The Implementation of QR Code Application

Andy Liu

July 24, 2022

1 Introduction

This document briefly introduces the implementation of QR Code application, which is developed by React JS and TypeScript. This document requires basic knowledge of React JS and TypeScript.

In the following sections, I firstly present the functionalities of QR Code application in Sec.2. The overview of the application, i.e. the included React components and their relationships, is illustrated in Sec.3. Sec.4 describes the details of the implementations, and Sec.5 would discuss some areas to be enhanced.

2 Main functionalities

The QR Code application consists of the following functionalities:

- Allow the user to log in using Google Sign-In.
- Verify the user credentials and display "Hello <firstname>", e.g. "Hello Andy!"
- Loads a page that shows a dropdown with fixed values (can be fetched from an API or can be pre-selected values) to generate a QR Code. The dropdown to consist of only 5 values for user to choose from.
- As soon as a user selects the value, press a button to generate a QR Code in orange colour.
- Display a thank you message after code generation.

3 Overview of the application

The QR Code application is composed of several React components, within which `SignInComponent` and `GenerateQRCodeComponent` are two most important ones. Fig.1 gives an overview of the components, which describes the parent-child component relationships as parent-child nodes in a tree. A child component is rendered in the parent component. For the purpose of simplicity, component names are shortened in the remainder of this document. E.g. `SignInComponent` is named as `SignIn` for short.

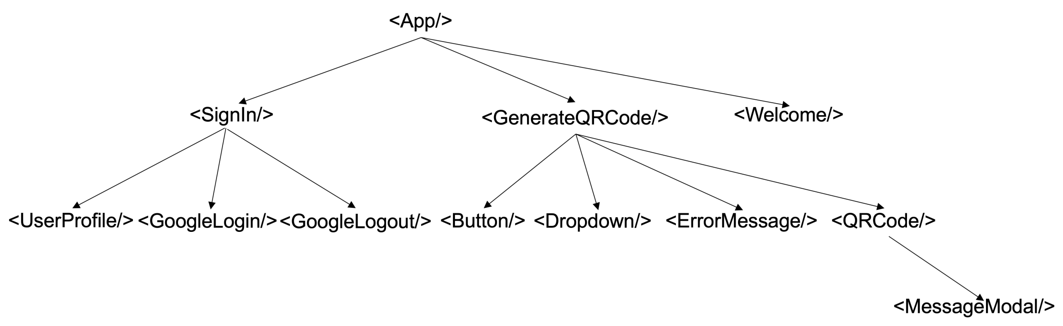
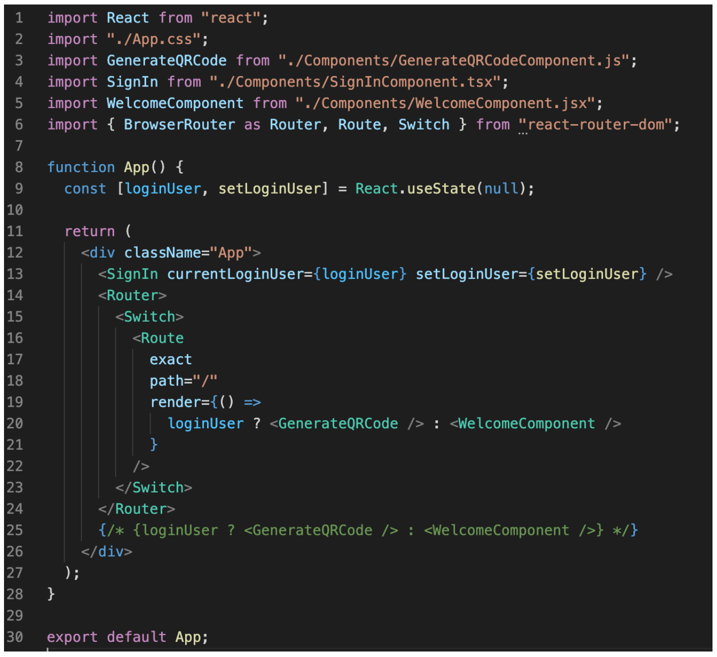


Figure 1: The overview of the components.



```
1  import React from "react";
2  import "./App.css";
3  import GenerateQRCode from "./Components/GenerateQRCodeComponent.js";
4  import SignIn from "./Components/SignInComponent.tsx";
5  import WelcomeComponent from "./Components/WelcomeComponent.jsx";
6  import { BrowserRouter as Router, Route, Switch } from "react-router-dom";
7
8  function App() {
9    const [loginUser, setLoginUser] = React.useState(null);
10
11   return (
12     <div className="App">
13       <SignIn currentLoginUser={loginUser} setLoginUser={setLoginUser} />
14       <Router>
15         <Switch>
16           <Route
17             exact
18             path="/"
19             render={() =>
20               loginUser ? <GenerateQRCode /> : <WelcomeComponent />
21             }
22           />
23         </Switch>
24       </Router>
25       {/* {loginUser ? <GenerateQRCode /> : <WelcomeComponent />} */}
26     </div>
27   );
28 }
29
30 export default App;
```

Figure 2: The codes of App.js.

`App` is the parent component of `SignIn`, `GenerateQRCode` and `Welcome`, and it would render `GenerateQRCode` or `Welcome` based on user login status. `SignIn` consists of `UserProfile`, `GoogleLogin` and `GoogleLogout` buttons, and it is in charge of user's login/logout and displaying user profile. `GenerateQRCode` is responsible for selecting the value and rendering the generated QR code, which consists of a 'Generate QR Code' button, `Dropdown`, `QRCode` and `ErrorMessage`. `MessageModal` is implemented for showing message to the users, and it's rendered in `QRCode`. The implementation details would be introduced in Sec.4.

4 The implementation

4.1 App component

Fig.2 shows the codes of App.js.

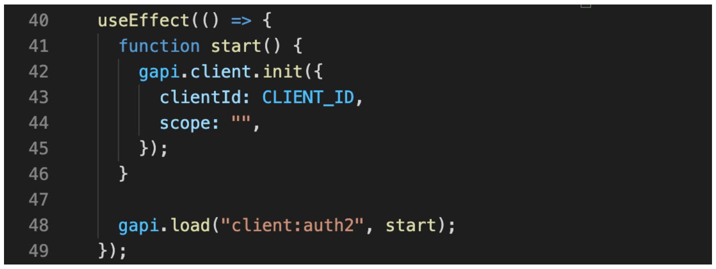
React Hooks is used to define $loginUser$ and $setLoginUser()$ (Line 9), where $loginUser$ is initialised with $null$, indicating user is not logged in. Both $loginUser$ and $setLoginUser()$ are passed to `SignIn` (Line 13), where $loginUser$ would be updated with user's profile by $setLoginUser()$ when a user is logged in. By this way, with the call of a function passed with props, the child component can update the parent's state.

React Router is used to define the root URL (Lines 14-24). If $loginUser$ is not $null$, indicating the user is logged in, then the root URL would be linked to `GenerateQRCode`. Otherwise, the root URL would be linked to `Welcome`.

4.2 SignIn component

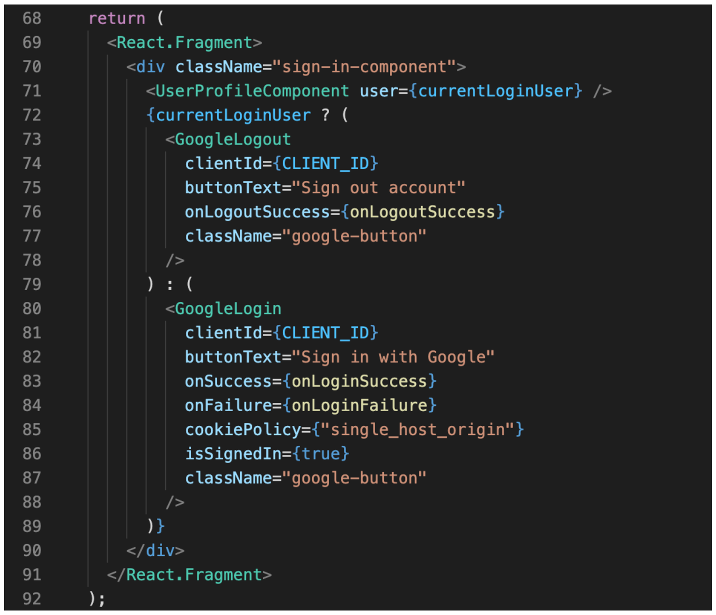
Google Sign-in functionality is implemented following the examples described in How to integrate Google API into your React app and Google OAuth 2.0 Login for React in 5 minutes. Fig.3 shows the initialization of Google authentication, which is called in $useEffect()$. We can then use `GoogleLogin` and `GoogleLogout` for a user to sign in or sign out.

Fig.4 shows the render part of `SignIn`. With passed in $currentLoginUser$, `UserProfile` renders user's profile image and a hello message with the format of 'Hello {user.givenName}!'. `GoogleLogout` would be rendered if $currentLoginUser$ is not $null$. Otherwise, `GoogleLogin` would be rendered. $isSignedIn = true$ is passed to `GoogleLogin` to keep the user logged in. Once the user logged in



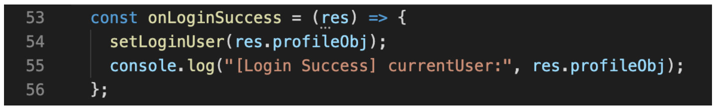
```
useEffect(() => {
  function start() {
    gapi.client.init({
      clientId: CLIENT_ID,
      scope: "",
    });
  }

  gapi.load("client:auth2", start);
});
```

Figure 3: The initialization of Google authentication.



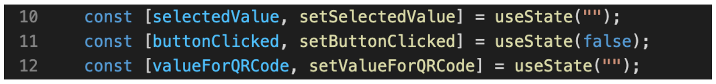
```
return (
  <React.Fragment>
    <div className="sign-in-component">
      <UserProfileComponent user={currentLoginUser} />
      {currentLoginUser ? (
        <GoogleLogout
          clientId={CLIENT_ID}
          buttonText="Sign out account"
          onLogoutSuccess={onLogoutSuccess}
          className="google-button"
        />
      ) : (
        <GoogleLogin
          clientId={CLIENT_ID}
          buttonText="Sign in with Google"
          onSuccess={onLoginSuccess}
          onFailure={onLoginFailure}
          cookiePolicy={"single_host_origin"}
          isSignedIn={true}
          className="google-button"
        />
      )}
    </div>
  </React.Fragment>
);
```

Figure 4: The render part of `SignIn`.



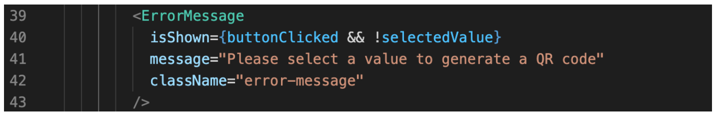
```
const onLoginSuccess = (res) => {
  setLoginUser(res.profileObj);
  console.log("[Login Success] currentUser:", res.profileObj);
};
```

Figure 5: The *onLoginSuccess* function.



```
const [selectedValue, setSelectedValue] = useState("");
const [buttonClicked, setButtonClicked] = useState(false);
const [valueForQRCode, setValueForQRCode] = useState("");
```

Figure 6: State variables in `GenerateQRCode`.



```
39          <ErrorMessage
40            isShown={buttonClicked && !selectedValue}
41            message="Please select a value to generate a QR code"
42            className="error-message"
43          />
```

Figure 7: Conditionally render error message.

successfully, a callback function *onLoginSuccess*() in Fig.5 would be called to set the user profile using *setLoginUser*() function. Notice that *setLoginUser*() would change the local state variable *loginUser* of `App`, and the updated *loginUser* would be passed to `SignIn` as discussed before. Please reference SignInComponent.tsx for more details.

4.3 GenerateQRCode component

`GenerateQRCode` is implemented in GenerateQRCodeComponent.js.

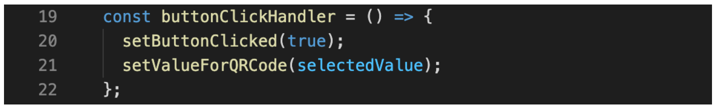
As shown in Fig.6, in order to monitor the status of `GenerateQRCode`, I used *React.useState*() to create three local state variables, which are *selectedValue*, *buttonClicked* and *valueQRCode*, and corresponding setter functions to update these states. Variable *selectedValue* refers to the selected value in the dropdown list, *buttonClicked* indicates if the 'Generate QR Code' button is clicked, and *valueForQRCode* refers to the value for generating QR code.

With these state variables, we can conditionally render the components. For instance, Line 40 in Fig.7 shows that `ErrorMessage` would be rendered only when there is no selected value and the button is clicked.

These state variables are updated by the setter functions. As shown in Fig.8, Line 20 sets *buttonClicked* as `true` in function *buttonClickHandler*(), which would be called by the onClick event of 'Generate QR Code' button. The implementation of `Dropdown` and `ErrorMessage` are quite straightforward, and please reference DropdownComponent.tsx and ErrorMessage.tsx for more details. Besides, I also implemented *ApiDropdown*, which renders a dropdown list of options fetched from an API. Corresponding codes could be found in ApiDropdownComponent.jsx.

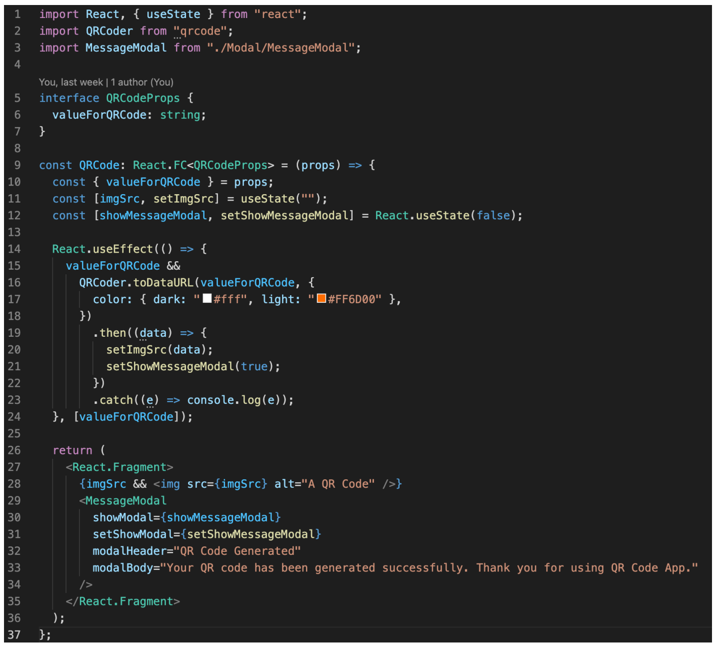
With passed in *valueForQRCode*, `QRCode` renders a generated QR code and show a 'Thank you' message in a modal. Fig.9 shows the codes of `QRCode` component. Library `qrcode` is imported to generate the QR code (Line 2). *React.useEffect*() is called to generate the QR code source each time when rendering `QRCode` (Lines 14-24). In order to avoid indefinite rendering, *valueForQRCode* is set as the dependency, indicating that re-render happens only when *valueForQRCode* changes (Line 24). Function *toDataURL*() is called to generate QR code source (Lines 16-23). The css styles of the QR code is specified (Line 17), and *toDataURL*() returns a `Promise` object. Once the QR code is generated successfully, the source URL would be assigned to *imgSrc* by *setImgSrc*() and also *setShowMessageModal*() sets *showMessageModal* as `true` (Lines 19-22). Then state variables *imgSrc* and *showMessageModal* could be used to control the rendering of the QR code and `MessageModal` (Lines 27-35).

Regarding `MessageModal` and `Modal` components, please reference MessageModal.jsx and Modal.jsx respectively for more details.



```
19    const buttonClickHandler = () => {
20      setButtonClicked(true);
21      setValueForQRCode(selectedValue);
22    };
```

Figure 8: Button click handler for 'Generate QR Code' button.



```tsx
import React, { useState } from "react";
import QRCoder from "qrcode";
import MessageModal from "./Modal/MessageModal";

You, last week | 1 author (You)
interface QRCodeProps {
  valueForQRCode: string;
}

const QRCode: React.FC<QRCodeProps> = (props) => {
  const { valueForQRCode } = props;
  const [imgSrc, setImgSrc] = useState("");
  const [showMessageModal, setShowMessageModal] = React.useState(false);

  React.useEffect(() => {
    valueForQRCode &&
      QRCoder.toDataURL(valueForQRCode, {
        color: { dark: "#fff", light: "#FF6D00" },
      })
        .then((data) => {
          setImgSrc(data);
          setShowMessageModal(true);
        })
        .catch((e) => console.log(e));
  }, [valueForQRCode]);

  return (
    <React.Fragment>
      {imgSrc && <img src={imgSrc} alt="A QR Code" />}
      <MessageModal
        showModal={showMessageModal}
        setShowModal={setShowMessageModal}
        modalHeader="QR Code Generated"
        modalBody="Your QR code has been generated successfully. Thank you for using QR Code App."
      />
    </React.Fragment>
  );
};
```

Figure 9: The codes of `QRCode`.

5 To be improved

The functionalities required in Sec.2 are implemented by above solutions. However, some areas still need improvements.

React Redux could be applied to manage the states globally for the application. In this application, if we want to share some state of a component with others, we have to pass this state as props, like what I did when sharing *loginUser* to `SignIn` in Sec.4.1. However, this way would become increasingly difficult with the scale of the application increases. A common way to solve this issue is to apply React Redux, which helps manage global states in Redux store and trigger an action globally. For example, instead of passing *userLogin* from `App` to `SignIn`, we could store the user's login status in Redux store, then each component could fetch this status from the store, which is convenient and easy to maintain.

We should write tests to cover different scenarios. In this implementation, no test is given. However, in real development, each component and functionality should be covered by tests, which is vital to save time because manually testing each functionality of an application to ensure nothing is broken could take extremely long. I usually write unit tests with Jest, Enzyme and React Testing Library. I also write Cypress tests for testing the UI of some important components and functionalities, although the performance of Cypress is lower than other libraries.