Protected Routes in React with Authorization

In this lesson, we will learn how to put restrictions on access to certain pages based on the user's privileges.

WE'LL COVER THE FOLLOWING

- Route Protections
- Redirecting Based on Authorization

So far, all the routes of our application are accessible to everyone whether the user is authenticated or not.

For instance, when we sign out on the *home* or *account* page, there is no redirection, even though these pages should only be accessible to authenticated users. There is no reason to show a non-authenticated user the account or home page because these are the places where a user accesses sensitive information.

Route Protections

In this section, we will implement the protection for these routes called **authorization**. The protection is a **broad-grained authorization**, which checks for authenticated users. If none are present, it will redirect from a *protected* to a *public* route. Otherwise, it will do nothing.

The condition is defined as follows:

```
const condition = authUser => authUser != null;

// short version
const condition = authUser => !!authUser;
```

In contrast, a more **fine-grained authorization** could be a *role-based* or *permission-based* authorization:

```
// role-based authorization
const condition = authUser => authUser.role === 'ADMIN';


// permission-based authorization
const condition = authUser => authUser.permissions.canEditAccount;
```

Fortunately, we'll implement it in a way that lets us define the authorization condition (predicate) with flexibility. With this, we can use one of the generalized authorization rules, namely, permission-based and role-based authorizations.

Like the `withAuthentication` higher-order component, there is a `withAuthorization` higher-order component used to shield the authorization business logic from our components. It can be used on a component that needs to be protected with authorization (e.g. home page, account page).

Let's add the higher-order component in a new `src/components/Session/withAuthorization.js` file:

```
import React from 'react';

const withAuthorization = () => Component => {
  class WithAuthorization extends React.Component {
    render() {
      return <Component {...this.props} />;
    }
  }

  return WithAuthorization;
};

export default withAuthorization;
```

Session/withAuthorization.js

So far, the higher-order component does nothing but take a component as input and return it as output.

However, the higher-order component should be able to receive a condition function passed as a parameter. We can decide if it should be a broad (authentication) or fine-grained (role-based, permission-based) authorization rule.

Secondly, based on the condition, it should redirect the user to a public page (public route) if the current page is protected.

Let's have a look at the implementation details for the higher-order component and go through them step-by-step:

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

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

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

    componentWillUnmount() {
      this.listener();
    }

    render() {
      return (
        <Component {...this.props} />
      );
    }
  }

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

export default withAuthorization;
```

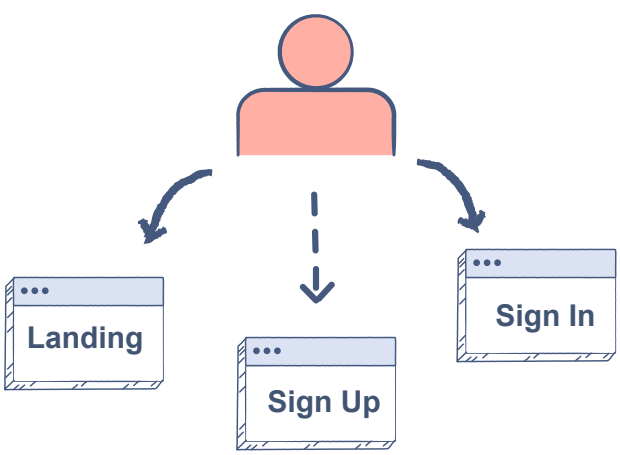
Session/withAuthorization.js

The `render` method displays the passed component (e.g. home page, account page) that should be protected by this higher-order component. We will refine this later.

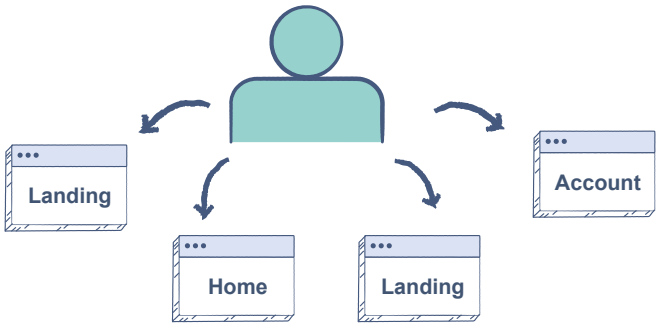
The real authorization logic happens in the `componentDidMount()` lifecycle method. Like the `withAuthentication()` higher-order component, it uses the Firebase listener to trigger a callback function every time the authenticated user changes.

user changes.

The authenticated user is either an `authUser` object or `null`. Within this function, the passed `condition()` function is executed with the `authUser`.



Unauthorized User



Authorized User

Redirecting Based on Authorization

If the authorization fails, for instance, because the authenticated user is `null`, the higher-order component redirects to the **sign-in** page.

If it doesn't fail, the higher-order component does nothing and renders the passed component (e.g. home page, account page).

To redirect a user, the higher-order component has access to the `history` object of the Router using the in-house `withRouter()` from the React Router library.

In the next lesson, we will export the higher-order component into the `src/components/Sessions/index.js`.