authentication databases express node react

Build a Basic CRUD App with Node and React

Braden Kelley

July 10, 2018

Last Updated: April 5, 2021

22 MIN READ

There are *a lot* of JavaScript frameworks out there today. It seems like I hear about a new one every month or so. They all have their advantages and are usually there to solve some sort of problem with an existing framework. My favorite to work with so far has been React. One of the best things about it is how many open source components and libraries there are in the React ecosystem, so you have a lot to choose from. This can be really difficult if you're indecisive, but if you like the freedom to do things your way then React may be the best option for you.

In this tutorial, I'll walk you through creating both a frontend web app in React and a backend REST API server in Node. The frontend will have a home page and a posts manager, with the posts manager hidden behind secure user authentication. As an added security measure, the backend will also not let you create or edit posts unless you're properly authenticated.

The tutorial will use Okta's OpenID Connect (OIDC) to handle authentication. On the frontend, the Okta React SDK will be used to request a token and provide it in requests to the server. On the backend, the Okta JWT Verifier will ensure that the user is properly authenticated, and throw an error otherwise.

Table of Contents

Why React?

React has been one of the most popular JavaScript libraries for the past few years. One of the biggest concepts behind it, and what makes it so fast, is to use a virtual DOM (the Document Object Model, or DOM, is what describes the layout of a web page) and make small updates in batches to the real DOM. React isn't the first library to do this, and there are quite a few now, but it certainly made the idea popular. The idea is that the DOM is slow, but JavaScript is fast, so you just say what you want the final output to look like and React will make those changes to the DOM behind the scenes. If no changes need to be made, then it doesn't affect the DOM. If only a small text field changes, it will just patch that one element.

React is also most commonly associated with JSX, even though it's possible to use React without JSX. JSX lets you mix HTML in with your JavaScript. Rather than using templates to define the HTML and binding those values to a view model, you can just write everything in JavaScript. Values can be plain JavaScript objects, instead of strings that need to be interpreted. You can also write reusable React components that then end up looking like any other HTML element in your code.

```
const Form = () => (
  <form>
    <label>
      Name
      <input value="Arthur Dent" />
    </label>
    <label>
      Answer to life, the universe, and everything
      <input type="number" value={42} />
    </label>
  </form>
);

const App = () => (
  <main>
    <h1>Welcome, Hitchhiker!</h1>
    <Form />
  </main>
);
```

...and here's what the same code would look like if you wrote it in plain JavaScript, without using JSX:

```
const Form = () => React.createElement(
  "form",
  null,
  React.createElement(
    "label",
    null,
    "Name",
    React.createElement("input", { value: "Arthur Dent" })
  ),
  React.createElement(
    "label",
    null,
    "Answer to life, the universe, and everything",
    React.createElement("input", { type: "number", value: 42 })
  )
);

const App = () => React.createElement(
  "main",
  null,
  React.createElement(
    "h1",
    null,
    "Welcome, Hitchhiker!"
  ),
  React.createElement(Form, null)
);
```

Create Your React App

The quickest way to get started with React is to use Create React App, a tool that generates a progressive web app (PWA) with all the scripts and boilerplate tucked away neatly behind something called `react-scripts`, so you can just focus on writing code. It has all kinds of nice dev features as well, like updating the code whenever you make changes, and scripts to compile it down for production. You can use `npm` or `yarn`, but I'll be using `yarn` in this tutorial.

To install `create-react-app` and `yarn`, simply run:

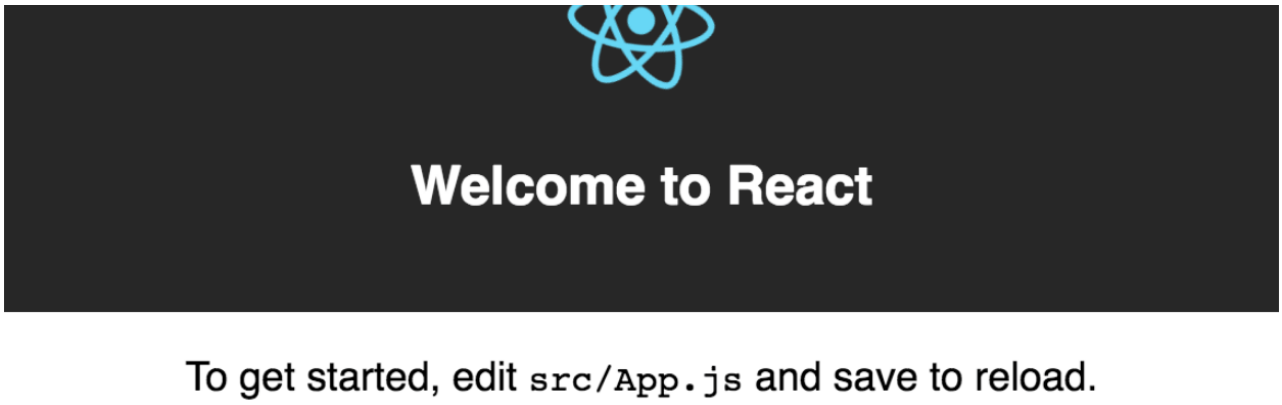
```
npm i -g create-react-app@3.2.0 yarn@1.19.1
```

NOTE: I'll be adding version numbers to help future-proof this post. In general though, you'd be fine leaving out the version numbers (e.g. `npm i -g create-react-app`).

Now bootstrap your application with the following commands:

```
create-react-app my-react-app
cd my-react-app
yarn start
```

The default app should now be running on port 3000. Check it out at `http://localhost:3000`.



Create a Basic Homepage in React with Material UI

To keep things looking nice without writing a lot of extra CSS, you can use a UI framework. Material UI is a great framework for React that implements Google's Material Design principles.

Add the dependency with:

```
yarn add @material-ui/core@4.5.1
```

Material recommends the Roboto font. You can add it to your project by editing `public/index.html` and adding the following line inside the `head` tag:

```
<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto:300,400,500">
```

You can separate components into separate files to help keep things organized. First, create a couple new folders in your `src` directory: `components`, and `pages`

Now create an `AppHeader` component. This will serve as the navbar with links to pages, as well as show the title and whether you're logged in.

src/components/AppHeader.js

```
import React from 'react';
import {
  AppBar,
  Toolbar,
  Typography,
} from '@material-ui/core';

const AppHeader = () => (
  <AppBar position="static">
    <Toolbar>
      <Typography variant="h6" color="inherit">
        My React App
      </Typography>
    </Toolbar>
  </AppBar>
);

export default AppHeader;
```

Also create a homepage:

src/pages/Home.js

```
import React from 'react';
import {
  Typography,
} from '@material-ui/core';

export default () => (
  <Typography variant="h4">Welcome Home!</Typography>
);
```

Now go ahead and actually just gut the sample app, replacing `src/App.js` with the following:

src/App.js

```
  CssBaseline,
  withStyles,
} from '@material-ui/core';

import AppHeader from './components/AppHeader';
import Home from './pages/Home';

const styles = theme => ({
  main: {
    padding: theme.spacing(3),
    [theme.breakpoints.down('xs')]: {
      padding: theme.spacing(2),
    },
  },
});

const App = ({ classes }) => (
  <Fragment>
    <CssBaseline />
    <AppHeader />
    <main className={classes.main}>
      <Home />
    </main>
  </Fragment>
);

export default withStyles(styles)(App);
```

Material UI uses JSS (one of many flavors in the growingly popular trend of CSS in JavaScript), which is what `withStyles` provides.

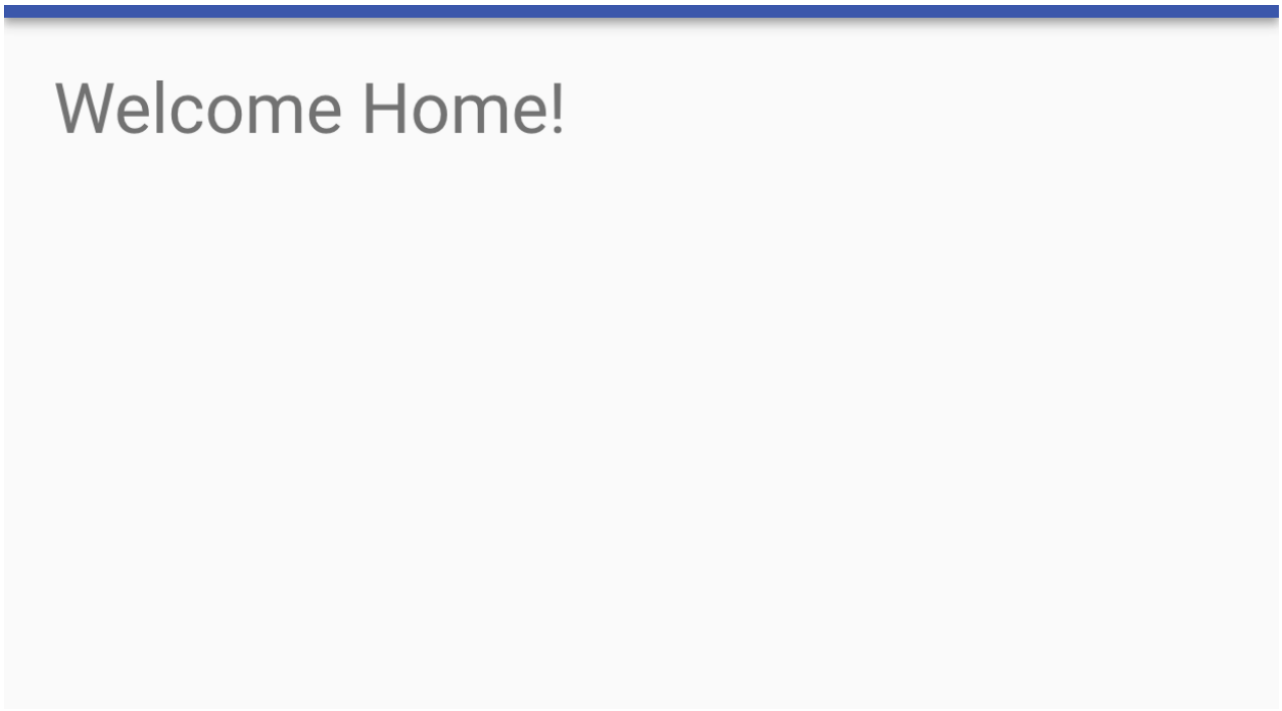
The `CssBaseline` component will add some nice CSS defaults to the page (e.g. removing margins from the body), so we no longer need `src/index.css`. You can get rid of a couple other files too, now that we've gotten rid of most of the `Hello World` demo app.

```
rm src/index.css src/App.css src/logo.svg
```

In `src/index.js`, remove the reference to `index.css` (the line that says `import './index.css';`). While you're at it, add the following as the very last line of `src/index.js` to turn on hot module reloading, which will make it so that changes you make automatically update in the app without needing to refresh the whole page:

```
if (module.hot) module.hot.accept();
```

At this point, your app should look like this:



Add Authentication to Your Node + React App with Okta

You would never ship your new app out to the Internet without secure identity management, right? Well, Okta makes that a lot easier and more scalable than what you're probably used to. Okta is a cloud service that allows developers to create, edit, and securely store user accounts and user account data, and connect them with one or multiple applications. Our API enables you to:

- Authenticate and authorize your users
- Store data about your users
- Perform password-based and social login
- Secure your application with multi-factor authentication
- And much more! Check out our product documentation

Before you begin, you'll need a free Okta developer account. Install the Okta CLI and run `okta register` to sign up for a new account. If you already have an account, run `okta login`. Then, run `okta apps create`. Select the default app name, or change it as you see fit. Choose **Single-Page App** and press **Enter**.

▶ What does the Okta CLI do?

Copy your **Client ID** and paste it as a variable into a file called `.env.local` in the root of your project. This will allow you to access the file in your code without needing to store credentials in source control. Your Okta domain is the first part of your issuer, before `/oauth2/default`.

Environment variables (other than `NODE_ENV`) need to start with `REACT_APP_` in order for Create React App to read them, so the file should end up looking like this:

.env.local

```
REACT_APP_OKTA_CLIENT_ID={clientId}
REACT_APP_OKTA_ORG_URL={yourOktaDomain}
```

The easiest way to add Authentication with Okta to a React app is to use Okta's React SDK. You'll also need to add routes, which can be done using React Router. I'll also have you start adding icons to the app (for now as an avatar icon to show you're logged in). Material UI provides Material Icons, but in another package, so you'll need to add that too. Run the following command to add these new dependencies:

```
yarn add @okta/okta-react@3.0.8 react-router-dom@5.1.2 @material-ui/icons@4.5.1
```

For routes to work properly in React, you need to wrap your whole application in a `Router`. Similarly, to allow access to authentication anywhere in the app, you need to wrap the app in a `Security` component provided by Okta. Okta also needs access to the router, so the `Security` component should be nested inside the router. You should modify your `src/index.js` file to look like the following:

src/index.js

```
import { BrowserRouter } from 'react-router-dom';
import { Security } from '@okta/okta-react';

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

const oktaConfig = {
  issuer: `${process.env.REACT_APP_OKTA_ORG_URL}/oauth2/default`,
  redirect_uri: `${window.location.origin}/login/callback`,
  client_id: process.env.REACT_APP_OKTA_CLIENT_ID,
};

ReactDOM.render(
  <BrowserRouter>
    <Security {...oktaConfig}>
      <App />
    </Security>
  </BrowserRouter>,
  document.getElementById('root'),
);

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();

if (module.hot) module.hot.accept();
```

Now in `src/App.js` you can use `Route`s. These tell the app to only render a certain component if the current URL matches the given `path`. Replace your `Home` component with a route that only renders the component when pointing at the root URL (`/`), and renders Okta's `LoginCallback` component for the `/login/callback` path.

src/App.js

```
@@ -1,4 +1,5 @@
 import React, { Fragment } from 'react';
+import { Route } from 'react-router-dom';
+import { LoginCallback } from '@okta/okta-react';
 import {
   CssBaseline,
   withStyles,
@@ -21,7 +23,8 @@ const App = ({ classes }) => (
     <CssBaseline />
     <AppHeader />
     <main className={classes.main}>
-      <Home />
+      <Route exact path="/" component={Home} />
+      <Route path="/login/callback" component={LoginCallback} />
     </main>
   </Fragment>
 );
```

Next, you need a login button. This file is a bit bigger because it contains some logic to check if the user is authenticated. I'll show you the whole component first, then walk through what each section is doing:

src/components/LoginButton.js

```
  IconButton,
  Menu,
  MenuItem,
  ListItemText,
} from '@material-ui/core';
import { AccountCircle } from '@material-ui/icons';
import { withOktaAuth } from '@okta/okta-react';

class LoginButton extends Component {
  state = {
    authenticated: null,
    user: null,
    menuAnchorEl: null,
  };

  componentDidUpdate() {
    this.checkAuthentication();
  }

  componentDidMount() {
    this.checkAuthentication();
  }

  async checkAuthentication() {
    const authenticated = this.props.authState.isAuthenticated;
    if (authenticated !== this.state.authenticated) {
      const user = await this.props.authService.getUser();
      this.setState({ authenticated, user });
    }
  }

  login = () => this.props.authService.login('/');
  logout = () => {
    this.handleMenuClose();
    this.props.authService.logout('/');
  };

  handleMenuOpen = event => this.setState({ menuAnchorEl: event.currentTarget });
  handleMenuClose = () => this.setState({ menuAnchorEl: null });

  render() {
    const { authenticated, user, menuAnchorEl } = this.state;

    if (authenticated === null) return null;
    if (!authenticated) return <Button color="inherit" onClick={this.login}>Login</Button>;

    const menuPosition = {
      vertical: 'top',
      horizontal: 'right',
    };

    return (
      <div>
        <IconButton onClick={this.handleMenuOpen} color="inherit">
          <AccountCircle />
```

```
            anchorOrigin={menuPosition}
            transformOrigin={menuPosition}
            open={!!menuAnchorEl}
            onClose={this.handleMenuClose}
          >
            <MenuItem onClick={this.logout}>
              <ListItemText
                primary="Logout"
                secondary={user && user.name}
              />
            </MenuItem>
          </Menu>
        </div>
      );
    }
  }

  export default withOktaAuth(LoginButton);
```

React components have a concept of state management. Each component can be passed props (in a component like `<input type="number" value={3} />`, `type` and `number` would be considered props). They can also maintain their own state, which has some initial values and can be changed with a function called `setState`. Any time the `props` or `state` changes, the component will rerender, and if changes need to be made to the DOM they will happen then. In a component, you can access these with `this.props` or `this.state`, respectively.

Here, you're creating a new React component and setting the initial state values. Until you query the `auth` prop, you don't know whether there's a user or not, so you set `authenticated` and `user` to `null`. Material UI will use `menuAnchorEl` to know where to anchor the menu that lets you log the user out.

```
  class LoginButton extends Component {
    state = {
      authenticated: null,
      user: null,
      menuAnchorEl: null,
    };

    // ...
  }
```

React components also have their own lifecycle methods, which are hooks you can use to trigger actions at certain stages of the component lifecycle. Here, when the component is first mounted you'll check to see whether or not the user has been authenticated, and if so

update the state when something is different, otherwise you'll get yourself into an infinite loop (the component updates, so you give the component new values, which updates the component, you give it new values, etc.). The `withOktaAuth()` function is a Higher Order Component (HOC) which wraps the original component and returns another one containing the `auth` prop.

```
class LoginButton extends Component {
  // ...

  componentDidUpdate() {
    this.checkAuthentication();
  }

  componentDidMount() {
    this.checkAuthentication();
  }

  async checkAuthentication() {
    const authenticated = this.props.authState.isAuthenticated;
    if (authenticated !== this.state.authenticated) {
      const user = await this.props.authService.getUser();
      this.setState({ authenticated, user });
    }
  }

  // ...
}

export default withOktaAuth(LoginButton);
```

The following functions are helper functions used later to log the user in or out, and open or close the menu. Writing the function as an arrow function ensures that `this` is referring to the instantiation of the component. Without this, if a function is called somewhere outside of the component (e.g. in an `onClick` event), you would lose access to the component and wouldn't be able to execute functions on it or access `props` or `state`.

```
  login = () => this.props.authService.login('/');
  logout = () => {
    this.handleMenuClose();
    this.props.authService.logout('/');
  };

  handleMenuOpen = event => this.setState({ menuAnchorEl: event.currentTarget });
}
```

All React components must have a `render()` function. This is what tells React what to display on the screen, even if it shouldn't display anything (in which case you can return `null`).

When you're not sure of the authentication state yet, you can just return `null` so the button isn't rendered at all. You can use `this.props.authState.isAuthenticated` to determine if a user is signed in. If it's `false`, you'll want to provide a `Login` button. If the user is logged in, you can instead display an avatar icon that has a dropdown menu with a Logout button.

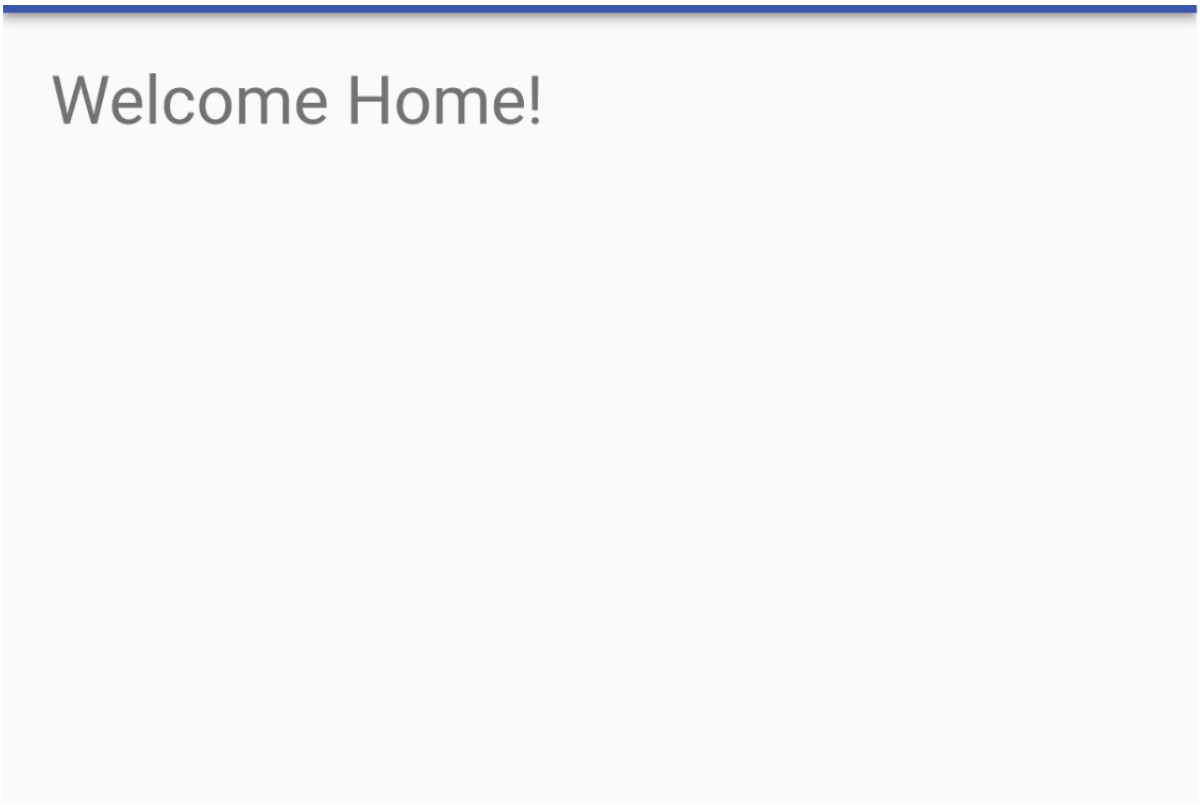
```
  render() {
    const { authenticated, user, menuAnchorEl } = this.state;

    if (authenticated == null) return null;
    if (!authenticated) return <Button color="inherit" onClick={this.login}>Login</Button>;

    const menuPosition = {
      vertical: 'top',
      horizontal: 'right',
    };

    return (
      <div>
        <IconButton onClick={this.handleMenuOpen} color="inherit">
          <AccountCircle />
        </IconButton>
        <Menu
          anchorEl={menuAnchorEl}
          anchorOrigin={menuPosition}
          transformOrigin={menuPosition}
          open={!!menuAnchorEl}
          onClose={this.handleMenuClose}
        >
          <MenuItem onClick={this.logout}>
            <ListItemText
              primary="Logout"
              secondary={user && user.name}
            />
          </MenuItem>
        </Menu>
      </div>
    );
  }
}
```

The next piece of the puzzle is to add this `LoginButton` component to your header. In order to display it on the right-hand side of the page, you can put an empty spacer `div` that has a `flex` value of 1. Since the other objects aren't told to flex, the spacer will take up as much space as it can. Modify your `src/components/AppHeader.js` file like so:

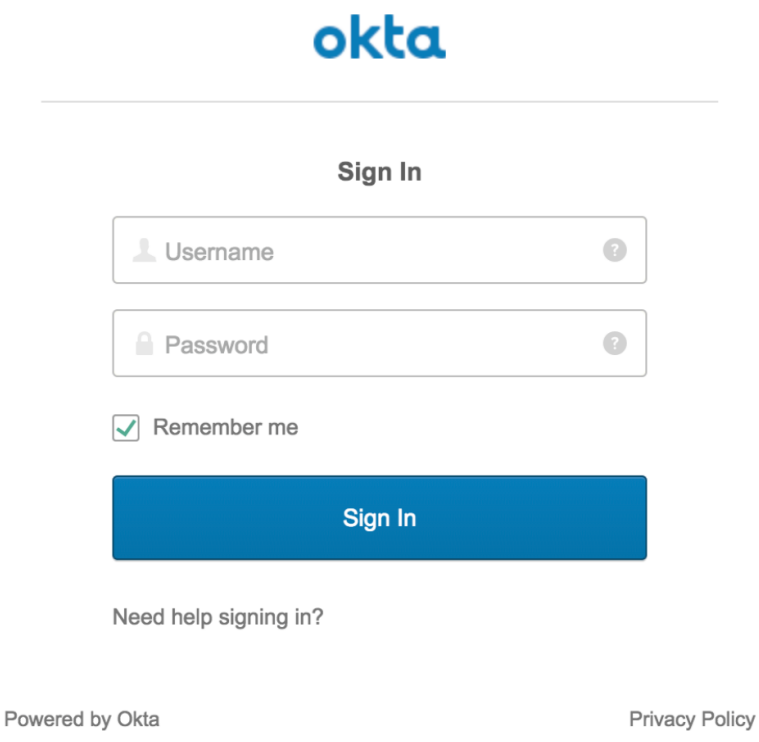
src/components/AppHeader.js

```
@@ -3,18 +3,27 @@ import {
   AppBar,
   Toolbar,
   Typography,
+  withStyles,
 } from '@material-ui/core';

-const AppHeader = () => (
+import LoginButton from './LoginButton';
+
+const styles = {
+  flex: {
+    flex: 1,
+  },
+};
+
+const AppHeader = ({ classes }) => (
   <AppBar position="static">
     <Toolbar>
       <Typography variant="h6" color="inherit">
         My React App
       </Typography>
+      <div className={classes.flex} />
+      <LoginButton />
     </Toolbar>
   </AppBar>
 );

-export default AppHeader;
+export default withStyles(styles)(AppHeader);
```

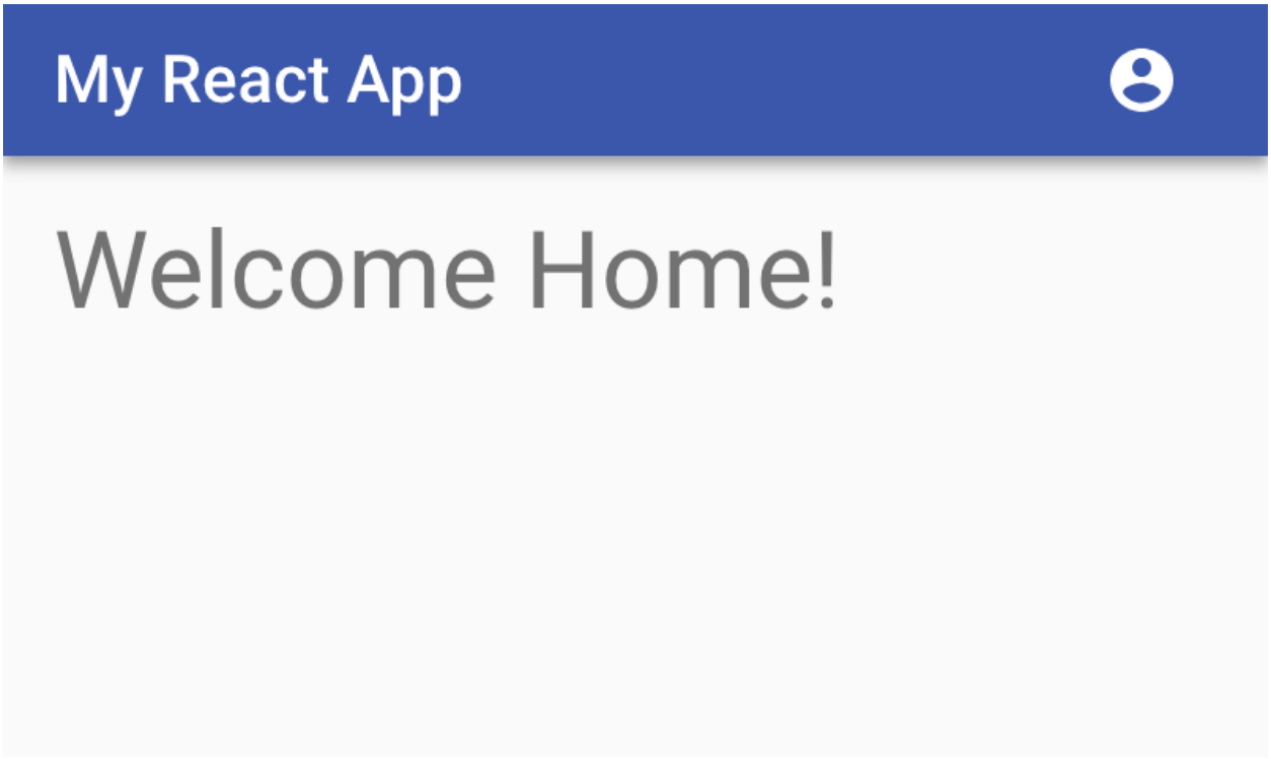
You should now be able to log in and out of your app using the button in the top right.

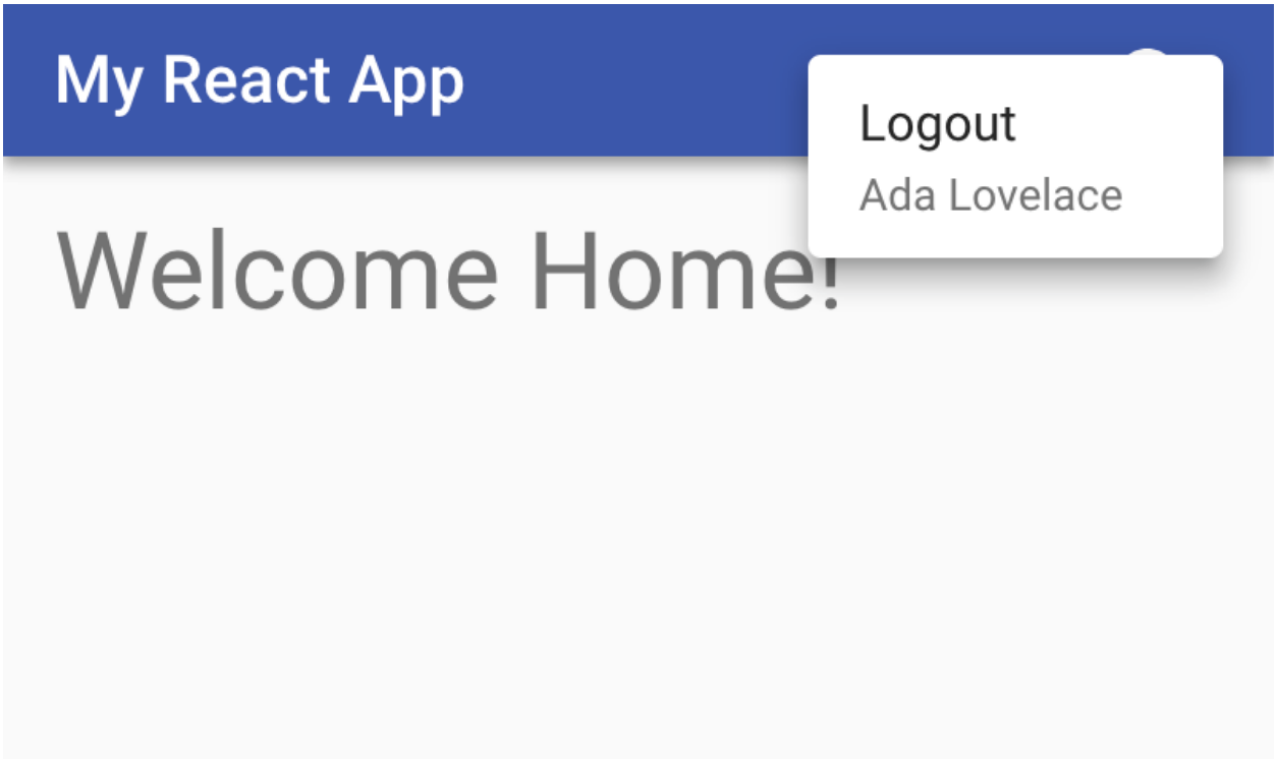


When you click the Login button, you'll be redirected to your Okta organization URL to handle authentication. You can log in with the same credentials you use in your developer console.



button. Clicking the button keeps you on the homepage but logs you out again.





Add a Node REST API Server

project at this point:

```
yarn add @okta/jwt-verifier@2.1.0 body-parser@1.19.0 cors@2.8.5 dotenv@8.2.0 finale-rest@1.0.6
yarn add -D npm-run-all@4.1.5
```

You'll also need to make sure you have SQLite installed on your machine. You can check if it's installed by running the following command:

```
which sqlite3
```

You should get something like `/usr/bin/sqlite3` or `/Users/me/bin/sqlite3`. If you don't get a response, then you'll need to install SQLite. On a Debian-based Linux system you should be able to run the command `sudo apt-get install sqlite3`. On macOS you could use homebrew with `brew install sqlite3`. If you're having trouble geting it installed, try visiting the SQLite homepage for more information.

Once you have SQLite ready, create a new folder for the server under the src directory:

```
mkdir src/server
```

Now create a new file `src/server/index.js`. To keep this simple we will just use a single file, but you could have a whole subtree of files in this folder. Keeping it in a separate folder lets you watch for changes just in this subdirectory and reload the server only when making changes to this file, instead of anytime any file in `src` changes. Again, I'll post the whole file and then explain some key sections below.

src/server/index.js

```
const express = require('express');
const cors = require('cors');
const bodyParser = require('body-parser');
const Sequelize = require('sequelize');
const finale = require('finale-rest');
const OktaJwtVerifier = require('@okta/jwt-verifier');

const oktaJwtVerifier = new OktaJwtVerifier({
  clientId: process.env.REACT_APP_OKTA_CLIENT_ID,
  issuer: `${process.env.REACT_APP_OKTA_ORG_URL}/oauth2/default`,
});

const app = express();
app.use(cors());
app.use(bodyParser.json());

app.use(async (req, res, next) => {
  try {
    if (!req.headers.authorization) throw new Error('Authorization header is required');

    const accessToken = req.headers.authorization.trim().split(' ')[1];
    await oktaJwtVerifier.verifyAccessToken(accessToken, 'api://default');
    next();
  } catch (error) {
    next(error.message);
  }
});

const database = new Sequelize({
  dialect: 'sqlite',
  storage: './test.sqlite',
});

const Post = database.define('posts', {
  title: Sequelize.STRING,
  body: Sequelize.TEXT,
});

finale.initialize({ app, sequelize: database });

finale.resource({
  model: Post,
  endpoints: ['/posts', '/posts/:id'],
});

const port = process.env.SERVER_PORT || 3001;

database.sync().then(() => {
  app.listen(port, () => {
    console.log(`Listening on port ${port}`);
  });
});
```

```
require('dotenv').config({ path: '.env.local' });
```

This sets up the HTTP server and adds some settings to allow for Cross-Origin Resource Sharing (CORS) and will automatically parse JSON.

```
const app = express();
app.use(cors());
app.use(bodyParser.json());
```

Here is where you check that a user is properly authenticated. First, throw an error if there is no `Authorization` header, which is how you will send the authorization token. The token will actually look like `Bearer aLongBase64String`. You want to pass the Base 64 string to the Okta JWT Verifier to check that the user is properly authenticated. The verifier will initially send a request to the issuer to get a list of valid signatures, and will then check locally that the token is valid. On subsequent requests, this can be done locally unless it finds a claim that it doesn't have signatures for yet.

If everything looks good, the call to `next()` tells Express to go ahead and continue processing the request. If however, the claim is invalid, an error will be thrown. The error is then passed into `next` to tell Express that something went wrong. Express will then send an error back to the client instead of proceeding.

```
app.use(async (req, res, next) => {
  try {
    if (!req.headers.authorization) throw new Error('Authorization header is required');

    const accessToken = req.headers.authorization.trim().split(' ')[1];
    await oktaJwtVerifier.verifyAccessToken(accessToken, 'api://default');
    next();
  } catch (error) {
    next(error.message);
  }
});
```

Here is where you set up Sequelize. This is a quick way of creating database models. You can Sequelize with a wide variety of databases, but here you can just use SQLite to get up and running quickly without any other dependencies.

```
  storage: './test.sqlite',
});

const Post = database.define('posts', {
  title: Sequelize.STRING,
  body: Sequelize.TEXT,
});
```

Finale works well with Sequelize and Express. It binds the two together like glue, creating a set of CRUD endpoints with just a couple lines of code. First, you initialize Finale with the Express app and the Sequelize database model. Next, you tell it to create your endpoints for the `Post` model: one for a list of posts, which will have `POST` and `GET` methods; and one for individual posts, which will have `GET`, `PUT`, and `DELETE` methods.

```
finale.initialize({ app, sequelize: database });

finale.resource({
  model: Post,
  endpoints: ['/posts', '/posts/:id'],
});
```

The last part of the server is where you tell Express to start listening for HTTP requests. You need to tell sequelize to initialize the database, and when it's done it's OK for Express to start listening on the port you decide. By default, since the React app is using `3000`, we'll just add one to make it port `3001`.

```
const port = process.env.SERVER_PORT || 3001;

database.sync().then(() => {
  app.listen(port, () => {
    console.log(`Listening on port ${port}`);
  });
});
```

Now you can make a couple small changes to `package.json` to make it easier to run both the frontend and backend at the same time. Replace the default `start` script and add a couple others, so your scripts section looks like this:

package.json

```
    "start:web": "react-scripts start",
    "start:server": "node src/server",
    "watch:server": "nodemon --watch src/server src/server",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
```

Now you can simply run `yarn start` and both the server and the React app will run at the same time, reloading whenever relevant changes are made. If you need to change the port for any reason, you can change the React app's port and the server's port with the `PORT` and `SERVER_PORT` environment variables, respectively. For example, `PORT=8080 SERVER_PORT=8081 yarn start`.

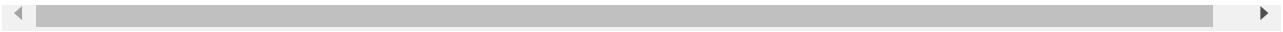
Add the Posts Manager Page to Your Node + React App

Now that you have a Node backend to manage your posts, you can link up the React frontend by adding another page. This will send requests to fetch, create, edit, and delete posts. It will also send the required authorization token along with each request so the server knows that you're a valid user.

One nice thing about React Router is that it lets you use variables in the URL. This will allow us to use the ID of a post in the URL, so you could go to `/posts/2` to view post number 2. With that in mind, you can create a modal that will be open whenever you are on that portion of the page, and to close the modal all you would need to do is navigate back to `/posts`.

Forms in React can be a bit of a pain. You can use a basic `form` element, but you would also need to listen for `onChange` events, update the state of the component, and set the new value on the `input` elements. To make forms easier, there are at least a few libraries out there, but I'll show you how to use React Final Form to cut out a lot of the boilerplate.

You'll also need recompose, lodash, and moment for some helper functions. You can install them all as dependencies with the following command:



Create a Post Editor Component

Create a `PostEditor` component which will be used in the Post Manager page. For now, the posts will just have `title` and `body` fields.

src/components/PostEditor.js

```
  Card,
  CardContent,
  CardActions,
  Modal,
  Button,
  TextField,
} from '@material-ui/core';
import { compose } from 'recompose';
import { withRouter } from 'react-router-dom';
import { Form, Field } from 'react-final-form';

const styles = theme => ({
  modal: {
    display: 'flex',
    alignItems: 'center',
    justifyContent: 'center',
  },
  modalCard: {
    width: '90%',
    maxWidth: 500,
  },
  modalCardContent: {
    display: 'flex',
    flexDirection: 'column',
  },
  marginTop: {
    marginTop: theme.spacing(2),
  },
});

const PostEditor = ({ classes, post, onSave, history }) => (
  <Form initialValues={post} onSubmit={onSave}>
    {({ handleSubmit }) => (
      <Modal
        className={classes.modal}
        onClose={() => history.goBack()}
        open
      >
        <Card className={classes.modalCard}>
          <form onSubmit={handleSubmit}>
            <CardContent className={classes.modalCardContent}>
              <Field name="title">
                {({ input }) => <TextField label="Title" autoFocus {...input} />}
              </Field>
              <Field name="body">
                {({ input }) => (
                  <TextField
                    className={classes.marginTop}
                    label="Body"
                    multiline
                    rows={4}
                    {...input}
                  />
                )}
              </Field>
```

```
            <Button size="small" onClick={() => history.goBack()}>Cancel</Button>
          </CardActions>
        </form>
      </Card>
    </Modal>
  )}
</Form>
);

export default compose(
  withRouter,
  withStyles(styles),
)(PostEditor);
```

Add Friendly Error Messages

In some circumstances, you may run into an issue with the server not working as expected. For example, if you don't have sqlite on your machine, the posts may fail to load and it could be unclear as to why. A way to help debug is to display a message to the user when you get an error. Material Design has a concept called a Snackbar, where messages will show up at the bottom of the screen.

The Snackbar component expects a unique ID for your rendered component. In order to create one, you can use the `uuid` package. Go ahead and install that now:

```
yarn add uuid@3.3.2
```

Now create a new file `src/components/ErrorSnackbar.js`:

src/components/ErrorSnackbar.js

```
  withStyles,
  Snackbar,
  SnackbarContent,
  IconButton,
} from '@material-ui/core';
import { Error as ErrorIcon, Close as CloseIcon } from '@material-ui/icons';
import { compose, withState } from 'recompose';
import uuid from 'uuid/v4';

const styles = theme => ({
  snackbarContent: {
    backgroundColor: theme.palette.error.dark,
  },
  message: {
    display: 'flex',
    alignItems: 'center',
  },
  icon: {
    fontSize: 20,
  },
  iconVariant: {
    opacity: 0.9,
    marginRight: theme.spacing(1),
  },
});

const ErrorSnackbar = ({ id, message, onClose, classes }) => (
  <Snackbar
    open
    autoHideDuration={6000}
    onClose={onClose}
  >
    <SnackbarContent
      className={`${classes.margin} ${classes.snackbarContent}`}
      aria-describedby={id}
      message={
        <span id={id} className={classes.message}>
          <ErrorIcon className={`${classes.icon} ${classes.iconVariant}`} />
          {message}
        </span>
      }
      action={[
        <IconButton key="close" aria-label="Close" color="inherit" onClick={onClose}>
          <CloseIcon className={classes.icon} />
        </IconButton>
      ]}
    />
  </Snackbar>
);

export default compose(
  withState('id', 'setId', uuid),
  withStyles(styles),
)(ErrorSnackbar);
```

Create the Posts Manager Page Component

You'll also need a page to render a list of posts, and to inject the post editor. Create a new file `src/pages/PostsManager.js`. Once again, I'll post the whole file then walk you through each section.

src/pages/PostsManager.js

```
import {
  withStyles,
  Typography,
  Fab,
  IconButton,
  Paper,
  List,
  ListItem,
  ListItemText,
  ListItemSecondaryAction,
} from '@material-ui/core';
import { Delete as DeleteIcon, Add as AddIcon } from '@material-ui/icons';
import moment from 'moment';
import { find, orderBy } from 'lodash';
import { compose } from 'recompose';

import PostEditor from '../components/PostEditor';
import ErrorSnackbar from '../components/ErrorSnackbar';

const styles = theme => ({
  posts: {
    marginTop: theme.spacing(2),
  },
  fab: {
    position: 'absolute',
    bottom: theme.spacing(3),
    right: theme.spacing(3),
    [theme.breakpoints.down('xs')]: {
      bottom: theme.spacing(2),
      right: theme.spacing(2),
    },
  },
});

const API = process.env.REACT_APP_API || 'http://localhost:3001';

class PostsManager extends Component {
  state = {
    loading: true,
    posts: [],
    error: null,
  };

  componentDidMount() {
    this.getPosts();
  }

  async fetch(method, endpoint, body) {
    try {
      const response = await fetch(`${API}${endpoint}`, {
        method,
        body: body && JSON.stringify(body),
        headers: {
          'content-type': 'application/json',
          accept: 'application/json',
```

```
    return await response.json();
  } catch (error) {
    console.error(error);

    this.setState({ error });
  }
}

async getPosts() {
  this.setState({ loading: false, posts: (await this.fetch('get', '/posts')) || [] });
}

savePost = async (post) => {
  if (post.id) {
    await this.fetch('put', `/posts/${post.id}`, post);
  } else {
    await this.fetch('post', '/posts', post);
  }

  this.props.history.goBack();
  this.getPosts();
}

async deletePost(post) {
  if (window.confirm(`Are you sure you want to delete "${post.title}"`)) {
    await this.fetch('delete', `/posts/${post.id}`);
    this.getPosts();
  }
}

renderPostEditor = ({ match: { params: { id } } }) => {
  if (this.state.loading) return null;
  const post = find(this.state.posts, { id: Number(id) });

  if (!post && id !== 'new') return <Redirect to="/posts" />;

  return <PostEditor post={post} onSave={this.savePost} />;
};

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

  return (
    <Fragment>
      <Typography variant="h4">Posts Manager</Typography>
      {this.state.posts.length > 0 ? (
        <Paper elevation={1} className={classes.posts}>
          <List>
            {orderBy(this.state.posts, ['updatedAt', 'title'], ['desc', 'asc']).map(post => (
              <ListItem key={post.id} button component={Link} to={`/posts/${post.id}`}>
                <ListItemText
                  primary={post.title}
                  secondary={post.updatedAt && `Updated ${moment(post.updatedAt).fromNow()}`}
                />
                <ListItemSecondaryAction>
                  <IconButton onClick={() => this.deletePost(post)} color="inherit">
```

```
                </ListItem>
              ))}
            </List>
          </Paper>
        ) : (
          !this.state.loading && <Typography variant="subtitle1">No posts to display</Typograph
        )}
        <Fab
          color="secondary"
          aria-label="add"
          className={classes.fab}
          component={Link}
          to="/posts/new"
        >
          <AddIcon />
        </Fab>
        <Route exact path="/posts/:id" render={this.renderPostEditor} />
        {this.state.error && (
          <ErrorSnackbar
            onClose={() => this.setState({ error: null })}
            message={this.state.error.message}
          />
        )}
      </Fragment>
    );
  }
}

export default compose(
  withOktaAuth,
  withRouter,
  withStyles(styles),
)(PostsManager);
```

The backend is set to run on port 3001 on your local machine by default, so this sets that as a fallback. However, if you'd like to run this on another server, or on another port, you'll need a way to edit that. You could run the app with `API=https://api.example.com yarn start:web` to override this.

```
const API = process.env.REACT_APP_API || 'http://localhost:3001';
```

When the component first mounts, you won't have any data yet. You may want some indicator that the page is still loading, so setting the state to `loading: true` lets you know that later on. Setting the initial posts to an empty array makes the code simpler later since you can just always assume you have an array, even if it's empty. Then you'll want to fetch the set of posts as soon as the component mounts.

```
    loading: true,
    posts: [],
    error: null,
  };

  componentDidMount() {
    this.getPosts();
  }

  // ...
}
```

Here you're setting up a simple helper function to send a request to the server. This uses the `fetch` function that's built into all modern browsers. The helper accepts a `method` (e.g. `get`, `post`, `delete`), an `endpoint` (here it would either be `/posts` or a specific post like `/posts/3`), and a `body` (some optional JSON value, in this case the post content).

This also sets some headers to tell the backend that any body it sends will be in JSON format, and it sets the authorization header by fetching the access token from Okta.

If you get an error, setting it in the state will allow you to display it in the Snackbar.

```
class PostsManager extends Component {
  // ...

  async fetch(method, endpoint, body) {
    try {
      const response = await fetch(`${API}${endpoint}`, {
        method,
        body: body && JSON.stringify(body),
        headers: {
          'content-type': 'application/json',
          accept: 'application/json',
          authorization: `Bearer ${await this.props.authService.getAccessToken()}`,
        },
      });
      return await response.json();
    } catch (error) {
      console.error(error);

      this.setState({ error });
    }
  }

  // ...
}
```

You have one function to fetch posts (`getPosts`), which will also set `loading` to `false` since it's the function that gets called when the component first loads.

There's another function to save posts, which handles the case of adding a new post as well as modifying an existing post. Since the posts will be loaded in a modal based on the route, once the post is updated the browser is told to go back to `/posts` .

The last function is to delete a post. The `confirm` function actually blocks the UI, so it's not normally recommended for an app like this, but it works well for demo purposes. It's a built-in browser function that simply gives a popup asking you to confirm, and returns either `true` or `false` depending on your answer.

After saving or deleting a post, the `getPosts` command is called again to make sure that all the posts are up to date.

```
class PostsManager extends Component {
  // ...

  async getPosts() {
    this.setState({ loading: false, posts: await this.fetch('get', '/posts') });
  }

  savePost = async (post) => {
    if (post.id) {
      await this.fetch('put', `/posts/${post.id}`, post);
    } else {
      await this.fetch('post', '/posts', post);
    }

    this.props.history.goBack();
    await this.getPosts();
  }

  async deletePost(post) {
    if (window.confirm(`Are you sure you want to delete "${post.title}"`)) {
      await this.fetch('delete', `/posts/${post.id}`);
      await this.getPosts();
    }
  }

  // ...
}
```

The `renderPostEditor` function will be passed into a `Route` so that it only renders when you're looking at a specific post. If you're still loading posts, you won't want to render anything just yet, so you can just return `null` . After the posts are loaded, you can use the

already deleted).

The only exception is for a special route `/posts/new`, which will be used to create a new post. In that case, you don't want to redirect. Now that you have a post model, you can render the `PostEditor` component from above and pass the model to it to render in a modal.

```
class PostsManager extends Component {
  // ...

  renderPostEditor = ({ match: { params: { id } } }) => {
    if (this.state.loading) return null;
    const post = find(this.state.posts, { id: Number(id) });

    if (!post && id !== 'new') return <Redirect to="/posts" />;

    return <PostEditor post={post} onSave={this.savePost} />;
  };

  // ...
}
```

Here is the main render function. When there are no posts, it should display a message "No posts to display", except when the posts are still loading. You could choose to render a loading symbol, but for now just rendering nothing will suffice.

When there are posts, it renders a simple list of them, with the main text being the title of the post, and some subtext saying when it was last updated. The updated text uses `moment` for rendering a user-friendly string like `10 minutes ago` instead of the raw timestamp.

By adding `component={Link}` and the `to` value, you are actually turning the list item into a link that takes you to the path of the post (e.g. `/posts/5`). You can do the same to send you to create a new post, by creating the Floating Action Button (FAB) that you see on many Material Design apps.

The ErrorSnackbar will display any error messages you receive when fetching posts.

```
  render() {
    const { classes } = this.props;

    return (
      <Fragment>
        <Typography variant="h4">Posts Manager</Typography>
        {this.state.posts.length > 0 ? (
          <Paper elevation={1} className={classes.posts}>
            <List>
              {orderBy(this.state.posts, ['updatedAt', 'title'], ['desc', 'asc']).map(post => (
                <ListItem key={post.id} button component={Link} to={`/posts/${post.id}`}>
                  <ListItemText
                    primary={post.title}
                    secondary={post.updatedAt && `Updated ${moment(post.updatedAt).fromNow()}`}
                  />
                  <ListItemSecondaryAction>
                    <IconButton onClick={() => this.deletePost(post)} color="inherit">
                      <DeleteIcon />
                    </IconButton>
                  </ListItemSecondaryAction>
                </ListItem>
              ))}
            </List>
          </Paper>
        ) : (
          !this.state.loading && <Typography variant="subtitle1">No posts to display</Typograph
        )}
        <Fab
          color="secondary"
          aria-label="add"
          className={classes.fab}
          component={Link}
          to="/posts/new"
        >
          <AddIcon />
        </Fab>
        <Route exact path="/posts/:id" render={this.renderPostEditor} />
        {this.state.error && (
          <ErrorSnackbar
            onClose={() => this.setState({ error: null })}
            message={this.state.error.message}
          />
        )}
      </Fragment>
    );
  }
}
```

In order to get access to the Okta SDK, you need to use the `withOktaAuth()` HOC again. This time there are actually a few other HOCs to add, so you can use a utility function called `compose` from to wrap your component with multiple HOCs.

```
  withRouter,
  withStyles(styles),
)(PostsManager);
```

Add the Route and Navigation Links

OK, you're in the home stretch now. You just need to tell the app when to render the Posts Manager page, and a link to get there.

Add the `PostsManager` page to `src/App.js`. Okta provides a `SecureRoute` component which is an extension of React Router's `Route` component. This will ensure that if you try to go to that page and aren't logged in, you'll be redirected to sign in. If you're on that page and you sign out, you'll be redirected home.

src/App.js

```
--- a/src/App.js
+++ b/src/App.js
@@ -1,6 +1,6 @@
 import React, { Fragment } from 'react';
 import { Route } from 'react-router-dom';
-import { LoginCallback } from '@okta/okta-react';
+import { SecureRoute, LoginCallback } from '@okta/okta-react';
 import {
   CssBaseline,
   withStyles,
@@ -8,6 +8,7 @@ import {

 import AppHeader from './components/AppHeader';
 import Home from './pages/Home';
+import PostsManager from './pages/PostsManager';

 const styles = theme => ({
   main: {
@@ -24,6 +25,7 @@ const App = ({ classes }) => (
     <AppHeader />
     <main className={classes.main}>
       <Route exact path="/" component={Home} />
+      <SecureRoute path="/posts" component={PostsManager} />
       <Route path="/login/callback" component={LoginCallback} />
     </main>
   </Fragment>
```

src/components/AppHeader.js

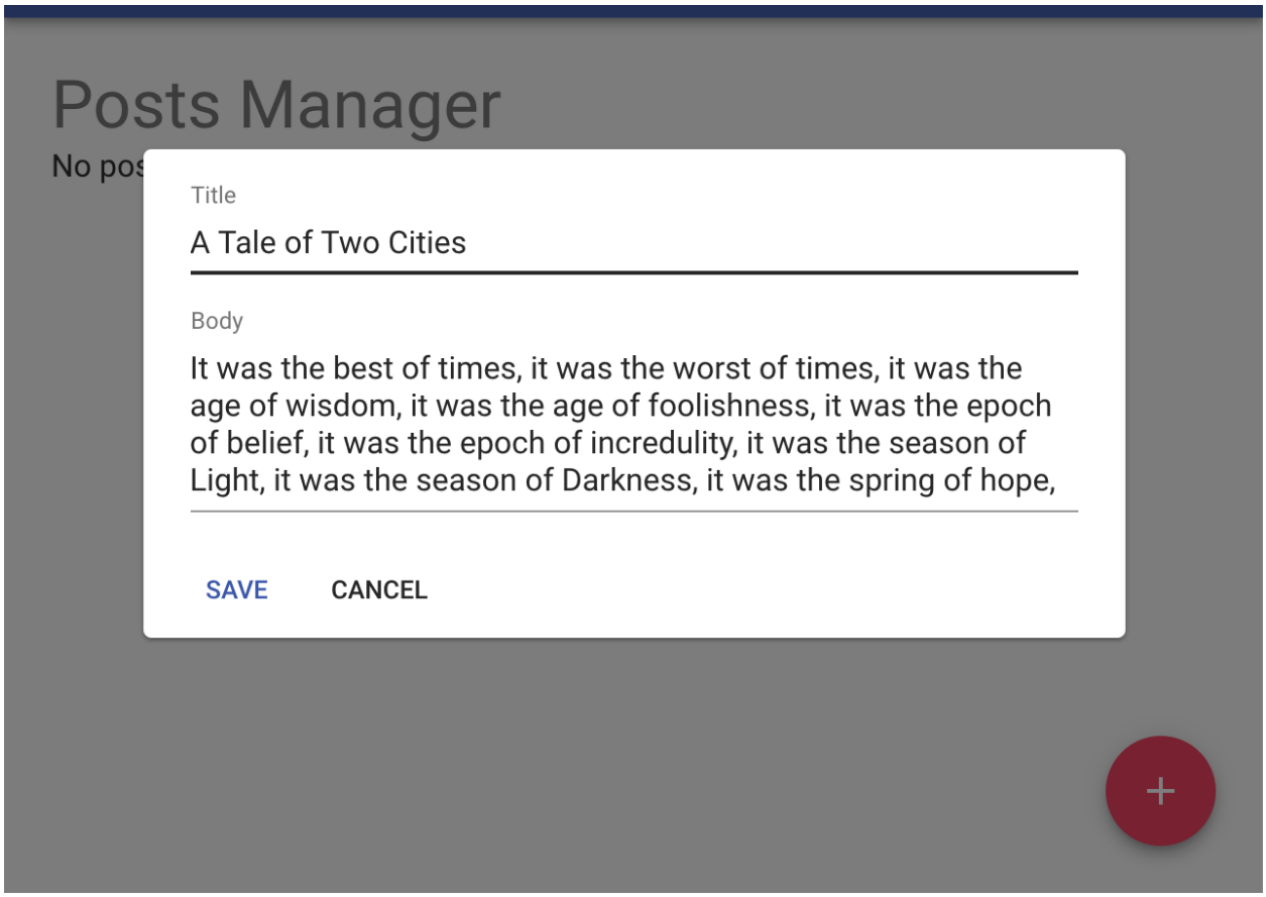
```
--- a/src/components/AppHeader.js
+++ b/src/components/AppHeader.js
@@ -1,6 +1,8 @@
 import React from 'react';
+import { Link } from 'react-router-dom';
 import {
   AppBar,
+  Button,
   Toolbar,
   Typography,
   withStyles,
@@ -20,6 +22,8 @@ const AppHeader = ({ classes }) => (
         <Typography variant="h6" color="inherit">
           My React App
         </Typography>
+        <Button color="inherit" component={Link} to="/">Home</Button>
+        <Button color="inherit" component={Link} to="/posts">Posts Manager</Button>
         <div className={classes.flex} />
         <LoginButton />
       </Toolbar>
```

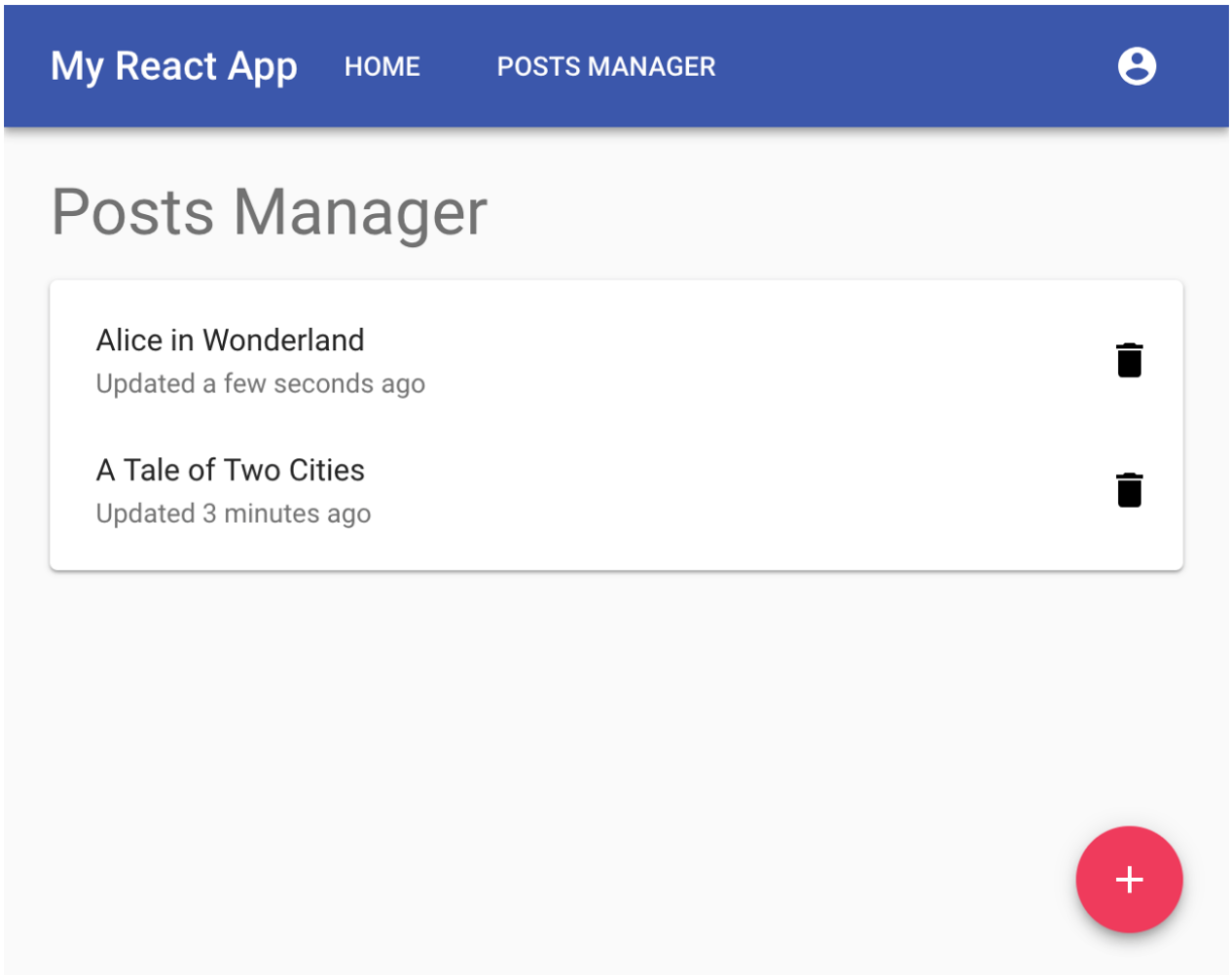
Test your React + Node CRUD App

You now have a fully functioning Single Page App, connected to a REST API server, secured with authentication via Okta's OIDC.

Go ahead and test out the app now. If they're not already running, make sure to start the server and the frontend. In your terminal run `yarn start` from your project directory.

Navigate to `http://localhost:3000`. You should be able to add, edit, view, and delete posts to your heart's desire!


Posts Manager
No pos
Title
A Tale of Two Cities
Body
It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope,
SAVE
CANCEL


My React App
HOME
POSTS MANAGER
Posts Manager
Alice in Wonderland
Updated a few seconds ago
A Tale of Two Cities
Updated 3 minutes ago

Learn More About React, Node, and Okta

Hopefully you found this article helpful. If you're new to React, maybe you're one step closer to deciding whether you love it or hate it. If you're a React veteran, maybe you found out how easy it can be to add authentication to a new or existing app. Or maybe you learned a bit about Node.

If you'd like to view the source code for the example application in this post, you can find it at https://github.com/oktadeveloper/okta-react-node-example.

If you're still aching for more content, there is a plethora of great posts on the Okta developer blog. This post was not-so-loosely based on Build a Basic CRUD App with Vue.js and Node, which I would definitely recommend checking out if you're interested in learning more about Vue.js. Here are some other great articles to check out as well:

- The Ultimate Guide to Progressive Web Applications
- Build User Registration with Node, React, and Okta
- Build a React Application with User Authentication in 15 Minutes
- Build a React Native Application and Authenticate with OAuth 2.0
- Tutorial: Build a Basic CRUD App with Node.js

And as always, we'd love to hear from you. Hit us up with questions or feedback in the comments, or on Twitter @oktadev.

Changelog:

- Apr 5, 2021: Updated to use Okta JWT Verifier v2.1.0 and Finale instead of Epilogue. You can see the changes in the example app or in this blog post.
- Oct 22, 2020: Updated to use Okta React v3.0.8. You can see the changes in the example app or in this blog post.
- Nov 1, 2019: Added an error snackbar to help with debugging, added some information about installing SQLite, and updated a majority of the dependencies. Changes to this post can be viewed in okta-blog#58.

 Braden Kelley

---

Okta Developer Blog Comment Policy

We welcome relevant and respectful comments. Off-topic comments may be removed.