A Firebase in React Tutorial for Beginners [2019]

NOVEMBER 20, 2018 BY ROBIN WIERUCH - EDIT THIS POST

Interested in reading this tutorial as one of many chapters in my advanced React with Firebase book? Checkout the entire The Road to Firebase book that teaches you to create business web applications without the need to create a backend application with a database yourself.

This comprehensive tutorial walks you through a real-world application using React and Firebase. React is used to display applications in web browsers and to store local state in components, while Firebase is used for authentication, authorization, and managing a realtime database.

After you've mastered the basics of React, I always recommend moving on to advanced topics like authentication, authorization, and connecting React applications to databases. These operations make up the fundamentals real business applications need. Don't worry about implementing the backend application that manages it all yourself, as Firebase provides the perfect alternative. I have seen real

businesses go from zero to profitable with only React and Firebase as their tools, myself included. No backend application with Node.js was needed, and this tutorial was created to show you how.

50% of this tutorial's outcome can seen here. Security reasons prevent me from showing everything there, though the remaining material can be found in the book. To keep the guide updated, here is a list of the primary libraries and their versions used in this tutorial:

- React 16.7
- React Router 4
- Firebase 4

Please let me know if the tutorial needs any updates for others learning about the topic, and don't hesitate to point out improvements in the comments, or you can visit the article directly on GitHub to open issues or pull requests.

REQUIREMENTS

The requirements for this tutorial are a working editor or IDE/terminal, and recent versions of node and npm. You should have learned about React in the first place. The Road to learn React is a free ebook that provides all the fundamentals of React. You will build a larger application in plain React, and transition from JavaScript ES5 to JavaScript ES6 and beyond. This tutorial will not dive into all the details taught in the ebook, so take the chance to grab your copy of it to learn those first.

TABLE OF CONTENTS

- React Application Setup: create-react-app
- React Router for Firebase Auth
- Firebase in React Setup
- Provide Firebase in React
- Firebase's Authentication API
- Sign Up with React and Firebase
- Sign In with React and Firebase
- Sign Out with React and Firebase
- Session Handling with Firebase/React
- Session Handling with Higher-Order Components
- Password Reset and Password Change with Firebase
  - Password Forget
  - Password Change

- Protected Routes in React with Authorization
- Firebase Realtime Database in React
- Manage Users with Firebase's Realtime Database in React

REACT APPLICATION SETUP: CREATE-REACT-APP

Let's get started with the React + Firebase application we are going to build together. The application should be the perfect starter project to realize your ideas. It should be possible to display information with React, to navigate from URL to URL with React Router and to store and retrieve data with Firebase. Also the application will have everything that's needed to register, login and logout users. In the end, you should be able to implement any feature on top of this application to create well-rounded React applications.

If you lack information on how to setup your React development environment, checkout these setup guides for MacOS and Windows. Now, there are two ways to begin with this application: either follow my guidance in this section; or find a starter project in this GitHub repository and follow its installation instructions. This section will show how to set up the same project from scratch, whereas the starter project grants instant access without setting up the folder/file structure yourself.

The application we are going to build with React and Firebase will be set up with Facebook's official React boilerplate project, called create-react-app. You can set up your project with it on the command line whereas the name for the project is up to you. Afterward, navigate on the command line into the project:

```
npx create-react-app react-firebase-authentication
cd react-firebase-authentication
```

Now you have the following command on your command line to start your application. You can start your application and visit it in the browser:

```
npm start
```

Now we'll set up the project for our needs. First, get rid of the files from the boilerplate React project, since we won't be using them. From the command line, head to your *src/* folder and execute it:

```
cd src
rm App.js App.test.js App.css logo.svg
```

Second, create a *components/* folder in your application's *src/* folder on the command line. This is where all your components will be implemented. Also, the App component that you have removed in the previous step will be recreated here:

```
mkdir components
```

Create a dedicated folder for each component we will implement for this application. For the sake of readability, I split up the commands into multiple lines:

```
cd components
mkdir Account Admin App Home Landing SignIn SignOut SignUp
mkdir Navigation PasswordChange PasswordForget
mkdir Session Firebase
```

In each folder, create an *index.js* file for the component. Navigate into a folder, create the file, and navigate out again. Repeat these steps for every component. You can choose to name your folders/files differently, but that's how I liked to do it for my applications.



```
cd App
touch index.js
cd ..
```

Next, implement a basic React component for each file you created. For the App component in *src/components/App/index.js*, it could look like the following:

```
import React from 'react';

const App = () => (
  <div>
    <h1>App</h1>
  </div>
);

export default App;
```

Fix the relative path to the App component in the *src/index.js* file. Since you have moved the App component to the *src/components* folder, you need to add the */components* subpath to it.

```
import React from 'react';
import ReactDOM from 'react-dom';

import './index.css';
```

```
import * as serviceWorker from './serviceWorker';

import App from './components/App';

ReactDOM.render(<App />, document.getElementById('root'));

serviceWorker.unregister();
```

Then, create one more folder in your *src/* folder:

```
mkdir constants
```

The folder should be located next to *src/components/*. Move into *src/constants/* and create two files for the application's routing and roles management later:

```
cd constants
touch routes.js roles.js
cd ..
```

The application with its folders and files is set up, and you can verify this by running it on the command line and accessing it through a browser. Check the starter project on GitHub I linked in the beginning of this section to verify whether you have set up everything properly.

Exercises:

- Familiarize yourself with the folder structure of a project.
- Optionally, introduce a test for your App component and test the application.
- Optionally, introduce CSS Modules, SASS or Styled Components and style the application.
- Optionally, introduce Git and keep track of your changes by having your project on GitHub.

REACT ROUTER FOR FIREBASE AUTH

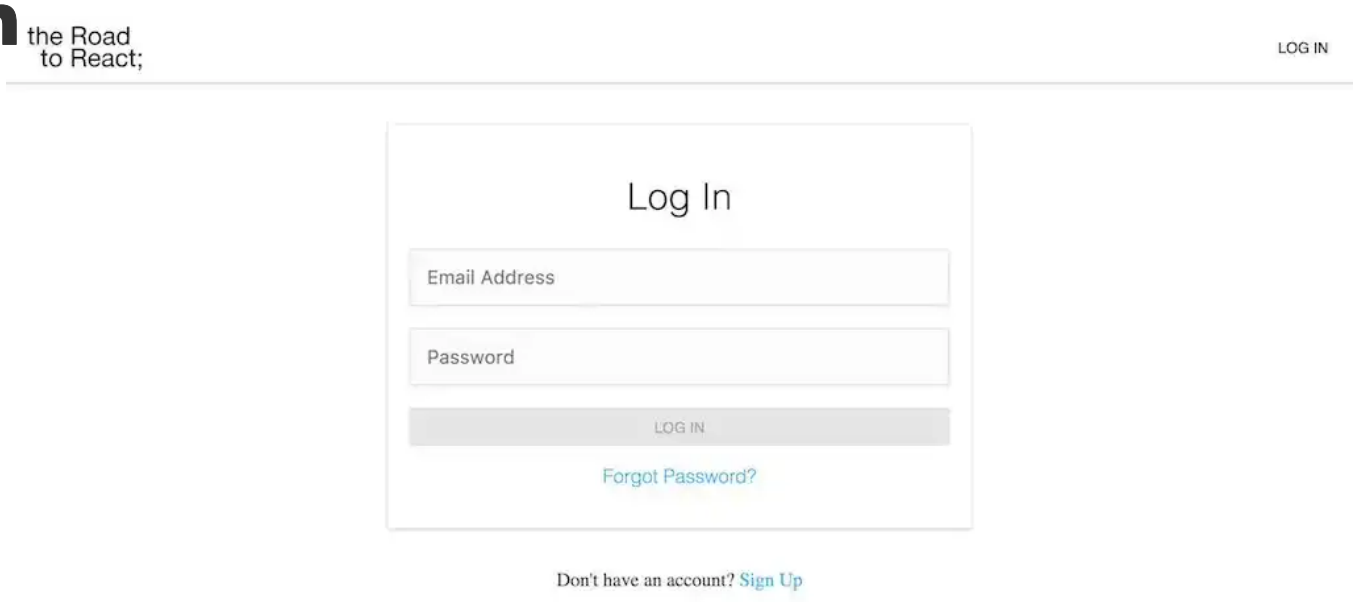
Since we are building a larger application in the following sections, it would be great to have a couple of pages (e.g. landing page, account page, admin page, sign up page, sign in page) to split the application into multiple URLs (e.g. /landing, /account, /admin). These URLs or subpaths of a domain are called routes in a client-side web application. Let's implement the routing with React Router before we dive into Firebase for the realtime database and authentication/authorization. If you haven't used React Router before, it should be straightforward to pick up the basics throughout building this application.

The application should have multiple routes. For instance, a user should be able to visit a public landing page, and also use sign up and sign in pages to enter the application as an authenticated user. If a user is authenticated, it is possible to visit protected pages like account or admin pages whereas the latter is only accessible by authenticated users with an admin role. You can consolidate all the routes of your application in a well-defined *src/constants/routes.js* constants file:

```
export const LANDING = '/';
export const SIGN_UP = '/signup';
export const SIGN_IN = '/signin';
export const HOME = '/home';
export const ACCOUNT = '/account';
export const ADMIN = '/admin';
export const PASSWORD_FORGET = '/pw-forget';
```

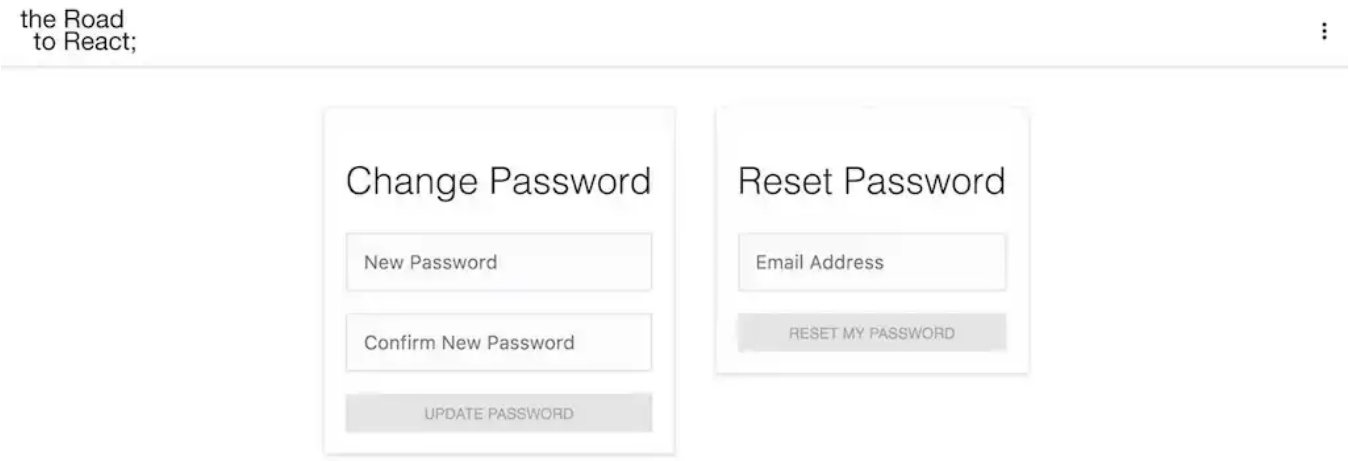
Each route represents a page in your application. For instance, the sign up page should be reachable in development mode via *http://localhost:3000/signup* and in production mode via *http://yourdomain/signup*.

First, you will have a **sign up page** (register page) and a **sign in page** (login page). You can take any web application as the blueprint to structure these routes for a well-rounded authentication experience. Take the following scenario: A user visits your web application, is convinced by your service, and finds the button in the top-level navigation to sign in to your application. But the user has no account yet, so a sign up button is presented as an alternative on the sign in page.

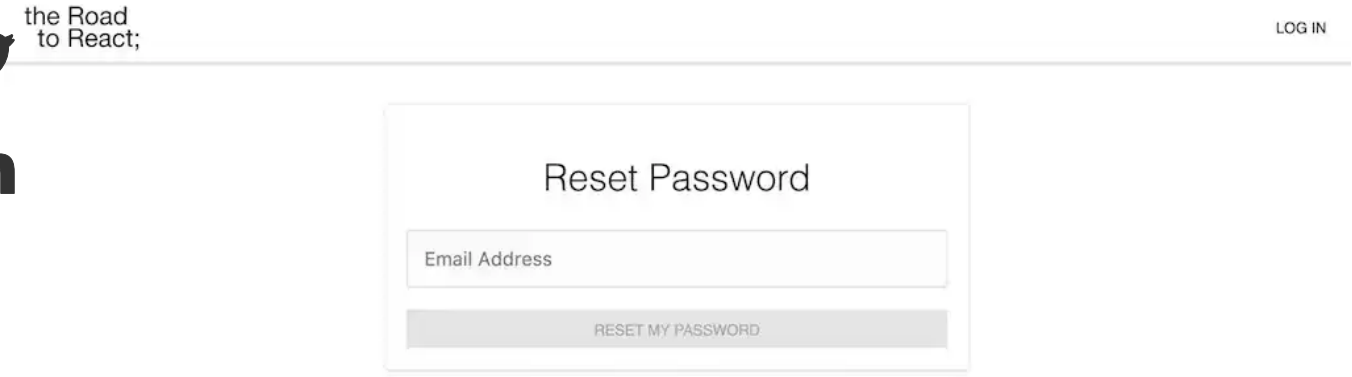


Second, there will be a **landing page** and a **home page**. The landing page is your default route (e.g. *http://yourdomain/*). That's the place where a user ends up when visiting your web application. The user doesn't need to be authenticated to go this route. On the other hand, the home page is a **protected route**, which users can only access if they have been authenticated. You will implement the protection of the route using authorization mechanisms for this application.

Third, next to the **home page**, there will be protected **account page** and **admin page** as well. On the account page, a user can reset or change a password. It is secured by authorization as well, so it is only reachable for authenticated users. On the admin page, a user authorized as admin will be able to manage this application's users. The admin page is protected on a more fine-grained level, because it is only accessible for authenticated admin users.



Lastly, the **password forget** component will be exposed on another non-protected page, a **password forget page**, as well. It is used for users who are not authenticated and forgot about their password.



We've completed the routes for this React with Firebase application. I find it exciting to build a well-rounded application with you, because it can be used as a boilerplate project that gives you authentication, authorization, and a database. These are foundational pillars for any web-based application.

Now, all these routes need to be accessible to the user. First, you need a router for your web application, which is responsible to map routes to React components. React Router is a popular package to enable routing, so install it on the command line:

```
npm install react-router-dom
```

The best way to start is implementing a Navigation component that will be used in the App component. The App component is the perfect place to render the Navigation component, because it

always renders the Navigation component but replaces the other components (pages) based on the routes. Basically, the App component is the container where all your fixed components are going (e.g. navigation bar, side bar, footer), but also your components that are displayed depending on the route in the URL (e.g. account page, login page, password forget page).

First, the App component will use the Navigation component that is not implemented yet. Also, it uses the Router component provided by React Router. The Router makes it possible to navigate from URL-to-URL on the client-side application without another request to a web server for every route change. The application is only fetched once from a web server, after which all routing is done on the client-side with React Router.

In *src/components/App/index.js* file:

```
import React from 'react';
import { BrowserRouter as Router } from 'react-router-dom';

import Navigation from '../Navigation';

const App = () => (
  <Router>
    <Navigation />
  </Router>
);

export default App;
```

Second, implement the Navigation component. It uses the Link component of React Router to enable navigation to different routes. These routes were defined previously in your constants file. Let's import all of them and give every Link component a specific route.

In *src/components/Navigation/index.js* file:

```
import React from 'react';
import { Link } from 'react-router-dom';

import * as ROUTES from '../../constants/routes';

const Navigation = () => (
  <div>
    <ul>
      <li>
        <Link to={ROUTES.SIGN_IN}>Sign In</Link>
      </li>
      <li>
        <Link to={ROUTES.LANDING}>Landing</Link>
      </li>
      <li>
        <Link to={ROUTES.HOME}>Home</Link>
```

```
        </li>
        <li>
          <Link to={ROUTES.ACCOUNT}>Account</Link>
        </li>
        <li>
          <Link to={ROUTES.ADMIN}>Admin</Link>
        </li>
      </ul>
    </div>
);

export default Navigation;
```

Now, run your application again and verify that the links show up in your browser, and that once you click a link, the URL changes. Notice that even though the URL changes, the displayed content doesn't change. The navigation is only there to enable navigation through your application. But no one knows what to render on each route. That's where the *route to component* mapping comes in. In your App component, you can specify which components should show up according to corresponding routes with the help of the Route component from React Router.

In *src/components/App/index.js* file:

```
import React from 'react';
import {
  BrowserRouter as Router,
  Route,
} from 'react-router-dom';

import Navigation from '../Navigation';
import LandingPage from '../Landing';
import SignUpPage from '../SignUp';
import SignInPage from '../SignIn';
import PasswordForgetPage from '../PasswordForget';
import HomePage from '../Home';
import AccountPage from '../Account';
import AdminPage from '../Admin';

import * as ROUTES from '../../constants/routes';

const App = () => (
  <Router>
    <div>
      <Navigation />

      <hr />

      <Route exact path={ROUTES.LANDING} component={LandingPage} />
      <Route path={ROUTES.SIGN_UP} component={SignUpPage} />
      <Route path={ROUTES.SIGN_IN} component={SignInPage} />
      <Route path={ROUTES.PASSWORD_FORGET} component={PasswordForgetPage} />
      <Route path={ROUTES.HOME} component={HomePage} />
      <Route path={ROUTES.ACCOUNT} component={AccountPage} />
```

```
        <Route path={ROUTES.ADMIN} component={AdminPage} />
      </div>
    </Router>
  );

  export default App;
```

If a route matches a path prop, the respective component will be displayed; thus, all the page components in the App component are exchangeable by changing the route, but the Navigation component stays fixed independently of any route changes. This is how you enable a static frame with various components (e.g. Navigation) around your dynamic pages driven by routes. It's all made possible by React's powerful composition.

Previously, you created basic components for each page component used by our routes. Now you should be able to start the application again. When you click through the links in the Navigation component, the displayed page component should change according to the URL. The routes for the PasswordForget and SignUp components are not used in the Navigation component, but will be defined elsewhere later. For now, you have successfully implemented fundamental routing for this application.

Exercises:

- Learn more about React Router
- Confirm your source code for the last section

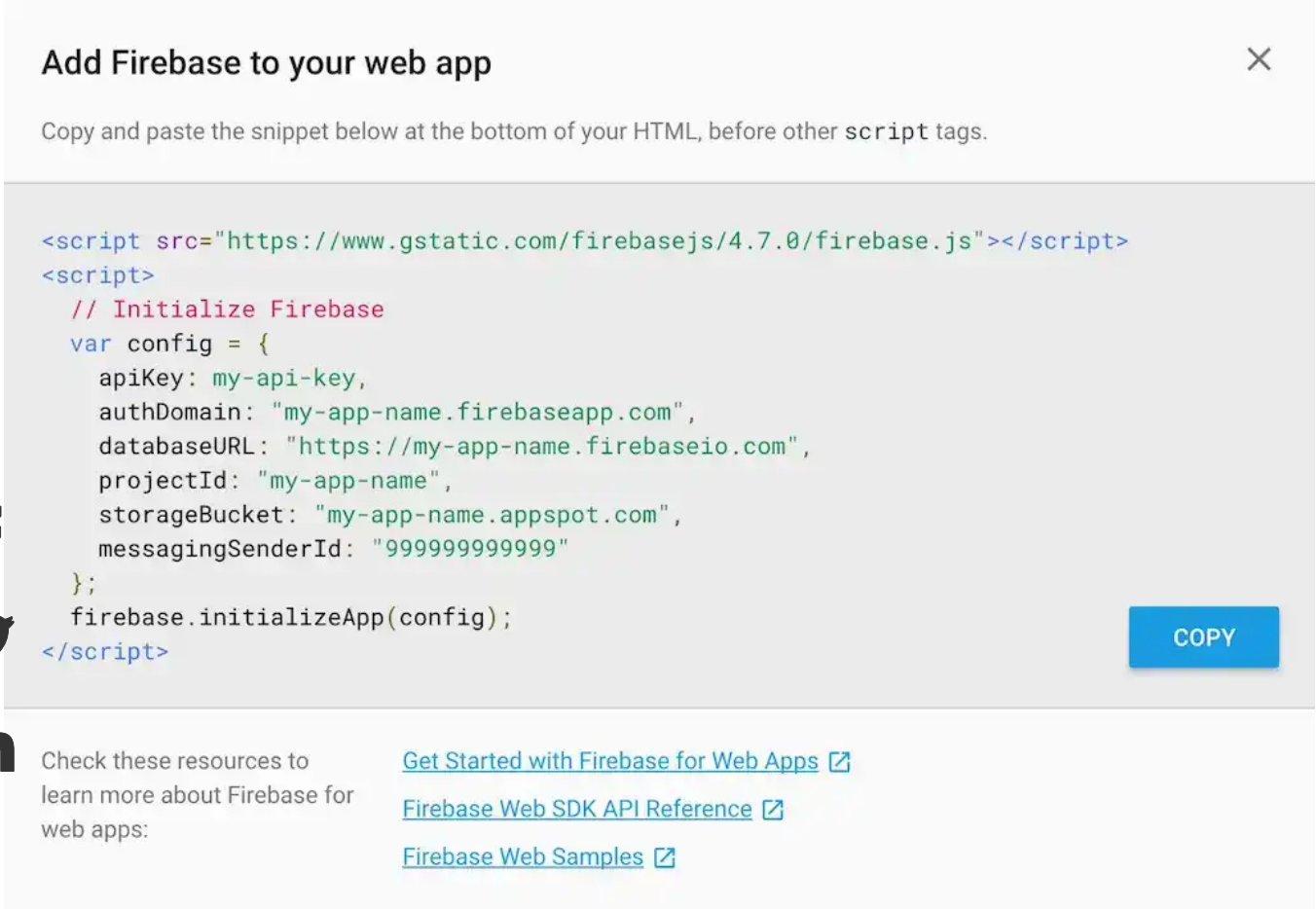
FIREBASE IN REACT SETUP

The main focus here is using Firebase in React for the application we'll build together. Firebase, bought by Google in 2014, enables realtime databases, extensive authentication and authorization, and even for deployment. You can build real-world applications with React and Firebase without worrying about implementing a backend application. All the things a backend application would handle, like authentication and a database, is handled by Firebase. Many businesses use React and Firebase to power their applications, as it is the ultimate combination to launch an MVP.

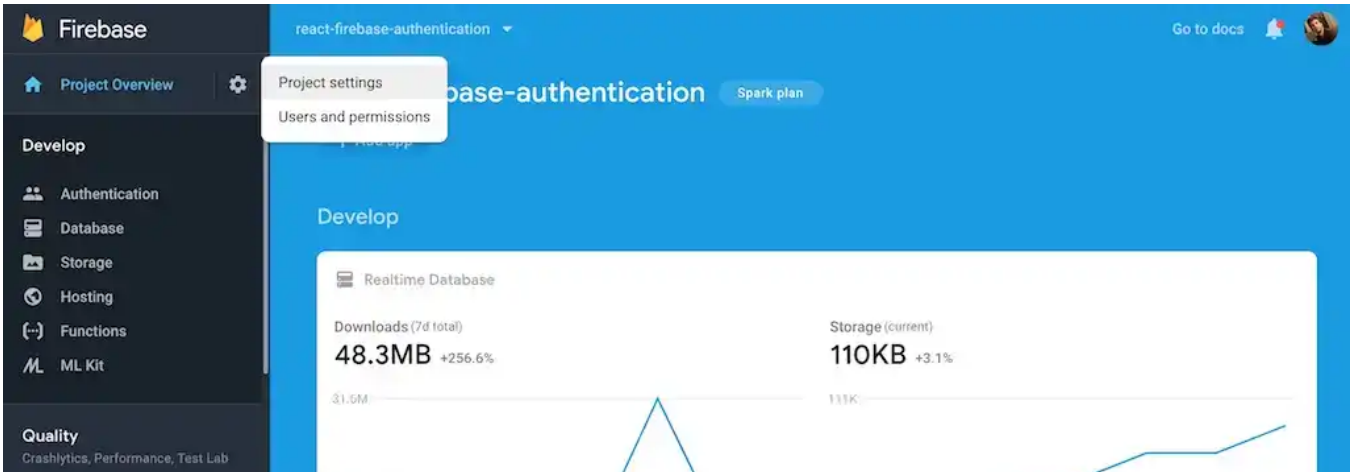
To start, sign up on the official Firebase website. After you have created a Firebase account, you should be able to create projects and be granted access to the project dashboard. We'll begin by creating a project for this application on their platform whereas the project can have any name. In the case of this application, run it on the free pricing plan. If you want to scale your application later, you can change the plan. Follow this visual Firebase setup and introduction guide to learn more about

Firebase's dashboard and features. It would also give you first guidelines on how to acivate Firebase's Realtime Database instead of Firebase's Cloud Firestore.

Next, find the project's configuration in the settings on your project's dashboard. There, you'll have access to all the necessary information: secrets, keys, ids and other details to set up your application. Copy these in the next step to your React application.



Sometimes the Firebase website doesn't make it easy to find this page. Since it's moved around with every iteration of the website, I cannot give you any clear advice where to find it on your dashboard. This is an opportunity to familiarize yourself with Firebase project's dashboard while you search for the configuration.



Now that we've completed the Firebase setup, you can return to your application in your editor/IDE to add the Firebase configuration. First, install Firebase for your application on the command line:

```
npm install firebase
```

Next, we'll create a new file for the Firebase setup. We will use a JavaScript class to encapsulate all Firebase functionalities, realtime database, and authentication, as a well-defined API for the rest of the application. You need only instantiate the class once, after which it can use it then to interact with the Firebase API, your custom Firebase interface.

Let's start by copying the configuration from your Firebase project's dashboard on their website to your application as a configuration object in a new *src/components/Firebase/firebase.js* file. Make sure to replace the capitalized keys with the corresponding keys from your copied configuration:

```
const config = {
  apiKey: YOUR_API_KEY,
  authDomain: YOUR_AUTH_DOMAIN,
  databaseURL: YOUR_DATABASE_URL,
  projectId: YOUR_PROJECT_ID,
  storageBucket: '',
  messagingSenderId: YOUR_MESSAGING_SENDER_ID,
};
```

As alternative, you can also use environment variables in React applications, but you have to use the `REACT_APP` prefix when you use create-react-app to set up the application:

```
const config = {
  apiKey: process.env.REACT_APP_API_KEY,
  authDomain: process.env.REACT_APP_AUTH_DOMAIN,
  databaseURL: process.env.REACT_APP_DATABASE_URL,
  projectId: process.env.REACT_APP_PROJECT_ID,
  storageBucket: process.env.REACT_APP_STORAGE_BUCKET,
  messagingSenderId: process.env.REACT_APP_MESSAGING_SENDER_ID,
};
```

Now you can define the environmental variables in a new *.env* file in your project's root folder. The *.env* file can also be added to your *.gitginore* file (in case you are using git), so your Firebase credentials are not exposed publicly on a platform like GitHub.

```
REACT_APP_API_KEY=XXXXxxxx
REACT_APP_AUTH_DOMAIN=xxxxXXXX.firebaseapp.com
REACT_APP_DATABASE_URL=https://xxxXXXX.firebaseio.com
REACT_APP_PROJECT_ID=xxxxXXXX
REACT_APP_STORAGE_BUCKET=xxxxXXXX.appspot.com
REACT_APP_MESSAGING_SENDER_ID=xxxxXXXX
```

Both ways work. You can define the configuration inline in source code or as environment variables. Environmental variables are more secure, and should be used when uploading your project to a version control system like git, though we will be continuing with the Firebase setup. Import firebase from the library you installed earlier, and then use it within a new Firebase class to initialize firebase with the configuration:

```
import app from 'firebase/app';

const config = {
  apiKey: process.env.REACT_APP_API_KEY,
  authDomain: process.env.REACT_APP_AUTH_DOMAIN,
  databaseURL: process.env.REACT_APP_DATABASE_URL,
  projectId: process.env.REACT_APP_PROJECT_ID,
  storageBucket: process.env.REACT_APP_STORAGE_BUCKET,
  messagingSenderId: process.env.REACT_APP_MESSAGING_SENDER_ID,
};

class Firebase {
  constructor() {
    app.initializeApp(config);
  }
}

export default Firebase;
```

That's all that is needed for a firebase configuration in your application. Optionally, you can create a second Firebase project on the Firebase website to have one project for your development environment and one project for your production environment. That way, you never mix data in the Firebase database in development mode with data from your deployed application (production mode). If you decide to create projects for both environments, use the two configuration objects in your Firebase setup and decide which one you take depending on the development/production environment:

```
import app from 'firebase/app';

const prodConfig = {
  apiKey: process.env.REACT_APP_PROD_API_KEY,
  authDomain: process.env.REACT_APP_PROD_AUTH_DOMAIN,
  databaseURL: process.env.REACT_APP_PROD_DATABASE_URL,
```

```
  projectId: process.env.REACT_APP_PROD_PROJECT_ID,
  storageBucket: process.env.REACT_APP_PROD_STORAGE_BUCKET,
  messagingSenderId: process.env.REACT_APP_PROD_MESSAGING_SENDER_ID,
};

const devConfig = {
  apiKey: process.env.REACT_APP_DEV_API_KEY,
  authDomain: process.env.REACT_APP_DEV_AUTH_DOMAIN,
  databaseURL: process.env.REACT_APP_DEV_DATABASE_URL,
  projectId: process.env.REACT_APP_DEV_PROJECT_ID,
  storageBucket: process.env.REACT_APP_DEV_STORAGE_BUCKET,
  messagingSenderId: process.env.REACT_APP_DEV_MESSAGING_SENDER_ID,
};

const config =
  process.env.NODE_ENV === 'production' ? prodConfig : devConfig;

class Firebase {
  constructor() {
    app.initializeApp(config);
  }
}

export default Firebase;
```

An alternate way to implement this is to specify a dedicated *.env.development* and *.env.production* file for both kinds of environment variables in your project. Each file is used to define environmental variables for the matching environment. Defining a configuration becomes straightforward again, because you don't have to select the correct configuration yourself.

```
import app from 'firebase/app';

const config = {
  apiKey: process.env.REACT_APP_API_KEY,
  authDomain: process.env.REACT_APP_AUTH_DOMAIN,
  databaseURL: process.env.REACT_APP_DATABASE_URL,
  projectId: process.env.REACT_APP_PROJECT_ID,
  storageBucket: process.env.REACT_APP_STORAGE_BUCKET,
  messagingSenderId: process.env.REACT_APP_MESSAGING_SENDER_ID,
};

class Firebase {
  constructor() {
    app.initializeApp(config);
  }
}

export default Firebase;
```

Whether you used environment variables, defined the configuration inline, used only one Firebase project, or multiple projects for each environment, you configured Firebase for your React application.

The next section will show you how a Firebase instance created from the Firebase class is used in React.

Exercises:

- Read more about the Firebase Setup for Web Applications
- Read more about Firebase's Pricing Plans to know better about the limitations of the free plan.
- Confirm your source code for the last section

PROVIDE FIREBASE IN REACT

You created a Firebase class, but you are not using it in your React application yet. In this section, we'll connect the Firebase with the React world. The simple approach is to create a Firebase instance with the Firebase class, and then import the instance (or class) in every React component where it's needed. That's not the best approach though, for two reasons:

- It is more difficult to test your React components.
- It is more error prone, because Firebase should only be initialized once in your application (singleton) and by exposing the Firebase class to every React component, you could end up by mistake with multiple Firebase instances.

An alternative way is to use React's Context API to provide a Firebase instance once at the top-level of your component hierarchy. Create a new *src/components/Firebase/context.js* file in your Firebase module and provide the following implementation details:

```
import React from 'react';

const FirebaseContext = React.createContext(null);

export default FirebaseContext;
```

The `createContext()` function essentially creates two components. The `FirebaseContext.Provider` component is used to provide a Firebase instance once at the top-level of your React component tree, which we will do in this section; and the `FirebaseContext.Consumer` component is used to retrieve the Firebase instance if it is needed in the React component. For a well-encapsulated Firebase module, we'll define a *index.js* file in our Firebase folder that exports all necessary functionalities (Firebase class, Firebase context for Consumer and Provider components):

```
import FirebaseContext from './context';
```

```
import Firebase from './firebase';

export default Firebase;

export { FirebaseContext };
```

The Firebase Context from the Firebase module (folder) is used to provide a Firebase instance to your entire application in the *src/index.js* file. You only need to create the Firebase instance with the Firebase class and pass it as value prop to the React's Context:

```
import React from 'react';
import ReactDOM from 'react-dom';

import './index.css';
import * as serviceWorker from './serviceWorker';

import App from './components/App';
import Firebase, { FirebaseContext } from './components/Firebase';

ReactDOM.render(
  <FirebaseContext.Provider value={new Firebase()}>
    <App />
  </FirebaseContext.Provider>,
  document.getElementById('root'),
);

serviceWorker.unregister();
```

Doing it this way, we can be assured that Firebase is only instantiated once and that it is injected via React's Context API to React's component tree. Now, every component that is interested in using Firebase has access to the Firebase instance with a `FirebaseContext.Consumer` component. Even though you will see it first-hand later for this application, the following code snippet shows how it would work:

```
import React from 'react';

import  { FirebaseContext } from '../Firebase';

const SomeComponent = () => (
  <FirebaseContext.Consumer>
    {firebase => {
      return <div>I've access to Firebase and render something.</div>;
    }}
  </FirebaseContext.Consumer>
);

export default SomeComponent;
```

Firebase and React are now connected, the fundamental step to make the layers communicate with each other. Next, we will implement the interface for the Firebase class on our side to communicate with the Firebase API.

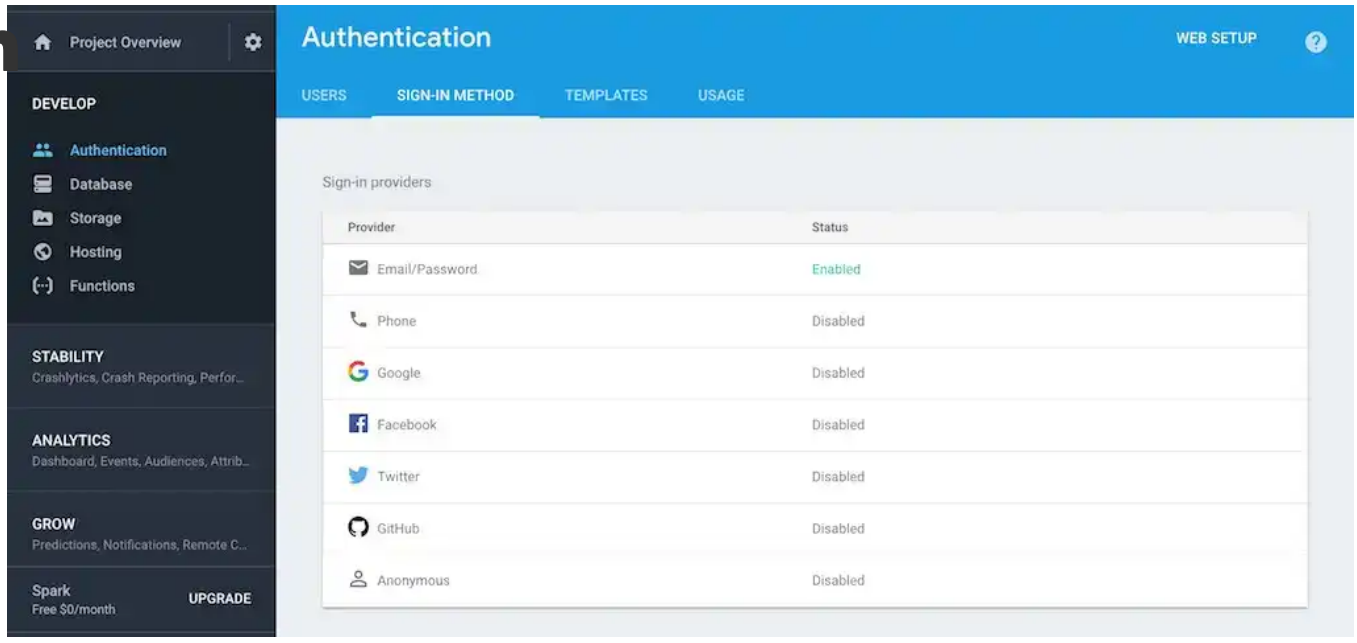
Exercises:

- Read more about React's Context API
- Confirm your source code for the last section

FIREBASE'S AUTHENTICATION API

In the previous section, you created a Firebase project on the official Firebase website. This section will implement the interface of your Firebase class that enables communication between the class and the Firebase authentication API. In the sections afterward, you will use the interface of the Firebase class in your React components.

First, we need to activate one of the available authentication providers on Firebase's website. On your project's Firebase dashboard, you can find a menu item which says "Authentication". Select it and click "Sign-In Method" menu item afterward. There you can enable the authentication with Email/Password:



Second, we will implement the authentication API for our Firebase class. Import and instantiate the package from Firebase responsible for all the authentication in your *src/components/Firebase/firebase.js* file:

```
import app from 'firebase/app';
```

```
import 'firebase/auth';

const config = {
  apiKey: process.env.REACT_APP_API_KEY,
  authDomain: process.env.REACT_APP_AUTH_DOMAIN,
  databaseURL: process.env.REACT_APP_DATABASE_URL,
  projectId: process.env.REACT_APP_PROJECT_ID,
  storageBucket: process.env.REACT_APP_STORAGE_BUCKET,
  messagingSenderId: process.env.REACT_APP_MESSAGING_SENDER_ID,
};

class Firebase {
  constructor() {
    app.initializeApp(config);

    this.auth = app.auth();
  }
}

export default Firebase;
```

Let's define all the authentication functions as class methods step by step. They will serve our communication channel from the Firebase class to the Firebase API. First, the sign up function (registration) takes email and password parameters for its function signature and uses an official Firebase API endpoint to create a user:

```
import app from 'firebase/app';
import 'firebase/auth';

const config = { ... };

class Firebase {
  constructor() {
    app.initializeApp(config);

    this.auth = app.auth();
  }

  // *** Auth API ***

  doCreateUserWithEmailAndPassword = (email, password) =>
    this.auth.createUserWithEmailAndPassword(email, password);
}

export default Firebase;
```

We'll also set up the login/sign-in function, which takes email and password parameters, as well:

```
import app from 'firebase/app';
import 'firebase/auth';
```

```
const config = { ... };

class Firebase {
  constructor() {
    app.initializeApp(config);

    this.auth = app.auth();
  }

  // *** Auth API ***

  doCreateUserWithEmailAndPassword = (email, password) =>
    this.auth.createUserWithEmailAndPassword(email, password);

  doSignInWithEmailAndPassword = (email, password) =>
    this.auth.signInWithEmailAndPassword(email, password);
}

export default Firebase;
```

These endpoints are called asynchronously, and they will need to be resolved later, as well as error handling. For instance, it is not possible to sign in a user who is not signed up yet since the Firebase API would return an error. In case of the sign out function, you don't need to pass any argument to it, because Firebase knows about the currently authenticated user. If no user is authenticated, nothing will happen when this function is called.

```
import app from 'firebase/app';
import 'firebase/auth';

const config = { ... };

class Firebase {
  constructor() {
    app.initializeApp(config);

    this.auth = app.auth();
  }

  // *** Auth API ***

  doCreateUserWithEmailAndPassword = (email, password) =>
    this.auth.createUserWithEmailAndPassword(email, password);

  doSignInWithEmailAndPassword = (email, password) =>
    this.auth.signInWithEmailAndPassword(email, password);

  doSignOut = () => this.auth.signOut();
}

export default Firebase;
```

There are two more authentication methods to reset and change a password for an authenticated user:

```
import app from 'firebase/app';
import 'firebase/auth';

const config = { ... };

class Firebase {
  constructor() {
    app.initializeApp(config);

    this.auth = app.auth();
  }

  // *** Auth API ***

  doCreateUserWithEmailAndPassword = (email, password) =>
    this.auth.createUserWithEmailAndPassword(email, password);

  doSignInWithEmailAndPassword = (email, password) =>
    this.auth.signInWithEmailAndPassword(email, password);

  doSignOut = () => this.auth.signOut();

  doPasswordReset = email => this.auth.sendPasswordResetEmail(email);

  doPasswordUpdate = password =>
    this.auth.currentUser.updatePassword(password);
}

export default Firebase;
```

That's the authentication interface for your React components that will connect to the Firebase API. In the next section, we will consume all the methods of your Firebase class in your React components.

Exercises:

- Read more about Firebase Authentication for Web
- Confirm your source code for the last section

SIGN UP WITH REACT AND FIREBASE

We set up all the routes for your application, configured Firebase and implemented the authentication API for your Firebase class. It's also possible to use Firebase within your React components. Now it's time to use the authentication functionalities in your React components, which we'll build from

scratch. I try to put most of the code in one block, because the components are not too small, and splitting them up step by step might be too verbose. Nevertheless, I will guide you through each code block afterward. The code blocks for forms can become repetitive, so they will be explained once well.

Let's start with the sign up page (registration page). It consists of the page, a form, and a link. The form is used to sign up a new user to your application with username, email, and password. The link will be used on the sign in page (login page) later if a user has no account yet. It is a redirect to the sign up page, but not used on the sign up page itself. Implement the *src/components/SignUp/index.js* file the following way:

```
import React, { Component } from 'react';
import { Link } from 'react-router-dom';

import * as ROUTES from '../../constants/routes';

const SignUpPage = () => (
  <div>
    <h1>SignUp</h1>
    <SignUpForm />
  </div>
);

class SignUpForm extends Component {
  constructor(props) {
    super(props);
  }

  onSubmit = event => {

  }

  onChange = event => {

  };

  render() {
    return (
      <form onSubmit={this.onSubmit}>

      </form>
    );
  }
}

const SignUpLink = () => (
  <p>
    Don't have an account? <Link to={ROUTES.SIGN_UP}>Sign Up</Link>
  </p>
);

export default SignUpPage;
```

```
export { SignUpForm, SignUpLink };
```

The SignUpForm component is the only React class component in this file, because it has to manage the form state in React's local state. There are two pieces missing in the current SignUpForm component: the form content in the render method in terms of input fields to capture the information (email address, password, etc.) of a user and the implementation of the `onSubmit` class method when a user signs up eventually.

First, let's initialize the state of the component. It will capture the user information such as username, email, and password. There will be a second password field/state for a password confirmation. In addition, there is an error state to capture an error object in case of the sign up request to the Firebase API fails. The state is initialized by an object destructuring. This way, we can use the initial state object to reset the state after a successful sign up.

```
...

const INITIAL_STATE = {
  username: '',
  email: '',
  passwordOne: '',
  passwordTwo: '',
  error: null,
};

class SignUpForm extends Component {
  constructor(props) {
    super(props);

    this.state = { ...INITIAL_STATE };
  }

  ...

}

...
```

Let's implement all the input fields to capture the information in the render method of the component. The input fields need to update the local state of the component by using a `onChange` handler.

```
...

class SignUpForm extends Component {

  ...
```

```
  onChange = event => {
    this.setState({ [event.target.name]: event.target.value });
  };

  render() {
    const {
      username,
      email,
      passwordOne,
      passwordTwo,
      error,
    } = this.state;

    return (
      <form onSubmit={this.onSubmit}>
        <input
          name="username"
          value={username}
          onChange={this.onChange}
          type="text"
          placeholder="Full Name"
        />
        <input
          name="email"
          value={email}
          onChange={this.onChange}
          type="text"
          placeholder="Email Address"
        />
        <input
          name="passwordOne"
          value={passwordOne}
          onChange={this.onChange}
          type="password"
          placeholder="Password"
        />
        <input
          name="passwordTwo"
          value={passwordTwo}
          onChange={this.onChange}
          type="password"
          placeholder="Confirm Password"
        />
        <button type="submit">Sign Up</button>

        {error && <p>{error.message}</p>}
      </form>
    );
  }
}

...
```

Let's take the last implemented code block apart. All the input fields implement the unidirectional data flow of React; thus, each input field gets a value from the local state and updates the value in the local state with a `onChange` handler. The input fields are controlled by the local state of the component and don't control their own states. They are controlled components.

In the last part of the form, there is an optional error message from an error object. The error objects from Firebase have this message property by default, so you can rely on it to display the proper text for your application's user. However, the message is only shown when there is an actual error using a conditional rendering.

One piece in the form is missing: validation. Let's use an `isInvalid` boolean to enable or disable the submit button.

```
...

class SignUpForm extends Component {

  ...

  render() {
    const {
      username,
      email,
      passwordOne,
      passwordTwo,
      error,
    } = this.state;

    const isInvalid =
      passwordOne !== passwordTwo ||
      passwordOne === '' ||
      email === '' ||
      username === '';

    return (
      <form onSubmit={this.onSubmit}>
        <input
        ...
        <button disabled={isInvalid} type="submit">
          Sign Up
        </button>

        {error && <p>{error.message}</p>}
      </form>
    );
  }
}

...
```

The user is only allowed to sign up if both passwords are the same, and if the username, email and at least one password are filled with a string. This is password confirmation in a common sign up process.

You should be able to visit the */signup* route in your browser after starting your application to confirm that the form with all its input fields shows up. You should also be able to type into it (confirmation that the local state updates are working) and able to enable the submit button by providing all input fields a string (confirmation that the validation works).

What's missing in the component is the `onSubmit()` class method, which will pass all the form data to the Firebase authentication API via your authentication interface in the Firebase class:

```
...

class SignUpForm extends Component {

  ...

  onSubmit = event => {
    const { username, email, passwordOne } = this.state;

    this.props.firebase
      .doCreateUserWithEmailAndPassword(email, passwordOne)
      .then(authUser => {
        this.setState({ ...INITIAL_STATE });
      })
      .catch(error => {
        this.setState({ error });
      });

    event.preventDefault();
  };

  ...
}

...
```

The code is not working yet, but let's break down what we have so far. All the necessary information passed to the authentication API can be destructured from the local state. You will only need one password property, because both password strings should be the same after the validation.

Next, call the sign up function defined in the previous section in the Firebase class, which takes the email and the password property. The username is not used yet for the sign up process, but will be used later.

If the request resolves successfully, you can set the local state of the component to its initial state to empty the input fields. If the request is rejected, you will run into the catch block and set the error

object in the local state. An error message should show up in the form due to the conditional rendering in your component's render method.

Also, the `preventDefault()` method on the event prevents a reload of the browser which otherwise would be a natural behavior when using a submit in a form. Note that the signed up user object from the Firebase API is available in the callback function of the then block in our request. You will use it later with the username.

You may have also noticed that one essential piece is missing: We didn't make the Firebase instance available in the SignUpForm component's props yet. Let's change this by utilizing our Firebase Context in the SignUpPage component, and by passing the Firebase instance to the SignUpForm.

```
import React, { Component } from 'react';
import { Link } from 'react-router-dom';

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

const SignUpPage = () => (
  <div>
    <h1>SignUp</h1>
    <FirebaseContext.Consumer>
      {firebase => <SignUpForm firebase={firebase} />}
    </FirebaseContext.Consumer>
  </div>
);

const INITIAL_STATE = { ... };

class SignUpForm extends Component {
  ...
}

...
```

Now the registration of a new user should work. However, I'd like to make one improvement on how we access the Firebase instance here. Rather than using a render prop component, which is automatically given with React's Context Consumer component, it may be simpler to use a higher-order component. Let's implement this higher-order component in the *src/components/Firebase/context.js*:

```
import React from 'react';

const FirebaseContext = React.createContext(null);

export const withFirebase = Component => props => (
  <FirebaseContext.Consumer>
    {firebase => <Component {...props} firebase={firebase} />}
```

```
  </FirebaseContext.Consumer>
);

export default FirebaseContext;
```

Next, make it available via our Firebase module in the *src/components/Firebase/index.js* file:

```
import FirebaseContext, { withFirebase } from './context';
import Firebase from './firebase';

export default Firebase;

export { FirebaseContext, withFirebase };
```

Now, instead of using the Firebase Context directly in the SignUpPage, which doesn't need to know about the Firebase instance, use the higher-order component to wrap your SignUpForm. Afterward, the SignUpForm has access to the Firebase instance via the higher-order component. It's also possible to use the SignUpForm as standalone without the SignUpPage, because it is responsible to get the Firebase instance via the higher-order component.

```
import React, { Component } from 'react';
import { Link } from 'react-router-dom';

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

const SignUpPage = () => (
  <div>
    <h1>SignUp</h1>
    <SignUpForm />
  </div>
);

const INITIAL_STATE = { ... };

class SignUpFormBase extends Component {
  ...
}

const SignUpLink = () => ...

const SignUpForm = withFirebase(SignUpFormBase);

export default SignUpPage;

export { SignUpForm, SignUpLink };
```

When a user signs up to your application, you want to redirect the user to another page. It could be the user's home page, a protected route for only authenticated users. You will need the help of React Router to redirect the user after a successful sign up.

```
import React, { Component } from 'react';
import { Link, withRouter } from 'react-router-dom';

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

...

class SignUpFormBase extends Component {

  ...

  onSubmit = (event) => {
    const { username, email, passwordOne } = this.state;

    this.props.firebase
      .doCreateUserWithEmailAndPassword(email, passwordOne)
      .then(authUser => {
        this.setState({ ...INITIAL_STATE });
        this.props.history.push(ROUTES.HOME);
      })
      .catch(error => {
        this.setState({ error });
      });

    event.preventDefault();
  }

  ...
}

...

const SignUpForm = withRouter(withFirebase(SignUpFormBase));

export default SignUpPage;

export { SignUpForm, SignUpLink };
```

Let's take the previous code block apart again. To redirect a user to another page programmatically, we need access to React Router to redirect the user to another page. Fortunately, the React Router node package offers a higher-order component to make the router properties accessible in the props of a component. Any component that goes in the `withRouter()` higher-order component gains access to all the properties of the router, so when passing the enhanced SignUpFormBase component to the `withRouter()` higher-order component, it has access to the props of the router. The relevant

property from the router props is the `history` object, because it allows us to redirect a user to another page by pushing a route to it.

The history object of the router can be used in the `onSubmit()` class method eventually. If a request resolves successfully, you can push any route to the history object. Since the pushed */home* route is defined in our App component with a matching component to be rendered, the displayed page component will change after the redirect.

There is one improvement that we can make for the higher-order components used for the SignUpForm. Nesting functions (higher-order components) into each other like we did before can become verbose. A better way is to compose the higher-order components instead. To do this, install recompose for your application on the command line:

```
npm install recompose
```

You can use recompose to organize your higher-order components. Since the higher-order components don't depend on each other, the order doesn't matter. Otherwise, it may be good to know that the compose function applies the higher-order components from right to left.

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

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

...

const SignUpForm = compose(
  withRouter,
  withFirebase,
)(SignUpFormBase);

export default SignUpPage;

export { SignUpForm, SignUpLink };
```

Run your application again. If you signed up a user successfully, it should redirect to the home page. If the sign up fails, you should see an error message. Try to sign up a user with the same email address twice and verify that a similar error message shows up: "The email address is already in use by another account.". Congratulations, you signed up your first user via Firebase authentication.

Exercises:

- Read more about data fetching in React

- Read more about higher-order components in React
- Read more about render prop components in React
- Confirm your source code for the last section

SIGN IN WITH REACT AND FIREBASE

A sign up automatically results in a sign in/login by the user. We cannot rely on this mechanic, however, since a user could be signed up but not signed in. Let's implement the login with Firebase now. It is similar to the sign up mechanism and components, so this time we won't split it into so many code blocks. Implement the *src/components/SignIn/index.js* file:

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

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

const SignInPage = () => (
  <div>
    <h1>SignIn</h1>
    <SignInForm />
    <SignUpLink />
  </div>
);

const INITIAL_STATE = {
  email: '',
  password: '',
  error: null,
};

class SignInFormBase extends Component {
  constructor(props) {
    super(props);

    this.state = { ...INITIAL_STATE };
  }

  onSubmit = event => {
    const { email, password } = this.state;

    this.props.firebase
      .doSignInWithEmailAndPassword(email, password)
      .then(() => {
        this.setState({ ...INITIAL_STATE });
        this.props.history.push(ROUTES.HOME);
      })
```

```
      .catch(error => {
        this.setState({ error });
      });

    event.preventDefault();
  };

  onChange = event => {
    this.setState({ [event.target.name]: event.target.value });
  };

  render() {
    const { email, password, error } = this.state;

    const isInvalid = password === '' || email === '';

    return (
      <form onSubmit={this.onSubmit}>
        <input
          name="email"
          value={email}
          onChange={this.onChange}
          type="text"
          placeholder="Email Address"
        />
        <input
          name="password"
          value={password}
          onChange={this.onChange}
          type="password"
          placeholder="Password"
        />
        <button disabled={isInvalid} type="submit">
          Sign In
        </button>

        {error && <p>{error.message}</p>}
      </form>
    );
  }
}

const SignInForm = compose(
  withRouter,
  withFirebase,
)(SignInFormBase);

export default SignInPage;

export { SignInForm };
```

It is almost the same as the sign up form. Its input fields capture all the necessary information like username and password. A validation step makes sure the email and password are set before performing the request by enabling or disabling the submit button. The authentication API is used

again, this time with a function to sign in the user rather than sign them up. If sign in succeeds, the local state is updated with the initial state and the user is redirected again. If the sign in fails, an error object is stored in the local state and an error message appears. The SignUpLink, which was defined earlier in the SignUp module, is used on the sign in page. It lets users sign up if they don't have an account, and it is found on the sign in page.

Exercises:

- Familiarize yourself with the SignIn and SignInForm components.
  - If they are mysterious to you, checkout the previous section with the implementation of the SignUpForm again
- Confirm your source code for the last section

SIGN OUT WITH REACT AND FIREBASE

To complete the authentication loop, next we'll implement the sign out component. The component is just a button that appears within the Navigation component. Since we can use the previously-defined authentication API to sign out a user, passing functionality to a button in a React component is fairly straightforward. Implement the SignOutButton component in the *src/components/SignOut/index.js* file:

```
import React from 'react';

import { withFirebase } from '../Firebase';

const SignOutButton = ({ firebase }) => (
  <button type="button" onClick={firebase.doSignOut}>
    Sign Out
  </button>
);

export default withFirebase(SignOutButton);
```

The SignOutButton has access to the Firebase instance using the higher-order component again. Now, use the SignOutButton in the Navigation component in your *src/components/Navigation/index.js* file:

```
import React from 'react';
import { Link } from 'react-router-dom';

import SignOutButton from '../SignOut';
import * as ROUTES from '../../constants/routes';

const Navigation = () => (
```

```
    <div>
      <ul>
        <li>
          <Link to={ROUTES.SIGN_IN}>Sign In</Link>
        </li>
        <li>
          <Link to={ROUTES.LANDING}>Landing</Link>
        </li>
        <li>
          <Link to={ROUTES.HOME}>Home</Link>
        </li>
        <li>
          <Link to={ROUTES.ACCOUNT}>Account</Link>
        </li>
        <li>
          <Link to={ROUTES.ADMIN}>Admin</Link>
        </li>
        <li>
          <SignOutButton />
        </li>
      </ul>
    </div>
  );

export default Navigation;
```

Regarding components, everything is set to fulfil a full authentication roundtrip. Users can sign up (register), sign in (login), and sign out (logout).

Exercises:

- Read more about Firebase Authentication with E-Mail/Password
- Confirm your source code for the last section

SESSION HANDLING WITH FIREBASE/REACT

This section is the most important one for the authentication process. You have all the components needed to fulfil an authentication roundtrip in React, and all that's missing is an overseer for the session state. Logic regarding the current authenticated user needs to be stored and made accessible to other components. This is often the point where developers start to use a state management library like Redux or MobX. Without these, we'll make due using global state instead of state management libraries.

Since our application is made under the umbrella of App component, it's sufficient to manage the session state in the App component using React's local state. The App component only needs to keep track of an authenticated user (session). If a user is authenticated, store it in the local state and pass

the authenticated user object down to all components that are interested in it. Otherwise, pass the authenticated user down as `null`. That way, all components interested in it can adjust their behavior (e.g. use conditional rendering) based on the session state. For instance, the Navigation component is interested because it has to show different options to authenticated and non authenticated users. The SignOut component shouldn't show up for a non authenticated user, for example.

We handle session handling in the App component in the *src/components/App/index.js* file. Because the component handles local state now, you have to refactor it to a class component. It manages the local state of a `authUser` object, and then passes it to the Navigation component.

```
import React, { Component } from 'react';
import { BrowserRouter as Router, Route } from 'react-router-dom';

...

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      authUser: null,
    };
  }

  render() {
    return (
      <Router>
        <div>
          <Navigation authUser={this.state.authUser} />

          <hr/>

          ...
        </div>
      </Router>
    );
  }
}

export default App;
```

The Navigation component can be made aware of authenticated user to display different options. It should either show the available links for an authenticated user or a non authenticated user.

```
import React from 'react';
import { Link } from 'react-router-dom';

import SignOutButton from '../SignOut';
import * as ROUTES from '../../constants/routes';
```

```
const Navigation = ({ authUser }) => (
  <div>{authUser ? <NavigationAuth /> : <NavigationNonAuth />}</div>
);

const NavigationAuth = () => (
  <ul>
    <li>
      <Link to={ROUTES.LANDING}>Landing</Link>
    </li>
    <li>
      <Link to={ROUTES.HOME}>Home</Link>
    </li>
    <li>
      <Link to={ROUTES.ACCOUNT}>Account</Link>
    </li>
    <li>
      <SignOutButton />
    </li>
  </ul>
);

const NavigationNonAuth = () => (
  <ul>
    <li>
      <Link to={ROUTES.LANDING}>Landing</Link>
    </li>
    <li>
      <Link to={ROUTES.SIGN_IN}>Sign In</Link>
    </li>
  </ul>
);

export default Navigation;
```

Let's see where the `authUser` (authenticated user) comes from in the App component. Firebase offers a listener function to get the authenticated user from Firebase:

```
...

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

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      authUser: null,
    };
  }

  componentDidMount() {
```

```
    this.props.firebase.auth.onAuthStateChanged(authUser => {
      authUser
        ? this.setState({ authUser })
        : this.setState({ authUser: null });
    });
  }

  ...

}

export default withFirebase(App);
```

The helper function `onAuthStateChanged()` receives a function as parameter that has access to the authenticated user. Also, the passed function is called every time something changes for the authenticated user. It is called when a user signs up, signs in, and signs out. If a user signs out, the `authUser` object becomes null, so the `authUser` property in the local state is set to null and all components depending on it adjust their behavior (e.g. display different options like the Navigation component).

We also want to avoid memory leaks that lead to performance issues, so we'll remove the listener if the component unmounts.

```
...

class App extends Component {
  ...

  componentDidMount() {
    this.listener = this.props.firebase.auth.onAuthStateChanged(
      authUser => {
        authUser
          ? this.setState({ authUser })
          : this.setState({ authUser: null });
      },
    );
  }

  componentWillUnmount() {
    this.listener();
  }

  ...

}

export default withFirebase(App);
```

Start your application and verify that your sign up, sign in, and sign out functionality works, and that the Navigation component displays the options depending on the session state (authenticated user).

Congratulations, you have successfully implemented the authentication process with Firebase in React. Everything in the following sections regarding authentication is considered extra, to improve the developer's experience and add a couple of useful features along the way.

Exercises:

- Read more about Firebase's Authenticated User
- Confirm your source code for the last section

SESSION HANDLING WITH HIGHER-ORDER COMPONENTS

We added a basic version of session handling in the last section. However, the authenticated user still needs to be passed down from the App component to interested parties. That can become tedious over time, because the authenticated user has to be passed through all components until it reaches all the leaf components. You used the React Context API to pass down the Firebase instance to any component before. Here, you will do the same for the authenticated user. In a new *src/components/Session/context.js* file, place the following new React Context for the session authenticated user):

```
import React from 'react';

const AuthUserContext = React.createContext(null);

export default AuthUserContext;
```

Next, import and export it from the *src/components/Session/index.js* file that is the entry point to this module:

```
import AuthUserContext from './context';

export { AuthUserContext };
```

The App component can use the new context to provide the authenticated user to components that are interested in it:

```
...
```

```
import { AuthUserContext } from '../Session';

class App extends Component {
  ...

  render() {
    return (
      <AuthUserContext.Provider value={this.state.authUser}>
        <Router>
          <div>
            <Navigation />

            <hr />

            ...
          </div>
        </Router>
      </AuthUserContext.Provider>
    );
  }
}

export default withFirebase(App);
```

The `authUser` doesn't need to be passed to the Navigation component anymore. Instead, the Navigation component uses the new context to consume the authenticated user:

```
...

import { AuthUserContext } from '../Session';

const Navigation = () => (
  <div>
    <AuthUserContext.Consumer>
      {authUser =>
        authUser ? <NavigationAuth /> : <NavigationNonAuth />
      }
    </AuthUserContext.Consumer>
  </div>
);
```

The application works the same as before, except any component can simply use React's Context to consume the authenticated user. To keep the App component clean and concise, I like to extract the session handling for the authenticated user to a separate higher-order component in a new *src/components/Session/withAuthentication.js* file:

```
import React from 'react';

const withAuthentication = Component => {
```

```
  class WithAuthentication extends React.Component {
    render() {
      return <Component {...this.props} />;
    }
  }

  return WithAuthentication;
};

export default withAuthentication;
```

Move all logic that deals with the authenticated user from the App component to it:

```
import React from 'react';

import AuthUserContext from './context';
import { withFirebase } from '../Firebase';

const withAuthentication = Component => {
  class WithAuthentication extends React.Component {
    constructor(props) {
      super(props);

      this.state = {
        authUser: null,
      };
    }

    componentDidMount() {
      this.listener = this.props.firebase.auth.onAuthStateChanged(
        authUser => {
          authUser
            ? this.setState({ authUser })
            : this.setState({ authUser: null });
        },
      );
    }

    componentWillUnmount() {
      this.listener();
    }

    render() {
      return (
        <AuthUserContext.Provider value={this.state.authUser}>
          <Component {...this.props} />
        </AuthUserContext.Provider>
      );
    }
  }

  return withFirebase(WithAuthentication);
};
```

```
export default withAuthentication;
```

As you can see, it also uses the new React Context to provide the authenticated user. The App component will not be in charge of it anymore. Next, export the higher-order component from the *src/components/Session/index.js* file, so that it can be used in the App component after:

```
import AuthUserContext from './context';
import withAuthentication from './withAuthentication';

export { AuthUserContext, withAuthentication };
```

The App component becomes a function component again, without the additional business logic for the authenticated user. Now, it uses the higher-order component to make the authenticated user available for all other components below of the App component:

```
import React from 'react';
import { BrowserRouter as Router, Route } from 'react-router-dom';

import Navigation from '../Navigation';
import LandingPage from '../Landing';
import SignUpPage from '../SignUp';
import SignInPage from '../SignIn';
import PasswordForgetPage from '../PasswordForget';
import HomePage from '../Home';
import AccountPage from '../Account';
import AdminPage from '../Admin';

import * as ROUTES from '../../constants/routes';
import { withAuthentication } from '../Session';

const App = () => (
  <Router>
    <div>
      <Navigation />

      <hr />

      <Route exact path={ROUTES.LANDING} component={LandingPage} />
      <Route path={ROUTES.SIGN_UP} component={SignUpPage} />
      <Route path={ROUTES.SIGN_IN} component={SignInPage} />
      <Route
        path={ROUTES.PASSWORD_FORGET}
        component={PasswordForgetPage}
      />
      <Route path={ROUTES.HOME} component={HomePage} />
      <Route path={ROUTES.ACCOUNT} component={AccountPage} />
      <Route path={ROUTES.ADMIN} component={AdminPage} />
    </div>
  </Router>
);
```

```
export default withAuthentication(App);
```

Start the application and verify that it still works. You didn't change any behavior in this section, but shielded away the more complex logic into a higher-order component. Also, the application now passes the authenticated user implicitly via React's Context, rather than explicitly through the component tree using props.

Exercises:

- Check again your Firebase Context and higher-order component implementation in the *src/components/Firebase* module, which is quite similar to what you have done in this section.
- Confirm your source code for the last section

PASSWORD RESET AND PASSWORD CHANGE WITH FIREBASE

Let's take a step back from the higher-order components, React Context API, and session handling. In this section, we will implement two additional features available in the Firebase authentication API, the ability to retrieve (password forget) and change a password.

Password Forget

Let's start by implementing the password forget feature. Since you already implemented the interface in your Firebase class, you can use it in components. The following file adds most of the password reset logic in a form again. We already used a couple of those forms before, so it shouldn't be different now. Add this in the *src/components/PasswordForget/index.js* file:

```
import React, { Component } from 'react';
import { Link } from 'react-router-dom';

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

const PasswordForgetPage = () => (
  <div>
    <h1>PasswordForget</h1>
    <PasswordForgetForm />
  </div>
);

const INITIAL_STATE = {
  email: '',
  error: null,
```

```
};

class PasswordForgetFormBase extends Component {
  constructor(props) {
    super(props);

    this.state = { ...INITIAL_STATE };
  }

  onSubmit = event => {
    const { email } = this.state;

    this.props.firebase
      .doPasswordReset(email)
      .then(() => {
        this.setState({ ...INITIAL_STATE });
      })
      .catch(error => {
        this.setState({ error });
      });

    event.preventDefault();
  };

  onChange = event => {
    this.setState({ [event.target.name]: event.target.value });
  };

  render() {
    const { email, error } = this.state;

    const isInvalid = email === '';

    return (
      <form onSubmit={this.onSubmit}>
        <input
          name="email"
          value={this.state.email}
          onChange={this.onChange}
          type="text"
          placeholder="Email Address"
        />
        <button disabled={isInvalid} type="submit">
          Reset My Password
        </button>

        {error && <p>{error.message}</p>}
      </form>
    );
  }
}

const PasswordForgetLink = () => (
  <p>
    <Link to={ROUTES.PASSWORD_FORGET}>Forgot Password?</Link>
  </p>
```

```
);

export default PasswordForgetPage;

const PasswordForgetForm = withFirebase(PasswordForgetFormBase);

export { PasswordForgetForm, PasswordForgetLink };
```

The code is verbose, but it it's no different from the sign up and sign in forms from previous sections. The password forget uses a form to submit the information (email address) needed by the Firebase authentication API to reset the password. A class method (onSubmit) ensures the information is send to the API. It also resets the form's input field on a successful request, and shows an error on an erroneous request. The form is validated before it is submitted as well. The file implements a password forget link as a component which isn't used directly in the form component. It is similar to the SignUpLink component that we used on in the SignInPage component. This link is the same, and it's still usable. If a user forgets the password after sign up, the password forget page uses the link in the *src/components/SignIn/index.js* file:

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

import { SignUpLink } from '../SignUp';
import { PasswordForgetLink } from '../PasswordForget';
import { withFirebase } from '../Firebase';
import * as ROUTES from '../../constants/routes';

const SignInPage = () => (
  <div>
    <h1>SignIn</h1>
    <SignInForm />
    <PasswordForgetLink />
    <SignUpLink />
  </div>
);


...
```

The password forget page is already matched in the App component, so you can drop the PasswordForgetLink component in the sign in page and know the mapping between route and component is complete. Start the application and reset your password. It doesn't matter if you are authenticated or not. Once you send the request, you should get an email from Firebase to update your password.

Password Change

Next we'll add the password change feature, which is also in your Firebase interface. You only need a form component to use it. Again, the form component isn't any different from the sign in, sign up, and password forget forms. In the *src/components/PasswordChange/index.js* file add the following component:

```
import React, { Component } from 'react';

import { withFirebase } from '../Firebase';

const INITIAL_STATE = {
  passwordOne: '',
  passwordTwo: '',
  error: null,
};

class PasswordChangeForm extends Component {
  constructor(props) {
    super(props);

    this.state = { ...INITIAL_STATE };
  }

  onSubmit = event => {
    const { passwordOne } = this.state;

    this.props.firebase
      .doPasswordUpdate(passwordOne)
      .then(() => {
        this.setState({ ...INITIAL_STATE });
      })
      .catch(error => {
        this.setState({ error });
      });

    event.preventDefault();
  };

  onChange = event => {
    this.setState({ [event.target.name]: event.target.value });
  };

  render() {
    const { passwordOne, passwordTwo, error } = this.state;

    const isInvalid =
      passwordOne !== passwordTwo || passwordOne === '';

    return (
      <form onSubmit={this.onSubmit}>
        <input
          name="passwordOne"
          value={passwordOne}
          onChange={this.onChange}
          type="password"
```

```
            placeholder="New Password"
          />
          <input
            name="passwordTwo"
            value={passwordTwo}
            onChange={this.onChange}
            type="password"
            placeholder="Confirm New Password"
          />
          <button disabled={isInvalid} type="submit">
            Reset My Password
          </button>

          {error && <p>{error.message}</p>}
        </form>
      );
    }
  }

  export default withFirebase(PasswordChangeForm);
```

The component updates its local state using **onChange** handlers in the input fields. It validates the state before submitting a request to change the password by enabling or disabling the submit button, and it shows again an error message when a request fails.

So far, the PasswordChangeForm is not matched by any route, because it should live on the Account page. The Account page could serve as the central place for users to manage their account, where it shows the PasswordChangeForm and PasswordResetForm, accessible by a standalone route. You already created the *src/components/Account/index.js* file and matched the route in the App component. You only need to implement it:

```
import React from 'react';

import { PasswordForgetForm } from '../PasswordForget';
import PasswordChangeForm from '../PasswordChange';

const AccountPage = () => (
  <div>
    <h1>Account Page</h1>
    <PasswordForgetForm />
    <PasswordChangeForm />
  </div>
);

export default AccountPage;
```

The Account page doesn't have any business logic. It uses the password forget and password change forms in a central place. In this section, your user experience improved significantly with the password

forget and password change features, handling scenarios where users have trouble remembering passwords.

Exercises:

- Consider ways to protect the Account page and make it accessible only for authenticated users.
- Confirm your source code for the last section

PROTECTED ROUTES IN REACT WITH AUTHORIZATION

So far, all of your application's routes are accessible by everyone. It doesn't matter whether the user is authenticated or not authenticated. For instance, when you sign out on the home or account page, there is no redirect, even though these pages should be only accessible for authenticated users. There is no reason to show a non authenticated user the account or home page in the first place, because these are the places where a user accesses sensitive information. In this section, so you will implement a protection for these routes called authorization. The protection is a **broad-grained authorization**, which checks for authenticated users. If none is present, it redirects from a protected to a public route; else, it will do nothing. The condition is defined as:

```
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 implement it in a way that lets you define the authorization condition (predicate) with flexibility, so that you can use a more generalized authorization rule, permission-based or role-based authorizations.

Like the `withAuthentication` higher-order component, there is a `withAuthorization` higher-order component to shield the authorization business logic from your components. It can be used on

any component that needs to be protected with authorization (e.g. home page, account page). Let's start to 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;
```

So far, the higher-order component is not doing anything but taking a component as input and returning it as output. However, the higher-order component should be able to receive a condition function passed as parameter. You can decide if it should be a broad or fine-grained (role-based, permission-based) authorization rule. Second, it has to decide based on the condition whether it should redirect to a public page (public route), because the user isn't authorized to view the current protected page (protected route). Let's paste the implementation details for the higher-order component and go through it 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;
```

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. The authenticated user is either a `authUser` object or `null`. Within this function, the passed `condition()` function is executed with the `authUser`. 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()` higher-order component from the React Router library.

Remember to export the higher-order component from your session module into the *src/components/Sessions/index.js* file:

```
import AuthUserContext from './context';
import withAuthentication from './withAuthentication';
import withAuthorization from './withAuthorization';

export { AuthUserContext, withAuthentication, withAuthorization };
```

In the next step, you can use the higher-order component to protect your routes (e.g. /home and /account) with authorization rules using the passed `condition()` function. To keep it simple, the following two components are only protected with a broad authorization rule that checks if the `authUser` is not `null`. First, enhance the HomePage component with the higher-order component and define the authorization condition for it:

```
import React from 'react';

import { withAuthorization } from '../Session';

const HomePage = () => (
```

```
    <div>
      <h1>Home Page</h1>
      <p>The Home Page is accessible by every signed in user.</p>
    </div>
);

const condition = authUser => !!authUser;

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

Second, enhance the AccountPage component with the higher-order component and define the authorization condition. It similar to the previous usage:

```
import React from 'react';

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

const AccountPage = () => (
  <div>
    <h1>Account Page</h1>
    <PasswordForgetForm />
    <PasswordChangeForm />
  </div>
);

const condition = authUser => !!authUser;

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

The protection of both pages/routes is almost done. 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 = this.props.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;
```

The improvement in the render method was needed to avoid showing the protected page before the redirect happens. You want to show nothing if the authenticated user doesn't meet the condition's criteria. Then it's fine if the listener is too late to redirect the user, 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. You 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);
```

You can try it by signing out from your application and trying to access the */account* or */home* routes. Both should redirect you to the */signin* route. It should also redirect you automatically when you stay on one of the routes while you sign out.

You can imagine how this technique gives control over authorizations, not just by broader authorization rules, but 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[ROLES.ADMIN];

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

Don't worry about this yet, because we'll implement a role-based authorization for this application later. For now, you have successfully implemented a full-fledged authentication mechanisms with Firebase in React, added neat features such as password reset and password change, and protected routes with dynamic authorization conditions.

Exercises:

- Research yourself how a role-based or permission-based authorization could be implemented.
- Confirm your source code for the last section

FIREBASE REALTIME DATABASE IN REACT

So far, only Firebase knows about your users. There is no way to retrieve a single user or a list of users for your application from their authentication database. They are stored internally by Firebase to keep

the authentication secure. That's good, because you are never involved in storing sensitive data like passwords. However, you can introduce the Firebase realtime database to keep track of user entities yourself. It makes sense, because then you can associate other domain entities (e.g. a message, a book, an invoice) created by your users to your users. You should keep control over your users, even though Firebase takes care about all the sensitive data. This section will explain how to store users in your realtime database in Firebase. First, initialize the realtime database API for your Firebase class as you did earlier for the authentication API:

```
import app from 'firebase/app';
import 'firebase/auth';
import 'firebase/database';

const config = { ... };

class Firebase {
  constructor() {
    app.initializeApp(config);

    this.auth = app.auth();
    this.db = app.database();
  }

  // *** Auth API ***

  ...
}

export default Firebase;
```

Second, extend the interface for your Firebase class for the user entity. It defines two new functions: one to get a reference to a user by identifier (uid) and one to get a reference to all users:

```
import app from 'firebase/app';
import 'firebase/auth';
import 'firebase/database';

const config = { ... };

class Firebase {
  constructor() {
    app.initializeApp(config);

    this.auth = app.auth();
    this.db = app.database();
  }

  // *** Auth API ***

  doCreateUserWithEmailAndPassword = (email, password) =>
    this.auth.createUserWithEmailAndPassword(email, password);
```

```
    doSignInWithEmailAndPassword = (email, password) =>
      this.auth.signInWithEmailAndPassword(email, password);

    doSignOut = () => this.auth.signOut();

    doPasswordReset = email => this.auth.sendPasswordResetEmail(email);

    doPasswordUpdate = password =>
      this.auth.currentUser.updatePassword(password);

    // *** User API ***

    user = uid => this.db.ref(`users/${uid}`);

    users = () => this.db.ref('users');
}

export default Firebase;
```

The paths in the `ref()` method match the location where your entities (users) will be stored in Firebase's realtime database API. If you delete a user at "users/5", the user with the identifier 5 will be removed from the database. If you create a new user at "users", Firebase creates the identifier for you and assigns all the information you pass for the user. The paths follow the REST philosophy where every entity (e.g. user, message, book, author) is associated with a URI, and HTTP methods are used to create, update, delete and get entities. In Firebase, the RESTful URI becomes a simple path, and the HTTP methods become Firebase's API.

Exercises:

- Activate Firebase's Realtime Database on your Firebase Dashboard
  - Set your Database Rules on your Firebase Project's Dashboard to `{ "rules": { ".read": true, ".write": true } }` to give everyone read and write access for now.
- Read more about Firebase's realtime database setup for Web
- Confirm your source code for the last section

MANAGE USERS WITH FIREBASE'S REALTIME DATABASE IN REACT

Now, use these references in your React components to create and get users from Firebase's realtime database. The best place to add user creation is the SignUpForm component, as it is the most natural place to save users after signing up via the Firebase authentication API. Add another API request to create a user when the sign up is successful. In *src/components/SignUp/index.js* file:

```
...

class SignUpFormBase extends Component {
  constructor(props) {
    super(props);

    this.state = { ...INITIAL_STATE };
  }

  onSubmit = event => {
    const { username, email, passwordOne } = this.state;

    this.props.firebase
      .doCreateUserWithEmailAndPassword(email, passwordOne)
      .then(authUser => {
        // Create a user in your Firebase realtime database
        return this.props.firebase
          .user(authUser.user.uid)
          .set({
            username,
            email,
          });
      })
      .then(() => {
        this.setState({ ...INITIAL_STATE });
        this.props.history.push(ROUTES.HOME);
      })
      .catch(error => {
        this.setState({ error });
      });

    event.preventDefault();
  };

  ...
}

...
```

There are two important things happening for a new sign up via the submit handler:

- (1) It creates a user in Firebase's internal authentication database that is only limited accessible.
- (2) If (1) was successful, it creates a user in Firebase's realtime database that is accessible.

To create a user in Firebase's realtime database, it uses the previously created reference from the Firebase class by providing the identifier (uid) of the user from Firebase's authentication database. Then the `set()` method can be used to provide data for this entity which is allocated for "users/uid". Finally, you can use the `username` as well to provide additional information about your user.

Note: It is fine to store user information in your own database. However, you should make sure not to store the password or any other sensitive data of the user on your own. Firebase already deals with

the authentication, so there is no need to store the password in your database. Many steps are necessary to secure sensitive data (e.g. encryption), and it could be a security risk to perform it on your own.

After the second Firebase request that creates the user resolves successfully, the previous business logic takes place again: reset the local state and redirect to the home page. To verify the user creation is working, retrieve all the users from the realtime database in one of your other components. The admin page may be a good choice for it, because it can be used by admin users to manage the application-wide users later. First, make the admin page available via your Navigation component:

```
...

const NavigationAuth = () => (
  <ul>
    <li>
      <Link to={ROUTES.LANDING}>Landing</Link>
    </li>
    <li>
      <Link to={ROUTES.HOME}>Home</Link>
    </li>
    <li>
      <Link to={ROUTES.ACCOUNT}>Account</Link>
    </li>
    <li>
      <Link to={ROUTES.ADMIN}>Admin</Link>
    </li>
    <li>
      <SignOutButton />
    </li>
  </ul>
);

...
```

Next, the AdminPage component's `componentDidMount()` lifecycle method in *src/components/Admin/index.js* is the perfect place to fetch users from your Firebase realtime database API:

```
import React, { Component } from 'react';

import { withFirebase } from '../Firebase';

class AdminPage extends Component {
  constructor(props) {
    super(props);

    this.state = {
      loading: false,
      users: {},
```

```
    };
  }

  componentDidMount() {
    this.setState({ loading: true });

    this.props.firebase.users().on('value', snapshot => {
      this.setState({
        users: snapshot.val(),
        loading: false,
      });
    });
  }

  render() {
    return (
      <div>
        <h1>Admin</h1>
      </div>
    );
  }
}

export default withFirebase(AdminPage);
```

We are using the users reference from our Firebase class to attach a listener. The listener is called `on()`, which receives a type and a callback function. The `on()` method registers a continuous listener that triggers every time something has changed, the `once()` method registers a listener that would be called only once. In this scenario, we are interested to keep the latest list of users though.

Since the users are objects rather than lists when they are retrieved from the Firebase database, you have to restructure them as lists (arrays), which makes it easier to display them later:

```
...

class AdminPage extends Component {
  constructor(props) {
    super(props);

    this.state = {
      loading: false,
      users: [],
    };
  }

  componentDidMount() {
    this.setState({ loading: true });

    this.props.firebase.users().on('value', snapshot => {
      const usersObject = snapshot.val();

      const usersList = Object.keys(usersObject).map(key => ({
```

```
          ...usersObject[key],
          uid: key,
        }));

        this.setState({
          users: usersList,
          loading: false,
        });
      });
  }

  ...
}

export default withFirebase(AdminPage);
```

Remember to remove the listener to avoid memory leaks from using the same reference with the `off()` method:

```
...

class AdminPage extends Component {
  ...

  componentWillUnmount() {
    this.props.firebase.users().off();
  }

  ...
}

export default withFirebase(AdminPage);
```

Render your list of users in the AdminPage component or in a child component. In this case, we are using a child component:

```
...

class AdminPage extends Component {
  ...

  render() {
    const { users, loading } = this.state;

    return (
      <div>
        <h1>Admin</h1>

        {loading && <div>Loading ...</div>}

        <UserList users={users} />
```

```
      </div>
    );
  }
}

const UserList = ({ users }) => (
  <ul>
    {users.map(user => (
      <li key={user.uid}>
        <span>
          <strong>ID:</strong> {user.uid}
        </span>
        <span>
          <strong>E-Mail:</strong> {user.email}
        </span>
        <span>
          <strong>Username:</strong> {user.username}
        </span>
      </li>
    ))}
  </ul>
);

export default withFirebase(AdminPage);
```

You have gained full control of your users now. It is possible to create and retrieve users from your realtime database. You can decide whether this is a one-time call to the Firebase realtime database, or if you want to continuously listen for updates as well.

Exercises:

- Read more about how to read and write data to Firebase's realtime database
- Confirm your source code for the last section

. . .

Everything essential is in place for Firebase authentication and Firebase realtime database for user management. I am interested in seeing what you will build on top of it! If you want to continue to follow this tutorial, get the whole book to finish this application with plenty of powerful features.

What's else will be included in the book?

- Role-based Authorization: So far, you have only authorized your application on a broad level, by checking for an authenticated user. In the book, you will learn how to assign roles to your users and how to give them additional privileges.
- User Management: In order to get more control over your users, I will show you how to merge authentication user and database user. Then you can always assign new properties to your database

user while having access to it on your user after authentication too.

- Users and Messages: Next to the user management, you will introduce a second entity for messages to your application. By using both entities, user and message, we can build a chat application.
- Read and Write Operations: In the application, you created a user and display a list of users with real-time capabilities. The book continuous with the usual delete and update operations to organize your users in the realtime database.
- Offline, Double Opt-In, Social Logins: The book adds more Firebase attributes ike offline capabilities, double opt-in sign ups, and social sign ups/ins via third-parties like Facebook or Google.
- Firebase Deployment: The final step in the book is to deploy an application with Firebase. The books walks you through the process step-by-step to see your project online.
- Firestore: Firebase's Firestore is the new Firebase Realtime Database. In the book, I may show you a way to migrate to this new tech stack. Then it is up to you whether you want to use Firestore or Firebase's Realtime Database.
- Source Code Projects: This application is only build with React and Firebase. But what about taking it on the next level to enable it for real businesses? That's where I want to show you how to migrate the project to Redux, MobX, or Gatsby.js. You will get access to variations of this application that will have additional tech when choosing the course instead of only the book:
  - Gatsby + Firebase
  - React + Redux + Firebase
  - React + MobX + Firebase
  - React + Semantic UI + Firebase
  - React + Cloud Firestore

KEEP READING ABOUT FIREBASE ›

HOW TO TEST FIREBASE WITH JEST

Every time I used Firebase, I ran into the problem of how to test Firebase's database and authentication. Since I am using Jest as my default testing environment, I figured everything I needed...

A VISUAL FIREBASE TUTORIAL

This short visual Firebase tutorial should help you to create your first Firebase application that can be used with any web framework/library such as React, Angular or Vue. For instance, you can use...