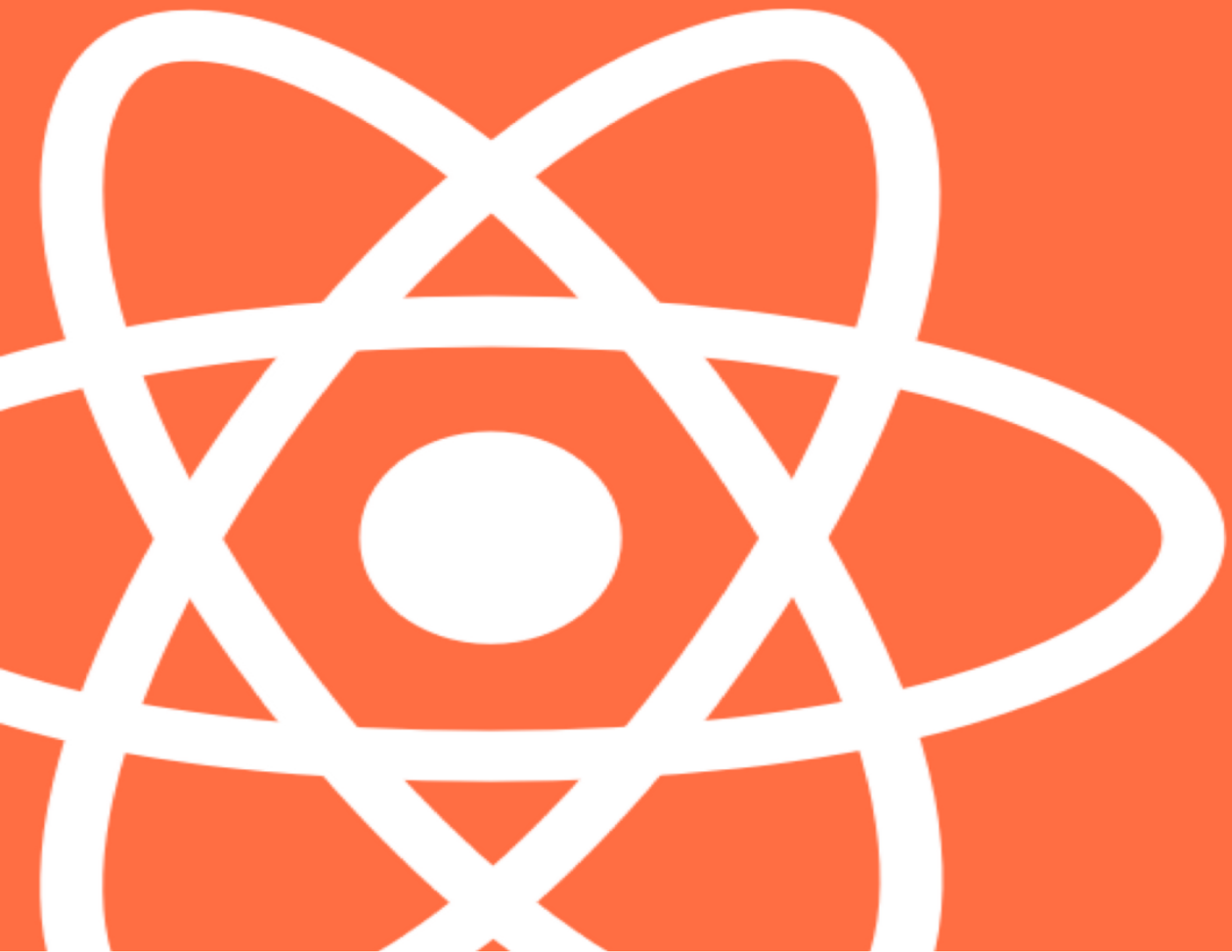
by robin wieruch

the Road to React with Firebase



The Road to React with Firebase

Your journey to master Firebase in React

Robin Wieruch

This book is for sale at http://leanpub.com/the-road-to-react-with-firebase

This version was published on 2018-06-19

Tweet This Book!

Please help Robin Wieruch by spreading the word about this book on Twitter!

The suggested hashtag for this book is #ReactJs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#ReactJs

Contents

Introduction . . . . . **i**
About the Author iii
Requirements iv
FAQ v
How to read the book? vii
Tips & Tricks vii

React Application Setup: create-react-app **1**

React Router and Routes **3**
Firebase Authentication in React 9
Firebase in React Setup 10
Firebase's Authentication API 15
Sign Up with React and Firebase 17
Sign In with React and Firebase 25
Sign Out with React and Firebase 28
Session Handling in React Components 30
Session Handling in React with Higher Order Components 33
Password Reset and Password Change with Firebase 39

Authorization in React **45**
Protected Routes in React with Authorization 46

Firebase Database in React **50**
User Management with Firebase's Database in React 51

What's next? **56**

Authentication in React, Firebase and Redux **58**
Authentication in React, Firebase and MobX 67

Outline **75**

Introduction

Note: The book is WIP and thus you may find a couple of TODO annotations along the way. Furthermore, it is a **one to one** copied version from my website[1] to make it accessible for readers in PDF, EPUB and MOBI formats. There is no formatting regarding the wording made yet (e.g. it mentions "tutorial" or "article" here, but it should be "book"; it mentions "section", but it should be "chapter"). Furthermore, there is no proper book foreword or outline in the end of the book. Nevertheless, I am curious if you like the draft of the book so far. If there is enough interest in it, and people will buy it as well, I would love to continue writing it to make it a comprehensive Firebase in React journey. Everything that follows now is the version taken from the website. If you find any potential improvements, please open up Issues or Pull Requests on GitHub[2]. If you feel any section needs more explanations, please open an Issue on GitHub too. I would very much appreciate your contribution to it.

The topic about authentication in React keeps popping up from time to time. When people approach me with this question, most often after they have learned React.js or any other SPA solution (Angular.js, Vue.js), I usually tell them to start out with Firebase. It is the simplest way to learn about the essential parts of authentication in React from a frontend perspective. You can learn about it without worrying about any backend implementations. Firebase handles it for you. By only learning about the authentication from one perspective, in the beginning, you keep the level of complexity low and thus keep yourself motivated to learn about it. Once you understand authentication from a client-perspective, you can continue to build your own authentication backend on the server-side.

Authentication can be a complex topic when learning about web development in general. The puristic frontend and backend implementation of an authentication mechanism can be quite overwhelming. How to handle the session on the client- and the server-side? If a RESTful server is stateless, where goes the session on the backend? What about cookies or the native session storage in the browser? Should Passport.js be used on the server-side? All these questions let you run in circles and you never start implementing. Therefore my advice: Take it step by step and use only Firebase in your React application in the beginning. The following tutorial gives you a complete walkthrough of how to use Firebase for authentication in React. The outcome of it can be seen in a GitHub repository[3]. It is not styled, but that's not what the tutorial is about. Instead, it implements a whole authentication flow in Firebase and React with you. The styling of the application is up to you.

In order to keep the guide updated, here is a list of the main node packages and their versions which are used in this tutorial.

- React 16

[1] https://www.robinwieruch.de/complete-firebase-authentication-react-tutorial
[2] https://github.com/rwieruch/the-road-to-react-with-firebase
[3] https://react-firebase-authentication.wieruch.com/

- React Router 4
- Firebase 4

Please help me out if the tutorial needs any updates in order to keep it reliable for other people learning about the topic as well. In general, don't hesitate to point out improvements in the comments or visit the article directly on GitHub to open issues or pull requests.

You may wonder that there is no word about MobX or Redux. Indeed, it could be used to manage the client-side state for the authenticated user. But it isn't necessary to use one of those libraries. React's local state is absolutely sufficient to handle the state for the authenticated user. I just wanted to point it out again, because **a lot of people associate authentication in React immediately with Redux or MobX**. **Yet the article will showcase it initially without using any of these state management libraries**. However, in the end, the article will show you as bonus how to upgrade your application using Redux or MobX for the session handling.

About the Author

Robin Wieruch is a german software and web engineer who is dedicated to learn and teach programming in JavaScript. After graduating from university with a masters degree in computer science, he hasn't stopped learning every day on his own. His experiences from the startup world, where he used JavaScript excessively during his professional time and spare time, gave him the opportunity to teach others about these topics.

For a few years, Robin worked closely with a great team of engineers at a company called Small Improvements[4] on a large scale application. The company builds a SaaS product enabling customers to create a feedback culture at their company. Under the hood, the application worked with JavaScript in the frontend and Java in the backend. In the frontend, the first iteration was written in Java with the Wicket Framework and jQuery. When the first generation of SPAs became popular, the company migrated to Angular 1.x for the frontend application. After using Angular for more than 2 years, it became clear that Angular wasn't the best solution to work with state intense applications back in the days. That's why the company made the final jump to React and Redux that has enabled it to operate on a large scale successfully.

During his time in the company, Robin regularly wrote articles about web development on his personal website. He noticed that people would give him great feedback on his articles that allowed him to improve his writing and teaching style. Article after article, Robin grew in his ability to teach others. Whereas the first article was packed with too much stuff that could be quite overwhelming for students, the articles improved over time by focussing and teaching only one subject.

Nowadays, Robin is self-employed to teach others. He finds it a fulfilling activity to see students thrive by giving them clear objectives and a short feedback loop. That's one thing you would learn at a feedback company, wouldn't you? But without coding himself he wouldn't be able to teach things. That's why he invests his remaining time in programming. You can find more information about Robin and ways to support and work with him on his website[5].

[4] https://www.small-improvements.com/
[5] https://www.robinwieruch.de/about

Requirements

The requirements for this tutorial are a working editor or IDE, a running command line[6], and installed recent versions of node and npm[7]. In addition, you should have learned about React in the first place. The Road to learn React[8] is a free ebook which gives you all the fundamentals of React. You will build a larger application along the way in plain React and transition smoothly 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 about those first. In addition, please follow the referenced articles in this tutorial to understand many of the underlying implementation details in case things are not explained in depth. Two of them are important to understand, but they will be linked in the appropriate sections later on.

- React's Context API[9]
- A gentle introduction to React's Higher Order Components[10]

[6] https://www.robinwieruch.de/developer-setup/
[7] https://nodejs.org/en/
[8] https://www.robinwieruch.de/the-road-to-learn-react/
[9] https://www.robinwieruch.de/react-context-api/
[10] https://www.robinwieruch.de/gentle-introduction-higher-order-components/

FAQ

How to get updates? I have two channels where I share updates about my content. Either you can subscribe to updates by email[11] or follow me on Twitter[12]. Regardless of the channel, my objective is to only share qualitative content. You will never receive any spam. Once you get the update that the book has changed, you can download the new version of it.

How to get access to the source code projects and screencasts series? If you have bought one of the extended packages that gives you access to the source code projects, screencast series or any other addon, you should find these on your course dashboard[13]. If you have bought the course somewhere else than on the official Road to React[14] course platform, you need to create an account on the platform, go to the Admin page and reach out to me with one of the email templates. Afterward I can enroll you to the course. If you haven't bought one of the extended packages, you can reach out any time to upgrade your content to access the source code projects and screencast series.

How can I get help while reading the book? The book has a Slack Group[15] for people who are reading the book. You can join the channel to get help or to help others. After all, helping others can internalize your learnings, too. If there is no one out to help you, you can always reach out to me.

Is there any troubleshoot area? If you run into problems, please join the Slack Group. In addition, you could have a look into the open issues on GitHub[16] for the book. Perhaps your problem was already mentioned and you can find the solution for it. If your problem wasn't mentioned, don't hesitate to open a new issue where you can explain your problem, maybe provide a screenshot, and some more details (e.g. book page, node version). After all, I try to ship all fixes in next editions of the book.

What should I do when I cannot afford to pay for the course? If you cannot afford the course but want to learn about the topic, you can reach out to me. It could be that you are still a student or that the course would be too expensive in your country. In addition, I want to support any cause to improve the diversity in our culture of developers. If you belong to a minority or are in an organization that supports diversity, please reach out to me.

Can I help to improve the content? Yes, I would love to hear your feedback. You can simply open an issue on GitHub[17]. These can be improvements technical wise yet also about the written word. I am no native speaker that's why any help is appreciated. You can open pull requests on the GitHub page as well.

How to support the project? If you believe in the content that I create, you can support me[18]. Furthermore, I would be grateful if you spread the word about this book after you read it and enjoyed reading it. Furthermore, I would love to have you as my Patron on Patreon[19].

[11] https://www.getrevue.co/profile/rwieruch
[12] https://twitter.com/rwieruch
[13] https://roadtoreact.com/my-courses
[14] https://roadtoreact.com
[15] https://slack-the-road-to-learn-react.wieruch.com/
[16] https://github.com/rwieruch/the-road-to-react-with-firebase/issues
[17] https://github.com/rwieruch/the-road-to-react-with-firebase
[18] https://www.robinwieruch.de/about/
[19] https://www.patreon.com/rwieruch

Is there a money back guarantee? Yes, there is 100% money back guarantee for two months if you don't think it's a good fit. Please reach out to me to get a refund.

What's your motivation behind the book? I want to teach about this topic in a consistent way. You often find material online that doesn't receive any updates or only teaches a small part of a topic. When you learn something new, people struggle to find consistent and up-to-date resources to learn from. I want to give you this consistent and up-to-date learning experience. In addition, I hope I can support minorities with my projects by giving them the content for free or by having other impacts[20]. In addition, in the recent time, I found myself fulfilled when teaching others about programming. It's a meaningful activity for me that I prefer over any other 9 to 5 job at any company. That's why I hope to pursue this path in the future.

Is there a call to action? Yes. I want you to take a moment to think about a person who would be a good match to learn about React with Firebase. The person could have shown the interest already, could be in the middle of learning React with Firebase or might not yet be aware about wanting to learn it. Reach out to that person and share the book. It would mean a lot to me. The book is intended to be given to others.

[20] https://www.robinwieruch.de/giving-back-by-learning-react/

How to read the book?

One thing is certain, no one learned programming by just reading a book. Programming is about practical experiences. It is about conquering challenges to strengthen your skills as a developer in the long term. No one learned a programming paradigm, such as functional programming or object-oriented programming, in the short term. No one learned a concept, such as state management in modern web applications, on a weekend. No one learned yet another library, such as React or Express, over one night. You will learn things only by deliberately practicing them.

In the book, I want to give you these hands on experiences and challenges to grow. The challenges are meant to create a flow experience, a scenario where the challenge meets your skills and tools at hand. Otherwise, you would feel either overwhelmed or bored. If the book accomplishes to keep this balance of challenging you and respecting your level of skill, you might experience a state of flow[21]. Personally I found this insight astonishing when I read about myself, so I hope that I can induce it in this book. It would be the perfect learning experience.

Tips & Tricks

As mentioned, there are practical tasks in the book. There, you will be guided to solve problems by using the techniques you have learned in the previous sections. It should give you the experience of applying your learnings beyond just reading this book. The book is intended to be a guide that let's you apply your learnings in practical applications in order to deepen your understanding.

In addition, make sure to **internalize each lesson learned before you continue** with the next chapter. The book is written in a way that the learnings build up on each other. Your knowledge about these topics will not only scale horizontally by using different techniques side by side but also vertically by using technique on technique. That's why it is important to internalize each learning before you continue to read.

Another hint is **making notes while you read the book**. You can write down questions where the book doesn't give you an answer and look them up afterward. Or you can write down your learnings to internalize them. That is how I do it when I read a book. Last but not least, if you write down feedback about the book, you can send me your notes afterward. I am highly interested to improve the book all the time for keeping the quality up.

Moreover I encourage you to **write the code yourself**. Even though the book shows you the source code, don't copy and paste it, but try to write it yourself. Only this way you will get a better understanding of it. When introducing bugs this way, you have to figure out yourself what went wrong in order to fix it. So it is encouraged to introduce bugs, otherwise everything goes flawless when following only the instructions in the book, but you will not know how to come up with your own solutions in the world out there after reading the book. Only this way you will grow your skills as a developer.

Last but not least, perhaps you are able to find time in between **tinkering on your own applications for applying your recent learnings** from the book. It is the best way of learning about things in the

[21] https://www.robinwieruch.de/lessons-learned-deep-work-flow/

programming world, because without ever applying the learnings yourself, you will never master them. Personally I highly recommend to come up with your own project on the side. If you have difficulties to come up with a project idea, checkout this article[22].

It should be obvious by now that you will have the best outcome of this book by having a laptop on your side. That way you can directly apply your new learnings in your editor/IDE and confront yourself with the challenges from the book. As mentioned before, no one learned something by just reading a book. You are in charge to get more than only white pages with black letters out of it. You are filling it with color in between.

[22] https://www.robinwieruch.de/how-to-learn-framework/

React Application Setup: create-react-app

You are going to implement a whole authentication mechanism in React with sign up, sign in and sign out. Furthermore, it should be possible to reset a password or change a password as a user. The latter option is only available for an authenticated user. Last, but not least, it should be possible to protect certain routes (URLs) to be accessible by authenticated users only. Therefore you will build a proper authorization solution around it.

The application will be bootstrapped with Facebook's official React boilerplate project create-react-app[23]. You can install it once globally on the command line, and make use of it whenever you want afterward.

Command Line

```
npm install -g create-react-app
```

After you have installed it, you can bootstrap your project with it on the command line. You don't need to give it the same name.

Command Line

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

Now you have the following commands on your command line to start and test your application. Unfortunately, the tutorial doesn't cover testing yet.

Command Line

```
npm start
npm test
```

You can start your application and visit it in the browser. Afterward, let us install a couple of libraries on the command line which are needed for the authentication and the routing in the first place. You will use the official firebase node package for the authentication / database and react-router-dom (React Router) to enable multiple routes for your application.

[23] https://github.com/facebookincubator/create-react-app

Command Line

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

In addition, create a *components/* folder in your application's *src/* folder on the command. That's where all your components will be implemented eventually.

Command Line

```
cd src
mkdir components
```

Next, move the App component and all its related files to the *components/* folder. That way, you will start off with a well-structured folder/file hierarchy.

Command Line

```
mv App.js components/
mv App.test.js components/
mv App.css components/
mv logo.svg components/
```

Last, but not least, fix the relative path to the App component in the *src/index.js* file. Since you have moved the App component, you need to add the */components* subpath.

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './components/App';
import registerServiceWorker from './registerServiceWorker';

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

Next, run your application on the command line again. It should work again and be accessible in the browser for you. Make yourself familiar with it if you haven't used create-react-app before.

Before implementing the authentication in React, it would be great to have a couple of pages (e.g. sign-up page, sign-in page) to split up the application into multiple URLs (routes). Therefore, let's implement the routing with React Router first before diving into Firebase and the authentication afterward.

React Router and Routes

The following application should have multiple routes for the authentication mechanism but also for the application domain itself. Therefore, you can create a file to consolidate all the routes of your application in well-defined constants. Keep it in a constants folder to add more of those files eventually.

Command Line: src/

```
mkdir constants
cd constants
touch routes.js
```

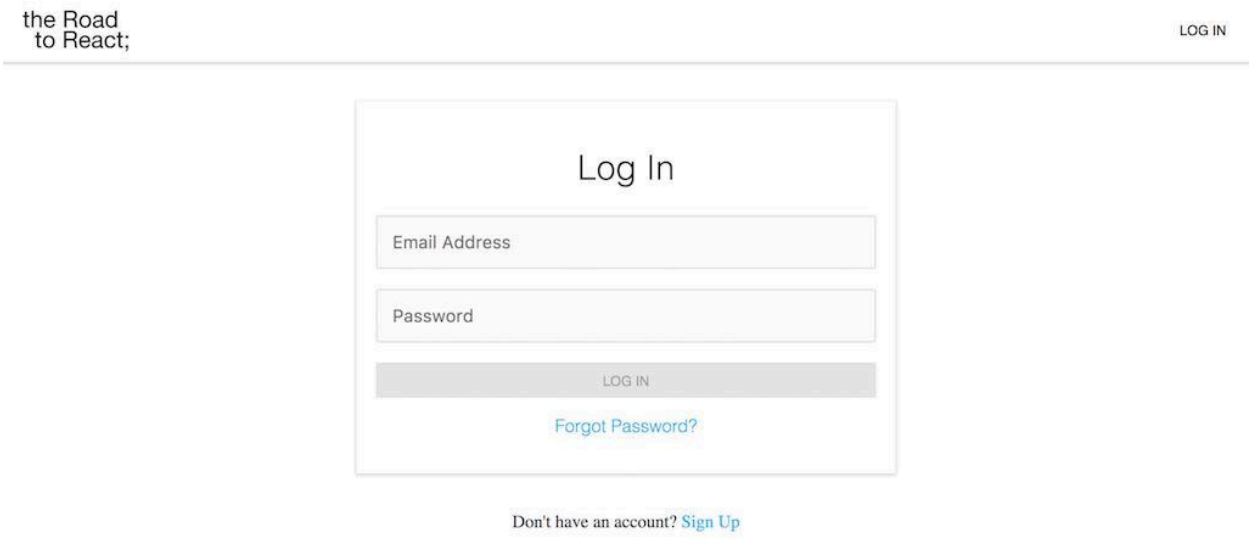
In the newly created file, define all the necessary routes for this tutorial.

src/constants/routes.js

```
export const SIGN_UP = '/signup';
export const SIGN_IN = '/signin';
export const LANDING = '/';
export const HOME = '/home';
export const ACCOUNT = '/account';
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*. Let's walk through the routes step by step.

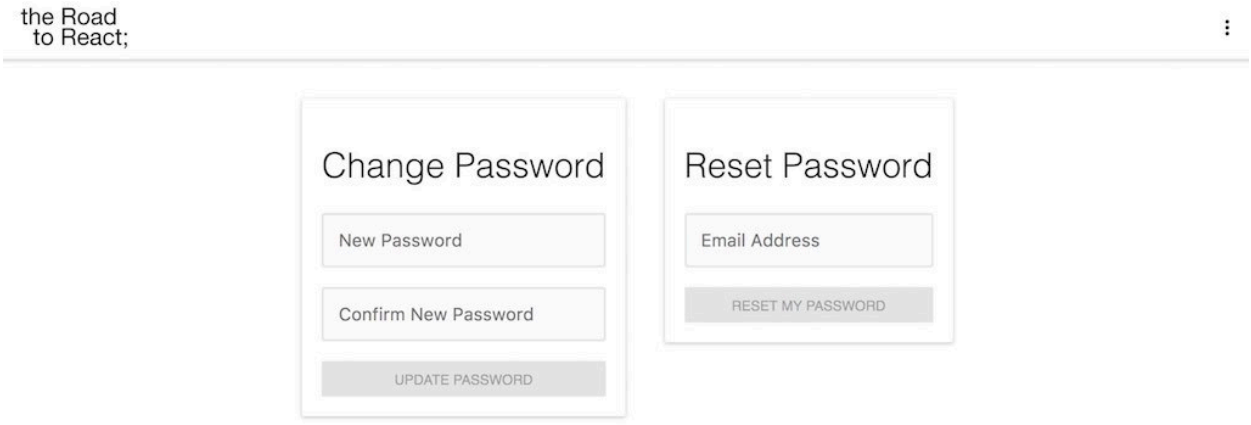
First of all, you will have a **sign-up page** and a **sign-in page**. You can take any web application out there as the blueprint to structure these routes for an authentication mechanism. For instance, take the following scenario: A user visits your web application. The user is convinced by your service and finds the button in the navigation bar 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.



Sign In

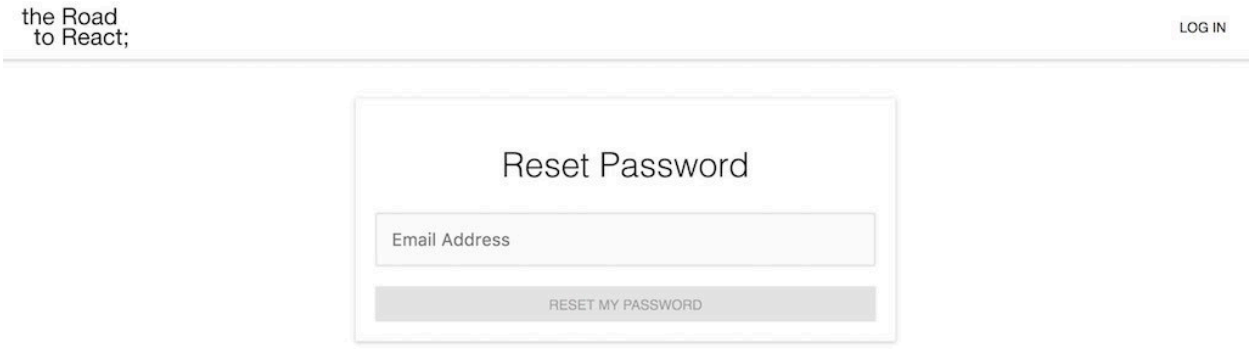
Second, there will be a **landing page** and a **home page**. The landing page is your root route. That's the place where a user ends up when visiting your web application by default. The user doesn't need to be authenticated to visit this route. On the other hand, the home page is a so-called **protected route**. The user can only access it when being authenticated. You will implement the protection of the route by using authorization in this tutorial.

Third, there will be a protected **account page** as well. On this page, a user can reset a password or change a password. It is secured by authorization as well, and thus only reachable for authenticated users.



Account Page

Last, but not least, 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.



Password Forget Page

Now, all of these routes need to be accessible to the user. How to get started with the routing in React? The best way to start is by implementing a Navigation component which is used in the App component. The App component is the perfect place to render the Navigation component, because it will always render the Navigation component but replace the other components (pages) based on the mapped route. Basically, the App component is the container where all your fixed components are going (navigation bar, side bar, footer) but also your components which are displayed depending on the route in the URL.

First, refactor your App component as follows: It will use the Navigation component and wraps it already in the Router component provided by React Router. The Router makes it possible to navigate from URL-to-URL on the client-side application, without making requests to a web server. Thus, applications need to be requested only once from the server, with the process of handling requested routes handled on the client side.

src/components/App.js

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

import Navigation from './Navigation';

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

export default App;
```

In addition, you can remove the *logo.svg* image, because it isn't used anymore. Moreover, it is up to you to keep the *App.css* file up to date, to enhance your application's appearance, as we proceed through this tutorial.

Command Line: src/components/

```
rm logo.svg
```

Second, create the Navigation file on the command line.

Command Line: src/components/

```
touch Navigation.js
```

And third, implement the Navigation component. It uses the Link component of React Router to link the application to different routes. These routes were defined previously in your constants file.

src/components/Navigation.js

```
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>
    </ul>
  </div>

export default Navigation;
```

Now, run your application again and verify two things: The links need to show up in your browser; and once you click a link, the URL has to change. However, notice that even though the URL changes, the displayed content doesn't change. Let's implement this behavior.

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.

src/components/App.js

```
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 * as routes from '../constants/routes';

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

      <hr/>

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

```
export default App;
```

So if a route matches a path, 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 change. Next, you have to create all these page components. You can do it again on the command line.

Command Line: src/components/

```
touch Landing.js Home.js Account.js SignUp.js SignIn.js SignOut.js PasswordForget.js\
 PasswordChange.js
```

In each component, define a simple boilerplate component as a functional, stateless component. That's sufficient for now. These components will be filled with business logic later on. For instance, the component for the Landing page could be defined as follows.

src/components/Landing.js

```
import React from 'react';

const LandingPage = () =>
  <div>
    <h1>Landing Page</h1>
  </div>

export default LandingPage;
```

After you have done so for the other pages, you should be able to start the application again. Now, when you click through the links in the Navigation component, the displayed page component should change accordingly to the URL. Note that the routes for the PasswordForget page, PasswordChange page and SignUp page are not used yet but will be defined somewhere else later on. For now, you have successfully implemented the larger part of the routing for this application.

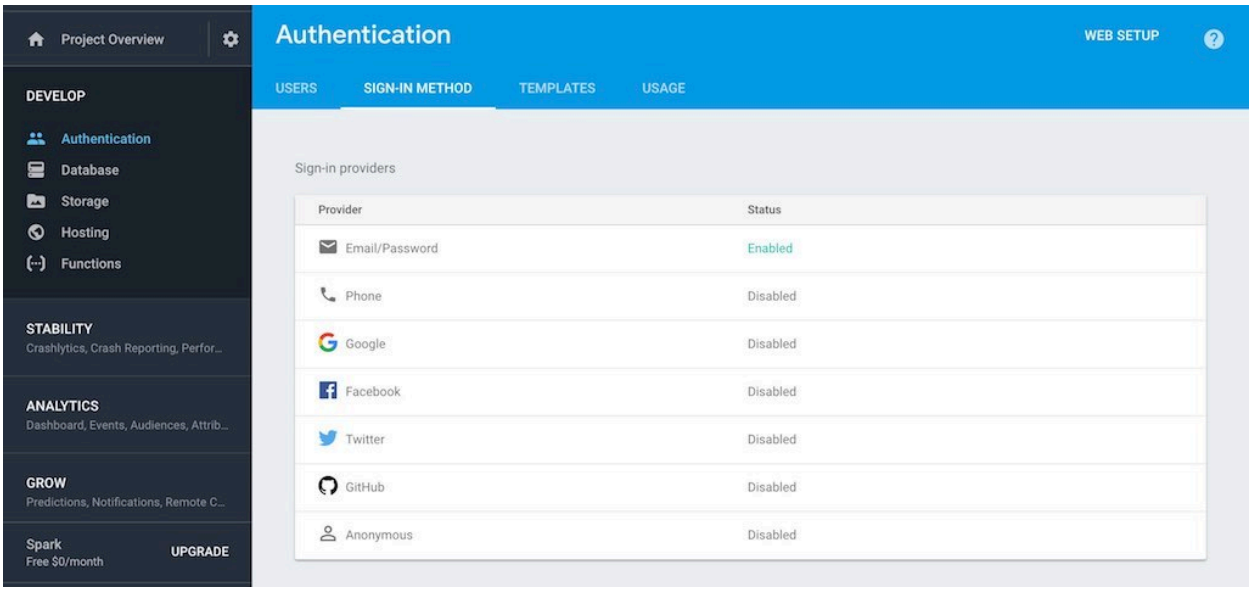
Firebase Authentication in React

TODO: Write introdcution to this larger section.

Firebase in React Setup

It's time to sign up for an account on the Firebase website[24]. After you have created an account, you should be able to create a new project that will be used for your application on their platform. You can give your project any name. Furthermore, you can run it on the free plan.

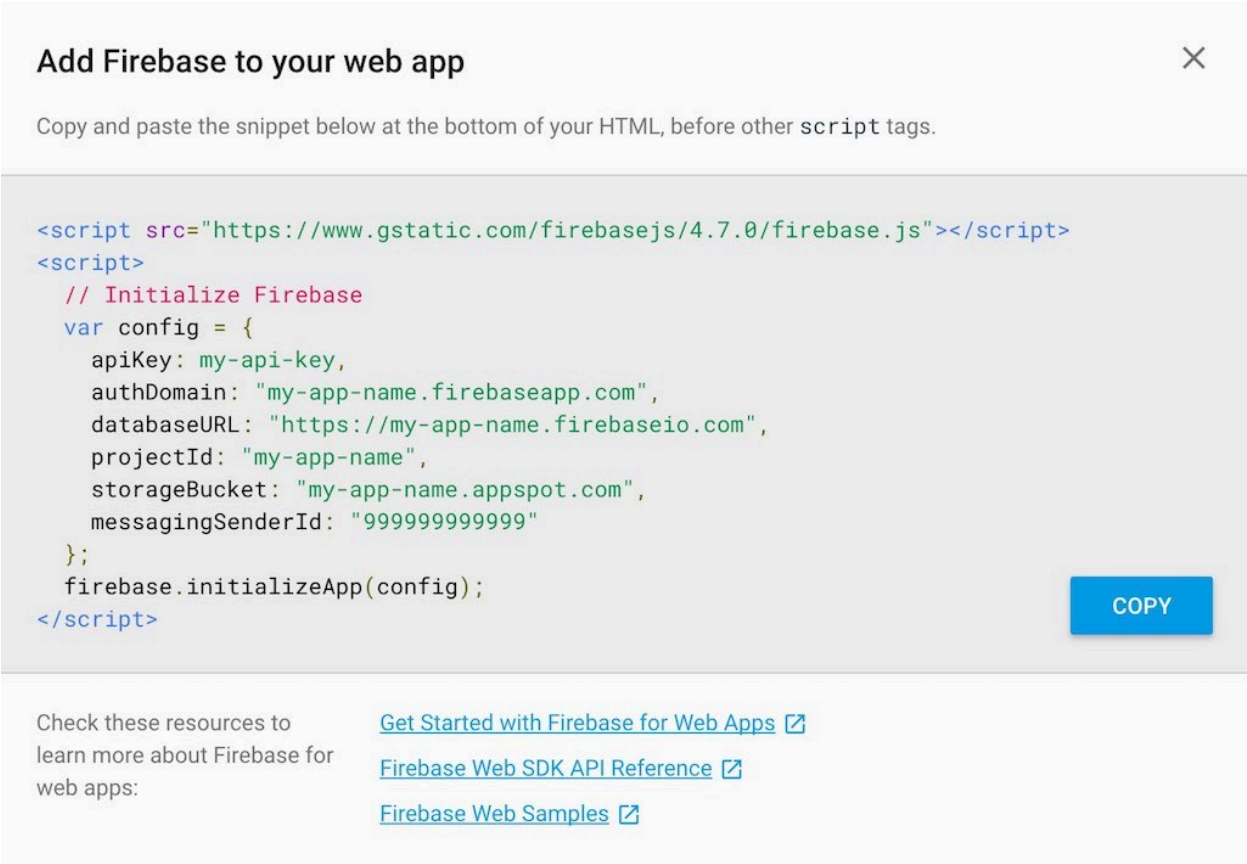
Once your project is created on their website, you should have a dashboard for it. There you can find a menu item which says "Authentication". Select it, and click the "Sign-In Method" menu item afterward. There you can enable the Email/Password authentication method. This tutorial will cover the basic authentication with email and password, but feel free to add other authentication methods later on. If you do so and open source your project, you can link it in the comments below for other readers of this guide.



Firebase Website - Authentication Methods

Next, you need to find your configuration in the project settings on your dashboard. There you have access to all the necessary information: secrets, keys, ids and other properties. You will copy these in the next step to your React application.

[24] https://firebase.google.com/



Firebase Website - Project Configuration

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. But it is somewhere there! Take it as opportunity to get to know your Firebase project dashboard while searching for this mysterious configuration.

That's it for the Firebase website setup. Now you can return to your application in your editor to add the Firebase configuration. So, create a couple of files in a new dedicated Firebase folder.

Command Line: src/

```
mkdir firebase
cd firebase
touch index.js firebase.js auth.js
```

So what's the deal with all those files? Here comes an overview from top to bottom:

- **index.js:** It's a simple entry point file to the Firebase module (*src/firebase/* folder) by grouping and exposing all the functionalities from the module to other modules in one file. Thus it shouldn't be necessary for other modules in your application to access any other file than this one to use its functionalities.

- **firebase.js**: The file where all the configuration goes that you have seen previously on your Firebase dashboard. In addition, Firebase itself will be instantiated in this file.
- **auth.js**: The file where the Firebase authentication API will be defined to sign up, sign in, sign out etc. a user in your application. It is the interface between the official Firebase API and your React application.

Let's start with the configuration. First, copy the configuration from your Firebase dashboard on their website to your application in a configuration object. Make sure to replace the capitalized keys with the corresponding keys from your own copied configuration.

src/firebase/firebase.js

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

Second, import the firebase object from the firebase node package which you have already installed in the very beginning of this tutorial. Afterward, initialize it, if it isn't already initialized, with the configuration object.

src/firebase/firebase.js

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

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

if (!firebase.apps.length) {
  firebase.initializeApp(config);
}
```

Third, initialize the auth object. That's the part of the Firebase API which will be used in the *src/firebase/auth.js* file and thus needs to be exported. In a later part of this tutorial, you will initialize the database object the same way too. But for now, the auth object will be sufficient.

src/firebase/firebase.js

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

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

if (!firebase.apps.length) {
  firebase.initializeApp(config);
}

const auth = firebase.auth();

export {
  auth,
};
```

That's it for the configuration part. There is one last optional step. On the Firebase website, you could create a second project. Afterward, your first project could be used as your development database and your second project as your production database. That way, you never mix up your data from development mode with your data from your deployed application (production mode). The step is optional.

src/firebase/firebase.js

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

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

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

```
};

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

if (!firebase.apps.length) {
  firebase.initializeApp(config);
}

const auth = firebase.auth();

export {
  auth,
};
```

Great! You have initialized Firebase in your application. In the next part, you will define the interface for the Firebase API which will be consumed by your React components.

Firebase's Authentication API

In the previous section, you have created a Firebase project on the official Firebase website and enabled the authentication with email and password. Now you will implement the authentication API. You can read up on all functionalities that are exposed by the API in the official Firebase documentation.

Initially, import the previously instantiated auth object from the Firebase configuration file.

src/firebase/auth.js

```
import { auth } from './firebase';
```

Now, let's define all the authentication functions step by step. First, the sign-up function. It takes email and password parameters in its function signature and uses an official Firebase endpoint from the firebase object to create a user.

src/firebase/auth.js

```
import { auth } from './firebase';

// Sign Up
export const doCreateUserWithEmailAndPassword = (email, password) =>
  auth.createUserWithEmailAndPassword(email, password);
```

Second, the sign-in function which takes email and password parameters, as well. It also uses a Firebase endpoint to sign in a user.

src/firebase/auth.js

```
import { auth } from './firebase';

// Sign Up
export const doCreateUserWithEmailAndPassword = (email, password) =>
  auth.createUserWithEmailAndPassword(email, password);

// Sign In
export const doSignInWithEmailAndPassword = (email, password) =>
  auth.signInWithEmailAndPassword(email, password);
```

At this point, note that these endpoints are called asynchronously. They need to be resolved later on. In addition, there needs to be error handling for it. For instance, it is not possible to sign in a user who is not signed up yet. The Firebase API would return an error. You will implement all of this in a later part of this tutorial.

Third, the sign-out function. You don't need to pass any argument to it, because the auth object itself knows about the currently authenticated user (if a user is authenticated in the first place).

src/firebase/auth.js

```
import { auth } from './firebase';

// Sign Up
export const doCreateUserWithEmailAndPassword = (email, password) =>
  auth.createUserWithEmailAndPassword(email, password);

// Sign In
export const doSignInWithEmailAndPassword = (email, password) =>
  auth.signInWithEmailAndPassword(email, password);

// Sign out
export const doSignOut = () =>
  auth.signOut();
```

Last, but not least, the two optional functions to reset and change a password of an authenticated user:

src/firebase/auth.js

```
import { auth } from './firebase';

...

// Password Reset
export const doPasswordReset = (email) =>
  auth.sendPasswordResetEmail(email);

// Password Change
export const doPasswordUpdate = (password) =>
  auth.currentUser.updatePassword(password);
```

That's it for the whole authentication interface for your React components to be connected to the Firebase API. It covers all the use cases for the purpose of this tutorial. Finally, you should expose the implemented authentication methods and the Firebase functionalities itself from your Firebase module.

src/firebase/index.js

```
import * as auth from './auth';
import * as firebase from './firebase';

export {
  auth,
  firebase,
};
```

That way, consumers (React components in our case) should be only allowed to access the *index.js* file as the sole interface to the entire Firebase module (*src/firebase/*), and should not access the auth or firebase files directly.

Sign Up with React and Firebase

In the previous sections, you have set up all the routes for your application, configured Firebase and implemented the authentication API. Now it is about time to use the authentication functionalities in your React components. Let's build the components from scratch. I try to put most of the code in one block at this point, because the components are not too small and splitting them up step by step could be too verbose. Nevertheless, I will guide you through each code block afterward. At some point, the code blocks for forms can become repetitive. Thus they will be explained once well enough in the beginning, but later in a similar version reused.

Let's start with the sign-up page. It consists of the page, a form, and a link. Whereas the form is used to sign up a new user to your application, the link will be used later on the sign-in page when a user has no account yet. It is only a redirect to the sign-up page, but not used on the sign-up page itself. Still, it shares the same domain and therefore shares the same file as the sign-up page and sign-up form.

src/components/SignUp.js

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

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

const SignUpPage = () =>
  <div>
    <h1>Sign Up</h1>
    <SignUpForm />
  </div>

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

  onSubmit = (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,
};
```

In the next step, let's focus on the SignUpForm component. It is the only React ES6 class component in the 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 terms of input fields to capture the information (email address, password, etc.) 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 two password fields for a password confirmation step. 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.

src/components/SignUp.js

```
...

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

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

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

  ...

}

...
```

Second, let's implement all the input fields to capture those information in the render method of the component. The input fields need to update the local state of the component by using their onChange handler.

src/components/SignUp.js

```
...

const byPropKey = (propertyName, value) => () => ({
  [propertyName]: value,
});

class SignUpForm extends Component {

  ...

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

    return (
      <form onSubmit={this.onSubmit}>
        <input
          value={username}
          onChange={event => this.setState(byPropKey('username', event.target.value)\
)}
          type="text"
          placeholder="Full Name"
        />
        <input
          value={email}
          onChange={event => this.setState(byPropKey('email', event.target.value))}
          type="text"
          placeholder="Email Address"
        />
        <input
          value={passwordOne}
          onChange={event => this.setState(byPropKey('passwordOne', event.target.val\
ue))}
          type="password"
          placeholder="Password"
        />
        <input
          value={passwordTwo}
          onChange={event => this.setState(byPropKey('passwordTwo', event.target.val\
ue))}
          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 (controlled components).

Let's take a look at the abstracted function which is used in the `setState()` method. It is a higher order function which takes a key value and the actual value that is typed into the input field. In the `byPropKey()` function, the key value is used as dynamic key[25] to allocate the actual value in the local state object. That's it for the whole update mechanism to handle all the input fields in the form.

In the last part of the form, there is an optional error message that is used from an error object. These error objects from Firebase have this message property by nature, so you can rely on it to display a proper text for your application user. However, the message is only shown when there is an error by using a conditional rendering[26].

One piece in the form is missing: validation. Let's use a `isInvalid` boolean to enable or disable the submit button. When you think about a validation condition for the form, what would it be? It is shown in the next code snippet.

src/components/SignUp.js

```
...

class SignUpForm extends Component {

  ...

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

    const isInvalid =
      passwordOne !== passwordTwo ||
```

[25] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer
[26] https://www.robinwieruch.de/conditional-rendering-react/

```
      passwordOne === '' ||
      email === '' ||
      username === '';

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

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

...
```

Of course, the user is only allowed to sign up if both passwords are the same (part of the password confirmation in a common sign up process) and if the username, email and at least one password are filled with a string. 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. Furthermore, you should be able to type into it (confirmation that the local state updates are working) and you should be 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 previously set authentication interface in the firebase folder.

src/components/SignUp.js

```
...
import { auth } from '../firebase';
import * as routes from '../constants/routes';

...

class SignUpForm extends Component {

  ...

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

    auth.doCreateUserWithEmailAndPassword(email, passwordOne)
```

```
      .then(authUser => {
        this.setState(() => ({ ...INITIAL_STATE }));
      })
      .catch(error => {
        this.setState(byPropKey('error', error));
      });

    event.preventDefault();
  }

  ...
}

...
```

Let's break down what happens in the previous code block. All the necessary information that is 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 anyway.

Next, you can call the sign up function which you have defined in the previous section. It takes the email and the password properties. The username is not used yet, but will be in a later section of this tutorial. It's up to you to remove it for now as well if you want to get rid of the noisy "unused variable" warning on the command line.

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. Thus an error message should show up in the form.

In addition, 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. Moreover, 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 on in this tutorial.

If you want to read more about asynchronous requests in React components, checkout this article which explains the concept in more detail by fetching data from a third-party API[27].

So what's next? When a user signs up to your application, you want to redirect the user to another page. Perhaps it should be the Home page of the user which will be a protected route for only authenticated users at some point of this tutorial. Therefore, you will need the help of React Router to redirect the user after a successful sign up.

[27] https://www.robinwieruch.de/react-fetching-data/

src/components/SignUp.js

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

import { auth } from '../firebase';
import * as routes from '../constants/routes';

const SignUpPage = ({ history }) =>
  <div>
    <h1>SignUp</h1>
    <SignUpForm history={history} />
  </div>

...

class SignUpForm extends Component {

  ...

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

    const {
      history,
    } = this.props;

    auth.doCreateUserWithEmailAndPassword(email, passwordOne)
      .then(authUser => {
        this.setState(() => ({ ...INITIAL_STATE }));
        history.push(routes.HOME);
      })
      .catch(error => {
        this.setState(byPropKey('error', error));
      });

    event.preventDefault();
  }

  ...
}

...

export default withRouter(SignUpPage);
```

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

Let's take the previous code block apart again. In order to redirect a user to another page programmatically, we need access to React Router. Somehow it needs to be possible to tell React Router that it should redirect the user to another page. Fortunately, the React Router node package offers a neat higher order component](https://www.robinwieruch.de/gentle-introduction-higher-order-components/) to make the router properties accessible in the props of a component. Any component which goes in the `withRouter()` higher order component gets access to all the properties of the router. Thus when passing the SignUpPage component to the `withRouter()` higher order component, it has in its own function signature access to the props of the React Router. The relevant property for us from the router props is the `history` object. That's the part which enables us to redirect a user to another page. It can be used to push routes to it for these redirects. That's why the history is passed down to the SignUpForm component.

Just in case: There is no particular reason why I wrapped the SignUpPage and not the SignUpForm with the higher order component.

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

You can run your application now and checkout if the sign up process works for you. If you signed up a user successfully, you should be redirected 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 the following or 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.

Sign In with React and Firebase

A sign up automatically results into a sign in of a user. That's something you will notice later on eventually. However, of course, we cannot rely on it, because a user could already be signed up but not signed in to your application.

Let's implement the sign in now. It is similar to the sign up mechanism and components, so this time I will not split up any code blocks. It's straight forward if you went through the previous sign up form.

src/components/SignIn.js

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

import { SignUpLink } from './SignUp';
import { auth } from '../firebase';
import * as routes from '../constants/routes';

const SignInPage = ({ history }) =>
  <div>
    <h1>SignIn</h1>
    <SignInForm history={history} />
    <SignUpLink />
  </div>

const byPropKey = (propertyName, value) => () => ({
  [propertyName]: value,
});

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

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

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

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

    const {
      history,
```

```
    } = this.props;

    auth.doSignInWithEmailAndPassword(email, password)
      .then(() => {
        this.setState(() => ({ ...INITIAL_STATE }));
        history.push(routes.HOME);
      })
      .catch(error => {
        this.setState(byPropKey('error', error));
      });

    event.preventDefault();
  }

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

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

    return (
      <form onSubmit={this.onSubmit}>
        <input
          value={email}
          onChange={event => this.setState(byPropKey('email', event.target.value))}
          type="text"
          placeholder="Email Address"
        />
        <input
          value={password}
          onChange={event => this.setState(byPropKey('password', event.target.value)\
)}
          type="password"
          placeholder="Password"
        />
        <button disabled={isInvalid} type="submit">
          Sign In
        </button>

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

export default withRouter(SignInPage);
```

```
export {
  SignInForm,
};
```

Basically, it is almost the same as the sign up form. The form with its input fields captures all the necessary information such as username and password. A validation step makes sure that email and password are set before performing the request by enabling or disabling the submit button. The authentication API is used again, but this time with the function to sign in a user rather than the function to sign up a user. If the 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 messages should show up. There is one difference though: The SignUpLink which you have defined in the previous section is used on the sign in page. It gives the user an alternative way in case the person isn't signed up yet but happens to be on the sign in page.

Sign Out with React and Firebase

In order to complete the authentication loop, let's implement as last part the sign out component. It doesn't need any form and is only a button which shows up in the Navigation component in the next steps. Since we can use the already defined authentication API to sign out a user, it is fairly straight forward to pass the functionality to a button in a React component.

src/components/SignOut.js

```
import React from 'react';

import { auth } from '../firebase';

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

export default SignOutButton;
```

The button can be used in the Navigation component now.

src/components/Navigation.js

```
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><SignOutButton /></li>
    </ul>
  </div>

export default Navigation;
```

From a component perspective of the application, everything is in place to fulfil a full authentication roundtrip now. A user is able to sign up, sign in and sign out. However, when you open the application, something feels odd. For instance, the "Sign Out" button should only show up when a

user is signed in. If the button is pressed, it should disappear. The simplest solution for this problem is to use a conditional rendering in the Navigation component. Based on an authenticated user object, the navigation changes its options. It has either all the options for an authenticated user or all the options for non authenticated users.

src/components/Navigation.js

```
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;
```

That way, the user only gets presented the options according to the state of the authentication. You may have noticed that the Navigation component has access to the authenticated user in its props. But where does this object come from? You will explore this in the next section!

Session Handling in React Components

The following section is the most crucial part to the authentication process. You have all components in place to fulfil the authentication roundtrip in React components, but no one knows about any session state yet. The question is: "Is there a currently signed in user?" There is no logic yet about an authenticated user. The information needs to be stored somewhere to make it accessible to other components (e.g. Navigation component).

Often that's the point where people start to use a state management library such as Redux or MobX[28]. But since our whole application is grouped under the umbrella App component, it's sufficient to manage the session state in the App component by using React's local state. It only needs to keep track of an authenticated user. 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 it down as `null`. That way, all components interested in it can use a conditional rendering to adjust their behavior based on the session state. For instance, the Navigation component from the previous section is interested in it, because it has to show different options to authenticated and non authenticated users.

Let's start to implement the session handling in the App component. Because the component handles local state now, you have to refactor it to an ES6 class component. It manages the local state of a `authUser` object, where we don't know yet where it comes from, and passes it to the Navigation component.

src/components/App.js

```
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} />
```

[28] https://www.robinwieruch.de/redux-mobx-confusion/

```
          <hr/>

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

export default App;
```

In the last section, you already made the Navigation component aware of the authenticated user to display different options depending on it. The crucial part comes now. Firebase offers a neat helper function which can be initialized in the `componentDidMount()` lifecycle method of the App component. It can be used as a listener for the authenticated user which is our implicit session state.

src/components/App.js

```
...

import * as routes from '../constants/routes';
import { firebase } from '../firebase';

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

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

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

  ...

}

export default App;
```

The helper function `onAuthStateChanged()` gets a function as input and this function has access to the authenticated user object. In addition, this passed function is called **every time** something changed for the authenticated user. It is called when a user signs up (because it results in a sign

in), signs in and signs out. If a user signs out, the `authUser` object becomes null. Thus the `authUser` property in the local state is set to null as well and as reaction components depending on it can display different options (e.g. Navigation component).

Start your application again and verify that your sign up, sign in and sign out works properly and that the Navigation components displays the options depending on the session state accordingly.

That's it! You have successfully implemented the authentication flow with Firebase in React. Everything that comes in the following sections is extra implementation sugar on top and a couple of neat features along the way. But you wouldn't need those things to continue with your own implementation.

The recent sections were quite a lot of content. I didn't go into all the details, because I teach those in the referenced articles and The Road to learn React[29]. So make sure to check out the ebook!

[29] https://www.robinwieruch.de/the-road-to-learn-react/

Session Handling in React with Higher Order Components

In this section, we will abstract the session handling away with higher order components and React's context API. It has two advantages:

- The higher order component fulfils only one purpose. It shields away the business logic from the App component. Thus the App component stays lightweight. There is no business logic mixed up in the component anymore. Higher order components are a great concept in React to extract logic from components, but you can use them later on to enhance components with it. Therefore, they are a great way to accomplish reusability, composability and often maintainability in React.
- React's context API[30] is a React concept which helps us to pass around properties in our application. Rather than passing props explicitly down to all components who are interested in them, you can pass these props implicitly down to these components without bothering the components in between of the hierarchy. Thus, in our case, the App component doesn't need to bother about the authenticated user object anymore, because it only passes it down to various other components.

First, you can revert the recent changes in the App component. It can become a functional stateless component again. It doesn't need to know about the authenticated user object anymore.

src/components/App.js

```
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 * as routes from '../constants/routes';

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

      <hr/>
```

[30] https://www.robinwieruch.de/react-context-api/

```
      <Route exact path={routes.LANDING} component={() => <LandingPage />} />
      <Route exact path={routes.SIGN_UP} component={() => <SignUpPage />} />
      <Route exact path={routes.SIGN_IN} component={() => <SignInPage />} />
      <Route exact path={routes.PASSWORD_FORGET} component={() => <PasswordForgetPag\
e />} />
      <Route exact path={routes.HOME} component={() => <HomePage />} />
      <Route exact path={routes.ACCOUNT} component={() => <AccountPage />} />
    </div>
  </Router>

export default App;
```

Next, you can wrap the App component up in a session handling higher order component. That's where all the business logic goes which you have just removed in the last step from the App component. Basically you enhance the App component with the higher order component.

src/components/App.js

```
...

import withAuthentication from './withAuthentication';

const App = () =>
  ...

...

export default withAuthentication(App);
```

That's how the higher order component makes its session handling logic available to the App component. We didn't implement the higher order component yet. First, you have to create a file for it on the command line.

Command Line: src/components/

```
touch withAuthentication.js
```

Second, implement the framework around the higher order component. Basically most of the higher order components start out like this.

src/components/withAuthentication.js

```
import React from 'react';

const withAuthentication = (Component) => {
  class WithAuthentication extends React.Component {
    render() {
      return (
        <Component />
      );
    }
  }

  return WithAuthentication;
}

export default withAuthentication;
```

Third, move all the previous session logic into the higher order component.

src/components/withAuthentication.js

```
import React from 'react';

import { firebase } from '../firebase';

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

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

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

    render() {
      return (
        <Component />
      );
    }
  }

export default withAuthentication;
```

If you are not familiar with higher order components, make sure to read this gentle introduction[31]. It gives you an approachable way to learn about them.

Fourth, there needs to be a mechanism to pass down the authenticated user object to the other components (e.g. Navigation component). As mentioned, we will use React's context API for it. Adjust your session handling higher order component to the following.

src/components/withAuthentication.js

```
import React from 'react';

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

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

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

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

    render() {
      const { authUser } = this.state;

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

export default withAuthentication;
```

As you may have noticed, there is some kind of mysterious Provider component coming from an imported `AuthUserContext` object. You will see in the next step where this object comes from and how it is implemented. For the previous code snippet, it is important to know that this Provider component can make its value accessible to all the components below. So if you would pass the App

[31] https://www.robinwieruch.de/gentle-introduction-higher-order-components/

component this higher order component, then the App component and all components below of it would have access to the value. Since the value is the authenticated user or null, all components below can act accordingly to it.

So where does the AuthUserContext come from? Let's implement it in a new file:

Command Line: src/components/

```
touch AuthUserContext.js
```

Now, you can put in the following content to create the context object by using React's context API. You have access to the API by using the `createContext()` function. You can pass an initial value argument to the function, but since the authenticated user should be null in the beginning, passing null to it is just fine.

src/components/AuthUserContext.js

```
import React from 'react';

const AuthUserContext = React.createContext(null);

export default AuthUserContext;
```

That's it. You have implemented a higher order component which got all the business logic for the authenticated user which was in the App component before. The higher order component is used already to enhance the App component with this same functionality. In addition, you have used React's context API to create a context object which exposes a Provider component for you. Once you use this Provider component, the value which you have passed to it becomes available to all components below the App component where the higher order component is used.

Last but not least, the only consumer of the authenticated user object so far, the Navigation component, needs to use the same `AuthUserContext` object to access the value of the context which you have passed to the Provider component. You can import the context object again, but this time use its Consumer component to make the value, which is the authenticated user, available to the Navigation component.

src/components/Navigation.js

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

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

const Navigation = () =>
  <AuthUserContext.Consumer>
    {authUser => authUser
      ? <NavigationAuth />
```

```
      : <NavigationNonAuth />
    }
  </AuthUserContext.Consumer>

...

export default Navigation;
```

Inside of the Consumer component, you are using a function instead of other components. That's called the render props pattern in React. If you are not familar with it, you can read up more about it over {{% a_blank "here" "https://reactjs.org/docs/render-props.html" %}}. What's important that it gives you access to the value which was passed before to the Provider pattern. Once the authenticated user in the `withAuthentication` higher order component changes, it changes as well as the passed value in the Provider component, and then also in the Consumer component. Notice that you don't need to pass the authenticated user down from the App component anymore. It is passed through it implicitly by using React's context.

Now, start your application again and verify that it still works the same as before. You didn't change any behavior of your application in this section, but only shielded away the more complex logic into a higher order component and added the convenience of passing the authenticated user implicitly via React's context rather than explicitly through the whole component tree by using props. These are two advanced patterns in React and you have used both in this last section.

Password Reset and Password Change with Firebase

There are two more neat features available in the Firebase authentication API and you have already implemented the interface for it in your Firebase module: password forget / reset and password change.

Let's start by implementing the password forget feature. Since you have already implemented the interface in your Firebase module, you can make use of it in a component. The following file implements the vast majority of the password reset logic in a form again. We already implemented a couple of those forms before, so it shouldn't be any different now.

src/components/PasswordForget.js

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

import { auth } from '../firebase';

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

const byPropKey = (propertyName, value) => () => ({
  [propertyName]: value,
});

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

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

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

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

    auth.doPasswordReset(email)
      .then(() => {
        this.setState(() => ({ ...INITIAL_STATE }));
      })
      .catch(error => {
        this.setState(byPropKey('error', error));
      });
```

```
    event.preventDefault();
  }

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

    const isInvalid = email === '';

    return (
      <form onSubmit={this.onSubmit}>
        <input
          value={this.state.email}
          onChange={event => this.setState(byPropKey('email', event.target.value))}
          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="/pw-forget">Forgot Password?</Link>
  </p>

export default PasswordForgetPage;

export {
  PasswordForgetForm,
  PasswordForgetLink,
};
```

Again it's a lot of code. But it isn't any different from the sign up and sign in forms from before. The password forget uses a form to submit the information (only email address) which is needed by the Firebase authentication API to reset the password. A class method (onSubmit) makes sure that the information is sent to the API. Furthermore, it resets the form's input field on a successful request or shows an error on an erroneous request. In addition, the form is validated as well before it can be submitted.

Moreover, 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 which was used on in the SignInPage

component. This link is not different. You can already make use of it. In case a user forgot about the password during the sign in process, the password forget page isn't far away by using the link.

src/components/SignIn.js

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

import { SignUpLink } from './SignUp';
import { PasswordForgetLink } from './PasswordForget';
import { auth } from '../firebase';
import * as routes from '../constants/routes';

const SignInPage = ({ history }) =>
  <div>
    <h1>SignIn</h1>
    <SignInForm history={history} />
    <PasswordForgetLink />
    <SignUpLink />
  </div>

...
```

Remember that the password forget page is already mapped in the App component. So you can simply drop the PasswordForgetLink component in the sign in page.

You can try it out yourself now. 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.

Now let's get to the second component: the password change component. You have implemented this functionality already in your Firebase interface as well. You only need a form component to make use of it. Again, the form component isn't any different from the sign in, sign up and password forget forms. Once you have learned about how to implement a form in React, the other forms are pretty similar to it.

src/components/PasswordChange.js

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

import { auth } from '../firebase';

const byPropKey = (propertyName, value) => () => ({
  [propertyName]: value,
});

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;

    auth.doPasswordUpdate(passwordOne)
      .then(() => {
        this.setState(() => ({ ...INITIAL_STATE }));
      })
      .catch(error => {
        this.setState(byPropKey('error', error));
      });

    event.preventDefault();
  }

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

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

    return (
      <form onSubmit={this.onSubmit}>
        <input
          value={passwordOne}
          onChange={event => this.setState(byPropKey('passwordOne', event.target.val\
ue))}
          type="password"
          placeholder="New Password"
        />
        <input
          value={passwordTwo}
          onChange={event => this.setState(byPropKey('passwordTwo', event.target.val\
ue))}
          type="password"
          placeholder="Confirm New Password"
        />
        <button disabled={isInvalid} type="submit">
          Reset My Password
```

```
          </button>

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

export default PasswordChangeForm;
```

The component updates its local state by using `onChange` handlers in the input fields. Furthermore, it validates the state before submitting a request to change the password by enabling or disabling the submit button. Last but not least, it shows again an error message when a request fails.

Now there is another neat implementation of your application. The password change component isn't reachable yet. On the other hand, it's great that the password forget form is reachable from the sign in page. But what about a central place to make those functionalities available for an authenticated user? An account page would be the perfect place. You have already created a file for such a page on the command line and mapped the route in the App component. Now you only need to implement it.

src/components/Account.js

```
import React from 'react';

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

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

export default AccountPage;
```

The AccountPage component isn't complicated and doesn't have any business logic. It merely uses the password forget and password change forms in a central place. In addition, it gets access to the authenticated user object via React's context by using the Consumer component again. It's identical to the Navigation component. So as you can see, you can use multiple Consumer components for the same context, but you have to ensure to have one Provider component somewhere above in your component hierarchy.

That's it. Your user experience has improved significantly with the password forget and password change features. Users who have trouble with their password can use these features now.

Authorization in React

TODO: Write introdcution to this larger section.

Protected Routes in React with Authorization

When you sign out from the home or account page, you will not get any redirect even though these pages should be only accessible for authenticated users. However, it makes no sense to show a non authenticated user the account page. In this section, you will implement a protection for these routes in case a user signs out. This process is called authorization.

The protection you are going to implement is a form of a **general authorization** for your application. It checks whether there is an authenticated user. If there isn't an authenticated user, it will redirect to a public route. Otherwise, it will do nothing. The condition could be defined as simple as: `authUser != null`. In contrast, a more **specific authorization** could be role or permission based authorization. For instance, `authUser.role === 'ADMIN'` would be a role based authorization and `authUser.permission.canEditAccount === true` would be a permission based authorization. Fortunately, we will implement it in a way that you can define the authorization condition (predicate) in a flexible way so that you have full control over it in the long run.

Similar to the higher order `withAuthentication` component, there will be a higher order `withAuthorization` component to shield away from the logic. It is not used on the App component, but can be used on all components which are associated with a route in the App component. Thus it can be reused for the HomePage and AccountPage components. What's the task of the higher order component? First of all, it gets the condition passed as configurational parameter. That way, you can decide on your own if it should be a general or specific authorization rule. Second, it has to decide based on the condition whether it should redirect to a public page because the user isn't authorized to view the current page.

Let's start to implement the higher order component. First, create it on the command line.

Command Line: src/components/

```
touch withAuthorization.js
```

Second, you have again the common framework for the higher order component.

src/components/withAuthorization.js

```
import React from 'react';

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

  return WithAuthorization;
}

export default withAuthorization;
```

Third, let me paste the whole implementation details for the higher order component and explain it afterward.

src/components/withAuthorization.js

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

import AuthUserContext from './AuthUserContext';
import { firebase } from '../firebase';
import * as routes from '../constants/routes';

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

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

  return withRouter(WithAuthorization);
}

export default withAuthorization;
```

Let's break it down. First, have a look at the render method. It renders either the passed component (e.g. HomePage, AccountPage) or nothing. That's just a fallback in case there is no authenticated user passed by the Consumer component from React's context object. It increases the protection of the component by rendering simply nothing. However, the real logic happens in the `componentDidMount()` lifecycle method. Same as the `withAuthentication()` higher order component, it uses the Firebase listener to trigger a callback function in case the authenticated user object changes. Every time the `authUser` changes, it checks the passed `authCondition()` function with the `authUser`. If the authorization fails, the higher order component redirects to the sign in page. If it doesn't fail, the higher order component does nothing. In order to be able to redirect, the higher order component has access to the history object of the Router by using the in-house `withRouter()` higher order component from the React Router library.

In the next step, you can use the higher order component to protect your routes (e.g. /home and

/account) with authorization rules. For the sake of keeping it simple, the following two components are only protected with a general authorization rule that checks if the `authUser` object is not null.

First, wrap the HomePage component in the higher order component and define the authorization condition for it. As mentioned, it checks if the user object is not null.

src/components/Home.js

```
import React from 'react';

import withAuthorization from './withAuthorization';

const HomePage = () =>
  <div>
    <h1>Home Page</h1>
    <p>The Home Page is accessible by every signed in user.</p>
  </div>

const authCondition = (authUser) => !!authUser;

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

Second, wrap the AccountPage component in the higher order component and define the authorization condition as well. It isn't any different from the previous usage of the higher order component.

src/components/Account.js

```
import React from 'react';

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

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

const authCondition = (authUser) => !!authUser;

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

That's it, your routes are protected now. You can try it yourself by signing out from your application and trying to access */account* or */home*. It should redirect you.

I guess you can imagine how this technique gives you full control over your authorizations in your application. Not only by using general authorization rules, but by being more specific with role and permission based authorizations. For instance, an admin page, which is only available for users with the admin role, could be protected as follows.

Code Playground

```
import React from 'react';

import AuthUserContext from './AuthUserContext';

const AdminPage = () =>
  <AuthUserContext.Consumer>
    {authUser =>
      <div>
        <h1>Admin</h1>
        <p>Restricted area! Only users with the admin rule are authorized.</p>
      </div>
    }
  </AuthUserContext.Consumer>

const authCondition = (authUser) => authUser.role === 'ADMIN';

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

Congratulations! You 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. In the next section, you will explore how you can manage the users who sign up in your application. So far, only Firebase knows about them but you have no own database running to store them yourself.

Firebase Database in React

TODO: Write introdcution to this larger section.

User Management with Firebase's Database in React

So far, only Firebase knows about your users. There is no way to retrieve the single user or a list of users for you. They are stored internally by Firebase to keep the authentication secured. That's good, because you are never involved in storing sensible data such as passwords. However, you can introduce the Firebase realtime database to keep track of the user entities yourself. And you should do it, because otherwise you never have a way to associate other domain entities (e.g. a todo item) created by your users to your users. Therefore, follow this section to store users in your realtime database in Firebase during the sign up process.

First of all, create a file for your Firebase realtime database API. It goes into the firebase folder next to your file for the authentication API.

Command Line: src/firebase/

```
touch db.js
```

In the file, you will implement the interface to your Firebase realtime database for the user entity. The file defines two functions: one to create a user and one to retrieve all users.

src/firebase/db.js

```
import { db } from './firebase';

// User API

export const doCreateUser = (id, username, email) =>
  db.ref(`users/${id}`).set({
    username,
    email,
  });

export const onceGetUsers = () =>
  db.ref('users').once('value');

// Other Entity APIs ...
```

In the first asynchronous function, the user is created as an object with the username and email properties. Furthermore, it is stored on the `users/${id}` resource path. So whenever you would want to retrieve a specific user from the Firebase database, you could get the one user via its unique identifier and the entity resource path.

In the second asynchronous function, the users are retrieved from the general user's entity resource path. The function will return all users from the Firebase realtime database.

Note: In the future, you could consider to split up the file into multiple files for different domain entities (e.g. *db/user.js*, *db/todos.js*, ...) in one module (e.g. *db/* folder).

You might have noticed that the file imports a database object from the *src/firebase/firebase.js* file which isn't defined nor declared yet. Let's create it now.

src/firebase/firebase.js

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

...

const db = firebase.database();
const auth = firebase.auth();

export {
  db,
  auth,
};
```

Here you can see again how Firebase separates its authentication and database API. You followed the same best practice in both *db.js* and *auth.js* files. Last but not least, don't forget to make the new functionalities from your Firebase database API available in your Firebase module's entry point.

src/firebase/index.js

```
import * as auth from './auth';
import * as db from './db';
import * as firebase from './firebase';

export {
  auth,
  db,
  firebase,
};
```

Finally, you can use those functionalities in your React components to create and retrieve users. Let's start with the user creation. The best place to do it would be the SignUpForm component. It is the most natural place to create a user in the database after the sign up through the Firebase authentication API. You can simply add another API request to create a user when the user signed up successfully.

src/components/SignUp.js

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

import { auth, db } from '../firebase';
import * as routes from '../constants/routes';

...
```

```
class SignUpForm extends Component {

  ...

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

    const {
      history,
    } = this.props;

    auth.doCreateUserWithEmailAndPassword(email, passwordOne)
      .then(authUser => {

        // Create a user in your own accessible Firebase Database too
        db.doCreateUser(authUser.user.uid, username, email)
          .then(() => {
            this.setState(() => ({ ...INITIAL_STATE }));
            history.push(routes.HOME);
          })
          .catch(error => {
            this.setState(byPropKey('error', error));
          });

      })
      .catch(error => {
        this.setState(byPropKey('error', error));
      });

    event.preventDefault();
  }

  ...
}
```

Notice how all the previous business logic from the first then block moves into the second then block. The previous logic is only called after the second asynchronous API call resolves successfully. In addition, see how the `authUser` object's `uid` and the `username` property from the local state can be used to pass the necessary parameters to your Firebase database API.

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 sensible data of the user on your own. Firebase already deals with the authentication and thus there is no need to store the password in your database. There are a bunch of steps necessary to secure such sensible data (e.g. encryption). It would be a security risk to do it on your own, so don't do it if someone else already handles it for you.

That's it for the user creation process. Now you are creating a user once a user signs up in your application. By now it is a lot of business logic in your component's lifecycle method and you could consider extracting the logic on your own to a separate place to keep the component lightweight. If you are going to add tests for your application after the tutorial, it is simpler to test the logic when it is extracted.

Next, only to verify that the user creation is working, you can retrieve all the users from the database in one of your other components. Since the HomePage component isn't of any use yet, you can do in this component to display your users stored in the realtime database of Firebase. The `componentDidMount()` lifecycle method is the perfect place to fetch users from your database API.

src/components/Home.js

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

import withAuthorization from './withAuthorization';
import { db } from '../firebase';

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

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

  componentDidMount() {
    db.onceGetUsers().then(snapshot =>
      this.setState(() => ({ users: snapshot.val() }))
    );
  }

  render() {
    return (
      <div>
        <h1>Home</h1>
        <p>The Home Page is accessible by every signed in user.</p>
      </div>
    );
  }
}

const authCondition = (authUser) => !!authUser;

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

Afterward, you can display a couple of properties of your list of users. Since the users are an object and not a list when they are retrieved from the Firebase database, you have to use the `Object.keys()` helper function to map over the keys in order to display them.

src/components/Home.js

```
...

class HomePage extends Component {
  ...

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

    return (
      <div>
        <h1>Home</h1>
        <p>The Home Page is accessible by every signed in user.</p>

        { !!users && <UserList users={users} /> }
      </div>
    );
  }
}

const UserList = ({ users }) =>
  <div>
    <h2>List of Usernames of Users</h2>
    <p>(Saved on Sign Up in Firebase Database)</p>

    {Object.keys(users).map(key =>
      <div key={key}>{users[key].username}</div>
    )}
  </div>

const authCondition = (authUser) => !!authUser;

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

That's it for the user entity management. You are in full control of your users now. It is possible to retrieve a user entity or a list of user entities. Furthermore, you can create a user in the realtime database. It is up to you to implement the other CRUD operations[32] as well in order to update a user, to remove a user and to get a single user entity from the database.

[32] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

What's next?

By now, everything is in place in terms of authentication and user management for your application. You could continue to implement your own domain logic. I am keen to see what you are implementing on top of this authentication boilerplate for Firebase in React, so don't hesitate to reach out to me.

You have started to implement your first entity which is managed by the Firebase realtime database. You retrieve a list of these entities from the Firebase database and created single entities in the database too. You can extend it with further CRUD operations. In addition, you can start to implement your own domain specific entities (e.g. todo items in a ToDo application).

There is one caveat you should keep in mind. In the very beginning of this tutorial, it says that you will learn everything about the client-sided authentication in React in this guide. That's true in terms of using Firebase. But to be honest, that was only half of the story when it comes to your own authentication implementation without Firebase. If you would implement your own authentication backend (e.g. Passport.js in a Node.js server) from scratch, there are a couple of more topics that you would need to consider for the client-side application. For instance, when using Firebase, you can sign in to your application, close the browser tab, open it up again and you will see yourself still signed in to the application. Firebase keeps this information stored for you and the listener in your `withAuthentication()` higher order component knows about the authenticated user object. However, if you would implement your own authentication mechanism from scratch, you would have to use cookies or the native local storage of the browser[33] on the client-side to keep a user authenticated over multiple browser sessions. That's only one caveat to the story, but shows that implementing an own authentication solution from scratch can be a complex adventure.

So what's next for this tutorial? Often people are using React with a state management library such as Redux or MobX. **The following sections will showcase you how to implement the session state handling in MobX or Redux**. You can follow one of these sections to learn about it. **They will not build up on each other**. Instead, both use this section as their boilerplate to continue with their implementation. However, keep two things in mind when you continue with one of these solutions:

- It works in plain React as well. So if you don't have any good reason to introduce Redux or MobX[34], consider to keep it like it is right now.
- This tutorial used the learnings from The Road to learn React[35] as foundation. The same applies for the next chapters and the course: Taming the State in React[36]. It is full of useful information about Redux and MobX and teaches state management in React from scratch by building

[33] https://www.robinwieruch.de/local-storage-react/
[34] https://www.robinwieruch.de/learn-react-before-using-redux/
[35] https://www.robinwieruch.de/the-road-to-learn-react/
[36] https://roadtoreact.com/course-details?courseId=TAMING_THE_STATE

applications along the way. So it is highly recommended to learn at least the basics about Redux or MobX from the course.

Authentication in React, Firebase and Redux

The section dives into using Redux on top of React and Firebase for the state management. Basically, you will exchange React's local state (user management: e.g. list of users on home page) and React's context (session management: e.g. authenticated user object) with Redux. The section builds on top of the last section which concluded the authentication and authorization in plain React and Firebase.

This section is divided into two parts. The first part will setup Redux and all the necessary parts for it. You will implement the state layer separately from the view layer. The second part exchanges the current state layer (local state for users, the context for an authenticated user) with the Redux state layer. It is the part where the new state layer is connected to the view layer.

First of all, you should install redux[37] and react-redux[38] on the command line.

Command Line

```
npm install redux react-redux
```

Furthermore, you will have to install recompose[39] on the command line to compose more than one higher order component on a component. You will enhance your component not only once, but multiple times by using the composing functionality of recompose.

Command Line

```
npm install recompose
```

Now let's setup the Redux state layer. First of all, you need the Redux store implementation. Therefore, create a folder and file for it.

Command Line: src/

```
mkdir store
cd store
touch index.js
```

Second, implement the store as singleton instance, because there should be only one Redux store. The store creation takes a root reducer which isn't defined yet. You will do it in the next step.

[37] https://redux.js.org/
[38] https://github.com/reactjs/react-redux
[39] https://github.com/acdlite/recompose

src/store/index.js

```
import { createStore } from 'redux';
import rootReducer from '../reducers';

const store = createStore(rootReducer);

export default store;
```

Third, create a dedicated module for the reducers. You will have a reducer for the session state (e.g. authenticated user) and a reducer for the user state (e.g. list of users from the database). In addition, you will have an entry point file to the module to combine those reducers as root reducer to pass it to the Redux store which you already did in the previous step.

Command Line: src/

```
mkdir reducers
cd reducers
touch index.js session.js user.js
```

Now, implement the two reducers. First, the session reducer which manages simply the `authUser` object. Remember that the authenticated user represents our session in the application.

src/reducers/session.js

```
const INITIAL_STATE = {
  authUser: null,
};

const applySetAuthUser = (state, action) => ({
  ...state,
  authUser: action.authUser
});

function sessionReducer(state = INITIAL_STATE, action) {
  switch(action.type) {
    case 'AUTH_USER_SET' : {
      return applySetAuthUser(state, action);
    }
    default : return state;
  }
}

export default sessionReducer;
```

Second, the user reducer which deals with the list of users from the Firebase realtime database:

src/reducers/user.js

```
const INITIAL_STATE = {
  users: {},
};

const applySetUsers = (state, action) => ({
  ...state,
  users: action.users
});

function userReducer(state = INITIAL_STATE, action) {
  switch(action.type) {
    case 'USERS_SET' : {
      return applySetUsers(state, action);
    }
    default : return state;
  }
}

export default userReducer;
```

Finally, combine both reducers in a root reducer to make it accessible for the store creation.

src/reducers/index.js

```
import { combineReducers } from 'redux';
import sessionReducer from './session';
import userReducer from './user';

const rootReducer = combineReducers({
  sessionState: sessionReducer,
  userState: userReducer,
});

export default rootReducer;
```

You have already passed the root reducer with all its reducers to the Redux store creation. The Redux setup is done. Now, you can connect your state layer with your view layer. The Redux store can be provided to the component hierarchy by using Redux's bridging Provider component. It's not the Provider component you have created before on your own by using React's context API, but this time a Provider component from a Redux library passing down the whole store instead of the authenticated user. The store will keep track of the authenticated user for you.

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import './index.css';
import App from './components/App';
import store from './store';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);

registerServiceWorker();
```

Now comes the refactoring part where you exchange a part of the React state layer with the Redux state layer. You will replace React's context entirely in this part by passing the state down the component tree with the Redux store.

Let's start with the simpler part of the refactoring: the user state layer. On the home page, you will use Redux's bridging library to connect the state via `mapStateToProps()` to the component. In addition, you will connect actions to your component as well via `mapDispatchToProps()` to store the users coming from the Firebase realtime database to your Redux store.

src/components/Home.js

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { compose } from 'recompose';

import withAuthorization from './withAuthorization';
import { db } from '../firebase';

class HomePage extends Component {
  componentDidMount() {
    const { onSetUsers } = this.props;

    db.onceGetUsers().then(snapshot =>
      onSetUsers(snapshot.val())
    );
  }

  render() {
    const { users } = this.props;

    return (
      <div>
```

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

        { !!users && <UserList users={users} /> }
      </div>
    );
  }
}

...

const mapStateToProps = (state) => ({
  users: state.userState.users,
});

const mapDispatchToProps = (dispatch) => ({
  onSetUsers: (users) => dispatch({ type: 'USERS_SET', users }),
});

const authCondition = (authUser) => !!authUser;

export default compose(
  withAuthorization(authCondition),
  connect(mapStateToProps, mapDispatchToProps)
)(HomePage);
```

Now the users are managed with Redux rather than in React's local state. You have connected the state and actions of Redux with the view layer.

What about the session state layer which should be handled by the session reducer? Essentially you will refactor it the same way as the user state layer before. You will replace your own Provider and Consumer components, where the authenticated user was reached through all components by using React's context API, with the state layer from Redux where the authenticated user will be stored in the Redux store instead. Thus, instead of passing the authenticated user object down via React's context, you pass it down via the global Redux store by providing the store in a parent component. You already provided the store in a parent component by using the custom Provider component from react-redux. Afterward, you can connect all the components that care about the authenticated user (e.g. Navigation, Account) to it.

The most important component to store the authenticated user object in the Redux store rather than in React's context is the `withAuthentication()` higher order component. We can refactor it to use the Redux store instead of React's context by connecting it to the state layer.

src/components/withAuthentication.js

```
import React from 'react';
import { connect } from 'react-redux';

import { firebase } from '../firebase';

const withAuthentication = (Component) => {
  class WithAuthentication extends React.Component {
    componentDidMount() {
      const { onSetAuthUser } = this.props;

      firebase.auth.onAuthStateChanged(authUser => {
        authUser
          ? onSetAuthUser(authUser)
          : onSetAuthUser(null);
      });
    }

    render() {
      return (
        <Component />
      );
    }
  }

  const mapDispatchToProps = (dispatch) => ({
    onSetAuthUser: (authUser) => dispatch({ type: 'AUTH_USER_SET', authUser }),
  });

  return connect(null, mapDispatchToProps)(WithAuthentication);
}

export default withAuthentication;
```

Now you have the authenticated user available in the Redux store. As consequence all components which rely on the authenticated user in React's context need to be refactored now. It's similar to the Home component which uses the list of users from the Redux store instead of React's local state.

In the Navigation component, the authenticated user is used to display different routing options. Thus you will need to refactor the component to connect it to the Redux store.

src/components/Navigation.js

```
import React from 'react';
import { connect } from 'react-redux';
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>

const mapStateToProps = (state) => ({
  authUser: state.sessionState.authUser,
});

export default connect(mapStateToProps)(Navigation);
```

In the Account component, the authenticated user is used to display the email address of the user. There you need to connect it to the Redux store as well.

src/components/Account.js

```
import React from 'react';
import { connect } from 'react-redux';
import { compose } from 'recompose';

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

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

const mapStateToProps = (state) => ({
  authUser: state.sessionState.authUser,
});

const authCondition = (authUser) => !!authUser;

export default compose(
  withAuthorization(authCondition),
  connect(mapStateToProps)
)(AccountPage);
```

Furthermore, don't forget that the authorization higher order component used the authenticated user from React's context as well for the fallback conditional rendering. You have to refactor it too.

src/components/withAuthorization.js

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

import { firebase } from '../firebase';
import * as routes from '../constants/routes';

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

    render() {
```

```
      return this.props.authUser ? <Component /> : null;
    }
  }

  const mapStateToProps = (state) => ({
    authUser: state.sessionState.authUser,
  });

  return compose(
    withRouter,
    connect(mapStateToProps),
  )(WithAuthorization);
}

export default withAuthorization;
```

That's it. In this section, you have introduced Redux as state management library to manage your session and user state. Instead of relying on React's context API for the authenticated user object and React's local state for the list of users from the Firebase database, you are storing these objects in the Redux store. You can find the project with a slightly different folder structure in this GitHub repository[40].

[40] https://github.com/rwieruch/react-redux-firebase-authentication

Authentication in React, Firebase and MobX

The section dives into using MobX on top of React and Firebase for the state management. Basically, you will exchange React's local state (user management: e.g. list of users on home page) and React's context (session management: e.g. authenticated user object) with MobX.

Note: None of the Redux changes from the previous section are reflected here. We will start with a clean plate from one section before where we didn't use Redux but only plain React and Firebase.

This section is divided into two parts. The first part will setup MobX and all the necessary parts for it. You will implement the state layer separately from the view layer. The second part exchanges the current state layer (local state for users, the context for an authenticated user) with the MobX state layer. It is the part where the new state layer is connected to the view layer.

First of all, you should follow this short guide to enable decorators in create-react-app[41]. You can also take the path of not using decorators, to avoid the eject process, but this tutorial only reflects the usage **with decorators**.

You should have installed mobx[42] and mobx-react[43] by now. Furthermore, you will have to install recompose on the command line to compose more than one higher order component on a component. You will enhance your component not only once, but multiple times by using the composing functionality of recompose.

Command Line

```
npm install recompose
```

Now let's setup the MobX state layer. First of all, you need to implement the MobX stores. Therefore, create a folder and files for it.

Command Line: src/

```
mkdir stores
cd stores
touch index.js sessionStore.js userStore.js
```

You will have a store for the session state (e.g. authenticated user) and a store for the user state (e.g. list of users from the database). In addition, you will have an entry point file to the module to combine those stores as root store. First, the session store which manages simply the authUser object. Remember that the authenticated user represents our session in the application.

[41] https://www.robinwieruch.de/create-react-app-mobx-decorators/
[42] https://mobx.js.org/
[43] https://github.com/mobxjs/mobx-react

src/stores/sessionStore.js

```
import { observable, action } from 'mobx';

class SessionStore {
  @observable authUser = null;

  constructor(rootStore) {
    this.rootStore = rootStore;
  }

  @action setAuthUser = authUser => {
    this.authUser = authUser;
  }
}

export default SessionStore;
```

Second, the user store which deals with the list of users from the Firebase realtime database:

src/stores/userStore.js

```
import { observable, action } from 'mobx';

class UserStore {
  @observable users = [];

  constructor(rootStore) {
    this.rootStore = rootStore;
  }

  @action setUsers = users => {
    this.users = users;
  }
}

export default UserStore;
```

Finally, combine both stores in a root store. This can be used to make the stores communicate to each other, but also to provide a way to import only one store (root store) to have access to all of its combined stores later on.

src/stores/index.js

```
import { configure } from 'mobx';

import SessionStore from './sessionStore';
import UserStore from './userStore';

configure({ enforceActions: true });

class RootStore {
  constructor() {
    this.sessionStore = new SessionStore(this);
    this.userStore = new UserStore(this);
  }
}

const rootStore = new RootStore();

export default rootStore;
```

The MobX setup is done. Now, you can connect your state layer with your view layer. All the MobX stores can be provided to the component hierarchy by using MobX's bridging Provider component. It's not the Provider component you have created before on your own by using React's context API, but this time a Provider component from a MobX library passing down the all stores instead of the authenticated user. The stores (e.g. user store) will keep track of the authenticated user for you. We will use an object spread operator here to pass all combined stores to the Provider component. Keep in mind that this way the root store isn't available in the component hierarchy, but only the combined stores. That's sufficient for the tutorial, but if you want to have access to the root store as well, you need to pass it to the Provider component too.

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'mobx-react';
import './index.css';
import App from './components/App';
import store from './stores';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(
  <Provider { ...store }>
    <App />
  </Provider>,
  document.getElementById('root')
);

registerServiceWorker();
```

Now comes the refactoring part where you exchange a part of the React state layer with the MobX state layer. You will replace React's context entirely in this part by passing the state down the component tree with the MobX stores.

Let's start with the simpler part of the refactoring: the user state layer. On the home page, you will use MobX's bridging library to inject the needed store via `inject()` to the component. In addition, you will make the component observable via `observer()` from the bridging library for MobX reactions. If the MobX state changes, the component will react to it. In addition, the user store is used to store the users coming from the Firebase realtime database.

src/components/Home.js

```
import React, { Component } from 'react';
import { inject, observer } from 'mobx-react';
import { compose } from 'recompose';

import withAuthorization from './withAuthorization';
import { db } from '../firebase';

class HomePage extends Component {
  componentDidMount() {
    const { userStore } = this.props;

    db.onceGetUsers().then(snapshot =>
      userStore.setUsers(snapshot.val())
    );
  }

  render() {
    const { users } = this.props.userStore;

    return (
      <div>
        <h1>Home</h1>
        <p>The Home Page is accessible by every signed in user.</p>

        { !!users && <UserList users={users} /> }
      </div>
    );
  }
}

...

const authCondition = (authUser) => !!authUser;

export default compose(
  withAuthorization(authCondition),
  inject('userStore'),
  observer
)(HomePage);
```

Now the users are managed with MobX rather than in React's local state. You have connected the state from MobX with the view layer.

What about the session state layer which should be handled by the session store? Essentially you will refactor it the same way as the user state layer before. You will replace your own Provider and Consumer components, where the authenticated user was reached through all components by using React's context API, where the authenticated user will be stored in the session store. Thus, instead of passing the authenticated user object down via React's context, you pass it down via the MobX's session store by providing the store in a parent component. You already provided the stores in a parent component by using the custom Provider component from mobx-react. Afterward, you can connect all the components that care about the authenticated user (e.g. Navigation, Account) to it.

The most important component to store the authenticated user object in the MobX session store rather than in React's context is the `withAuthentication()` higher order component. We can refactor it to use the MobX session store instead of React's context by connecting it to the state layer.

src/components/withAuthentication.js

```
import React from 'react';
import { inject } from 'mobx-react';

import { firebase } from '../firebase';

const withAuthentication = (Component) => {
  class WithAuthentication extends React.Component {
    componentDidMount() {
      const { sessionStore } = this.props;

      firebase.auth.onAuthStateChanged(authUser => {
        authUser
          ? sessionStore.setAuthUser(authUser)
          : sessionStore.setAuthUser(null);
      });
    }

    render() {
      return (
        <Component />
      );
    }
  }

  return inject('sessionStore')(WithAuthentication);
}

export default withAuthentication;
```

Now you have the authenticated user available in the MobX session store. As consequence all components which rely on the authenticated user in React's context need to be refactored now.

It's similar to the Home component which uses the list of users from the MobX user store instead of React's local state.

In the Navigation component, the authenticated user is used to display different routing options. Thus you will need to refactor the component to inject the MobX session store instead.

src/components/Navigation.js

```
import React from 'react';
import { inject, observer } from 'mobx-react';
import { compose } from 'recompose';
import { Link } from 'react-router-dom';

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

const Navigation = ({ sessionStore }) =>
  <div>
    { sessionStore.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 compose(
  inject('sessionStore'),
  observer
)(Navigation);
```

In the Account component, the authenticated user is used to display the email address of the user. There you need to inject the MobX session store as well.

src/components/Account.js

```
import React from 'react';
import { inject, observer } from 'mobx-react';
import { compose } from 'recompose';

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

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

const authCondition = (authUser) => !!authUser;

export default compose(
  withAuthorization(authCondition),
  inject('sessionStore'),
  observer
)(AccountPage);
```

Furthermore, don't forget that the authorization higher order component used the authenticated user from React's context as well for the fallback conditional rendering. You have to refactor it too.

src/components/withAuthorization.js

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

import { firebase } from '../firebase';
import * as routes from '../constants/routes';

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

    render() {
      return this.props.sessionStore.authUser ? <Component /> : null;
    }
  }
```

```
  return compose(
    withRouter,
    inject('sessionStore'),
    observer
  )(WithAuthorization);
}

export default withAuthorization;
```

That's it. In this section, you have introduced MobX as state management library to manage your session and user state. Instead of relying on React's context API for the authenticated user object and React's local state for the list of users from the Firebase database, you are storing these objects in the MobX stores. You can find the project with a slightly different folder structure in this GitHub repository[44].

[44] https://github.com/rwieruch/react-mobx-firebase-authentication

Outline

Hopefully, the guide has helped you to implement your own authentication and authorization mechanism in React with Firebase. If you have any suggestions or improvements, please reach out to me. Again, if the tutorial didn't explain everything in detail, then it was because most those things are already covered in my (partly free) courses[45] and the other referenced articles. The tutorial builds up on top of these learning materials, so make sure to check them out to learn everything about React, Redux, and MobX.

[45] https://roadtoreact.com/