Aug 31, 2017 · 4 min read

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

In the previous two parts, we created a simple backend and frontend with user authentication.

In this final part, we would perform our first authenticated call to the API and implement JWT Refresh Token workflow.

Calling an API

Let’s create a simple helper, that would add `Authentication` header with token from our state.

`reducers/index.js`

```
...
export function withAuth(headers={}) {
  return (state) => ({
    ...headers,
    'Authorization': `Bearer ${accessToken(state)}`
  })
}
```

Now we can create our first API action, in a new file `actions/echo.js`

```
import { RSAA } from 'redux-api-middleware';
import { withAuth } from '../reducers'


export const ECHO_REQUEST = '@@echo/ECHO_REQUEST';
export const ECHO_SUCCESS = '@@echo/ECHO_SUCCESS';
export const ECHO_FAILURE = '@@echo/ECHO_FAILURE';


export const echo = (message) => ({
  [RSAA]: {
      endpoint: '/api/echo/',
      method: 'POST',
      body: JSON.stringify({message: message}),
      headers: withAuth({ 'Content-Type':
'application/json' }),
      types: [
        ECHO_REQUEST, ECHO_SUCCESS, ECHO_FAILURE
      ]
  }
})
```

When action came into `redux-api-middleware` the headers function would be called, and the `Authorization` header would be expanded.

Save the API call result in the store with the simple reducer:

`reducers/echo.js`

```
import * as echo from '../actions/echo'

const initialState = {
  message: ""
}

export default (state=initialState, action) => {
  switch(action.type) {
    case echo.ECHO_SUCCESS:
      return {
        message: action.payload.message
      }
    default:
      return state
  }
}

export const serverMessage = (state) => state.message
```

Let’s call it. The simplest thing that we can do, is to perform API calls on `componentDidMount` just after when a user goes into the App page.

`App.js`

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

import {echo} from './actions/echo'
import {serverMessage} from './reducers'

class App extends Component {
  componentDidMount() {
      this.props.fetchMessage('Hi!')
  }

render() {
    return (
      <div>
        <h2>Welcome to React</h2>
        <p>{this.props.message}</p>
      </div>
    );
  }
}

export default connect(
  state => ({ message: serverMessage(state) }),
    { fetchMessage: echo }
  )(App);
```

We connect the App component with the state. As soon as component became mounted, the `echo` action would go to the `redux-api-middleware` that would perform the actual API call. On complete, `ECHO_SUCCEED` message would be dispatch down to the `echo` reducer. The message would be saved in the `state.echo.message` that would cause UI reconciliation.

Everything would work fine, till the Access token became expired. Just imagine the situation, when a user opens an application after a while. A user has a valid Refresh token, and the Access token already expired. API requests fire immediately from different components of an application. But we can’t execute them right now, till Access token became updated.

JWT Workflow

So, here is the trick. We going to wrap `redux-api-middleware` into out custom middleware, that postpone actions till valid access token received

```
import { isRSAA, apiMiddleware } from 'redux-api-
middleware';

import { TOKEN_RECEIVED, refreshAccessToken } from
'./actions/auth'
import { refreshToken, isAccessTokenExpired } from
'./reducers'

export function createApiMiddleware() {
  let postponedRSAAs = []

  return ({ dispatch, getState }) => {
    const rsaaMiddleware = apiMiddleware({dispatch,
getState})

return (next) => (action) => {
      const nextCheckPostoned = (nextAction) => {
        // Run postponed actions after token refresh
        if (nextAction.type === TOKEN_RECEIVED) {
          next(nextAction);
          postponedRSAAs.forEach((postponed) => {
            rsaaMiddleware(next)(postponed)
          })
          postponedRSAAs = []
        } else {
          next(nextAction)
        }
      }

      if(isRSAA(action)) {
        const state = getState(),
              token = refreshToken(state)

        if(token && isAccessTokenExpired(state)) {
          postponedRSAAs.push(action)
          if(postponedRSAAs.length === 1) {
            const action = refreshAccessToken(token)
            return rsaaMiddleware(nextCheckPostoned)
(action)
          } else {
            return
          }
        }
        return rsaaMiddleware(next)(action);
      }
      return next(action);
    }
  }
}

export default createApiMiddleware();
```

If everything is fine, we just delegate an action to the `rsaaMiddleware(next)`

But if we have a Refresh token and our Access token is expired, the action goes to the `postponedRSAAs` list, and `refresAccessToken` action fired instead. Since all middlewares works like a chain, and there is no loop in message processing, we catch `TOKEN_RECEIVED` in the custom `next-CheckPostponed` function that passed as `next` parameter to the `redux-api-middleware`

As soon as new Access Token received, we could fire postponed actions.

Now you can just replace an import `apiMiddleware` in the `store.js` to make everything works

```
import apiMiddleware from './middleware';
```

That’s our simple application.

Having JWT workflow implemented in the middleware allows to keep actions code free from authentication flow details.

The sample code available at the https://github.com/viewflow/cookbook/tree/master/_articles/redux_jwt_auth