# POC for SSO Integration with Java and ReactJS using Okta

Okta Single Sign-On (SSO) is a feature provided by Okta that allows users to access multiple applications with a single set of credentials. With SSO, users only need to log in once, and they can access various applications and services without repeatedly entering their username and password.

Here are steps to achieve okta SSO integration in our application having Java as the backend and React JS in frontend.
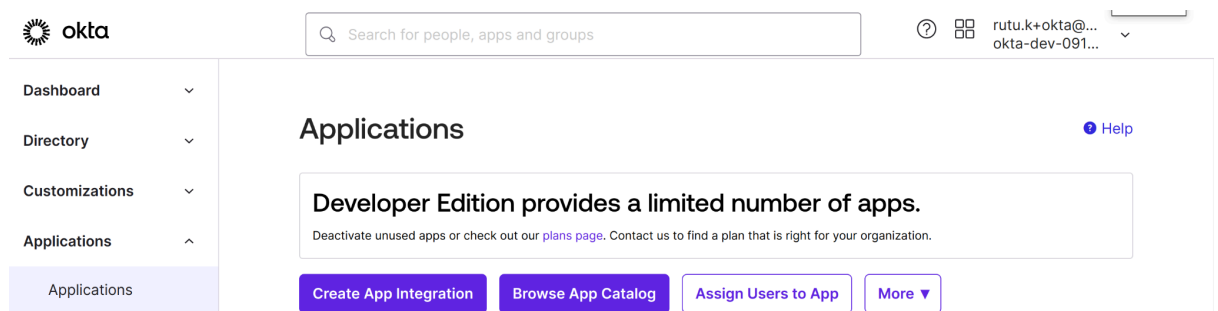
1. **Create Okta App Integration**

Firstly we'll need to Sign up for an Okta developer account at https://developer.okta.com/signup/ to set up your authorization server.

To sign up you will get the below options, which you will have to select an option that is in the red selected area:
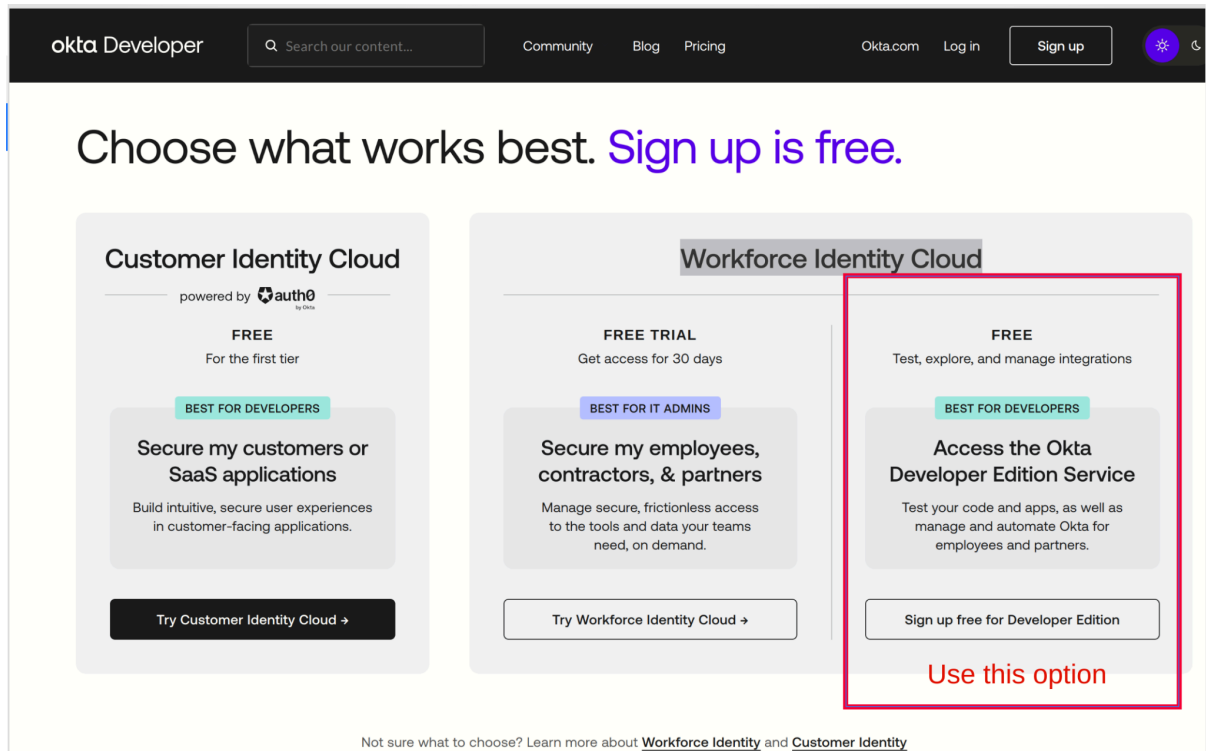
Then we will set the Okta integration app as follows:

From the menu select Applications → Create App Integration

For the **Sign-in** method select **OIDC — OpenID Connect**, for **Application type** select **Single-Page Application** and click Next.

Create a new app integration

Sign-in method
Learn More ↗

- ● **OIDC - OpenID Connect**
  Token-based OAuth 2.0 authentication for Single Sign-On (SSO) through API endpoints. Recommended if you intend to build a custom app integration with the Okta Sign-In Widget.

- ○ **SAML 2.0**
  XML-based open standard for SSO. Use if the Identity Provider for your application only supports SAML.

- ○ **SWA - Secure Web Authentication**
  Okta-specific SSO method. Use if your application doesn't support OIDC or SAML.

- ○ **API Services**
  Interact with Okta APIs using the scoped OAuth 2.0 access tokens for machine-to-machine authentication.

Application type

What kind of application are you trying to integrate with Okta?

Specifying an application type customizes your experience and provides the best configuration, SDK, and sample recommendations.

- ○ **Web Application**
  Server-side applications where authentication and tokens are handled on the server (for example, Go, Java, ASP.Net, Node.js, PHP)

- ● **Single-Page Application**
  Single-page web applications that run in the browser where the client receives tokens (for example, Javascript, Angular, React, Vue)

- ○ **Native Application**
  Desktop or mobile applications that run natively on a device and redirect users to a non-HTTP callback (for example, iOS, Android, React Native)

Cancel   **Next**

In the General Settings tab enter the name of app integration and select **Grant type → Authorization Code**, required to enable OAuth2 authentication.

Set the **Sign-in redirect URI**s to: http://localhost:3000/login/callback

Set the **Sign-out redirect URI**s to: http://localhost:3000

In the Assignments tab select: **Allow everyone in your organization to access**

New Single-Page App Integration

General Settings

App integration name
okta-client-demo

Logo (Optional)

Grant type

Learn More

Client acting on behalf of a user
☑ Authorization Code
☐ Refresh Token
☐ Implicit (hybrid)

Sign-in redirect URIs

Okta sends the authentication response and ID token for the user's sign-in request to these URIs

Learn More

☐ Allow wildcard * in sign-in URI redirect.

http://localhost:3000/login/callback ✕

+ Add URI

Sign-out redirect URIs (Optional)

After your application contacts Okta to close the user session, Okta redirects the user to one of these URIs.

Learn More

http://localhost:3000 ✕

+ Add URI

When using the OpenID Connect sign-in method, Okta redirects us to a hosted sign-in widget. To facilitate the return of user details to our application, we require a callback URL. In our case, the React app is served at http://localhost:3000.

By clicking save, a Client ID will be generated. This is a Public identifier for the client that is required for all OAuth flows and we'll need that in the next steps when configuring our spring boot app along with our client react app.

2. Setup Spring Boot App

Create a new spring boot application. For this example, we'll use Java 17 and Spring-boot 2.5.8

We'll add okta dependency in our pom.xml file:

```xml
<dependency>
    <groupId>com.okta.spring</groupId>
    <artifactId>okta-spring-boot-starter</artifactId>
    <version>3.0.6</version>
</dependency>
```

Now, We'll need to application.properties file with your `issuer` and `client_id` as configured in step 1.

```
okta.oauth2.issuer= https://{yourOktaDomain}/oauth2/default
okta.oauth2.client-id= {yourClientId}
```

To be able to access our API from different origins, we'll add some cors mapping to allow all origins in our **CorsConfig** file.

```java
@Configuration
public class CorsConfig {
    @Bean
    public CorsFilter corsFilter() {
        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        CorsConfiguration config = new CorsConfiguration();
        config.addAllowedOrigin("http://localhost:3000");
            // Allow requests from this origin
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");
        source.registerCorsConfiguration("/**", config);
        return new CorsFilter(source);
    }
}
```

The full code of this application is given [here](#).

3. Setup React App

Create the React app and add the required Okta
dependencies. We'll be using ant design for some basic
layout styles. In this example, we've used React 18. We'll
name our app: "client".

Start with your env.local file. Create it under your root
client project directory if not present and add your
variables. Variable names must have the "REACT_APP_"
prefix for react to fetch them. Set the values for Okta
ISSUER and CLIENT_ID to reflect your Okta settings as
configured in step 1 along with your spring-boot server
API_URL.

```
REACT_APP_API_URL = http://localhost:8080/api
REACT_APP_ISSUER = https://{yourOktaDomain}/oauth2/default
REACT_APP_CLIENT_ID = {yourClientId}
```

Create the **RequiredAuth.js** under components. This is where our main security logic will be held. It will be used along with the Routes to display the content of each component. If the user is not authenticated, it will redirect to the okta-hosted sign-in page.

In this **RequiredAuth.js** file, we'll add an API call to our spring boot application to validate the login of a user.

The full code of this application is also [here.](#)

4. Running your Apps

Run your spring server app and your client with npm start. You should be able to view your client application under localhost:3000

5. Summary

The default access token lifetime is 1 hour. You can review this under your Okta account. From the menu select Security → API. Click on the default Authorization Server and go to the tab Access Policies, here you will find a Default Policy Rule and by clicking the edit icon, you can change the access token lifetime.

default

Active ▾   Default

Settings   Scopes   Claims   **Access Policies**   Token Preview

Add New Access Policy

| 1 | Default Policy |

Default Policy

| Active ▾ | ✎ Edit | ✕ Delete |

Description   Default Policy for your Authorization Server

Assigned to clients   ◉ All Clients

Add rule

Priority	Rule Name	Scopes	Status	Actions		
1	Default Policy Rule	All	Active ▾	ℹ	✎	✕

Edit rule

Edit Rule

Rule Name

Default Policy Rule

IF Grant type is

Client acting on behalf of itself

☑ Client Credentials

Client acting on behalf of a user

☑ Authorization Code
☑ Implicit (hybrid)
☑ Resource Owner Password
☐ SAML 2.0 Assertion
☐ Device Authorization
☐ Token Exchange

AND User is

⦿ Any user assigned the app
◯ Assigned the app and a member of one of the following:

AND Scopes requested

⦿ Any scopes
◯ The following scopes:

THEN Use this inline hook

None (disabled) ▾

AND Access token lifetime is

1   Hours ▾

AND Refresh token lifetime is

Unlimited ▾

but will expire if not used every

7   Days ▾

Update rule   Cancel

While running your app you'll be able to retrieve your Bearer token from your application's local storage and use Okta's introspect API to check the status of your token among other information.

```
curl --location --request POST
'https://<your_okta_domain>/oauth2/default/v1/introspect' \
-H "Accept: application/json" \
-H "Content-Type: application/x-www-form-urlencoded" \
-d "client_id=<your_client_id>" \
-d "token_type_hint=access_token" \
-d "token= <your_access_token_value>"
```

If you sign out and run the above curl again, you'll be able to see that your token is inactive.

```
{
    "active": false
}
```

Here is a screen-cast of this whole integration demo: https://www.loom.com/share/c496ab3ab5cc456da24a958a607e a134?sid=069c91e0-24dc-41ea-81c7-35fcd1fb1046

Thank you for reading so far! Hopefully, this has helped some of you.