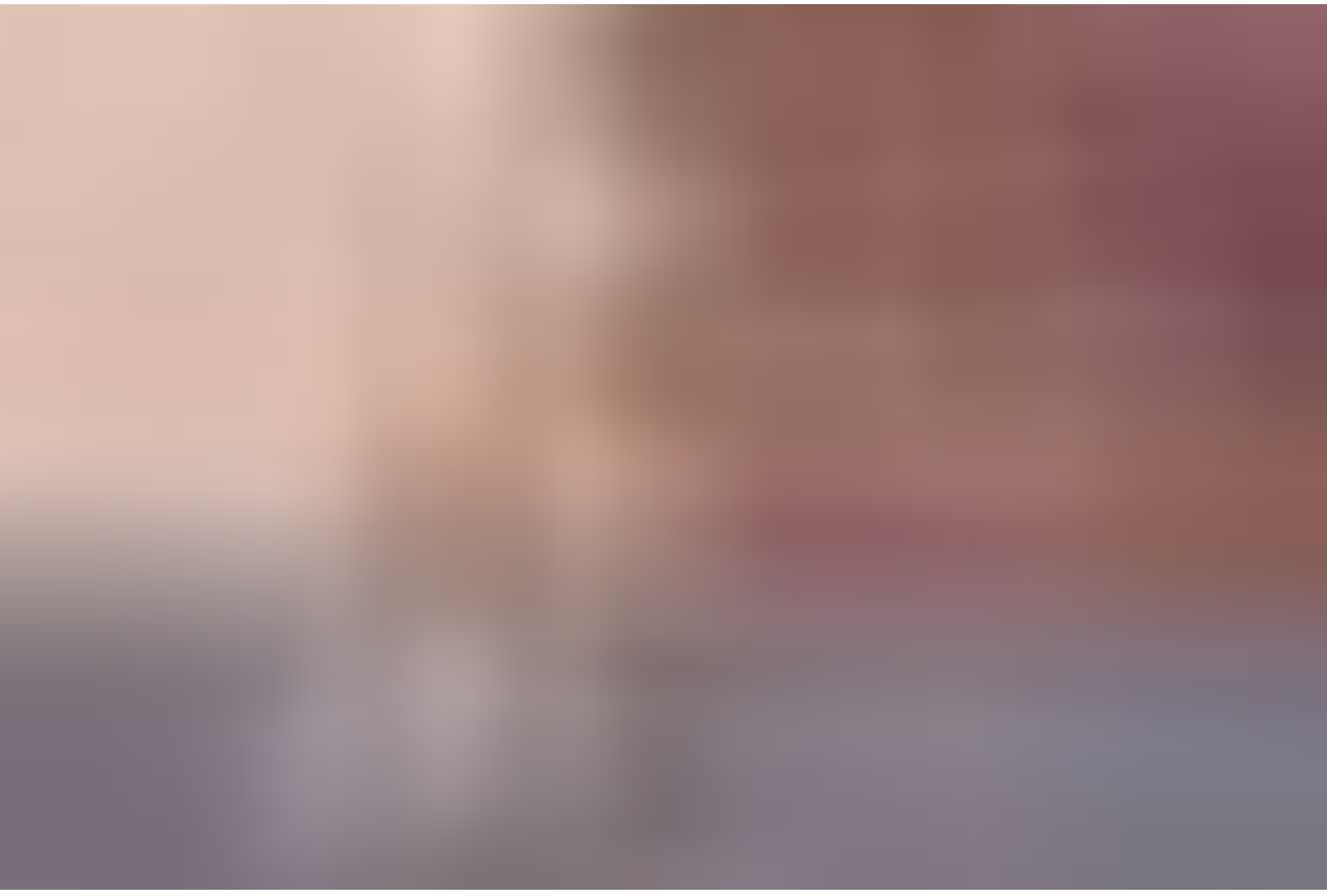
Aug 31, 2017 · 8 min read

Full stack Django: Quick start with JWT auth and React/Redux (Part II)



In the previous part, we created a simple django back-end with JSON Web token authentication. Now we going to make a frontend with React and Redux

To work over first part of the tutorial, you need to have nodejs, npm, and create-react-app installed.

Redux is a library helps to organize predictable, one-way data flow in a front-end application. Everything starts from a single source of truth—a store. A store could be modified only by firing actions. Actions go through Redux middleware down to the reducer functions. Reducer functions produce a new state. And then React library using Virtual DOM reconcile UI according to this new state.

If you have experience in the desktop UI development, the Redux naming could sound unnatural. Try to think about “actions” as “events”, “reducers” as “listeners”, and when later we meed with `mapStateToProps` function, it literally means “getComponentPropertiesFromState”

You can read more about Redux in it’s awesome documentation

Let’s start

```
$ mkdir frontend && cd frontend
$ create-react-app .
```

After a minute, we have a bare simple React application.

Choosing the front-end stack

React with Redux are the most trending libraries for the front-end development. To create a full-featured application we need to add few more dependencies.

Let’s choose also well-known and well-supported libraries

```
$ npm install --save redux react-redux
$ npm install --save react-router-redux@next history
$ npm install --save redux-persist@5.0.0-beta.7
$ npm install --save redux-persist-transform-filter
$ npm install --save redux-api-middleware@2.0.0-beta.3
```

```
$ npm install --save jwt-decode
$ npm install --save reactstrap react-transition-
group@1.1.2
$ npm install --save bootstrap@4.0.0-beta
```

React-router is the de facto standard router implementation for the React from ReactTraining. We installed Redux binding for it.

We going to keep JWT tokens in the Redux state, and persist the state with the redux-persist library, that has awesome API, multiple storage backends support and over 3k stars on GitHub. redux-persist-transform-filter used to whitelist subset of application state for persistence. Keeping tokens in the localStorage could be unsafe if your application includes javascript from other domains (including CDN). But it's the typical case for a complex Intranet software. And as the advantage of localStorage, we should not worry about CSRF.

Redux itself have no option to call an external API. We going to use redux-api-middleware that implements interface very similar to the redux/real-world example, but available as the package on npm. redux-api-middleware designed to do one thing only—to call an external API. That is the place we could put all our JWT Workflow logic.

jwt-decode is the lightweight JWT token decode library designed specifically for the browsers by Auth0 team.

And to make the UI shine, let's use reactstrap the bootstrap 4 library bindings for React.

Initial setup

First of all we need to combine all these libraries to play together. Start with simple new `reducers/index.js` file

```
import { combineReducers } from 'redux'
import { routerReducer } from 'react-router-redux'

export default combineReducers({
  router: routerReducer
})
```

That declares a root reducer that keeps a router state.

Now we can create a Redux store in `store.js`

```
import storage from 'redux-persist/es/storage'
import { apiMiddleware } from 'redux-api-middleware';
import { applyMiddleware, createStore } from 'redux'
import { createFilter   } from 'redux-persist-transform-
filter';
import { persistReducer, persistStore } from 'redux-
persist'
import { routerMiddleware } from 'react-router-redux'
import rootReducer from './reducers'


export default (history) => {
  const persistedFilter = createFilter(
    'auth', ['access', 'refresh']);


  const reducer = persistReducer(
    {
      key: 'polls',
      storage: storage,
      whitelist: ['auth'],
      transforms: [persistedFilter]
    },
    rootReducer)


  const store = createStore(
    reducer, {},
    applyMiddleware(
      apiMiddleware,
      routerMiddleware(history))
  )


  persistStore(store)


  return store
}
```

We just follow all libraries introductions and configure store with `redux-api-middleware` and persistence of the `store.auth` in a browser `localStorare`

Now we can modify `index.js` to make it work

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

```
import createHistory from 'history/createBrowserHistory'
import { ConnectedRouter } from 'react-router-redux'
import { Provider } from 'react-redux'

import './index.css';
import 'bootstrap/dist/css/bootstrap.css';
import App from './App';
import configureStore from './store'

const history = createHistory()
const store = configureStore(history)

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

We got bootstrap css imported, instantiate the Redux store and setup react-router.

You can run `npm start` at that time, and see the initial welcome page.

We finished with libraries boilerplate setup and can start working on the authentication

Authentication

First of all, we need two redux actions, one for login and one for token refresh in `actions/auth.js`

```
import { RSAA } from 'redux-api-middleware';

export const LOGIN_REQUEST = '@@auth/LOGIN_REQUEST';
export const LOGIN_SUCCESS = '@@auth/LOGIN_SUCCESS';
export const LOGIN_FAILURE = '@@auth/LOGIN_FAILURE';

export const TOKEN_REQUEST = '@@auth/TOKEN_REQUEST';
export const TOKEN_RECEIVED = '@@auth/TOKEN_RECEIVED';
export const TOKEN_FAILURE = '@@auth/TOKEN_FAILURE';

export const login = (username, password) => ({
  [RSAA]: {
    endpoint: '/api/auth/token/obtain/',
    method: 'POST',
```

```
    body: JSON.stringify({username, password}),
    headers: { 'Content-Type': 'application/json' },
    types: [
      LOGIN_REQUEST, LOGIN_SUCCESS, LOGIN_FAILURE
    ]
  }
})


export const refreshAccessToken = (token) => ({
  [RSAA]: {
    endpoint: '/api/auth/token/refresh/',
    method: 'POST',
    body: JSON.stringify({refresh: token}),
    headers: { 'Content-Type': 'application/json' },
    types: [
      TOKEN_REQUEST, TOKEN_RECEIVED, TOKEN_FAILURE
    ]
  }
})
```

That's how typical API calling actions looks like with `redux-api-middleware` The middleware would perform an actual call, and dispatch REQUEST/SUCCESS/FAILURE actions to the redux middleware pipeline down to reducer functions.

So, lets' create `reducers/auth.js`

```
import jwtDecode from 'jwt-decode'
import * as auth from '../actions/auth'


const initialState = {
  access: undefined,
  refresh: undefined,
  errors: {},
}


export default (state=initialState, action) => {
  switch(action.type) {
    case auth.LOGIN_SUCCESS:
      return {
        access: {
          token: action.payload.access,
          ...jwtDecode(action.payload.access)
        },
        refresh: {
          token: action.payload.refresh,
          ...jwtDecode(action.payload.refresh)
        },
        errors: {}
    }
    case auth.TOKEN_RECEIVED:
```

```
      return {
        ...state,
        access: {
          token: action.payload.access,
          ...jwtDecode(action.payload.access)
        }
      }
    case auth.LOGIN_FAILURE:
    case auth.TOKEN_FAILURE:
      return {
         access: undefined,
         refresh: undefined,
         errors:
             action.payload.response ||
                {'non_field_errors':
action.payload.statusText},
      }
    default:
      return state
    }
}
```

We keep in a state original Access and Refresh tokens, and decoded tokens payload. Let's add few state access methods to the same file

```
export function accessToken(state) {
    if (state.access) {
        return  state.access.token
    }
}

export function refreshToken(state) {
    if (state.refresh) {
        return  state.refresh.token
    }
}

export function isAccessTokenExpired(state) {
  if (state.access && state.access.exp) {
    return 1000 * state.access.exp - (new Date()).getTime()
< 5000
  }
  return true
}


export function isRefreshTokenExpired(state) {
  if (state.refresh && state.refresh.exp) {
    return 1000 * state.refresh.exp - (new
Date()).getTime() < 5000
  }
  return true
}


export function isAuthenticated(state) {
```

```
  return !isRefreshTokenExpired(state)
}


export function errors(state) {
   return  state.errors
}
```

Now we can include auth reducer to the root and export auth selectors functions in the `reducers/index.js`

```
import { combineReducers } from 'redux'
import { routerReducer } from 'react-router-redux'
import auth, * as fromAuth from './auth.js'


export default combineReducers({
  auth: auth,
  router: routerReducer
})


export const isAuthenticated =
 state => fromAuth.isAuthenticated(state.auth)
export const accessToken =
  state => fromAuth.accessToken(state.auth)
export const isAccessTokenExpired =
  state => fromAuth.isAccessTokenExpired(state.auth)
export const refreshToken =
  state => fromAuth.refreshToken(state.auth)
export const isRefreshTokenExpired =
  state => fromAuth.isRefreshTokenExpired(state.auth)
export const authErrors =
  state => fromAuth.errors(state.auth)
```

Login Form

Let's start work on the UI. First of all, we need a simple TextInput component in the `components/TextInput.js`

```
import React from 'react'
import { FormGroup, FormFeedback, Label, Input } from
'reactstrap';


export default ({name, label, error, type, ...rest}) => {
  const id = `id_${name}`,
        input_type = type?type:"text"
  return (
```

```
    <FormGroup color={error?"danger":""}>
      {label?<Label htmlFor={id}>{label}</Label>: ""}
      <Input type={input_type} name={name}
             id={id} className={error?"is-invalid":""}
             {...rest} />
      {error?
         <FormFeedback className="invalid-feedback">
           {error}
         </FormFeedback>
         : ""
      }
    </FormGroup>
  )
}
```

TextInput component provides an input box and renders field errors like we have in Django Forms. We can use this component to construct a `components/LoginForm.js`

```
import React, {Component} from 'react'
import { Alert, Button, Jumbotron,  Form } from
'reactstrap';
```

```
import TextInput from './TextInput'
```

```
export default class LoginForm extends Component {
  state = {
    username: '',
    password: ''
  }
```

```
  handleInputChange = (event) => {
    const target = event.target,
          value = target.type ===
            'checkbox' ? target.checked : target.value,
          name = target.name
```

```
    this.setState({
      [name]: value
    });
  }
```

```
  onSubmit = (event) => {
    event.preventDefault()
    this.props.onSubmit(this.state.username,
this.state.password)
  }
```

```
  render() {
    const errors = this.props.errors || {}
```

```
    return (
      <Jumbotron className="container">
        <Form onSubmit={this.onSubmit}>
          <h1>Authentication</h1>
          {
           errors.non_field_errors?
             <Alert color="danger">
                {errors.non_field_errors}
             </Alert>:""
          }
          <TextInput name="username" label="Username"
                     error={errors.username}
                     onChange={this.handleInputChange}
          <TextInput name="password" label="Password"
                     error={errors.password}
type="password"
                     onChange={this.handleInputChange}/>
          <Button type="submit" color="primary" size="lg"}>
              Log In
          </Button>
        </Form>
      </Jumbotron>
    )
  }
}
```

In the `LoginForm`, we keep username and password in a local state, without persisting it. Both `TextInput` and `LoginForm` are presentation components and have no connection to the Redux. We followed good practice to keep representational components separated.

Now we can create actual connected `Login` page container component in the `containers/Login.js`

```
import React from 'react'
import { connect } from 'react-redux'
import { Redirect } from 'react-router'
```

```
import LoginForm from '../components/LoginForm'
import {login} from  '../actions/auth'
import {authErrors, isAuthenticated} from '../reducers'


const Login = (props) => {
  if(props.isAuthenticated) {
    return (
      <Redirect to='/' />
    )
  } else {
    return (....
      <div className="login-page">
        <LoginForm {...props}/>
```

```
      </div>
    )
}
```

```
const mapStateToProps = (state) => ({
  errors: authErrors(state),
  isAuthenticated: isAuthenticated(state)
})
```

```
const mapDispatchToProps = (dispatch) => ({
  onSubmit: (username, password) => {
    dispatch(login(username, password))
  }
})

export default connect(mapStateToProps, mapDispatchToProps)
(Login);
```

Adding a little `index.css` can make everything looks better.

Running altogether

We almost ready to check our login functionality. Let's make a whole application available for authenticated users only. To do that, we need simple `containers/PrivateRoute.js` component

```
import React from 'react'
import { Route, Redirect } from 'react-router'
import { connect } from 'react-redux'
import * as reducers from '../reducers'
```

```
const PrivateRoute = ({ component: Component,
isAuthenticated, ...rest }) => (
  <Route {...rest} render={props => (
    isAuthenticated ? (
      <Component {...props}/>
    ) : (
      <Redirect to={{
        pathname: '/login',
        state: { from: props.location }
      }}/>
    )
  )}/>
)
```

```
const mapStateToProps = (state) => ({
  isAuthenticated: reducers.isAuthenticated(state)
})
```

```
export default connect(mapStateToProps, null)
(PrivateRoute);
```

The component performs authentication status check, and if user are not logged in, redirect to the login page.

Now open our main `src/index.js` and bind all together

```
...
import {Route, Switch} from 'react-router'

import Login from './containers/Login';
import PrivateRoute from './containers/PrivateRoute';

...

ReactDOM.render((
  <Provider store={store}>
    <ConnectedRouter history={history}>
      <Switch>
        <Route exact path="/login/" component={Login} />
        <PrivateRoute path="/" component={App}/>
      </Switch>
    </ConnectedRouter>
  </Provider>
), document.getElementById('root'));
```

Instead of just rendering an `<App/>` now we switch between login and app components depends on the application URL.

To allow the frontend to connect to our backend application, add proxy settings to the `package.json` file

```
{
  "name": "frontend",
  "version": "0.1.0",
  ....
  "proxy": "http://localhost:8000/"
}
```

It's time to run `npm start` again!

You can check that UI properly respond to the invalid inputs, then put right credentials and login.

We can open browser developer tools, and see that Access and Refresh tokens saved in the localStorage

It’s time to call our first protected API endpoint and implement JWT refresh token workflow. Go to the Part III