STACS
Designed for Finance

STACS Engineering -

Frontend development with ReactJS and OAuth 2.0

Transformative Technology for the Financial Industry

PREPARED BY:
Dexter Kwan
WEBSITE:
www.stacs.io
CONTACT:
info@stacs.io

Contents

Frontend development with ReactJS and OAuth 2.0

1. Introduction

Over the past seven months, I have been working at STACS as a Software Engineer Intern. One of the projects I worked on with the team was the blockchain browser, which was responsible for fetching and displaying data from the blockchain. Part of the work involved implementing HTTP API calls in React using the OAuth 2.0 standard, which is what I will be sharing on in this article.

2. Why React?

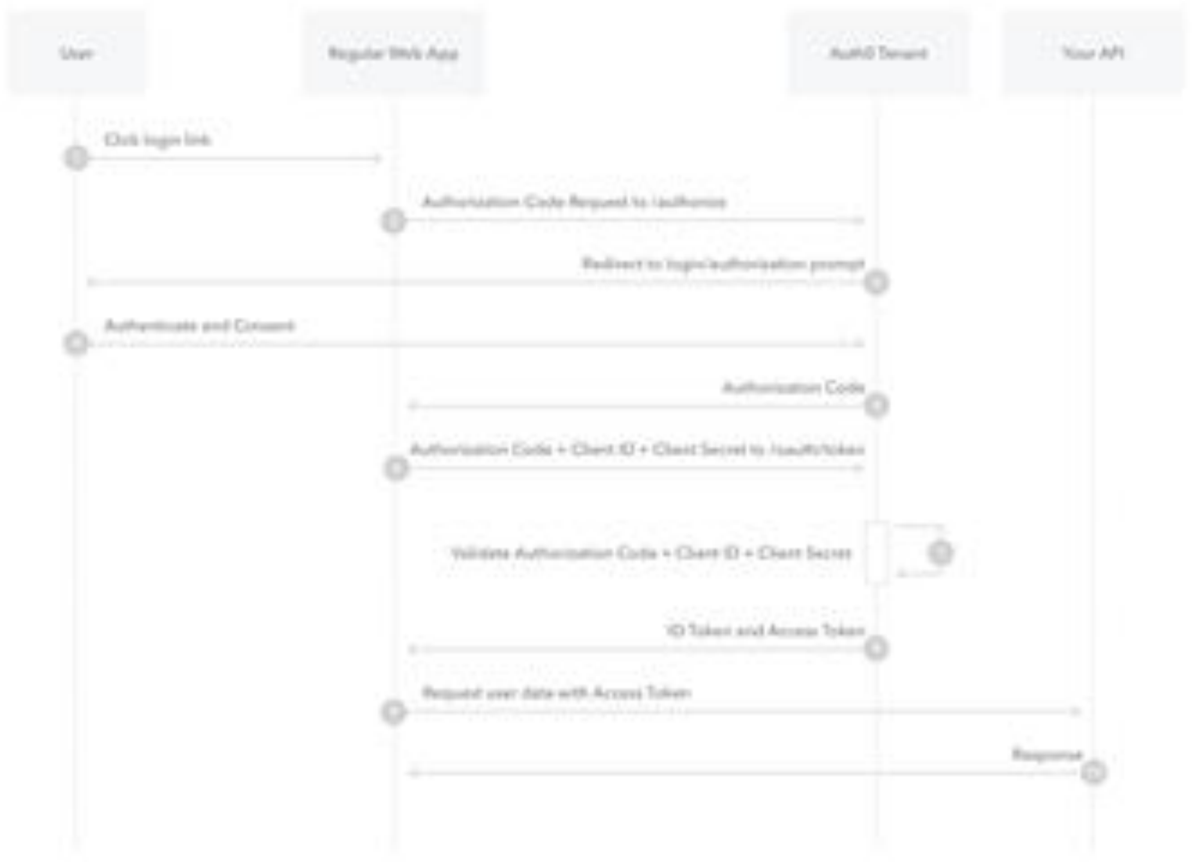
First of all, what is React? React is a JavaScript library built by Facebook that is used to build frontend applications. React uses components and data management, such as reducers and actions, along with providing JSX and a Virtual DOM, not to mention easily reusable library code that will help you build out the frontend to your application fast.

3. A Brief Guide on Redux

Redux is a store in React to manage the state of variables in the app. Redux achieves this with 3 main moving parts: Actions, Reducers and the Middleware. Actions are “signals” that get pushed into the system. The reducers and middleware will then pick up on these signals and run the appropriate code that matches what the action is calling for. Reducers handle the state, and act as pure functions. The middleware, on the other hand, is responsible whatever asynchronous functions we need. In our case, we use reducers to control the state variables, and use the middleware to make API calls.

4. What is OAuth 2.0?

OAuth 2.0 is an authorization framework that controls access to different users on a HTTP service. Below is a diagram from Auth0 on how the Authorization Code Flow works.


User
Regular Web App
Auth0 Tenant
Your API
Click login link
Authorization Code Request to /authorize
Redirect to login/authorization prompt
Authenticate and Consent
Authorization Code
Authorization Code + Client ID + Client Secret to /oauth/token
Validate Authorization Code + Client ID + Client Secret
ID Token and Access Token
Request user data with Access Token
Response

Upon authorization on the Authorization Code Flow, an Authorization Code is returned, which is combined together with the client ID and client secret on the application end to request for a ID Token and Access Token.

5. Authorization Code Flow vs. Implicit Flow

To understand the reasons and benefits of using authorization code flow, we need to first understand how it compares with implicit flow. The key difference between the two is that implicit flow returns the access token directly upon authentication, while authorization code flow has to return an authorization code, which will then be exchanged for the access token on the /token endpoint (steps 5 to 8 in the diagram above). This causes the authorization code flow to be more secure, since the access token no longer goes through the web flow, but rather directly through the /token API. This mitigates attacks that target the web stack.

6. Implementing Authorization Code Flow

The first thing to do is to request authorization, which is done through an authorization URL, which is usually in the format like:

```
https://authorization-server.com/oauth/authorize
?client_id=CLIENT_ID
&response_type=code
&redirect_uri=CALLBACK_URL
&scope=read
```

The user will then log in to the service to authenticate their identity. Afterwards, they will be redirected to the redirect URI, together with an *authorization code*. It looks something like this:

```
https://my-application.com/callback?code=AUTHORIZATION_CODE
```

Using the *authorization code*, the user will then request for an access token from the API by passing the authorization code along with the client ID and client secret, which can be done through a POST request.

```
https://my-
application.com/v1/oauth/token?client_id=CLIENT_ID&client_secret=CLIENT_SECRET&grant_type=authorizati
on_code&code=AUTHORIZATION_CODE&redirect_uri=CALLBACK_URL
```

The API will respond with the access token to the application, which can now be used to make API calls.

7. Making API calls in React

Now that we have our access token, we can make API calls to our backend. We first set up our redux by defining our action.

```
const BlockAction = {
 fetchTransactionsByBlockHeight: (data) => {
 // Fetch transactions
   return {
     type: ActionType.FETCH_TRANSACTIONS_BY_BLOCK_HEIGHT,
     payload: {
       data
     }
   }
 },
 fetchTransactionsByBlockHeightSuccess: (data, status) => {
 // Transactions fetched successfully
   return {
     type: ActionType.FETCH_TRANSACTIONS_BY_BLOCK_HEIGHT_SUCCESS,
     payload: {
       data,
       status
     }
   }
 },
 fetchTransactionsByBlockHeightFail: (error, status) => {
 // Fetching of transactions failed
   return {
     type: ActionType.FETCH_TRANSACTIONS_BY_BLOCK_HEIGHT_FAIL,
     payload: {
       error,
       status
     }
   }
 },
};
```

Following which, we create a reducer to handle our state.

```
const BlockReducer = (state = InitialState, action) => {
 switch (action.type) {
   case ActionType.FETCH_TRANSACTIONS_BY_BLOCK_HEIGHT:
   // Do nothing since API call happens in the middleware
     return {
       ...state,
     }
   case ActionType.FETCH_TRANSACTIONS_BY_BLOCK_HEIGHT_SUCCESS:
   // Success. Store data in state and update success flag.
     return {
       ...state,
       blockTransactionsData: action.payload.data,
       fetchBlockTransactionsStatus: action.payload.status,
     }
   case ActionType.FETCH_TRANSACTIONS_BY_BLOCK_HEIGHT_FAIL:
   // Error occured, update success flag.
     return {
       ...state,
       fetchBlockTransactionsStatus: action.payload.status,
     }
};
```

Then, we create our saga, which will be responsible for making the HTTP API calls.

```
// axios request function used to make API calls
async function axiosRequest(endpoint, payload = null, method = RequestMethod.GET) {
   const session = await Auth.currentSession();
   axios.defaults.baseURL = app.Host;
   // fetch access token from session
   axios.defaults.headers.common['Authorization'] = session.getAccessToken().jwtToken;
   return new Promise((resolve) => {
      axios[method.toLowerCase()](endpoint, payload).then(response => {
         resolve(response.data);
      }).catch(error => {
         if(endpoint === URI.userInfo) {
            localStorage.clear();
         }
         console.log('axiosRequest endpoint: ', endpoint, error);
      });
   })
}

async function fetchTransactionsByBlockHeight(action){
 try {
   return axiosRequest(URI.blockFetchTransactions, action.payload.data, RequestMethod.POST);
 } catch (err) {
   console.log('BlockSaga - fetchTransactionsByBlockHeight() failed: ', err);
 }
}
export function* runFetchTransactionsByBlockHeight(action){
 // runs whenever action FETCH_TRANSACTIONS_BY_BLOCK_HEIGHT is called.
 try {
   const blockData = yield fetchTransactionsByBlockHeight(action);
   yield put(BlockAction.fetchTransactionsByBlockHeightSuccess(blockData, ActionStatus.SUCCESS));
```

```
  } catch (err) {
    yield put(BlockAction.fetchTransactionsByBlockHeightFail(err, ActionStatus.ERROR));
  }
}

export default function* watchBlockSaga() {
  yield takeEvery(ActionType.FETCH_TRANSACTIONS_BY_BLOCK_HEIGHT,
runFetchTransactionsByBlockHeight);
}
```

With that, we have a fully functioning redux setup that can handle API calls to the backend, using the OAuth 2.0 Authorization Code Grant Flow. The setup is very much similar to implicit flow, with the main difference being the authorization flow requiring an additional step to obtain the access token.

Transformative Technology for the Financial Industry

STACS
Designed for Finance

Contact

STACS
www.stacs.io
info@stacs.io

Jin Ser
Solutions Architect Director
jin.ser@stacs.io