# OAUTH 2.0 DOCUMENTATION

ABCircle Systems Inc.

By: Gerard V. Ambrocio

**What is OAuth 2.0?**

OAuth 2.0 is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service, such as Facebook, GitHub, and DigitalOcean. It works by delegating user authentication to the service that hosts the user account, and authorizing third-party applications to access the user account. OAuth 2.0 provides authorization flows for web and desktop applications, and mobile devices.

**OAuth 2.0 terminology**

- **Resource Owner** (User): Entity that can grant access to a protected resource. Typically, this is the end-user.
- **Client** (Web App): Application requesting access to a protected resource on behalf of the Resource Owner.
- **Resource Server** (Your API): Server hosting the protected resources. This is the API you want to access.
- **Authorization Server**: Server that authenticates the Resource Owner and issues Access Tokens after getting proper authorization.
- **User Agent**: Agent used by the Resource Owner to interact with the Client (for example, a browser or a native application).

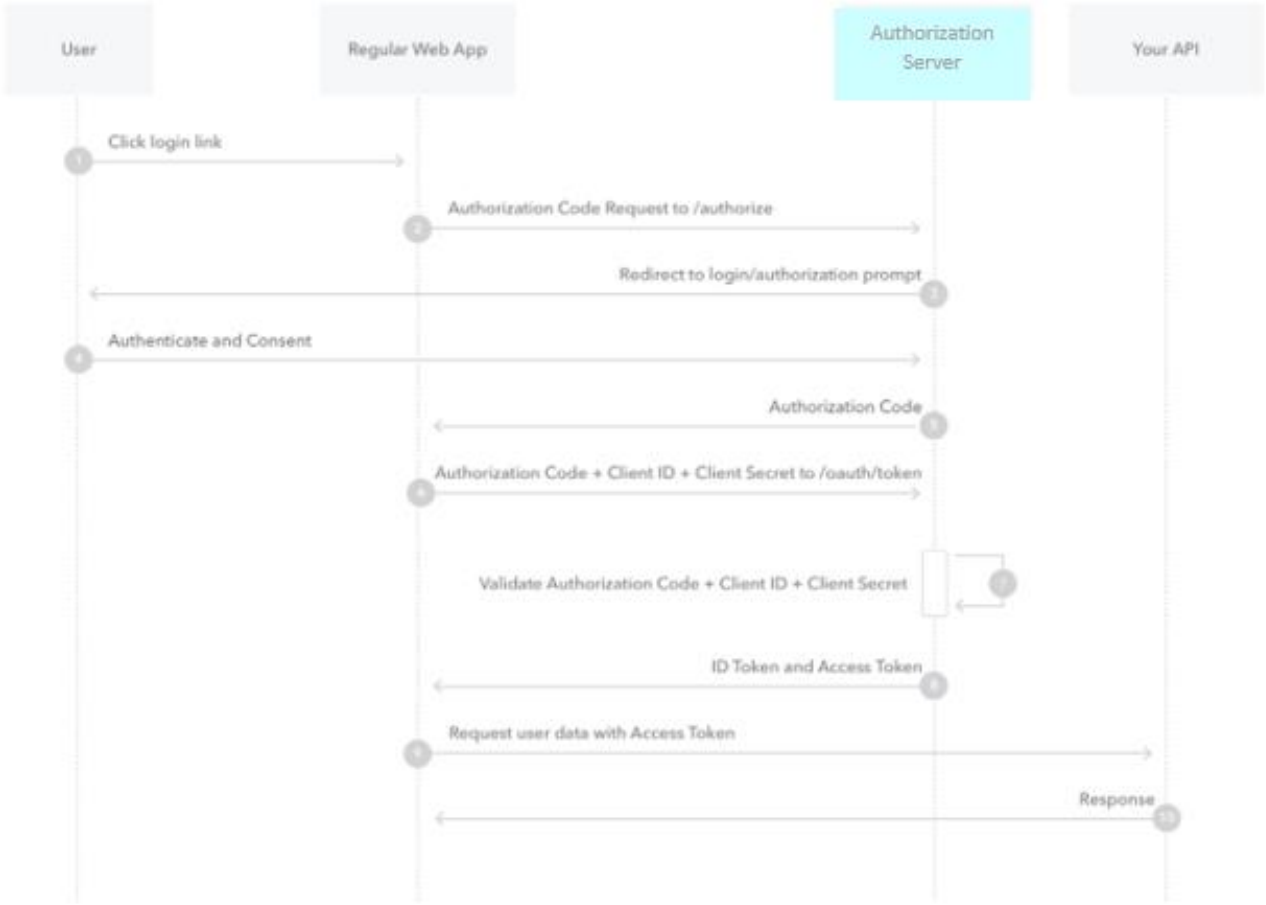
**Other important terms:**

- An OAuth 2.0 **"grant"** is the authorization given (or "granted") to the client by the user. Examples of grants are "authorization code" and "client credentials". Each OAuth grant has a corresponding flow.
- The **"access token"** is issued by the authorization server in exchange for the grant.

**What Flow are we using? –** Authorization Code Flow with PKCE extension **–** which obtains an access token to authorize API requests, for this flow the "authorization code" grant is used. The PKCE extension is recommended for additional security.

**Authorization code flow** is the most flexible of the supported authorization flows and is the recommended method of obtaining an access token for the API. This authorization flow is best suited to applications that have access to secure, private storage such as web applications deployed on a server.

**Standard Authorization Code Flow**




**Access tokens**, obtained using authorization code flow, provide permissions for your application to manipulate documents and other resources on behalf of the user and make requests for all API resources. Access tokens while having a limited lifetime, can be renewed with a *refresh* token. A **refresh token** is valid indefinitely and provides ability for your application to schedule tasks on behalf of a user without their interaction.

**PKCE**

When using the Authorization Code flow in a mobile app, or any other type of application that can't store a client secret, the PKCE extension is recommended. **Proof Key for Code Exchange or PKCE** (pronounced "pixie"), provides protections against other attacks where the authorization code may be intercepted.

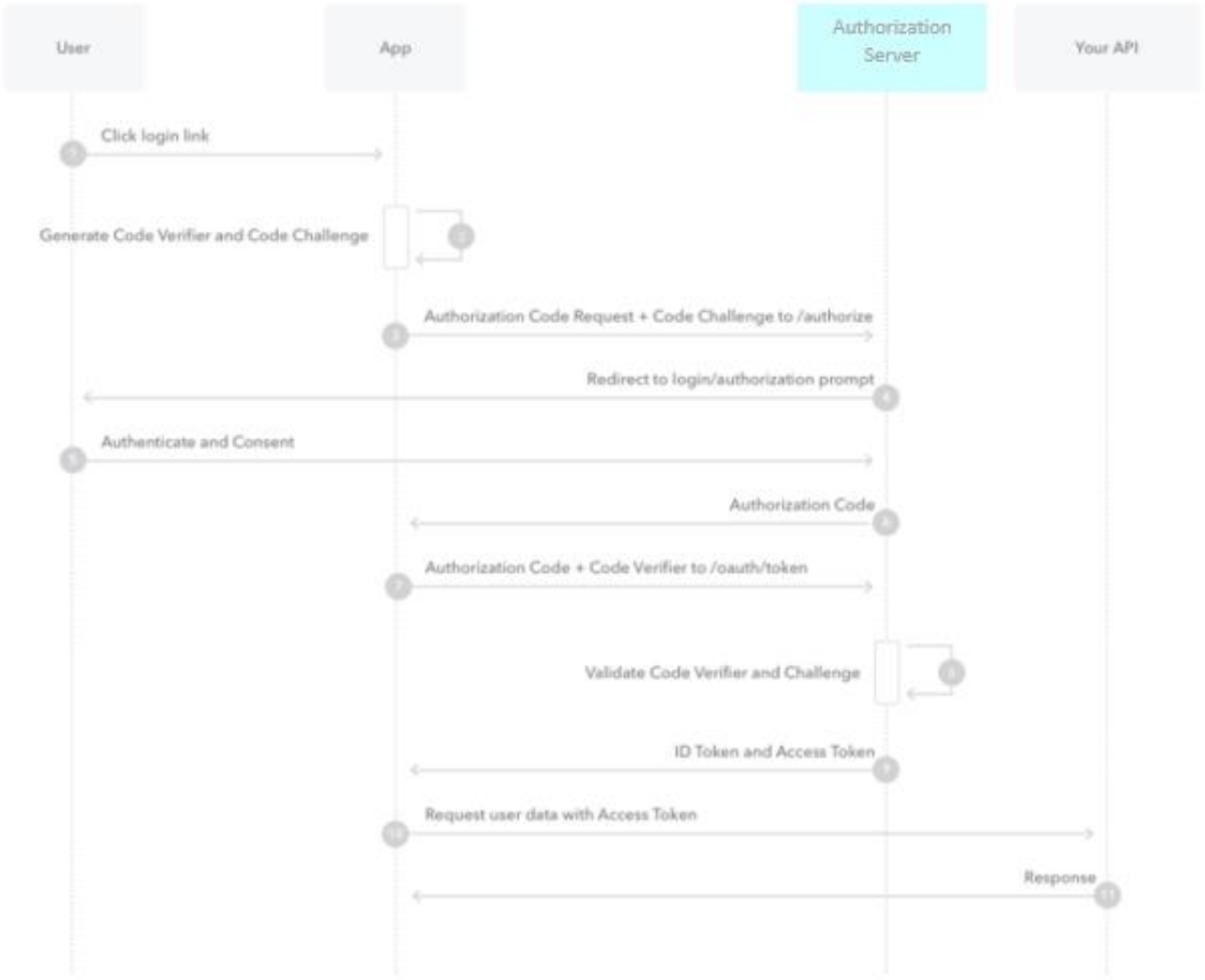
**The Difference vs. Standard Authorization Code Flow**

Compared to the standard authorization code flow, PKCE works by having the app generate a random value at the beginning of the flow called a **Code Verifier**. The app hashes the Code Verifier and the result is called the Code Challenge. The app then kicks off the flow in the normal way, except that it includes the Code Challenge in the query string for the request to the Authorization Server.

The Authorization Server stores the hashed value (the Code Challenge) for later verification and, after the user authenticates, redirects back to the app with an authorization code.

The app makes the request to exchange the code for tokens, only it sends the Code Verifier instead of a fixed secret. Now the Authorization Server can hash the Code Verifier and compare it to the hashed value it stored earlier. This is an effective, dynamic stand-in for a fixed secret. Assuming the hashed value matches, the Authorization Server will return the tokens.

**Authorization Code Flow with PKCE extension**




**Following the flow**

To demonstrate the Authorization code flow with PKCE extension, a sample code is provided to understand the basics of how to implement it. This is done through following the numbered scheme from the diagram above, where the "User" utilizing a "Client App" is trying to access a protected resource, "Your API" authorized through an "Authorization Server" which applies OAuth 2.0.

Beginning from step ⓪, where the applications are setup until step ⑪, where client application that has been given access, retrieves the protected resource.

⓪ Client App + Authorization Server + Your API – **Setting up the Applications**

Three web application projects must be made accordingly, a client application, an authorization server and your API / protected resource.

**Add the correct middleware to the three projects**

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints => {
        endpoints.MapDefaultControllerRoute();
    });
}
```

The most important ones are app.UseAuthentication(); and app.UseAuthorization(); added in the exact order as there have been known bugs for enabling one before the other. Also include app.UseHttpsRedirection(); and ensure the application uses HTTPS as it is required by OAuth 2.0.

**Configure the services of the Client Application**

```
public void ConfigureServices(IServiceCollection services)
{

   services.AddAuthentication(config => {
        config.DefaultAuthenticateScheme = "ClientCookie";
        config.DefaultSignInScheme = "ClientCookie";
        config.DefaultChallengeScheme = "OurServer";
    })
        .AddCookie("ClientCookie")
    A .AddOAuth("OurServer", config => {
            config.ClientId = "client_id";
            config.ClientSecret = "client_secret";
            config.CallbackPath = "/oauth/callback";
            config.UsePkce = true;
            config.AuthorizationEndpoint = "https://authenticationserver.local:447/oauth/authorize";
            config.TokenEndpoint = "https://authenticationserver.local:447/oauth/token";
            config.SaveTokens = true;

        B   config.Events = new OAuthEvents() {
                OnCreatingTicket = context => {
                    var accessToken = context.AccessToken;
                    var base64payload = accessToken.Split('.')[1];
                    var bytes = Decode.DecodeUrlBase64(base64payload);
                    var jsonPayload = Encoding.UTF8.GetString(bytes);
                    var claims = JsonConvert.DeserializeObject<Dictionary<string, string>>(jsonPayload);

                    foreach (var claim in claims)
                    {
                        context.Identity.AddClaim(new Claim(claim.Key, claim.Value));
                    }

                    return Task.CompletedTask;
                }
            };
        });

    C services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });
```

Initially, the default schemes must be named and established especially the default challenge scheme to authenticate the unauthorized route later on. A cookie scheme is set as well in order for easy access for the user through the cookie once the proper authentications have already been done.

A – In this section we added OAuth 2.0 and with the same name established, is set as our default challenge scheme. The configurations must be named accordingly, especially the Callback Path, Authorization and Token Endpoints. The ClientSecret which is supposed to be used to validate the client application in the standard authorization code flow is no longer necessary, while the ClientId will still be required for this grant type.  This is because of utilizing PKCE by setting config.UsePkce to be true, the code verifier is now the value to be validated later. Also, it is important to add config.SaveTokens to be true in order for the tokens to be saved when received by the client application.

B – Using an additional option of adding an event using the class OAuthEvents(), the access token may be repurposed. The received access token may be converted and collect the claims specified inside them.

C – Additional options to enable the storage of cookies in the browser.

**Configure the services of the Authentication Server**

```
public void ConfigureServices(IServiceCollection services)
{

A  services.AddDbContext<ApiDbContext>(options =>
       options.UseSqlServer(
           Configuration.GetConnectionString("DefaultConnection")));

B  services.AddIdentity<IdentityUser, IdentityRole>(options => options.SignIn.RequireConfirmedAccount = true)
       .AddEntityFrameworkStores<ApiDbContext>();

   services.ConfigureApplicationCookie(options => {
       options.LoginPath = $"/";
   });

   services.AddControllersWithViews()
       .AddRazorRuntimeCompilation();

   services.AddControllers();
}
```

A – It is important to configure the database settings properly as the user that logs in will be checked against what is stored inside the database. Properly set the connection string in appsettings.json.

B – ASP.NET Identity system is also used and configured in the authentication server code. This is important to configure in this manner as using the services.AddDefaultIdentity instead sets the redirect to Account/Login. Also, adding the login path to be “/” reenforces the login paths established by the client we set earlier.

**Configure the services of Your API**

```
public void ConfigureServices(IServiceCollection services)
{
A   services.AddDbContext<ApiDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

B   services.AddAuthentication("DefaultAuth")
        .AddScheme<AuthenticationSchemeOptions, CustomeAuthenticationHandler>("DefaultAuth", null);

C   services.AddAuthorization(config => {
        var defaultAuthBuilder = new AuthorizationPolicyBuilder();
        var defaultAuthPolicy = defaultAuthBuilder
            .AddRequirements(new JwtRequirement())
            .Build();

        config.DefaultPolicy = defaultAuthPolicy;
    });

    services.AddScoped<IAuthorizationHandler, JwtRequirementHandler>();

D   services.AddHttpClient()
        .AddHttpContextAccessor();

    services.AddControllers();

    services.AddSwaggerGen(c => {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "TodoAPI", Version = "v1" });
    });

}
```

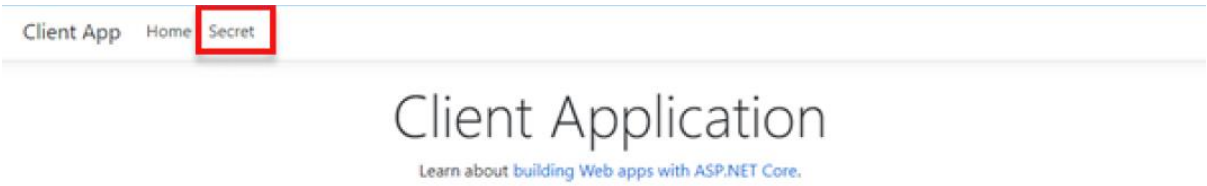
A – It is important to configure the database settings properly as the user that logs in will be checked against what is stored inside the database. Properly set the connection string in appsettings.json.

B – A custom authentication scheme is set as there is no actual need for authentication in this application but by default when a challenge is made, authentication will be triggered first.

C – Using an Authorization policy builder to implement an authorization policy where the requirements will be using a handler to authorize. Where in the next line this handler is set and configured in the JwtRequirementHandler class.

D – This will allow us to access the http context and extract the token from the header.

① User => Client App – **Click login Link or Protected Resource**




In this step, either trying to login or access a protected resource with the [Authorize] attribute in their route will trigger the default challenge scheme that was set.

② Client App – **Generate Code Verifier and Code Challenge**

In this step a code_challenge is generated using the code verifier and the code_challenge_method. In the startup.cs, while adding Oauth as the default challenge scheme, config.UsePkce was used and this automatically generates a code_challenge. Similarly, the GenerateCodeChallenge() method shown, is the exact conversion when using SHA256 which is the code_challenge_method.

```
private static string GenerateCodeChallenge(string codeVerifier)
{
    using var sha256 = SHA256.Create();
    var hash = sha256.ComputeHash(Encoding.UTF8.GetBytes(codeVerifier));
    var b64Hash = Convert.ToBase64String(hash);
    var code = Regex.Replace(b64Hash, "\\+", "-");
    code = Regex.Replace(code, "\\/", "_");
    code = Regex.Replace(code, "=+$", "");
    return code;
}
```

③ Client App => Authorization Server – **Authorization Code Request + Code Challenge to /authorize**

4.1.1. Authorization Request

The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the "application/x-www-form-urlencoded" format, per Appendix B:

response_type
REQUIRED. Value MUST be set to "code".

client_id
REQUIRED. The client identifier as described in Section 2.2.

redirect_uri
OPTIONAL. As described in Section 3.1.2.



scope
OPTIONAL. The scope of the access request as described by Section 3.3.

state
RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in Section 10.12.

Initially, the AuthorizationEndpoint was set to oauth/authorize which directs to our authorize route in the oauthcontroller and receives the parameters that is expected based on official specification. The parameters of note are the code_challenge_method and the code_challenge that was sent by the client application because of PKCE and it is identified by its client_id. Additionally, it is worth mentioning the response_type. “code”, which means that the client app informs the authorization server of the desired grant type it will be using, in this case the authorization code grant type. Finally, a view/page is returned to the client application to be signed-in by the user via the callback path or the redirect_uri.

```
[HttpGet]
0 references
public IActionResult Authorize(
    string code_challenge_method,
    string code_challenge,
    string response_type,
    string client_id,
    string redirect_uri,
    string scope,
    string state)
{

    if (code_challenge != null)
    {
        codeChallenge = code_challenge;
    }
    if (redirect_uri != null)
    {
        storeRedirectUri = redirect_uri;
    }
    if (state != null)
    {
        storeAuthState = state;
    }


    var query = new QueryBuilder();
    query.Add("redirectUri", storeRedirectUri);
    query.Add("state", storeAuthState);

    return View(model: query.ToString());
}
```

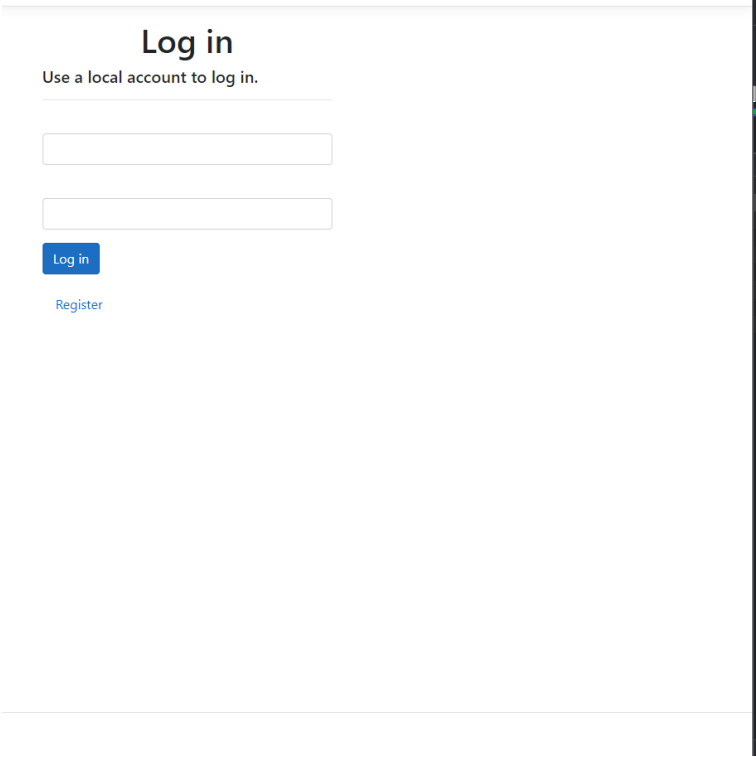
code_challenge_method: "5256"

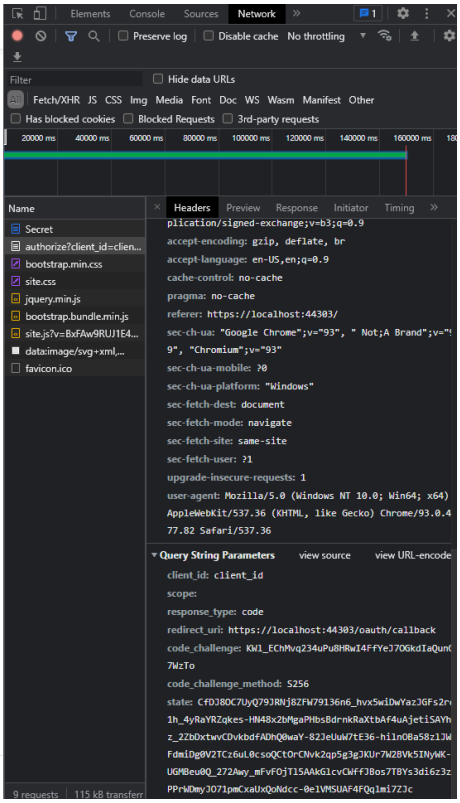
code_challenge: "RCI5V1tTSb2_EA8ZjEVVcqBGfhDqg0LyO7V5d_VmY9E"

response_type: "code"

client_id: "client_id"

④ Authorization Server => User – **Redirect to login/authorization prompt**



The user receives a login page where their credential should be inputted correctly. Alternatively, a register link should be included by the user when the current user is not in the database yet.

⑤ User => Authorization Server – **Authenticate and Consent**

```
[HttpPost]
0 references
public async Task<IActionResult> Authorize(
    string username,
    string password,
    string redirectUri
    string state)
{ ≤ 13,521ms elapsed

    var user = new UserLoginRequest();
    user.Email = username;
    user.Password = password;

A   if (ModelState.IsValid)
    {
        var existingUser = await _userManager.FindByEmailAsync(user.Email);

        if (existingUser == null)
        {
            return BadRequest(new RegistrationResponse()
            {
                Errors = new List<string>() {
                    "Invalid login request"
                },
                Success = false
            });
        }

        var isCorrect = await _userManager.CheckPasswordAsync(existingUser, user.Password);

        currentUser = existingUser;

        if (!isCorrect)
        {
            return BadRequest(new RegistrationResponse()
            {
                Errors = new List<string>() {
                    "Invalid login request"
                },
                Success = false
            });
        }

B       var code = GenerateRandom();

        authorizationCode = code;

        var query = new QueryBuilder();
        query.Add("code", code);
        query.Add("state", state);
    return Redirect($"{redirectUri}{query.ToString()}");
    }

    return BadRequest(new RegistrationResponse()
    {
        Errors = new List<string>() {
                "Invalid login request"
            },
        Success = false
    });
```

username: "e@e.com"
password: "password10A!"
redirectUri: "https://localhost:44303/oauth/callback"
state: "CfDJ8OgMEuOD1KdMhWhd_1YM1pHyUcR2HoLr21PgpM-WWcTlxhvGKTJpWWeLIle9fQJxmDA/

Initially, the parameters received by the authorization endpoint through its POST route shows that it accepts the username and password determined by the user. Additionally, the callback path or redirectUri is set, this was also used earlier and this route is not seen by the user, it is basically an intermediary route to assist in the flow. Shown above is an example when using the callback route in debug for Visual Studio, although based on our example for IIS deployment it should be: https://authenticationserver.local:447/oauth/callback.

A – This series of code validates the username and password against the set database. Also, the model state check is recommended to set the consistency of responses as well as an easy check to setup.

⑥ Authorization Server => Client App – **Authorization Code**

B – After the credentials are validated, the user will now generate an Authorization code that the client application should receive. It is then sent as an authorization response to the client where it saves the code it receives.

4.1.2. **Authorization Response**

If the resource owner grants the access request, the authorization server issues an authorization code and delivers it to the client by adding the following parameters to the query component of the redirection URI using the "application/x-www-form-urlencoded" format, per Appendix B:

code
REQUIRED. The authorization code generated by the authorization server. The authorization code MUST expire shortly after it is issued to mitigate the risk of leaks. A maximum authorization code lifetime of 10 minutes is RECOMMENDED. The client MUST NOT use the authorization code



more than once. If an authorization code is used more than once, the authorization server MUST deny the request and SHOULD revoke (when possible) all tokens previously issued based on that authorization code. The authorization code is bound to the client identifier and redirection URI.

state
REQUIRED if the "state" parameter was present in the client authorization request. The exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
          &state=xyz
```

The client MUST ignore unrecognized response parameters. The authorization code string size is left undefined by this specification. The client should avoid making assumptions about code value sizes. The authorization server SHOULD document the size of any value it issues.

⑦ Client App => Authorization Server – **Authorization Code + Code Verifier to /oauth/token**

The client then after it receives the authorization response it will send an access token request to the authorization server. This request routes to the TokenEndpoint that the client application set previously. With this step, there is blackbox or built-in sequence that uses again the callback path route to assist the flow and automatically redirect the client application to /oauth/token route. The parameters according the official documentation will be shown in the TokenEndpoint.

### 4.1.3. Access Token Request

The client makes a request to the token endpoint by sending the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type
: REQUIRED. Value MUST be set to "authorization_code".

code
: REQUIRED. The authorization code received from the authorization server.

redirect_uri
: REQUIRED, if the "redirect_uri" parameter was included in the authorization request as described in Section 4.1.1, and their values MUST be identical.

client_id
: REQUIRED, if the client is not authenticating with the authorization server as described in Section 3.2.1.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1.





For example, the client makes the following HTTP request using TLS (with extra line breaks for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The authorization server MUST:

- require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- authenticate the client if client authentication is included,
- ensure that the authorization code was issued to the authenticated confidential client, or if the client is public, ensure that the code was issued to "client_id" in the request,
- verify that the authorization code is valid, and
- ensure that the "redirect_uri" parameter is present if the "redirect_uri" parameter was included in the initial authorization request as described in Section 4.1.1, and if included ensure that their values are identical.

⑧ Authorization Server – **Validate Code Verifier and Challenge**

```
public async Task<IActionResult> Token(
    string code_verifier,
    string grant_type,
    string code,
    string redirect_uri,
    string client_id,
    string client_secret
    )
{
    var convertedToCodeChallenge = GenerateCodeChallenge(code_verifier);

A   if (code != authorizationCode && convertedToCodeChallenge != codeChallenge)
    {
        return BadRequest(new AuthResult()
        {
            Errors = new List<string>() {
                "Invalid Authorization Code / Code Verifier"
            },
            Success = false
        });
    }

    var user = currentUser;

    var claims = new[] {
    new Claim ("Id", user.Id),
    new Claim (JwtRegisteredClaimNames.Email, user.Email),
    new Claim (JwtRegisteredClaimNames.Sub, user.Email),
    new Claim (JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
};

B   var secretBytes = Encoding.UTF8.GetBytes(_jwtConfig.Secret);
    var key = new SymmetricSecurityKey(secretBytes);

    var algorithm = SecurityAlgorithms.HmacSha256;
    var signingCredentials = new SigningCredentials(key, algorithm);

C   var token = new JwtSecurityToken(
        Constants.Issuer,
        Constants.Audiance,
        claims,
        notBefore: DateTime.Now,
        expires: grant_type == "refresh_token"
            ? DateTime.Now.AddMinutes(5)
            : DateTime.Now.AddMilliseconds(1),
        signingCredentials);

    var access_token = new JwtSecurityTokenHandler().WriteToken(token);

D   var responseObject = new
    {
        access_token,
        token_type = "Bearer",
        raw_claim = "oauthTutorial",
        refresh_token = "RefreshTokenSampleValueSomething77"
    };

    var responseJson = JsonConvert.SerializeObject(responseObject);
    var responseBytes = Encoding.UTF8.GetBytes(responseJson);

    await Response.Body.WriteAsync(responseBytes, 0, responseBytes.Length);

    return Redirect(redirect_uri);
```

Initially, the parameters that are sent by the client application includes the code_verifier which needs to be converted to a code challenge by the authorization server. It is also seen that the grant_type is indeed “authorization_code”, from this, the code which is sent by the authentication server first is received again. Additionally, the client secret can also be accessed which again is more critical to the standard authorization code flow.

A – In this section the code_challenge is converted from the code_verifier and along with the code received from the client app, it is compared to the values obtained previously. Assuming that the both the values match from the parameters attained earlier in the flow, no error response will be returned.

B – This series of code involves hashing the client_secret which is supposedly to be included in the access_token. The idea of this is other applications that will utilize the authorization server need to contain the values of the client secret in their token, especially when renewing it.

C – The token is then generated through a JWT security handler. The contents of this token may be set using the TokenValidationParameters way which is not shown here or directly set as shown. The values within the token include claims, issuers, audience, the signing credentials as well as the expiry of the token. It is here where the grant type is set now to be “refresh_token”. For demonstration purposes the expiry of the access token is in miliseconds and the refresh_token for 5 minutes.

D – The response object is then set to include the access token, the token type to be set to “Bearer” as well as including a temporary refresh token which can be generated randomly. The response object must again be a json token encoded into UTF-8 shown in the official documentation.

4.1.4. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 5.1. If the request client authentication failed or is invalid, the authorization server returns an error response as described in Section 5.2.



An example successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

⑨ Authorization Server => Client App – **ID Token and Access Token**

```
[Authorize]
0 references
public async Task<IActionResult> Secret()
{
    var apiResponse = await AccessTokenRefreshWrapper(
    () => SecuredGetRequest("https://ownedapi.local:446/api/Todo"));

    var responseString = await apiResponse.Content.ReadAsStringAsync();

    List<ItemData> items = JsonConvert.DeserializeObject<List<ItemData>>(responseString);

    return View(items);
}

1 reference
private async Task<HttpResponseMessage> SecuredGetRequest(string url)
{
    var token = await HttpContext.GetTokenAsync("access_token");

    var id_token = await HttpContext.GetTokenAsync("id_token");

    var client = _httpClientFactory.CreateClient();

    client.DefaultRequestHeaders.Add("Authorization", $"Bearer {token}");

    return await client.GetAsync(url);
}

1 reference
public async Task<HttpResponseMessage> AccessTokenRefreshWrapper(
    Func<Task<HttpResponseMessage>> initialRequest)
{
    var response = await initialRequest();

    if (response.StatusCode == System.Net.HttpStatusCode.Unauthorized)
    {
        await RefreshAccessToken();
        response = await initialRequest();
    }

    return response;
}
```

As seen from the series of code above, the token can be retrieved through the http context. A series of code includes the method to refresh the access token when it is already expired. The token must then be included in the header before the request proceeds. The Request then continues to the route set in order to retrieve the protected resource.

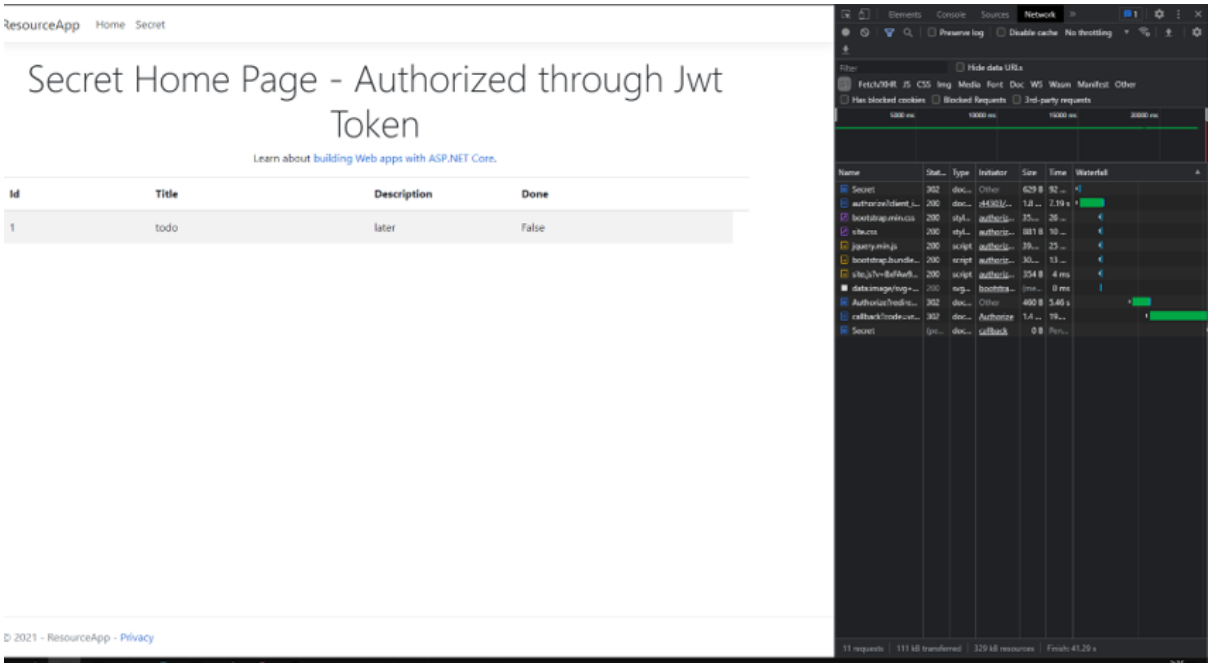
⑩ Client App => Your API or Protected Resource – **Request user data with Access Token**

```
public class JwtRequirement : IAuthorizationRequirement { }

2 references
public class JwtRequirementHandler : AuthorizationHandler<JwtRequirement>
{
    private readonly HttpClient _client;
    private readonly HttpContext _httpContext;

    0 references
    public JwtRequirementHandler(
        IHttpClientFactory httpClientFactory,
        IHttpContextAccessor httpContextAccessor)
    {
        _client = httpClientFactory.CreateClient(); // whats going to send the token to the server
        _httpContext = httpContextAccessor.HttpContext; // this is where we are getting the token
    }

    0 references
    protected override async Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        JwtRequirement requirement)
    {
        if (_httpContext.Request.Headers.TryGetValue("Authorization", out var authHeader))
        {
            var accessToken = authHeader.ToString().Split(' ')[1];

            var response = await _client
                .GetAsync($"https://localhost:44358/oauth/validate?access_token={accessToken}");

            if (response.StatusCode == System.Net.HttpStatusCode.OK)
            {
                context.Succeed(requirement);
            }
        }
    }
}
```

The access token is now retrieved by Your API, through the http context. The Authorization handler then processes the access token and includes it in a response to the authentication server to validate the token it just received through the established route. In that route the checks on the token are made by the authorization server and replies with an Ok response if the token is valid. But for the purposes of this sample code no checks were made.

```
[Authorize]
0 references
public IActionResult Validate()
{
    if (HttpContext.Request.Query.TryGetValue("access_token", out var accessToken))
    {

        return Ok();
    }

    return BadRequest();
}
```

⑪ Your API or Protected Resource => Client App – **Response**




Assuming the database configurations of Your API was set then the protected resource is returned to the client application. The user may then view or access the protected resource.

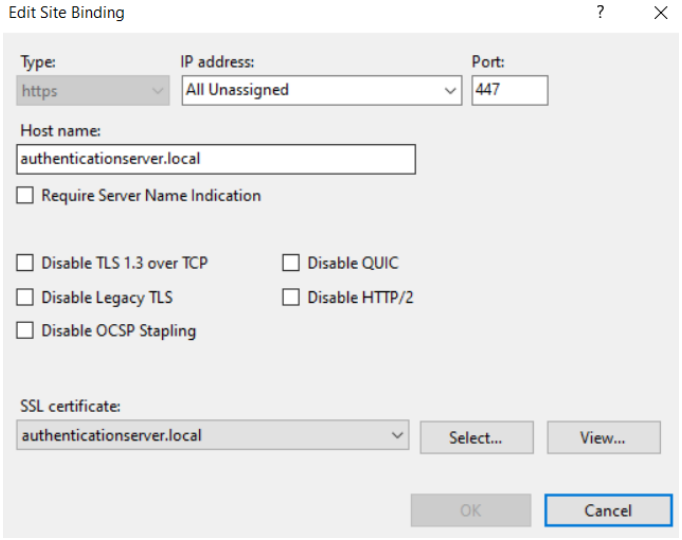
POTENTIAL ISSUES in IIS Deployment

- .Net 5

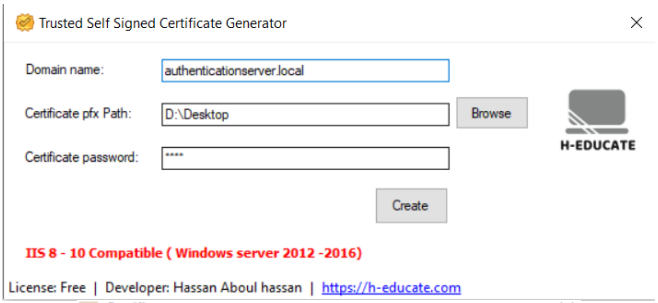
  There were some issues with .Net 5, that may or may not appear depending on the PC settings and configuration. It is therefore recommended for now to use .Net Core 3.1 for the Authorization Server Application.

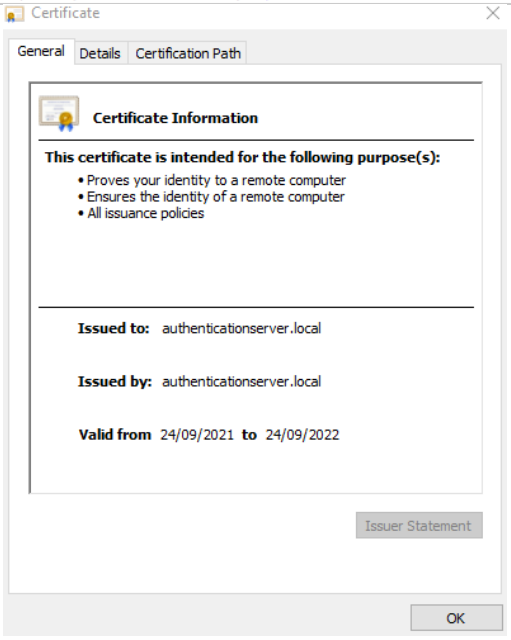
```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>


    <ItemGroup>
        <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="3.0.0" />
        <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="5.0.8" />

        <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="3.1.19" />
        <PackageReference Include="Microsoft.AspNetCore.Razor.Runtime" Version="2.2.0" />
        <PackageReference Include="Microsoft.EntityFrameworkCore" Version="5.0.8" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="5.0.8" />
        <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="5.0.8">
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
        </PackageReference>
        <PackageReference Include="Newtonsoft.Json" Version="13.0.1" />
    </ItemGroup>


    <ItemGroup>
      <Content Update="Views\Home\Index.cshtml">
        <ExcludeFromSingleFile>true</ExcludeFromSingleFile>
        <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
      </Content>
    </ItemGroup>


</Project>
```

- HTTPS
  OAuth 2.0 requires the use of HTTPS therefore when deploying in the IIS, it is then necessary to configure the bindings for https along with the SSLcertificate associated with it.




- SSL CERTIFICATE
  It is required as well by OAuth that the exchanges in the flow be secure that is why a valid SSL certificate is necessary. In IIS deployment, the simplest method I found was to use a third party application that configures your own self-signed certificate the proper values. Meaning that the “issued to:” and the “issued by:” in the certificate are for example: authentication server.local. It is recommended that the same is done for all the applications.



**Sources:**

Anton Wieslander's Project

https://github.com/T0shik/aspnetcore3-authentication

https://www.youtube.com/watch?v=Fhfvbl_KbWo&list=PLOeFnOV9YBa7dnrjpOG6lMpcyd7Wn7E8V

Mohamad Lawand's Project

https://www.youtube.com/watch?v=LgpC4tYtc6Y&t=3333s

https://github.com/mohamadlawand087/v7-RestApiNetCoreAuthentication

Official Documentation

https://datatracker.ietf.org/doc/html/rfc6749 - official Oauth 2.0 Framework docs

https://datatracker.ietf.org/doc/html/rfc7519 - official JWT docs

https://datatracker.ietf.org/doc/html/rfc7636 - official PKCE docs

https://datatracker.ietf.org/doc/html/rfc7617 - official basic http authentication scheme docs

Supporting Documentation

https://dev.mendeley.com/reference/topics/authorization_auth_code.html

https://developer.okta.com/docs/concepts/oauth-openid/#authentication-api-vs-oauth-2-0-vs-openid-connect

https://auth0.com/docs/authorization/flows/which-oauth-2-0-flow-should-i-use

Trust Self Signed certificate

https://www.youtube.com/watch?v=NuxlPZPDi_s

https://h-educate.com/trusted-self-signed-certificate-generator-tool/