Manage Users with Firebase's Real-time Database in React

In this lesson, we will manage the users in our React app using Firebase's Real-time Database.

WE'LL COVER THE FOLLOWING

In the last lesson, we created two new references. One for a specific user and the other for an object of all the users.

Let's pick up where we left off. We will use these references in our React components to create and get users from Firebase's Real-time database.

Creating a User

The best place to add user creation is the **SignUpForm component**, as it is the most natural place to save users after they have signed up via the Firebase authentication API. We add another API request to create a user when the sign up is successful.

In `src/components/SignUp/index.js` file, we add the following:

```
...

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

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

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

    this.props.firebase
      .doCreateUserWithEmailAndPassword(email, passwordOne)
```

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

    event.preventDefault();
  };

  ...
}

...
```

There are *two* important things happening with a new sign-up via the `submit handler`:

1. It creates a user in Firebase's internal authentication database which has **limited access**.
2. If (1) is successful, it creates a user in Firebase's real-time database which is accessible.

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

> **Note**
>
> It is fine to store user information in our own database. However, we should not store the password or any other sensitive data of the user. Firebase already deals with the authentication, so there is no need to store the password in our database. Several steps are required to secure sensitive data (e.g. encryption), and it can be a security risk to perform

> them on our own.

After the second Firebase request is resolved successfully, the previous business logic takes place again: *reset the local state and redirect to the home page*. To verify if the *"create user"* functionality is working, we retrieve all users from the real-time database in one of our other components. The admin page may be a good choice for this as it can be used by users with an admin role to manage the application-wide users later.

Accessing the Admin Page

First, we make sure that the admin page is available via the Navigation component.

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

```
...

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

...
```

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

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

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

class AdminPage extends Component {
```

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


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

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

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

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

export default withFirebase(AdminPage);
```

We are using the user's reference from our Firebase class to attach a listener. The listener is called `on()`, which receives a type and a callback function. The `on()` method registers a continuous listener that is triggered every time something changes, while the `once()` method registers a listener that would be called only once. In this scenario, we are only interested in keeping the latest list of users.

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

```
...

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

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

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

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

      const usersList = Object.keys(usersObject).map(key => ({
        ...usersObject[key],
        uid: key,
      }));

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

  ...
}

export default withFirebase(AdminPage);
```

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

```
...

class AdminPage extends Component {
  ...

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

  ...
}

export default withFirebase(AdminPage);
```

We render our list of users in the **AdminPage component** or in a **child component**.

In this case, we are using a child component as can be seen in the code snippet below:

```
...

class AdminPage extends Component {
  ...

  render() {
```

```
    const { users, loading } = this.state;

    return (

      <div>
        <h1>Admin</h1>

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

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

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

export default withFirebase(AdminPage);
```

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

We can finally test our entire app in the next section!