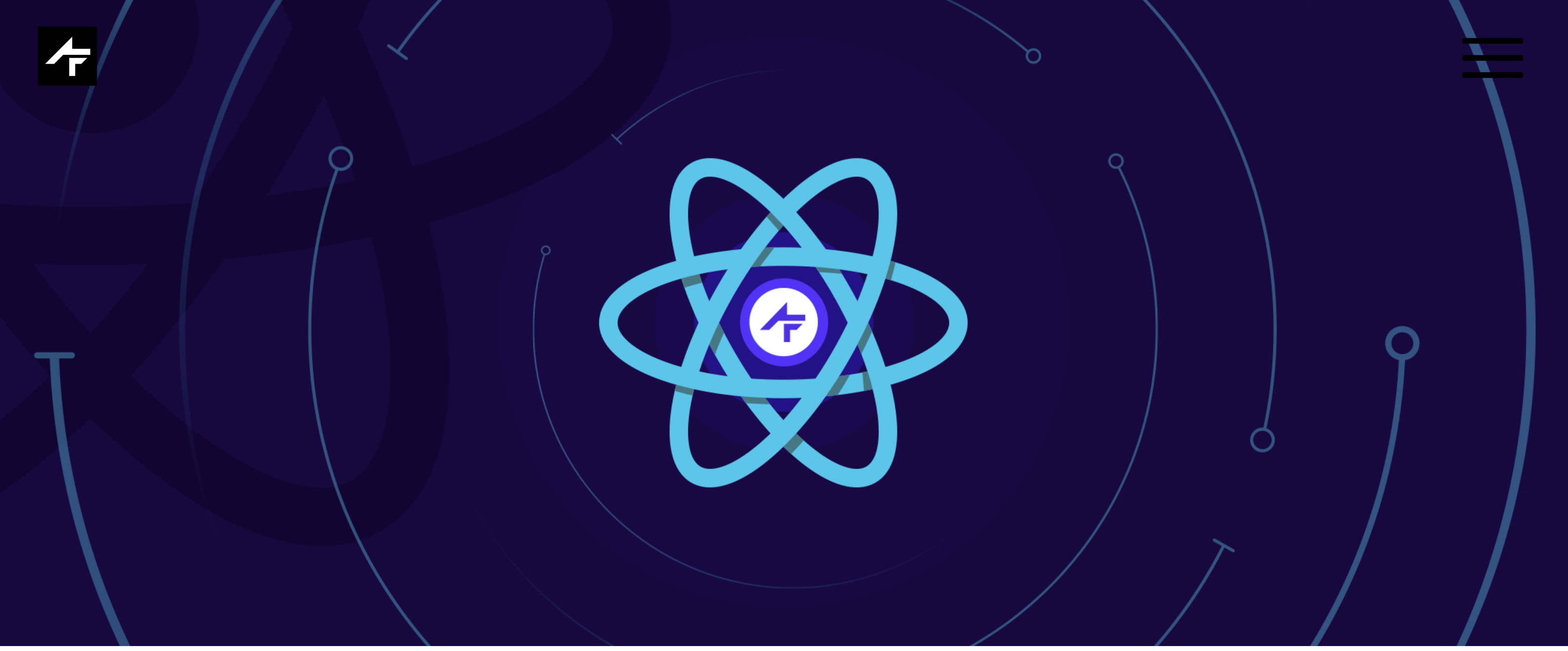

Authentication with ADAL in React Single Page Applications

One of the key features in Single Page Applications (SPAs) is a little thing known as *authentication*. The ability to login and make authenticated network requests to a backend API are often required, but not always easy to implement.

In the **first part of this tutorial**, we will cover how to implement basic **authentication** with Azure's Active Directory (**AAD**) and the Azure Directory Authentication Library (**ADAL**) for JavaScript, (also known as the adal-angular library on **npm**) in a Single Page Application (**SPA**) written with **React JS**.

In addition, we will make sure network requests to a (REST) API running on Azure always use a valid **bearer token**.

Not covered in this tutorial

- how to set up Azure AD
- how to set up a client web app on Azure
- how to set up an API backend on Azure

But I can give you this: you'll need to use OAuth2 implicit flow by setting *oauth2AllowImplicitFlowto* to "true" in the AAD manifest of the client application. And, you will need to use the **BearerStrategy** (as in, use an Authentication parameter in the request headers set to “Bearer <a valid access token>”) to authenticate network requests against the backend API.

Covered in this tutorial

- **First part:**
  - how to set up and initialize the adal-angular library

You can also **use this sample repo to follow the tutorial**.

It's a setup!

1. Set up an AD in Azure with a **user or two to test with**
2. Set up the necessary application project(s) in Azure, of which we will use the **tenant ID,** the **application ID** of the **client web app**, and the **application ID** of the **API app**.
3. Set up a **config file** in your React application where we can place our Azure IDs and other configuration parameters.

```
// src/config/AdalConfig.js
export default {
 clientId: 'ENTER THE APPLICATION ID OF THE REGISTERED WEB APP ON AZURE',
 endpoints: {
 // Necessary for CORS requests, for more info see https://github.com/AzureAD/azure-activedirectory-library-for-js/wiki/CORS-usage
 api: "ENTER THE APPLICATION ID OF THE REGISTERED API APP ON AZURE"
 },
 // 'tenant' is the Azure AD instance.
 tenant: 'ENTER YOUR TENANT ID',
 // 'cacheLocation' is set to 'sessionStorage' by default, for more info see https://github.com/AzureAD/azure-activedirectory-library-for-js/wiki/Config-authentication-context#configurable-options
 // We change it to'localStorage' because 'sessionStorage' does not work when our app is served on 'localhost' in development.
 cacheLocation: 'localStorage'
}
```

> TIP: Use custom environment variables here! See Create React App's guide for more information.

Then, we initialize the adal instance by combining the *AuthenticationContext* class, exported from the adal library, the *AdalConfig* we defined in the previous step.

```
// Initialize the authentication
export default new AuthenticationContext(AdalConfig)
```

> Don't worry if `export default new AuthenticationContext(AdalConfig)` would initialize a new instance each time you import it -> webpack will build all our javascript code in one file and *imports* will reference to single instances respectively.

Initialize axios instance

To make sure our network requests use the correct base url of the API, we create a config file with a certain *baseURL* parameter which we'll later use to initialize an axios instance.

```
// src/config/ApiConfig.js
export default {
  baseURL: "ENTER BASE URL OF API HERE" // something like "http://my-host-name.xyz/api"
}
```

Next, use the *ApiConfig* to initialize an axios instance like so:

```
// src/services/Api.js
import axios from 'axios'

import ApiConfig from '../config/ApiConfig'

const instance = axios.create(ApiConfig)
```

After we've initialized everything we need, we can start coding the logic to successfully render the React ication or to redirect the user to Microsoft's login page.

In index.js, import the AuthContext from our authentication service and the *AdalConfig* to be able to use the IDs.

```
// src/index.js
import AdalConfig from './config/AdalConfig'
import AuthContext from './services/Auth'
```

Add the following code to let the **adal library handle any possible callbacks** after logging in or (re-)acquiring tokens:

```
// Handle possible callbacks on id_token or access_token
AuthContext.handleWindowCallback()
```

Then we'll add some extra logic that we will only run when we are on the **parent window** and not in an iframe. If we were to allow it to run in iframes, which are used by adal to acquire tokens, then we would be stuck with multiple instances of our React app and we don't want that.

If we have **no logged in user** then we will **redirect the user to Microsoft's login page**. If we have a **logged in user** then we will **acquire an access token for our API** to see that everything works, **and** we will **render our React application**.

This results in the following code:

```
// extra callback logic, only in the actual application, not in iFrames in the app
if ((window === window.parent) && window === window.top &&
!AuthContext.isCallback(window.location.hash)) {
```

To make sure our network requests to the backend API remain authenticated, we will add some logic to our axios ance.

We will call adal's *acquireToken* function each time a network request is made. Adal will return the valid access token or it will asynchronously fetch a new one if it is invalid. Once the token is available we will add it to the *Authorization* header of the network request.

First, don't forget to add the necessary imports:

```
// src/services/Api.js
import AdalConfig from '../config/AdalConfig'
import AuthContext from './Auth'
```

Then we can place the re-acquiring of tokens in a request **interceptor** of the axios instance like so:

```
// src/services/Api.js
// Add a request interceptor
instance.interceptors.request.use((config) => {
  // Check and acquire a token before the request is sent
  return new Promise((resolve, reject) => {
    AuthContext.acquireToken(AdalConfig.endpoints.api, (message, token, msg) => {
      if (!!token) {
        config.headers.Authorization = `Bearer ${token}`
        resolve(config)
      } else {
          // Do something with error of acquiring the token
          reject(config)
      }
    })
```

Want to do adjustments to the session timeout?

Then follow **this link** to part 2 of this tutorial, where I explain how to add session management!

AppFoundry

AppFoundry is a Belgian **Digital Solutions** provider.
We design and build experiences for **iOS**, **Android**, **Web** and beyond.
Visit us in **Kontich**, **Merelbeke** or **Hasselt**!

Contact

Mail us
Call us
Visit us

Follow us

Facebook
Twitter
Linkedin
Github
Dribbble
Speaker Deck

Language: Nederlands **English**

© 2020 AppFoundry