

# Okta: Authentication and Protection

Created by William Crocker  
Last updated Apr 07, 2020

The purpose of this document is to describe, with examples, how to use Okta for user authentication and API protection. [REDACTED] has a corporate license for Okta which provides a Single Sign-On (SSO) platform. This enable [REDACTED] to secure enterprise products such as Jira, Confluence, Bitbucket, etc. Internally [REDACTED] developers can also take advantage of the Okta implementation to secure in-house built web applications and APIs.

[REDACTED] the module required for API access management. If that's the case, only web site authentication will be supported. The examples used in this article were coded against a developer account with supported both authentication and API access management.

## What is Okta?

Okta is an enterprise access management platform. It provides features such as SSO, MFA, API protection, universal directory, and more. It removes the need for companies to create directory services designed to manage users and authentication services. It supports Security Assertion Markup Language (SAML) and Open ID Connect (ODIC) protocols. For more information on Okta, visit [www.okta.com](http://www.okta.com) and [developer.okta.com](http://developer.okta.com).

## Proof of Concept

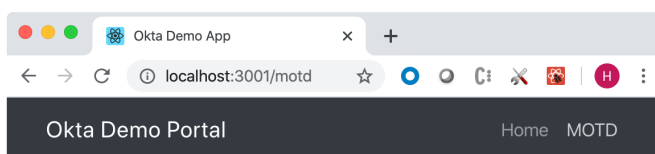
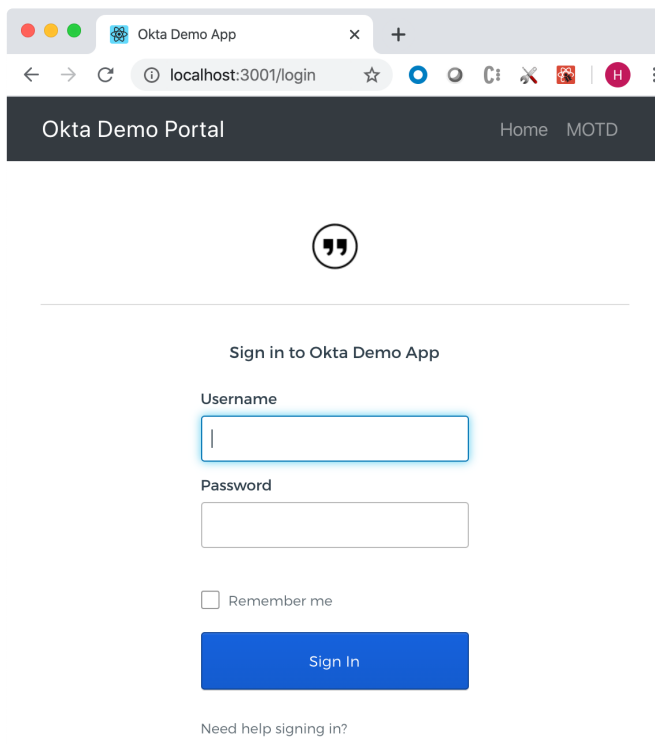
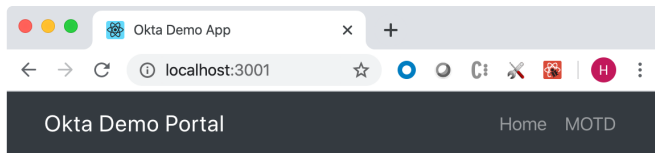
As part of the SPIKE [REDACTED] a PoC project was created to understand how to authenticate users and protect API calls. This knowledge can be used to enhance security for internally developed products as well as MuleSoft APIs.

## Process

The goal of this project is to create a secure web application that uses a protected API. The language of choice will be Node.js. The frontend (FE) user interface (UI) will be a React application. The web application will use Okta to authenticate the user and forward a token for API access. The backend (BE) application will use Okta to verify the token from the FE is valid and contains the elements required by the API. The sample code will use the Authorization Code flow. Compared to the Implicit flow, this is a more secure option. The following article discusses the choices in more depth: [developer.okta.com/blog/2019/08/22/okta-authjs-pkce](http://developer.okta.com/blog/2019/08/22/okta-authjs-pkce).

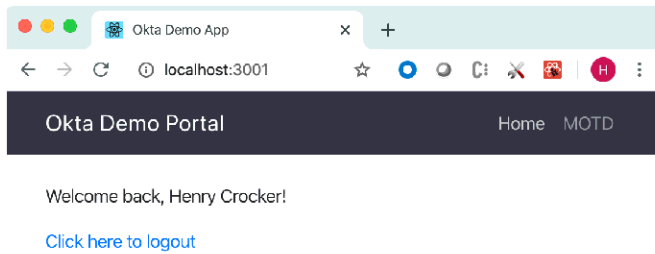
In order to have full control of the Okta environment, a developer account will be used. A managed application will be created along with an API authorization server. Developer accounts can be created at [developer.okta.com](https://developer.okta.com). Any IDs or user accounts found in this document will be deleted. Any user of these apps will be required to create an account and adjust the code to match the URIs and IDs to match applications and services for their account.

## Results



Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated.

— Edsger W. Dijkstra



## Okta Set Up

As mentioned above, an Okta developer account was used for this project. It allowed for administration of users and applications without affecting the [REDACTED] corporate Okta implementation.

### Process

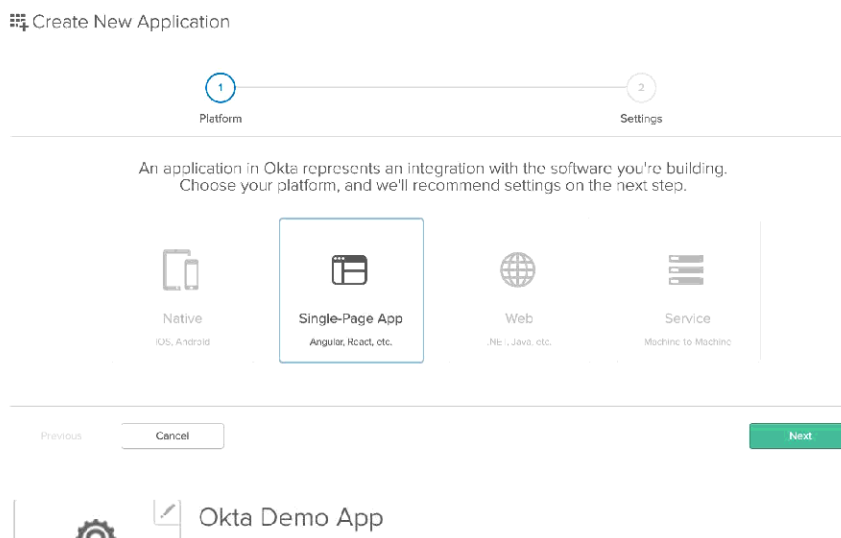
Use the following steps to configure the Okta environment for the project.


#### Create Account

Create a developer account at [developer.okta.com](https://developer.okta.com).


#### Add a new single page application (SPA)

For this project, a React application will be used. If “Web” were to be selected, the authentication workflow would follow PKCE.





Active


[View Logs](#)

[General](#)
[Sign On](#)
[Assignments](#)

General Settings Edit

APPLICATION

Application label

Okta Demo App

Application type

Single Page App (SPA)

Allowed grant types

Client acting on behalf of a user

☒ Authorization Code
 ☐ Implicit

LOGIN

Login redirect URIs

http://localhost:3001/redirect

Logout redirect URIs

http://localhost:3001

Login initiated by

App Only

Initiate login URI

http://localhost:3001/redirect

Client Credentials

Client ID

Public identifier for the client that is required for all OAuth flows.

Client authentication

☒ Use PKCE (for public clients)
 

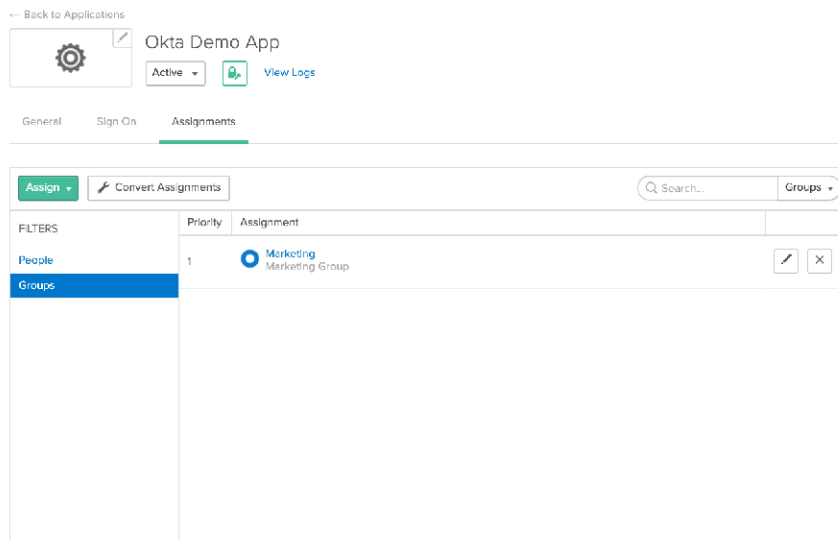
Uses Proof Key for Code Exchange (PKCE) instead of a client secret. A one-time key is generated by the client and sent with each request. Instead of verifying the identity of a client

Name the application anything you choose. Choose the port on which your application will be listening. In the case of the PoC, the API service will listen on port 3000 and the UI will listen on port 3001. Since we will use the Authorization Code with PKCE workflow, select "Authorization Code" for the "Grant type allowed" field. Set the 'Login redirect URIs' and 'Initiate login URI' to 'http://localhost:3001/redirect'. The path can be any value, but it must match the value used in the sign in widget and express route.

- `redirectUri: window.location.origin + "/redirect",`
- `<Route path="/redirect" component={LoginCallback} />`

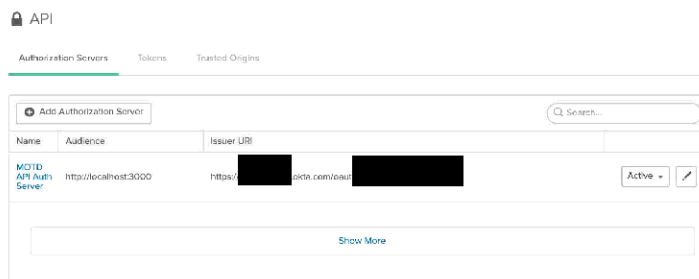
The user that created the application is automatically assigned. If more users are needed, create and assign to the application. In this project, only the creator will be used. In the case of this project, a group will be added. The API will check and only allow access from users in the

'Marketing' group. A new claim rule will be created later to ensure the user's groups are included in the access token.

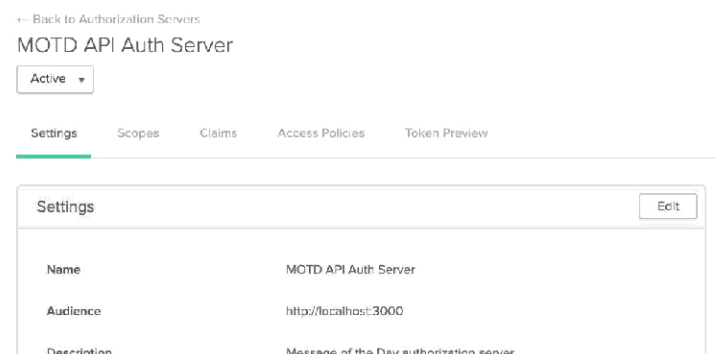


## Add API Authorization Sever

An authorization server is used to protect resources such as APIs. The authorization server will have associated IDs that are used to make reference to the server designed to authorize a resource. The developer account has a default server. In this project, a custom authorization was created. It's not necessary, i.e. the default server could have been used.



Give the server any name. The audience value should match the API protocol, host, and port. In this case, the API service is running on `http://localhost:3000`.



Issuer	https://demo.okta.com/oauth
Metadata URI	https://demo.okta.com/oauth2/aus5gxvqo7gfukqy24x6r/well-known/oauth-authorization-server
Signing Key Rotation	Automatic
Last Rotation	7 Apr 2020

The “Issuer” and “Audience” values are used by the API to verify a token sent by a client. It will verify the “Bearer” authorization token against the audience of the resource. If there is a match, access is granted.

The authorization flow will require CORS. Set the FE application as a trusted host.

API

Authorization Servers Tokens **Trusted Origins**

ADD ORIGINS

Add Origin URLs to redirect users to custom screens or enable browser based applications to access Okta APIs from Javascript (CORS).

Add Origin

Q Search...

FILTERS	Name	Origin URL	Type	Actions
All Origins	Demo Okta UI App	http://localhost:3001	CORS Redirect	
CORS				
Redirect				

For the sample application, access to the API will be limited to the ‘Marketing’ group and must have the scope of ‘motd:read’. This will require the creation of custom scopes and a custom claim that includes the groups to which the user is assigned.

← Back to Authorization Servers

MOTD API Auth Server [Help](#)

Active

Settings **Scopes** Claims Access Policies Token Preview

+ Add Scope

Name	Description	Default Scope	Metadata Publish	Actions
motd:read	Read only for Message of the Day API.	No	No	
openid	Signals that a request is an OpenID request.	No	Yes	
profile	The exact data varies based on what profile information you have provided, such as name, time zone, picture, or birthday.	No	Yes	
email	This allows the app to view your email address.	No	Yes	
address	This allows the app to view your address, such as: street address, city, state, and zip code.	No	Yes	
phone	This allows the app to view your phone number.	No	Yes	
offline_access	This keeps you signed in to the app, even when you are not using it.	No	Yes	

← Back to Authorization Servers

MOTD API Auth Server [Help](#)

Active

Settings **Scopes** **Claims** Access Policies Token Preview

+ Add Claim

CLAIM TYPE	Name	Value	Scopes	Type	Included	Actions
All	sub	{isspurator != null ? appuser.userName : appclaimid}	Any	always	Always	

ID	groups	groups: matches regex .*	Any	access	Always		
Access							

Edit Claim

Name

groups

Include in token type

Access Token

Always

Value type

Groups

Filter

Only include groups that meet the following condition.

Matches regex

.\*

Disable claim

☐ Disable claim

Include in

☒ Any scope

☐ The following scopes:

Save

Cancel

With this configuration, the access token may contain the 'motd:read' scope (if included in the login widget). The 'groups' claim will cause the user's groups to be included in the access token by default. Note, the Okta authentication object will have two tokens. The idToken contains identity information about the user and the accessToken will contain authorization or access limits for the user.

## Authentication

If a secure route is selected, Okta can verify authentication and challenge the user if not authenticated. Examples discussed here will be specifically for Node.js. Other languages are supported by Okta; however, even though the concept is similar, the implementation will be different.

## Define Routes

```

1 // snippet from App.js
2 import Login from "../components/auth/SignInWidget";
3 ....
4 <div className="container">
5   <Route path="/" exact component={Home} />
6   <Route path="/login" component={Login} />
7   <SecureRoute path="/motd" component={Motd} />
8   <Route path="/redirect" component={LoginCallback} />
9 </div>
10 ....

```

In the example above, a custom Login page will be used. It's identified by the /login route. This is not necessary and a default Okta page can be used. Providing a custom page allows for unique

styling, image, and text to appear on the Login page.

```
1 // snippet from config file
2 const config = {
3   oktaOrg: "https://dev-XXXXXXXXXX.okta.com",
4   issuerPath: "oauth2/XXXXXXXXXX",
5   clientId: "XXXXXXXXXX",
6   redirectUri: window.location.origin + "/redirect",
7   pkce: true,
8   scopes: ["openid", "profile", "email", "motd:read"],
9   display: "page",
10  responseType: "code",
11  responseMode: "query",
12  logo: "/quoteLogo.png",
13  i18n: {
14    en: {
15      "primaryauth.title": "Sign in to Okta Demo App"
16    }
17  }
18 };

1 // snippet from SignInWidget
2 ....
3 const signInConfig = {
4   baseUrl: config.oktaOrg,
5   clientId: config.clientId,
6   redirectUri: config.redirectUri,
7   logo: config.logo,
8   i18n: config.i18n,
9   authParams: {
10     issuer: `${config.oktaOrg}/${config.issuerPath}`,
11     display: config.display,
12     responseType: config.responseType,
13     responseMode: config.responseMode,
14     scopes: config.scopes,
15     pkce: config.pkce
16   }
17 };
18
19 const widget = new OktaSignIn(signInConfig);
20 ....
```

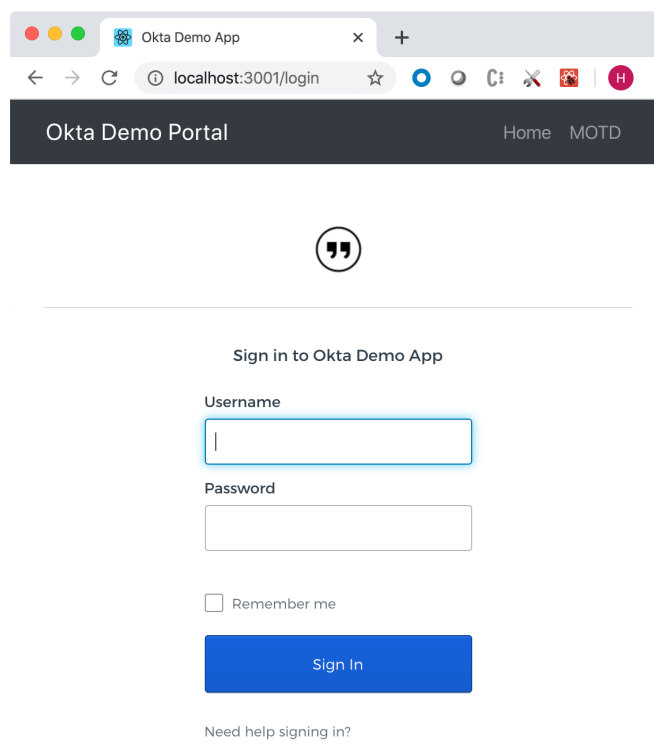
The code above builds on the Okta-provided sign-in component. The results of the above code produces the following page when the user is not logged in. Notice the custom logo and sign-in text. In this example, the MOTD menu item is protected. Clicking on MOTD will display the login screen if the user is not logged in.



The values from the config file should look familiar. They were created earlier when we set up the application and client service. Note the value for "pkce" - it must be "true" for the "Authorization Code" flow. The default Okta login callback will be used. It's possible to write a custom callback; however, that's beyond the scope of the project. This callback is responsible for taking the token values from the query string and storing for Okta use within the application.

The following values are required for successful Authorization Code flow. Initially, the values responseType and responseMode values were not set, i.e. the Okta default values were to be used. This caused a problem. The code would route to the /redirect path and then hang. The developer tools showed a call to 'codeverifier', but it didn't process properly. Setting the values manually solved the problem. Notice the values in the 'scopes' list. The new 'motd:read' scope is included. The API will check for this scope. If not there, API access is rejected.

```
1 scopes: ["openid", "profile", "email", "motd:read"],
2 pkce: true,
3 display: "page",
4 responseType: "code",
5 responseMode: "query",
```



Once logged in, Okta information is stored in the application's local storage. The Okta LoginCallback component is responsible for processing the response the /reject and storing the data. A custom callback can be created to replace the LoginCallback. In that case, the developer is responsible for storing the information properly for the authState hook.

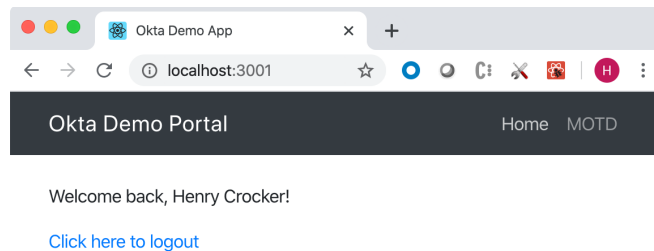
```
1 // snippet from App.js - this is the route responsible for storing the
```

```

2 // authentication data in local storage. It must use exactly the route
3 // defined below.
4 import { Security, SecureRoute, LoginCallback } from "@okta/okta-react";
5 ....
6 <Route path="/reject" component={LoginCallback} />
7 ....

```

There is an Okta component that can be used to get state and authorization functions. It's include in the root route and is used to display the name if logged in.



```

1 // snippet from Home.js
2 ....
3 const Home = () => {
4   const { authState, authService } = useOktaAuth();
5   const [userInfo, setUserInfo] = useState(null);
6
7   useEffect(() => {
8     if (!authState.isAuthenticated) {
9       setUserInfo(null);
10    } else {
11      authService.getUser().then(info => {
12        setUserInfo(info);
13      });
14    }
15    }, [authState, authService]);
16 ....

```

When the user logs out, the object is removed from local storage. Access to secure routes will cause another login request.

## API Protection

When the MOTD menu item is clicked, it makes a call to a protected API. The user has been authenticated and there are two tokens available in the Okta authentication object; idToken and

accessToken. When the API is called, the accessToken is included in the call as a 'Bearer' authorization header.

```
1 // snippet from utils.js
2 ....
3 const options = {
4     protocol: "http:",
5     hostname: "localhost",
6     port: 3000,
7     path: "/motd",
8     method: "GET",
9     headers: {
10         accept: "application/json",
11         authorization: `Bearer ${token}`
12     }
13 };
14 ....
```

When the API receives the request, it will verify the accessToken has the required data. If it does, access to the resource is granted. If not, an error is thrown. It expects the user to belong to the 'Marketing' group and have the 'motd:read' scope.

```
1 // snippet from API service auth.js
2 ....
3 const oktaJwtVerifier = new OktaJwtVerifier({
4     issuer: ISSUER,
5     clientId: CLIENT_ID,
6     assertClaims: {
7         'groups.includes': EXPECTED_GROUPS,
8         'scp.includes': EXPECTED_SCOPES
9     }
10 });
11
12 module.exports = (req, res, next) => {
13     try {
14         const { authorization } = req.headers;
15         ....
16         oktaJwtVerifier.verifyAccessToken(token, AUDIENCE)
17             .then(
18                 jwt => {
19                     req.jwt = jwt;
20                     next();
21                 }
22             )
23             .catch(
24                 err => {
25                     next(err);
```

```
26         }
27     );
28 }
29 ....
```

## Source Code

The following source code is written in Node.js. The BE (API resource) uses the express module. The FE application uses the React module. Any keys, client IDs, client secrets, issuers, etc. found in config files must be replaced with values from your Okta environment. The React app runs on port 3001 and the API listens on port 3000 by default. These port values can be changed by setting the PORT environment variable. Note, whatever port is defined must be reflected in the Okta application setup.

## Lessons Learned

While working with Okta, I ran into some difficult to solve problems. Most of the time, the answer could be found on the Okta developer website. There some obscure problems that I found answers to in forums or sites like StackOverflow. The list below has items that burned up some time to solve.


### Failed to fetch error

While testing in Postman, I received this error because the Bearer token I was using had a newline at the end.

### Does not match expected audience error

This was caused by a missing parameter in the token verifier. Make sure the audience matches the value defined in the Okta API application server configuration.

```
1 // snippet from auth.js - error will occur if the audience prop is missing
2 ....
3 oktaJwtVerifier.verifyAccessToken(token, AUDIENCE)
4 ....
```

The example in this article used a developer account. In order to make this work on the corporate Okta implementation, the application items will need to be added to the  account. See the internal references below for examples and Okta admin contact information.

## Internal References

## External References

- Okta Product Demos | Protect Your Modern App with API Access Management
  - <https://www.youtube.com/watch?v=4VuUEzmjkk&list=PLlid085fSVdvU8R1A4qDfM7Ky96OACNAF&index=22>
  - <https://github.com/oktadeveloper/okta-auth-js-pkce-example>
- <https://developer.okta.com/blog/2018/07/10/build-a-basic-crud-app-with-node-and-react>
- <https://developer.okta.com/blog/2019/08/22/okta-authjs-pkce>
- <https://github.com/okta/okta-signin-widget#showsignintogettokens>