# References

Code at : https://github.com/Ajay-Nallanagula/epampdp/tree/master/AUTH\_AUTORIZATION

* [Basic Auth, OAuth, OpenID Connect, Scopes & Refresh Tokens](https://www.youtube.com/watch?v=x6jUDfpESmA&list=PLP_rkG1reBjrCKy2Pb1bvjJKbKfantijk&index=3)
* \*\*\* Oauth+React+Node : <https://www.youtube.com/watch?v=GGGjnBkN8xk>
* [OAuth using React and Node](https://youtu.be/dyZmsz6usWk?feature=shared)
* <https://medium.com/strapi/protected-routes-and-authentication-with-react-and-node-js-d31d234644cd>
* Using JWT AUTH : <https://www.youtube.com/watch?v=nI8PYZNFtac>
* <https://auth0.com/blog/complete-guide-to-react-user-authentication/>

**🡪**  Get access Token: <https://auth0.com/docs/quickstart/spa/react/02-calling-an-api>

* <https://auth0.github.io/auth0-react/interfaces/GetTokenWithPopupOptions.html>

# Basic Authentication

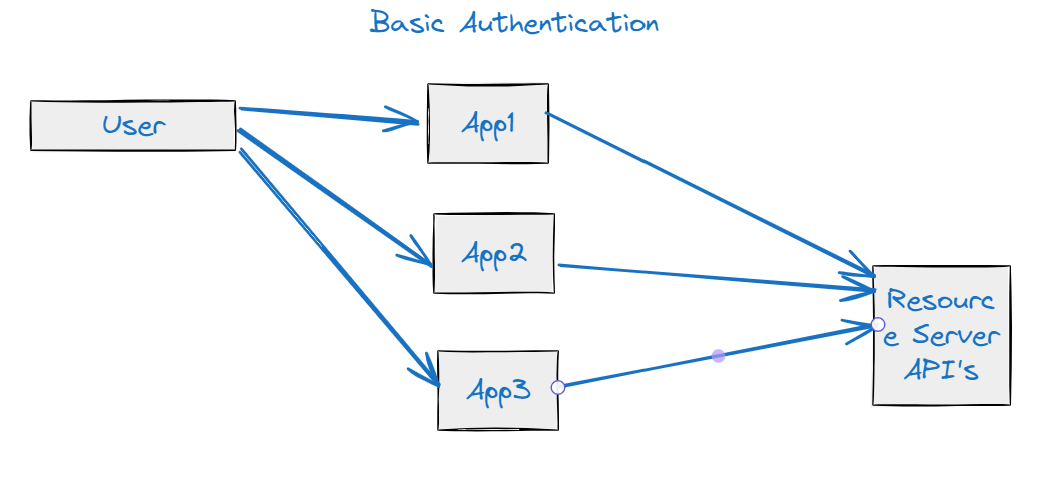
**Basic Authentication** : Where the username and password are encoded and sent on every request. The resource server, which is giving API’s sends the response . This is very naïve way of Authentication and certain dis advantages of it .

A diagram of a line

Description automatically generated

Disadvantages of Basic Authentication

1. The username and password is sent on every request this is risky. Password-Sharing is risky.
2. Now assume the resource server that offers API’s have grown popular and there are application(App1,App2,App3) that want to build on top of the API the resource server is providing … In that case as a user you will have to share the credentials with App1,App2,App3 these APPS will send the credentials to resource server on every request, Now regarding the access … consider resouce server is Twitter API, the user via App1,App2,App3 want to post a tweet and do no more.. In that case we will not have any control because App1,App2,App3 have the username and password, they can read time lines, direct-messages of our profile in Resource Server even though we didn’t give permissions … permission was only to write the messages .



1. There is no way to revoke access to an application, change in password will revoke access of all the three applications .

# OAUTH

OAUTH is limititing access to resources

1. OAUTH is granting access to applications without sharing the password to **applications** .
2. Ecah application should register with the API, so we can revoke access independently

## Authorization Code flow

A diagram of a software process

Description automatically generated

Assumptions:

**ACTORS**: “Resource Owner(User)”, “Client(Frontend+Backend Channel)”, “Authorization Server”, ”Resource Server(API Resources)”

1. There is an APP which want to talk to Resource server on Behalf of User.
2. That APP is APP1 Which has front Channel APP1(FC-APP1) and Back Channel APP1 (BC-APP1)
3. Auth-Server can be built from scratch or we can use AUTH0 Servers like OKTA, GOOGLE WORKSPACES , PingFederate etc.

Step 1: User access Front Channel of APP1,

Step 2: FC-APP1 sends a request to Authorization Server with following payload

Redirection URL: After auth flow success where to go the URL

Response Type: Code, the code here indicates this “Authorization Code Flow”.

Scope: FC-APP1 request for permissions, read/write/read-write etc

Step 3: Auth server prompts the User for his consent

Step 4: Assuming users give consent for the Prompt

Step 5: **Authorization code** is shared to FC-APP1

Step 6: Back channel communication , this happens backside cannot see these calls in the browser. These are internal calls , BC-APP1 takes the Authorization Code from FC-APP1 (Back Channel Communication)

Step 7: BC-APP1 sends **Authorization Code + SALT KEY** to Auth-Server .

Step 8: Auth-Server validates the Authorization Code and as response sends **Access Token** with required information **.**

Step 9: Using this Access Token, APP1 can communicate with Resource Server.

Step 10: Resorce server once recieves the access token, this will send to Authorization server for verification and recieves claims and user data

## How will Frontend Channel and Backend Channel Share Authorization Code(Step 6)?

* User Authorization :The client (typically a web application) sends a request for authorization to the Authorization Server, often through the front-end browser, redirecting the user to the authorization server's authorization endpoint. This request includes a redirect URI where the Authorization Server will send the user back to after they authorize the app.
* Server Approval: The user authorizes or denies the client's access request. If the user approves, the Authorization Server redirects the user to the redirect URI provided earlier by the client, and it includes an Authorization Code as a query parameter in this redirect URI.
* Code Exchange: This Authorization Code is then taken from the browser's address bar by the back-end part of the client, and the back-end uses this code to request an access token from the Authorization Server's token endpoint. This request is made from the server side (the back channel), and it contains the Authorization Code and the exact same redirect URI (for verification).
* Token Response: The Authorization Server validates the Authorization Code and redirect URI, and upon successful validation, it sends an Access Token (and optionally a Refresh Token) back to the client.

Sample code for Frontend Backend Channel

=====================================

//On Frontend

<!-- Redirect the user to OAuth server for authentication -->

<a href="http://localhost:3000/auth">Login</a>

On Backend:

var express = require('express');

var request = require('request');

var qs = require('querystring');

var app = express();

// Configuration

var client\_id = 'YOUR\_CLIENT\_ID';

var client\_secret = 'YOUR\_CLIENT\_SECRET';

var redirect\_uri = 'http://localhost:3000/callback';

var auth\_url = 'https://YOUR\_AUTH\_SERVER/authorize';

var token\_url = 'https://YOUR\_AUTH\_SERVER/token';

// Redirect user to OAuth server for authentication

app.get('/auth', function(req, res) {

res.redirect(auth\_url + '?' + qs.stringify({

client\_id: client\_id,

response\_type: 'code',

redirect\_uri: redirect\_uri

}));

});

// Callback endpoint, exchange the authorization code for an access token

app.get('/callback', function(req, res) {

var code = req.query.code; //Here the Authorization code is been picked from queryString

request.post(token\_url, {

form: {

grant\_type: 'authorization\_code',

code: code,

redirect\_uri: redirect\_uri,

client\_id: client\_id,

client\_secret: client\_secret

},

json: true

}, function(error, response, body) {

if (!error && response.statusCode === 200) {

// Successful token request, do something with the token (body.access\_token)

} else {

// Token request error

}

});

});

var server = app.listen(3000);

**How is Autorization Code Misused?**

**-----------------------------------------------------**

Authorization Code is Opaque key, without much information , BC-APP1 Shares Authorization Code + SALT Value with Auth-Server , Even If the Authorization Code is leaked , the salt value is known only to BC-APP1 and Authorization Server.

\*\*When OAUTH was introduced , It was mistakenly used for Authentication rather than delegated Authorization .

OAUTH was used to identify the user rather than focussing on access rights which is NOT CORRECT.

OAUTH at the nd gives us an access token which doesn’t contain any user/profile information like name, email,phone etc It contains only permissions for accessing the resources .

Everyone started writing hacks to authenticate using access token which is not the purpose of OAUTH this is not standarized way of OAUTH.

So to handle the authentication along with Authorization using OAUTH , **OPENID CONNECT was introduced.**

## OAUTH + OPENID CONNECT

**What is OpenID Connect?**

* Allows client to Verify the identity of end-users and get basic profile information about.
* OpenID Connect is a simple identity layer on top of OAUTH which tackles the identity problems i.e Authentication. OpenID Connect is not a separate protocol , Its meant to work along with OAUTH for authentication and Authorization.

Flow With OpenID Connect + OAUTH is as below shown

A diagram of a software process

Description automatically generated

Assumptions:

1. There is an APP which want to talk to Resource server on Behalf of User.
2. That APP is APP1 Which has front Channel APP1(FC-APP1) and Back Channel APP1 (BC-APP1)

Step 1: User access Front Channel of APP1,

Step 2: FC-APP1 sends a request to Authorization Server with following payload

Redirection URL: After auth flow success where to go the URL

Response Type: Code, the code here indicates this “Authorization Code Flow”.

Scope: FC-APP1 request for permissions, read/write/read-write and **“openid”**

Step 3: Auth server prompts the User for his consent

Step 4: Assuming users give consent for the Prompt

Step 5: **Authorization code** is shared to FC-APP1

Step 6: Back channel communication , this happens backside cannot see these calls in the browser. These are internal calls , BC-APP1 takes the Authorization Code from FC-APP1 (Back Channel Communication)

Step 7: BC-APP1 sends **Authorization Code + SALT KEY** to Auth-Server .

Step 8: Auth-Server validates the Authorization Code and as response sends **Access Token and ID TOKEN** with required information **.**

[Id Token is used for authentication Purposes and Access Token is used for Authorization purposes](https://auth0.com/blog/id-token-access-token-what-is-the-difference/)

Step 9:Using theses tokens, APP1 can communicate with Resource Server.

Step 10: Resorce server once recieves the access token, this will send to Authorization server for verification and recieves claims and user data

## Scopes

**What are Scopes?**

Scopes are used to limit access to user data

During Authorization Request , Auth-Provider will list down all the available scopes to the user, so that he understands what rights is he giving to the 3rd Party Application . This is Step-4 Typically where the scopes are listed.

Scopes should be more specific and not generic like read/write/read-write

Lets say you have to access Twitter API, which has resources like Timeline, Followers Messages, Tweets

If the scopes are generic , like say read.. the 3rd Party Application can read all the resources so rather than generic read we can give scopes like

Read-Timeline, Read-Tweets ,Read-Messages etc

Similarly for other writes etc …. We can use mix and match..

## Refresh Tokens

**What are Refresh Tokens**

------------------------------------------

Allows access tokens to be renewed

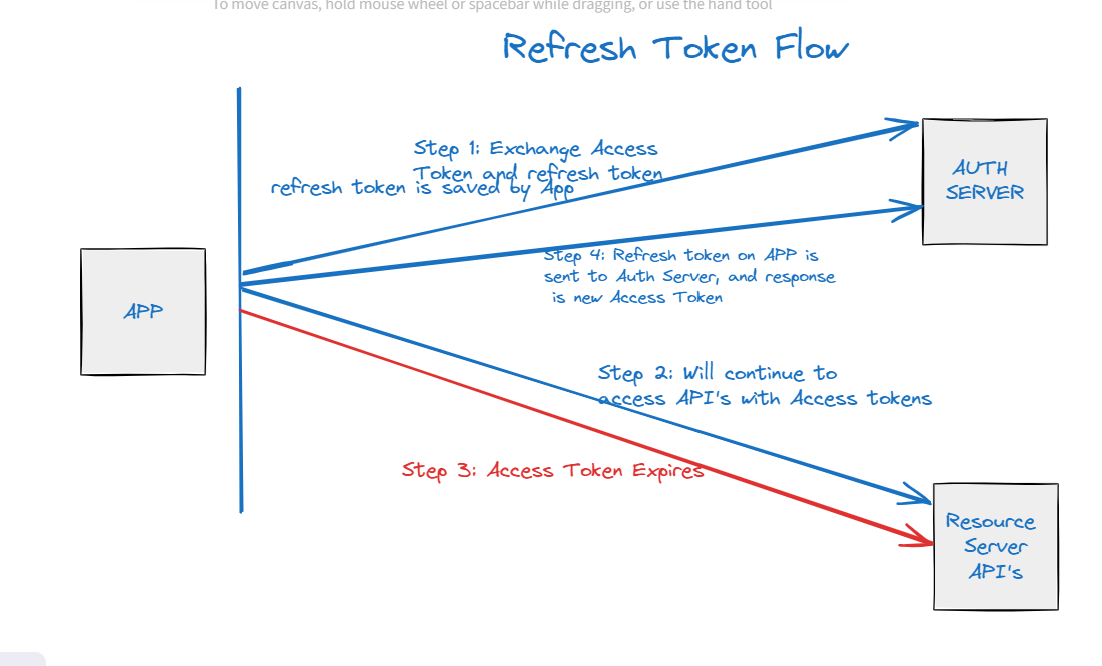
Access Tokens have a shelf life, even when compromised will not cause a devastating effect.

We can limit the Access token by providing a time periods , after a set time period 10mins or 30mins or an hour the access tokens are renewed . This provides another layer of security to our resources

Its Not a good idea to allow the access tokens live eternal…

The renew token is issued without the involvement of end user, the application(FC+BC) can provide its clientId and secret are passed to get new Access Token , It happens in back Channel only its risky to be on front channel.

ClientId is the unique Id using which the 3rd partyApplication registers with the Authorization Server



Step 1: Exchange Access Token and refresh token are sent by Auth Server. Refresh token is saved by Application user is accessing.

Step 2: Application will continue to access the Resource Server API with the access token that is provided by Auth Server.

Step 3: The Access token time elapsed and is expired .

Step 4: In that case the Application will send the refresh token to Auth server , Auth server in return will send the renewed Access Token

# WORKING OF CODE

## React Frontend

* \*\*\* Oauth+React+Node : <https://www.youtube.com/watch?v=GGGjnBkN8xk>
* Follow the example at C:\Ajay\_EPAM\PDP\_PRACTICSE\epampdp\AUTH\_AUTORIZATION\oAuth\auth-code-flow-demo on GitHub
* React Auth0 Packages Usage: <https://auth0.com/blog/complete-guide-to-react-user-authentication/>

Auth0 provides a package to work with authentication and Authorization with React

npm install @auth0/auth0-react

This @auth0/auth-react gives use few convenient methods like

 useAuth0, withAuthenticationRequired,Auth0Provider, useAuth0()

In short the auth0-react package will encapsulate all the logic that you have to do with Frontend-Channel and Backend-Channel and it will provide you with

Authorization-Code: getAccessTokenSilently or getAccessTokenWithPopup

Id Token: getIdTokenClaims

## NodeJs Backend

For NodeJs we need a package to be installed “npm i express-oauth2-jwt-bearer”

The auth method in the package will retrieve the bearer token and Id token and will give access to protected resources. All this is encapsulated

# Refresh Token Implementation

1. **Enable Refresh Token Rotation**: In the Auth0 dashboard, go to your application settings and enable "Refresh Token Rotation" under the "Advanced Settings" section.
2. **Set up API Scopes**: Ensure your API in Auth0 has the necessary scopes. You might need a scope like **offline\_access** for refresh tokens.
3. The "offline\_access" scope in OAuth is used when your application needs to access an API even when the user is not present or logged in. This might be the case for apps that need to interact with a user's data in the background, like scheduled email sync, database updates, backups, etc.
4. When you include the "offline\_access" scope in your request, the authorization server returns a refresh token along with the access token. This refresh token can be used to obtain a new access token once the current one expires.
5. Therefore, the "offline\_access" scope is not about offline access in the offline/online sense, but about being able to access the API on behalf of the user while they aren't actively using your application.
6. Do note, however, that usage of this scope might be subject to the user's approval, and the authorization server may also deny issuing refresh tokens based on its policy even when this scope is requested.

import createAuth0Client from '@auth0/auth0-spa-js';

const auth0 = await createAuth0Client({

domain: 'YOUR\_DOMAIN',

client\_id: 'YOUR\_CLIENT\_ID',

redirect\_uri: window.location.origin,

audience: 'YOUR\_API\_IDENTIFIER',

scope: 'openid profile email offline\_access',

});

// Log in the user and get the tokens

await auth0.loginWithRedirect();

const token = await auth0.getTokenSilently();

const refreshToken = auth0.getRefreshToken(); // Store this securely

Set up Axios Instance:

import axios from 'axios';

import createAuth0Client from '@auth0/auth0-spa-js';

// Create an axios instance

const api = axios.create({

baseURL: 'https://your-api-url.com',

});

// Function to get access token from Auth0

const getAccessToken = async () => {

return await auth0.getTokenSilently();

};

// Function to refresh access token

const refreshAccessToken = async () => {

const refreshToken = localStorage.getItem('refresh\_token'); // or from other secure storage

const { data } = await axios.post('https://your-domain/oauth/token', {

grant\_type: 'refresh\_token',

client\_id: 'YOUR\_CLIENT\_ID',

refresh\_token: refreshToken,

});

// Store new access token and refresh token

localStorage.setItem('access\_token', data.access\_token);

if (data.refresh\_token) {

localStorage.setItem('refresh\_token', data.refresh\_token);

}

return data.access\_token;

};

// Add a request interceptor

api.interceptors.request.use(

async (config) => {

const token = await getAccessToken();

if (token) {

config.headers['Authorization'] = `Bearer ${token}`;

}

return config;

},

(error) => {

return Promise.reject(error);

}

);

// Add a response interceptor

api.interceptors.response.use(

(response) => {

return response;

},

async (error) => {

const originalRequest = error.config;

// Check if the error is due to expired token

if (error.response.status === 401 && !originalRequest.\_retry) {

originalRequest.\_retry = true;

try {

// Attempt to refresh the token

const newAccessToken = await refreshAccessToken();

api.defaults.headers.common['Authorization'] = `Bearer ${newAccessToken}`;

originalRequest.headers['Authorization'] = `Bearer ${newAccessToken}`;

// Retry the original request

return api(originalRequest);

} catch (refreshError) {

// Handle refresh token failure (e.g., logout the user)

console.error('Token refresh failed:', refreshError);

// Redirect to login or logout user

}

}

return Promise.reject(error);

}

);

export default api;

### Complete Flow

1. **User logs in** and receives an access token and a refresh token.
2. **Access token is used** for API requests via Axios.
3. If an API request returns **401 Unauthorized**:
   * **Axios interceptor** catches this and uses the refresh token to get a new access token.
   * **New access token** is used to retry the failed request.

### Security Considerations

* **Token Storage**: Storing tokens in localStorage can expose them to XSS attacks. Consider more secure storage methods.
* **Token Rotation**: Regularly rotate refresh tokens to reduce the risk of token leakage.
* **Error Handling**: Ensure your application gracefully handles token refresh failures, potentially redirecting users to login.

# **Database Connectivity with OAUTH**

Here is a general outline for configuring an Auth0 Authorization Server with a database of users:

1. Database Setup: Auth0 allows you to use their own user-store or you can connect to your external user database. Auth0 user-store provides a ready-to-use Data Store.
2. Set Up Connection: To use the Auth0 user store, you just need to create a Database Connection on their dashboard. If you prefer to use your own user database, you need to create a "Custom Database" connection.
3. User Authentication: Auth0 comes with built-in username/password authentication so you don't have to set up your own authentication protocol.
4. Integration of Auth0 with Database: When creating the connection, you need to configure the custom database scripts that allow Auth0 to communicate with your database. These scripts are for actions like getting users, creating users, or verifying passwords.
5. Securing User Credentials: Auth0 already takes care of securing credentials, it encrypts passwords with bcrypt and you can also enable multi-factor authentication.
6. Configuration of Auth0 Authorization Server: The Auth0 dashboard allows easy configuration of client applications that need access and they provide an interface to handle user consents.
7. Building Authorization Login Pages: Auth0 includes a hosted login page that you can customize to match your brand.

Remember to replace **YOUR\_DOMAIN** in the authorization URL with your Auth0 domain.

This is a high-level overview, the actual steps needed will depend on the details of your particular system setup. Always refer to Auth0’s official documentation for the most accurate and detailed instructions.

## GENERAL FLOW

1. Database Setup: You'd need a user database that stores information about your users. This could be a traditional SQL database, a NoSQL database, or even a cloud identity service like Firebase or Okta. The database should contain user information (like usernames, emails, hashed passwords, and roles/permissions).
2. User Authentication: The Authorization Server needs to authenticate users before it can issue authorization codes. This is typically done by integrating an authentication protocol (e.g., LDAP, SAML) or an authentication service with the Authorization Server.
3. Integration of Authorization Server with Database: Connect your Authorization Server with your user database. It usually involves configuring your Authorization Server to be able to read from the user database. This can be achieved either through direct database connections (JDBC for Java, ODBC for .NET), proprietary APIs, or database drivers specific to your technology stack.
4. Securing User Credentials: Ensure user credentials are stored securely, usually by hashing and salting. The Authorization Server should never store plain text passwords.
5. Configuration of Authorization Server: You would need to configure your Authorization Server with a series of client applications it trusts. This trust is established by registering the client ID, the client secret and the redirect URIs of each client application with the Authorization Server.
6. Building Authorization Login Pages: An interface is required to capture user credentials. This usually manifests as a login form within a User Interface (UI), which communicates with the Authorization Server.

# MISC

## Different Code Types

1. Code: Used in Authorization Code and hybrid flows. This is the most common response type.
2. Token: Returns an Access Token to the client application directly. This is used in the Implicit flow.
3. ID\_token: The ID token is a type of token that is a JWT (JSON Web Token). This returns an ID Token directly to the client. It's typically used in OpenID Connect flows.
4. Code ID\_token: Returns both an authorization code and ID Token from the /authorize endpoint.
5. Code Token: Returns an authorization code and Access Token from the /authorize endpoint.
6. Code ID\_token Token: Returns an authorization code, ID Token and Access Token from the /authorize endpoint.
7. None: Provides a way for OAuth clients to perform RP-initiated (relying party-initiated) logouts.

The response type that's best for your application will depend on several factors, including what type of client it is (web-based, native, etc.), and whether it can securely store client secrets.

# How to Handle Session Timeout

import React, { useState, useEffect, useCallback } from 'react';

Create a useSessionTimeout custom hook to manage the session timeout logic.

// Custom Hook for session timeout

const useSessionTimeout = (timeoutDuration, warningDuration) => {

const [timeout, setTimeout] = useState(timeoutDuration);

const [warning, setWarning] = useState(false);

const resetTimeout = useCallback(() => {

setWarning(false);

setTimeout(timeoutDuration);

fetch('/extend-session', {

method: 'POST',

credentials: 'include'

})

.then(response => {

if (response.ok) {

console.log('Session extended');

} else {

console.log('Failed to extend session');

}

})

.catch(error => console.error('Error extending session:', error));

}, [timeoutDuration]);

useEffect(() => {

const warningTimer = setTimeout(() => {

setWarning(true);

}, timeoutDuration - warningDuration);

const timeoutTimer = setTimeout(() => {

// Add your logout logic here

alert('Session expired. Logging out...');

// Example: window.location.href = '/logout';

}, timeoutDuration);

return () => {

clearTimeout(warningTimer);

clearTimeout(timeoutTimer);

};

}, [timeoutDuration, warningDuration, resetTimeout]);

return [warning, resetTimeout];

};

#### //Step 2: Using the Hook in a Component

Use the custom hook in your main component or a higher-order component that wraps around your app.

const App = () => {

const [warning, resetTimeout] = useSessionTimeout(30 \* 60 \* 1000, 5 \* 60 \* 1000); // 30 min session, 5 min warning

useEffect(() => {

const handleActivity = () => {

resetTimeout();

};

window.addEventListener('mousemove', handleActivity);

window.addEventListener('keydown', handleActivity);

return () => {

window.removeEventListener('mousemove', handleActivity);

window.removeEventListener('keydown', handleActivity);

};

}, [resetTimeout]);

return (

<div>

{warning && (

<div className="session-warning">

<p>Your session is about to expire. Click anywhere to extend your session.</p>

</div>

)}

{/\* Your app components go here \*/}

</div>

);

};

export default App;

### Step 4: Backend Support

Ensure your backend supports session renewal. For instance, in an Express.js backend, you might have a route to extend the session.

app.post('/extend-session', (req, res) => {

if (req.session) {

req.session.cookie.maxAge = 30 \* 60 \* 1000; // 30 minutes

res.sendStatus(200);

} else {

res.sendStatus(401);

}

});