the response.

Define App State

```
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, userToken: action.token, isLoading: false};
   case 'SIGN_OUT': return {...prevState, userToken: null};
}
}, {
      isLoading: true,
      userToken: null,
});

// Same as

const [state, setState] = React.useState({isLoading: true, userToken: null});
```

Conditionally Define Screens

```
state.userToken ? (
  <>
    <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 name="ResetPassword" component={ResetPassword} />
</>
);
```

Restore the Token

```
React.useEffect(async() => {
     (async () => {
               // 2. try to read userToken from storage
               let userToken = await AsyncStorage.getItem('userToken');
               // 3. change state
               dispatch({ type: 'RESTORE_TOKEN', token: userToken });
               // OR setState(prev=>({…prev, token: userToken}));
     })();
}, []); // 1. in App.js on mount.
```

Provide Helpers

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

const authContext = React.useMemo(() => ({
      signIn: async data => {
        // We need to send some data (usually username, password) to server and get a token
        // After getting token, we need to persist the token using `AsyncStorage`
        dispatch({ type: 'SIGN_IN', token: 'dummy-auth-token' });
      },
      signOut: () => dispatch({ type: 'SIGN_OUT' }),
      signUp: async data => {
        // We need to send user data to server and get a token
        // After getting token, we need to persist the token using `AsyncStorage`
        dispatch({ type: 'SIGN_IN', token: 'dummy-auth-token' });
      },
}),[]);
```

Provide State Helpers with Context

```
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>
);
```

React Performance

Performance optimization usually comes at a complexity cost. In most cases, optimization is not worth the cost in complexity and maintainability.

Don't over-optimize until a bottleneck is found.

Measuring Performance

Be mindful of the environment setting of your application (dev, prod).

React Native Perf Monitor

- Shows you the refresh rate on both the UI and JS threads.
- Anything below 60 means frames are being dropped.

Chrome Performance Profiler

- Shows you a flame chart (graph chart for react components).
- Only available in development mode.

Common Inefficiencies

- Re-rendering too often.
- Unnecessarily changing props.
- Unnecessary logic in mount/update.

Re-rendering

Components will automatically re-render when they receive new props, but sometimes, a prop that isn't needed for the UI will change and cause an unnecessary re-render. If you use redux, only subscribe to the part of state that is necessary, context does not consider subscribing to a part of the state.

- Use keys in arrays/lists.
- Implement **`shouldComponentUpdate()`** and **`React.PureComponent,`** and **`useMemo(). PureComponent`** has a predefined **`shouldComponentUpdate()`** that does a shallow diff of props.

Unnecessarily Changing Props

Unnecessarily changing a value that is passed to a child could cause a re-render of the entire subtree.

If you have any object (or array, function, etc.) literals in your **`render()`** method, a new object will be created at each render, better to use constants, methods, or properties on the class instance.

```
render(){
    const dataElements = this.state.data.map();
    <PureButton onPress={()=>{...}} style={{width: '100%'}}/>
}
```

Unnecessary Logic in Mount/Update

Adding **properties** to class instance vs. **methods** on the class.

Properties are created at each mount for every instance whereas methods are one time.

```
class ButtonScreen extends React.Component{
    inc(){'this'}
    vs.
    inc = ()=>{'this'}  === constructor(){this.inc}
}
```

Testing

- **Unit tests**: Test an individual unit of code (function/class/method).
- **Integration/Service tests**: Test the integration of multiple pieces of code working together, independent of the UI.
- **UI/End-to-end tests**: Test a feature thoroughly including the UI, network calls, etc.

Deploying

Deploy to the appropriate store by building the app and uploading it to the store:

1. Set the correct metadata in **`app.json`** *https://docs.expo.io/workflow/configuration/*
2. Build the app using expo

   `expo login`

   `expo build, expo build:ios, expo build:android`

Expo will build your app in the cloud and upload the build to AWS-S3 and provide you with a link to the final **`.apk`** and **`.ipa`** files.

Run expo **`build:status`** and paste the URL in a browser to download.

Deploying, cont.

Upload to the appropriate store

https://docs.expo.io/distribution/building-standalone-apps/
https://docs.expo.io/distribution/app-stores/

You may deploy a new JS React Native app by republishing from expo, re-build the app and re-submit to store to change app metadata.