Final Enhancements

This is the last step in redirecting users to protected routes based on authorization.

Further Security

One refinement can be made in the `withAuthorization` higher-order component using the authenticated user from the context:

```
import React from 'react';
import { withRouter } from 'react-router-dom';
import { compose } from 'recompose';

import AuthUserContext from './context';
import { withFirebase } from '../Firebase';
import * as ROUTES from '../../constants/routes';

const withAuthorization = condition => Component => {
  class WithAuthorization extends React.Component {
    componentDidMount() {
      this.listener = firebase.auth.onAuthStateChanged(authUser => {
        if (!condition(authUser)) {
          this.props.history.push(ROUTES.SIGN_IN);
        }
      });
    }

    componentWillUnmount() {
      this.listener();
    }

    render() {
      return (
        <AuthUserContext.Consumer>
          {authUser =>
            condition(authUser) ? <Component {...this.props} /> : null
          }
        </AuthUserContext.Consumer>
      );
    }
  }

  return compose(
    withRouter,
    withFirebase,
  )(WithAuthorization);
};

export default withAuthorization;
```

Session/withAuthorization.js

The improvement in the render method was needed to avoid showing the protected page before the redirect happens. We want to show nothing if the authenticated user doesn't meet the condition's criteria.

In such a case, it's fine if the listener is too late to redirect the user. This is because the higher-order component didn't show the protected component.

Both routes are protected now, so we can render properties of the authenticated user in the `AccountPage` component without a `null` check for the authenticated user. We know the user should be there, otherwise, the higher-order component would redirect to a public route.

```
import React from 'react';

import { AuthUserContext, withAuthorization } from '../Session';
import { PasswordForgetForm } from '../PasswordForget';
import PasswordChangeForm from '../PasswordChange';

const AccountPage = () => (
  <AuthUserContext.Consumer>
    {authUser => (
      <div>
        <h1>Account: {authUser.email}</h1>
        <PasswordForgetForm />
        <PasswordChangeForm />
      </div>
    )}
  </AuthUserContext.Consumer>
);

const condition = authUser => !!authUser;

export default withAuthorization(condition)(AccountPage);
```

Account/index.js

Testing the Application

`/account` or `/home` routes when we are signed out. Both should redirect us to

the `/signin` route. It should also redirect us automatically if we are on one of the routes before signing out.

It is evident how this technique gives us control over authorizations, not just by broader authorization rules, but through more specific role-based and permission-based authorizations. For instance, an **admin** page available for users with the admin role could be protected as follows:

```
import React from 'react';

import * as ROLES from '../../constants/roles';

const AdminPage = () => (
  <div>
    <h1>Admin</h1>
    <p>
      Restricted area! Only users with the admin role are authorized.
    </p>
  </div>
);

const condition = authUser =>
  authUser && authUser.roles.includes(ROLES.ADMIN);

export default withAuthorization(condition)(AdminPage);
```

Admin/index.js

We don’t need to worry about this yet, because we’ll implement a role-based authorization for this application later.

For now, we have successfully implemented:

- A full-fledged authentication mechanism with Firebase in React.
- Added neat features such as password reset and password change.
- Protected routes with dynamic authorization conditions.

In the next section, we will run and verify our app after implementation of optimized Firebase Authorization.