Identity Server 4

The OpenID Connect Protocol in Practice

G. James McKenna

12/1/2021

This paper is written as a requirement for graduation; one the requirements for each of the two Internship Semesters. The topic I researched and wrote about for the first Internship Research Paper was the OpenID Connect Protocol. In that paper I wrote a lot about the protocols in theory. In this paper I want to discuss the OpenID Connect Protocol in practice: specifically an implementation of the open-source framework, Identity Server 4. I will focus on configuring the framework in an ASP.NET Application using companion .NET frameworks that integrate hand-in-glove with Identity Server 4 to provide a top-notch Authentication and Authorization service for applications following the Service Orientated Architecture.

Contents

Authentication and Authorization are the right and left hand of software security, and in today's world, almost every application reaches out to a backend of sorts. In this paper, I will demonstrate building and configuring a Secure Token Service (STS) so the reader will have a better understanding of what happens behind the curtain. How a token is built, how a token is used by a client application, how a token is passed to an API, and how the token is verified. Then what? Passing tokens is surface level, there's so much more happening behind the curtain.

With this paper, the reader will be better equipped when implementing an in-house, STS or configuring a Software-as-a-Service, STS. I will use the Identity Server 4, Open-source Framework to demonstrate many key topics. I will demonstrate the difference between an Identity Provider (IdP) and STS. I will show how an IdP and STS work together to provide Single Sign-on Authentication, Multi-Tenant Token Service and the difference between authorizing a User and authorizing a client application.

The Code

This paper is accompanied with a zip file named *GJM_BAIS_4992_Research_Report*. Throughout this paper I have included many screenshots of the code in an attempt to present concepts in a condensed way. The scope of Identity Server 4 and the OpenID Connect and OAuth 2.0 protocols in practice, could easily fill a large book.

Technologies

Table 1

Technologies Used in Example Applications

App Name	App Type	Opens In	URL
Identity Management	ASP.Net Core Identity/Kestral	Console	https://localhost:5005
AnAPI	ASP.NET 5 WebAPI/Kestral	Console	https://localhost:6001

STS	Identity Server 4 /Kestral	Console and Browser Window	https://localhost:5001
mvcClient	ASP.NET MVC/Kestral Identity Model	Console and Browser Window	https://localhost:5002
JSClient	JavaScript/Kestral Identity Model Oidc-client-js	Console and Browser Window	https://localhost:5003
ANativeClientOrService	Console Application	Console	N/A

Code Description

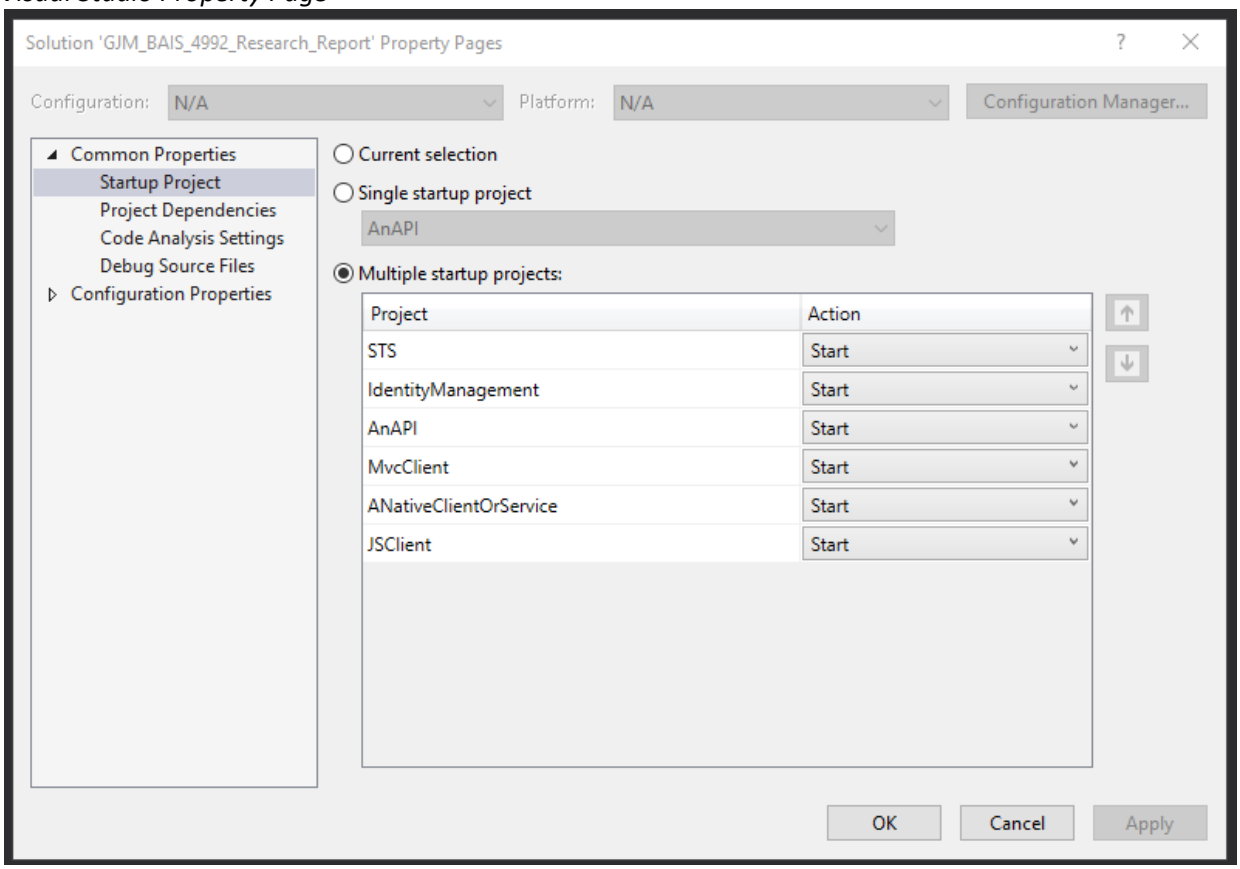
The working example code shows a basic console application requesting a token from the STS, afterwards calls an API, and the API returns the token contents to the console application. Included in the example code is a JavaScript client and ASP.NET MVC client set up to demonstrate the Single Sign-On functionality, as well as a basic ASP.NET app to demonstrate the Identity Management separation from the STS - a logged in user can manage their account. The JS client and the mvc client call an API, the API returns the token's content as json. Once logged into the mvc client, links will be available as a cursory demonstration.

Run the Code

Visual Studio will require the .NET 5 SDK to run the solution. The file was uploaded in the Zip Archive format and will need to be extracted before loading it into the Visual Studio, Integrated Development Environment. Once the GJM_BAIS_4992_Research_Report directory has been extracted from the zip archive, navigate into the folder and right-click on the GJM_BAIS_4992_Research_Report.sln file, then left-click and "Open With" Visual Studio 2019. Once Visual Studio opens the project, in the Solution Explorer window select "Solution 'GJM_BAIS_4992_Research_Report' (6 of 6 projects)," then right-click and left-click on "Set Startup Projects". A window will open showing the "Solution 'GJM_BAIS_4992_Research_Report' Property

Pages." Ensure that the "Multiple startup projects:" is selected and that the window is the same as the image below (project order matters).

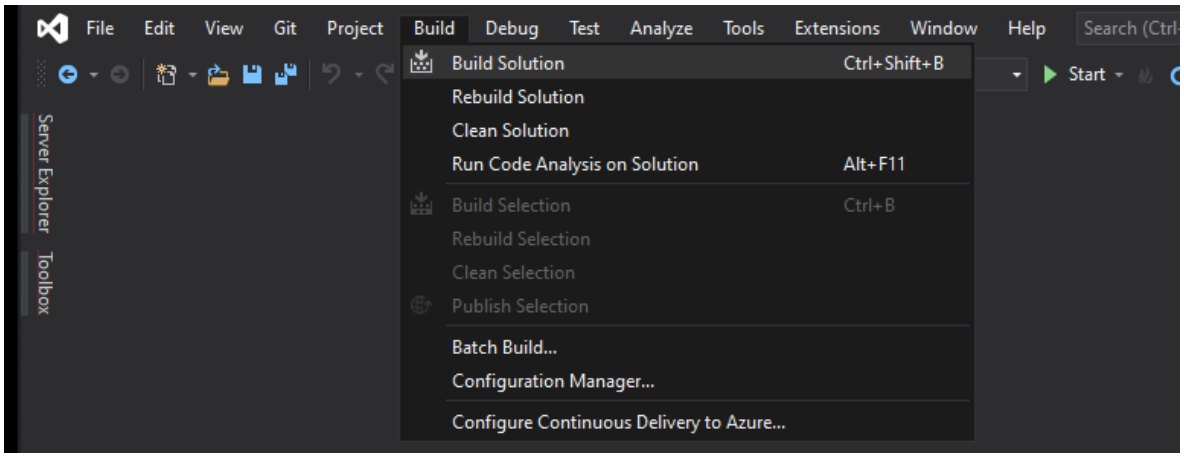
Figure 1
Visual Studio Property Page



Note: Screenshot taken of Visual Studio Window

Make any required adjustments so the projects are ordered as the image shows, and "Start" is selected for all the projects, then click "Apply" and "OK".

Figure 2
Visual Studio Navigation

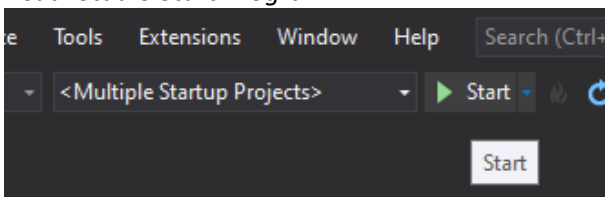


Note: Screenshot taken of Visual Studio Navigation Bar

From the top navigation bar of Visual Studio, select “Build”, then “Build Solution”. This should download all of the project’s dependencies and build the projects so they can be run. Once the build process has completed, press “Start”. Visual Studio 2019 created a tempkey.jwk as a development signing key and is part of the solution artifacts. Running these applications in development mode on a different computer might cause problems, and if asked, allow Visual Studio to create a tempkey.jwt for the machine that these applications are being run on (you might have to rename the existing to something like _tempkey.jwt). Visual Studio should create a signing certificate using the default certificates that come in Windows Certificate Store.

Figure 3

Visual Studio Start Program



Note: Screenshot taken in Visual Studio

The application will start opening six console windows and two web browser windows. When done, closing one of the browser windows should shut down all but one console window - it is fine to close it as well.

Resources

The real purpose of a secure token server (STS) is to control access to resources. It does so by creating tokens and passing them to client applications registered with the STS. A client application then passes the tokens to a secured service such as a RESTful API. A token is created for an authenticated client application and the client application uses the token to call a service on behalf of the user - or on behalf of itself.

There are two types of resource types that Identity Server 4 concerns itself with:

- Identity Resources represent claims about a logged in user such as an Unquie ID, name, date of birth and email address. Typically these resources can be made public, on other occasions a claim can be private and used internally in a secure way (such as healthcare number in a healthcare related application). There are standard claims defined by the OpenID Connect Protocol and custom claims can be created and retrieved from an IdP and used by the STS.
- API Resources are resources that a client application and/or a user want to access. Typically, a resource is a HTTP-based endpoint, such as a service or RESTful API, but could also be - for example - a message queuing endpoint. Identity Server 4 can be configured to secure pretty much any type of application service that is able to read the token create by, and handed out from the STS.

Identity Resources

Identity Server 4 is a STS, not an Identity Provider (IdP). When a user starts the log in process by (for example) clicking a “Login” link on a client application, the client application redirects the user to a webpage hosted on the STS. The user provides login credentials and the STS then reaches out to the IdP that the user is registered with. The IdP verifies the user and returns any claims and/or roles that the user has associated with their account - and that are specifically requested by the client application - to

the STS. The STS then uses those claims/roles to create an ID_Token and Access Token that is then passed back to the original client application. An IdP can be Google Accounts, Microsoft Accounts, Facebook or Twitter – many technology organizations have an identity management system and can act as an IdP for a STS. These types of IdP are known as an External Identity Provider. As it's the Identity Management System that fits hand and glove with Identity Server 4, Microsoft's ASP.NET Core Identity is used as the default IdP for this paper.

An identity resource is a named group of claims/roles that can be requested by a client application, or by making a request to the STS's UserInfo Endpoint and providing the required scope parameters. As mentioned earlier, the OpenID Connect specification has defined standard claims and they are mapped to claim types used by Identity Server 4, and can be used as needed by the application's requirements. With that being said, the openid scope (Scopes will be discussed later in this paper) is required by the OpenID Connect specification and must be included in the ID_Token returned from the STS. Below is an example defining an Identity Resources registered in an Identity Server 4 secure token server:

Figure 4
Identity Resources Configuration

```
public static IEnumerable<IdentityResource> IdentityResources =>
new List<IdentityResource>
{
    //Will be returned from UserInfo Endpoint
    //and/or included in the ID_Token
    new IdentityResources.OpenId(),
    new IdentityResources.Profile(),
    new IdentityResources.Email(),
    new IdentityResources.Address(),
    new IdentityResource {
        Name = "admin",
        DisplayName = "Identity Management Adminsitrator",
        UserClaims = new [] { "admin" },
        Description = "Identity Management Adminsitrator",
        ShowInDiscoveryDocument = true,
    },
    new IdentityResource
    {
        Name = "customClaim",
        DisplayName = "Your custom Identity Info added to tokens",
        UserClaims = new[] {"customClaim" },
        Description = "Your custom Identity Info added to tokens",
        ShowInDiscoveryDocument = true,
    }
};
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Config.cs

Once the identity resource is defined, a client application can request that resources be added to the access token returned by the STS (more on this when this paper addresses registering clients with the STS).

Scopes

Let's model something very simple, a system that has the logical operations of read and write. We define scopes with Identity Server 4 by using the ApiScope class as seen below in Figure 5.

Figure 5
API Scope Configuration

```
public static IEnumerable<ApiScope> ApiScopes =>
new List<ApiScope>
{
    //Flat scopes, no grouping based on logical API design
    //What the API is looking for when allowing access to an endpoint/resource
    new ApiScope(
        name: "AnAPI",
        displayName: "Administrator Access",
        userClaims: new [] { "admin", "name", "sub" }),
    //Read and rewite access to resource/endpoints
    new ApiScope(
        name: "invoiceManage",
        displayName: "Can read and write invoices",
        userClaims: new [] { "invoiceManage" }),
    new ApiScope(
        name: "identityManagementAdmin",
        displayName: "Can manage others identity",
        userClaims: new [] { "sub", "name", "email", "admin"}),
    //Read only access to resource/endpoints
    new ApiScope(
        name: "invoiceRead",
        displayName: "Can read invoices",
        userClaims: new [] { "invoiceRead" }),
    new ApiScope(
        name: "identityManagement",
        displayName : "Can manage your identity",
        userClaims : new[] { "sub, name, email" }),
};
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Config.cs

The scopes in the above image can then be assigned to registered client application in the Allowed Scopes property of the Client Class. Notice the ApiScopes named “invoiceManage” and “invoiceRead”. When these two apiscopes are requested by a client application, the STS will return an access token containing the user claims “invoiceManage” and “invoiceRead” (for both JWTs and Reference Tokens). This should be made clear: scopes are purely for authorizing clients; not users. If desired and warranted, more identity information about the user can be derived by including additional claims from the scope request. The previous scope definition tells the configuration system that when the “invoiceManage” scope gets granted, the claim should be added to the access token. This will pass the “invoiceManage” claim as a requested claim type to the profile service so that the consumer of the access token can use this data as for authorization decisions or business logic (render links to endpoint or not to).

API Resources

Defining and designing an API surface can be a complicated and daunting task, thankfully Identity Server 4 provides a couple of types to assist with that. "*The original OAuth 2.0 specification has the concept of scopes, which is just defined as the scope of access that the client requests. Technically speaking, the scope parameter is a list of space delimited values - you need to provide the structure and semantics of it."* (Baier & Brock, 2020)

In complex software systems, often there is more than one resource is introduced. This could be a service that calculates and returns the results to a user, a service to store persistent data, a messaging queue service or even a service to stream video. Each secured service/API will require scopes to access the resource, and some scopes might be exclusive to a specific resource. A client application might be granted permission to "upload" a video to a streaming site, and in order to do so, would need the "upload" scope included in the access token passed to the streaming service from the calling client application. Scopes could be shared across shared resources or granted only for specific resource use.

We started with simple scopes first. Now we'll have a look at how the APIResource class help structure scopes. When the API surface grows, a list of scopes might not be feasible and a better design would be to use name spacing with modifiers to organize scopes and assist with the logic layout of APIs. For more complicated APIs, Identity Server4 offers the APIResource class over APIScope class. The APIResource class allows both Identity Resources and access scopes to be grouped in a logical API design. Figure 6 below shows how to use the APIResource class to define two logical APIs. The client application that receives an access token with the scopes "identityManagementAdmin" and "identityManagement" will pass these token to the API in the request header. The API can then give access to the administration endpoint and the manage account endpoint to the user who has the appropriate claim type associated with their user account.

Figure 6

API Resource Configuration

```
public static IEnumerable<ApiResource> ApiResources =>
new List<ApiResource>
{
    new ApiResource
    {
        Name = "identityManagementAdmin",
        DisplayName = "Identity Management",
        //the user claim included in access token
        UserClaims = new [] { "sub", "name", "email", "admin" },
        //the endpoints user/client can access
        Scopes = new [] { "identityManagementAdmin" },
    },

    new ApiResource()
    {
        Name = "identityManagement",
        DisplayName = "Identity Management",
        Description = "Administrator Access",
        UserClaims = { "sub, name, email" },
        Scopes = { "identityManagement" }
    },
};
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Config.cs

Using API Resource grouping gives you the following additional features:

- Support for the JWT audience claim. The value(s) of the audience claim will be the name of the API resource(s)
- Support for adding common user claims across all contained scopes
- Support for introspection (Reference Token) by assigning an API secret to the resource
- Support for configuring the access token signing algorithm for the resource

Clients

Clients represent applications that can request tokens from the STS. The details vary from client application to client application, but typically the following common settings are defined:

- A unique client ID and application name
- Server Side clients should have a registered secret,
- The allowed interactions with the token service (OAuth Grant Type)
- A network location where identity and/or access token gets sent to (login/logout redirect URI)
- A list of scopes (resources) the client is allowed to access

Client Basics

Some properties (eg: Front and Back Channel) shown in Figure 7 are touched on in later parts of this paper, so for brevity, I rely on a picture telling 1000 words .

Figure 7

Client Configuration in the STS Part 1

```
new Client
{
    ClientId = "mvcClient",
    ClientName = "MvcClient",
    ClientSecrets = { new Secret("aDifferentSecret".Sha256()) },

    AllowedGrantTypes = GrantTypes.Code,

    RequirePkce = true,
    AllowPlainTextPkce = false,
    RequireConsent = false,

    RedirectUris = { "https://localhost:5002/signin-oidc" },
    FrontChannelLogoutSessionRequired = true,
    FrontChannelLogoutUri = "https://localhost:5002/signout-oidc",
    BackChannelLogoutSessionRequired = true,
    BackChannelLogoutUri = "https://localhost:5002/signout-oidc",
    PostLogoutRedirectUris = { "https://localhost:5002/signout-callback-oidc" },
```

Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Config.cs

Figure 8 shows how to configure the scopes that a client can request from the STS. When configuring a client with the STS, each client can have their own allowed scopes, or clients can use the same scopes and api resources as required.

Figure 8

Client Configuration in the STS Part 2 - Defining Scopes the Client Can Request

```
    AllowedScopes = {
        //Identity Resouces to include in access token the client app can request
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        IdentityServerConstants.StandardScopes.Address,
        "customClaim",  "admin",
        //APIResources Client can request
        "identityManagementAdmin", "identityManagement",
        //simple APIScopes
        "AnAPI", "offline_access", "invoiceManage", "invoiceRead"
    },
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Config.cs

Authentication

We must also register the use of an authentication and authorization service in the middleware pipeline – this will be a reoccurring task for all clients, APIs and even the STS application.

Figure 9

Client Configuration Part 3 – Client Application Middleware

```
    //Add Autnentication to request/response pipeline (middleware)
    app.UseAuthentication();
    //Add Authentication to request/response pipeline (middleware)
    app.UseAuthorization();
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > MvcClient> Startup.cs

Figure 10 demonstrates configuring a client application to use the OpenID Connect protocol and that the access token is held in a cookie.

Figure 10

Client Configuration Part 4 – Adding Services to the Client's Inversion of Control Container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    //Clear Microsoft default Claim mapping
    JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();
    //Adds user and client access token management with IdentityModel library
    services.AddAccessTokenManagement();

    services.AddAuthentication(options =>
    {
        //Token are in cookies
        options.DefaultScheme = "Cookies";
        //Use OpenID Connect protocol
        options.DefaultChallengeScheme = "oidc";
    })
    //Create this application's cookie to hold token
    .AddCookie("Cookies", options =>
    {
        options.Cookie.Name = "mvcClient";
        options.Events.OnSigningOut = async e =>
        {
            await e.HttpContext.RevokeUserRefreshTokenAsync();
        };

    })
```

Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > MvcClient> Startup.cs

Figure 11

Client Configuration Part 5 – Adding Services to the Client's Inversion of Control Container

```
//Authorization Handler, use OpenID Connect handler
.AddOpenIdConnect("oidc", options =>
{
    options.Authority = "https://localhost:5001";

    options.ClientId = "mvcClient";
    options.ClientSecret = "aDifferentSecret";

    //Indicates that the authentication session lifetime (e.g. cookies)
    //should match that of the authentication token.
    //If the token does not provide lifetime information then normal
    //session lifetimes will be used. This is disabled by default.
    options.UseTokenLifetime = true;

    //Don't include all User claims in id token,
    //a client needs to make a seperate request to
    //UserInfo Endpoint to get all a User's claims
    //True will put all Identity Resources registered for this client in the tokens
    //False only puts some Identity Resources in tokens
    //Get the rest from endpoint - see ProfileService covered in report
    options.GetClaimsFromUserInfoEndpoint = false;
    options.SaveTokens = true;
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > MvcClient> Startup.cs

Figure 11 is the authorization handler registered with the client application and is required for communication with the STS. Not only do we have to configure the allowed scopes in the client configuration of the STS, we also need to ask for those scopes from the STS. Figure 12 shows how we add the scopes that are to be requested by the client from the STS.

Figure 12

Client Configuration Part 6 – Adding Services to the Client's Inversion of Control Container

```
    //OAuth 2.0 / OpenID Connect GrantType registered with STS Config.cs
    options.ResponseType = "code";
    //Clean slate
    options.Scope.Clear();
    //Scopes the client is requesting in the access token must match STS Config.cs
    options.Scope.Add("AnAPI");
    options.Scope.Add("offline_access");
    options.Scope.Add("openid");
    options.Scope.Add("invoiceRead");
    options.Scope.Add("invoiceManage");
    options.Scope.Add("identityManagementAdmin");
    options.Scope.Add("identityManagement");
});
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > MvcClient> Startup.cs

Tokens

Figures 12, 13 and 14 explain away the confusion of setting cookie and token lifetimes. I am counting on a picture speaking 1000 words because without the images, I would need at least 1000 words. Further on in this paper, when STS configuration is touched on, there is more cookie and token lifetime configuration.

Figure 13

Token Life Time Part 1 – Secure Token Service Client Configuration

```
//The Token settings
AlwaysIncludeUserClaimsInIdToken = true,
AlwaysSendClientClaims = true,
RequireConsent = true,
RequirePkce = true,
AllowPlainTextPkce = false,

AllowAccessTokensViaBrowser = true,

//The refresh token should be long lived (at least longer than the access token).
//Once the refresh token expires, the user has to login again.
//Without sliding expiration the refresh token will expire in an absolute time,
//having the user to login again.


//A self contained token - verification with STS not required on every request
AccessTokenType = AccessTokenType.Jwt,
AllowOfflineAccess = true, //Allows Refresh Token

//Token lifetime - NOT COOKIE LIFETIME, NOT AUTHENTICATION LIFETIME.
//Just how long an access token can be used against an API (Resource registered with IS4)
//Default 300 seconds
IdentityTokenLifetime = 300,
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Config.cs

Figure 14

Token Life Time Part 2 – Secure Token Service Client Configuration

```
//Default 3600 seconds, 1 hour
AccessTokenLifetime = 300,

//Default 300 seconds: Once User consents,
//this token should no longer be needed until re-authorization.
//This AuthorizationCode is used to prove to IS4 that an access token and
//id token have been constented too
//and from there the refresh token takes over.
//So if using refresh tokens, AuthorizationCode shouldn't need a long lifetime.
AuthorizationCodeLifetime = 300,

//Defaults to 2592000 seconds / 30 days
//NOT GOOD FOR SPA's - 36000 = 10 hours
AbsoluteRefreshTokenLifetime = 36000,

//Each time a refresh token is requested, give a new, dispose the old
RefreshTokenUsage = TokenUsage.OneTimeOnly,
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Config.cs

Figure 15

Token Life Time Part 3 – Secure Token Service Client Configuration

```
//token will be refreshed only if this value has 50% elasped.
//If 50% elapsed, refresh will happen.
//Setting the accessTokenExpiringNotificationTime of the oidc-client/IdentityModel
//client to the same inactive timeout,
//will allow refresh on page navigation (assuming access and id tokens haven't already expired)
SlidingRefreshTokenLifetime = 150,
RefreshTokenExpiration = TokenExpiration.Sliding,

//Gets or sets a value indicating whether the access token (and its claims)
//should be updated on a refresh token request.
UpdateAccessTokenClaimsOnRefresh = true,

//The maximum duration (in seconds) since the last time the user authenticated.
//Defaults to null.
//You can adjust the lifetime of a session token to control when and how often a user is required
//to reenter credentials
//instead of being silently authenticated, when using a web application.
UserSsoLifetime = 300,
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Config.cs

APIs

Protecting an ASP.NET Core-based API is only a matter of adding the JWT bearer authentication handler.

API Basics

As with the client applications, we also need to add the CORS policy, authorization and authentication services to our API.

Figure 16

Adding Middleware to the API Request Pipeline

```
    //Register needed middleware
    app.UseCors("default");
    app.UseAuthentication();
    app.UseAuthorization();
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > AnAPI > Startup.cs

Figure 17

Adding Service to the API Inversion of Control Container Part 1

```
services.AddCors(options =>
{
    //this defines a CORS policy called "default"
    options.AddPolicy("default", policy =>
    {
        policy.WithOrigins("https://localhost:5003")
        //Set up a proper Content Security policy and DON'T allow any header or method
        //See STS project for example, too much for report's length contraints
            .AllowAnyHeader()
            .AllowAnyMethod();
    });
});
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > AnAPI > Startup.cs

Typically an API will have many authorization policies each stating required claims and/or roles that should be included in an access token. It is common for each “controller” to have their own authorization policy and the claims and roles required to call the controller’s action methods (.NET jargon).

Figure 18

Adding Service to the API Inversion of Control Container Part 2

```
//Only allow ApiScopes and APIResources configured with STS
//Block any request that don't have these scopes in the access token passed from client
services.AddAuthorization(options =>
{
    options.AddPolicy("AnAPIScope", policy =>
    {
        policy.RequireAuthenticatedUser();
        policy.RequireClaim("AnAPI");
        //Can also lock down API endpoints by requiring specific cliams in access token
        //policy.RequireClaim("requiredClaim1", "requiredClaim2");
    });
});
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > AnAPI > Startup.cs

Authentication

Authorization handlers can be a deep topic and any developer that is configuring application security should really understand them in depth. Figure 19 shows the simplest use of an Authorization handler setup for an API.

Figure 19

Adding Service to the API Inversion of Control Container Part 3

```
//Look for token in request's Authorization Header as a Bearer Token
services.AddAuthentication("Bearer").AddJwtBearer("Bearer", options =>
{
    //The STS who made the tokens and to verify token against
    options.Authority = "https://localhost:5001";
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateAudience = false
    };
});
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > AnAPI > Startup.cs

Tokens

If the choice is made to use reference tokens in the STS, an authentication handler that implements OAuth 2.0 token introspection endpoint needs to be used. The example Authorization Handler below supports both reference tokens and Jwt tokens. ASP.NET WebAPIs can be configured to dispatch tokens to the right handler based on the incoming token. Though not fully implemented in the example code, this paper would be remiss if it didn't touch on reference tokens.

Figure 20

Example of Authorization Handler in API Inversion of Control Container Part 4

```
//Further details out of project's lenght contraint
//Example of an Authorization Handler for Reference Tokens,
//causes multiple trips to and from STS introspection endpoint
//But more secure that self-contained access tokens
//.AddOAuth2Introspection("introspection", options =>
//{
//    options.Authority = STS URL;
//    //Allow a specifc client to call this API
//    options.ClientId = name of client registered with STS;
//    options.ClientSecret = that client's secret;
//})
//.AddIdentityServerAuthentication("IdentityServerAccessToken", options =>
//{
//    options.Authority = STS URL;
//    options.ApiName = Name of this API, must match name in APIResource registered STS Config.cs;
//    options.ApiSecret = Secret of this API, must match secret in APIResource registered STS Config.cs;

//    //Using Reference Tokens - lessons calls to IS4 to validate tokens
//    options.EnableCaching = true; //REQUIRES: caching strategy - in memory, redis ect...;
//    options.CacheDuration = TimeSpan.FromMinutes(2);
//    options.SupportedTokens = SupportedTokens.Both; This handler will support Jwts and Reference Tokens
//});
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > AnAPI > Startup.cs

Secure Token Server

I have touched on client and api configuration, now I should say a bit about configuring the STS. Again, I have included screenshots of the actual code, as a picture can convey 1000 words and this paper is running long.

Configuration

As with the client and API configuration, we also have to configure the Identity Server 4 middleware to make a Secure Token Server. We call it middleware because it sits between the application entry point and all requests pass though it before hitting the endpoints, and again as the response leaves the application back out onto the network and to the calling client application. Technically speaking, Identity Server 4 is middleware; not an application in and of itself. By only including Identity Server 4 middleware and endpoints, this application becomes the STS. Once other non-Identity Server 4 service endpoints are added, the application becomes more than an STS. It is

advised against making a STS do more than just create and manage tokens – it should be the only responsibility of an STS and therefore an application unto itself.

Figure 21

Registering Identity Server 4 Middleware

```
app.UseCookiePolicy();
app.UseIdentityServer();
app.UseAuthorization();
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Startup.cs

Add the obligatory authentication and authorization middleware, after adding the CORS policy (order matters).

Figure 22

Adding CORS to STS Dependency Inject Container

```
services.AddCors(options =>
{
    options.AddPolicy("STSCORS", policy =>
    {
        policy.WithOrigins(
            "https://localhost:5005",
            "https://localhost:6001",
            "https://localhost:5003",
            "https://localhost:5002");
    });
});
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Startup.cs

The default IdP (Identity Provider) we used in the example code is ASP.NET Identity Core – though external IdPs can be configured – and as such, we want to let Identity Server 4 know where we can verify the user credentials at login, and where the user claims and roles can be fetched from. Figure 23 shows how to configure Identity Core with the STS.

Figure 23

Adding a Default IdP STS Dependency Inject Container

```
//Configuring a default IdP
services.AddIdentity<ApplicationUser, IdentityRole>( options => {
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;

    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;

    options.SignIn.RequireConfirmedEmail = true;
    options.SignIn.RequireConfirmedPhoneNumber = false;

    options.User.AllowedUserNameCharacters =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._+";
    options.User.RequireUniqueEmail = true;
})
.AddEntityFrameworkStores<ApplicationDbContext>()
.AddDefaultTokenProviders();
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Startup.cs

We must not forget to actually add Identity Server 4 configuration. Figure 24 and 25, illustrate many of the configuration options and have code comments for a brief explanation.

Figure 24

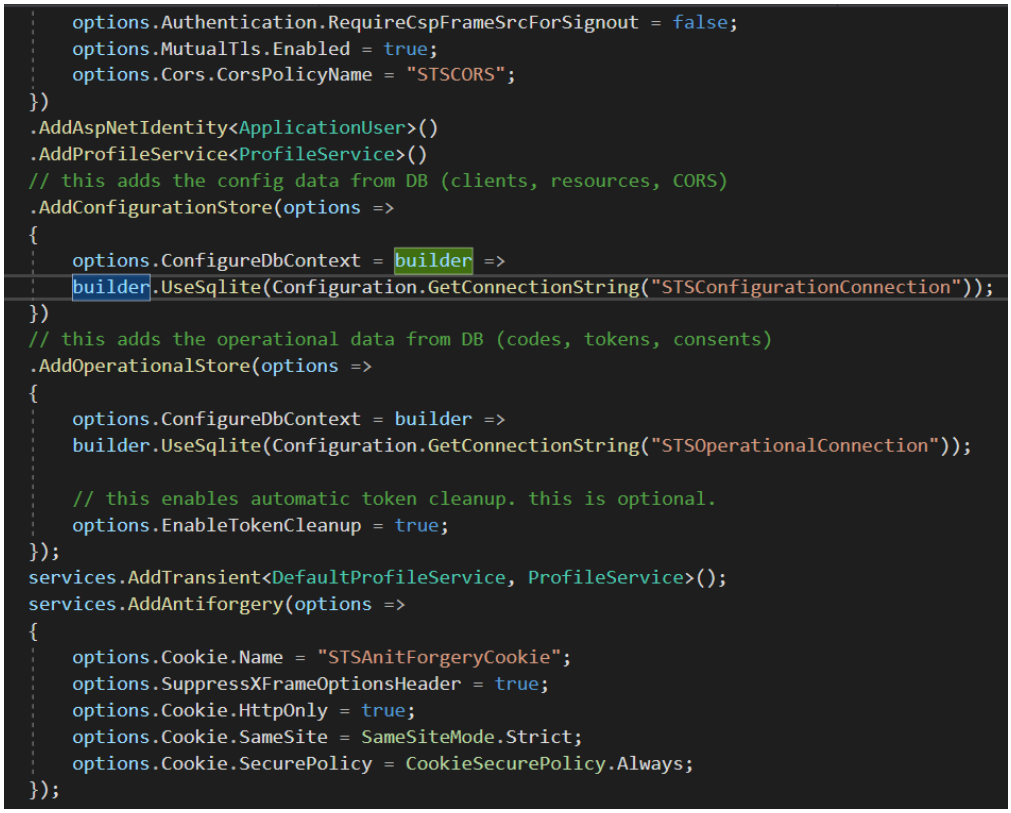
Adding Identity Server 4 Configuration to Dependency Injection Contain

```
//Adding IdentityServer4 Middleware to IOC and make this app a STS
var builder = services.AddIdentityServer(options =>
{
    options.Events.RaiseErrorEvents = true;
    options.Events.RaiseInformationEvents = true;
    options.Events.RaiseFailureEvents = true;
    options.Events.RaiseSuccessEvents = true;

    // see https://identityserver4.readthedocs.io/en/latest/topics/resources.html
    options.EmitStaticAudienceClaim = true;
    //Endpoints this app provides for Login and Logout
    options.UserInteraction = new UserInteractionOptions
    {
        LogoutUrl = "/Account/Logout",
        LoginUrl = "/Account/Login",
        LoginReturnUrlParameter = "returnUrl"
    };

    options.Csp.Level = IdentityServer4.Models.CspLevel.Two;

    //Session Cookie Configuration - NOT the Token cookies
    options.Authentication.CheckSessionCookieName = "STSSessionCookie";
    options.Authentication.CookieLifetime = TimeSpan.FromSeconds(3600);
    options.Authentication.CookieSlidingExpiration = true;

    options.Authentication.CookieSameSiteMode = SameSiteMode.Strict;
    options.Authentication.CheckSessionCookieSameSiteMode = SameSiteMode.Strict;
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Startup.cs

Figure 25

More Configuration for the STS's Dependency Inject Container



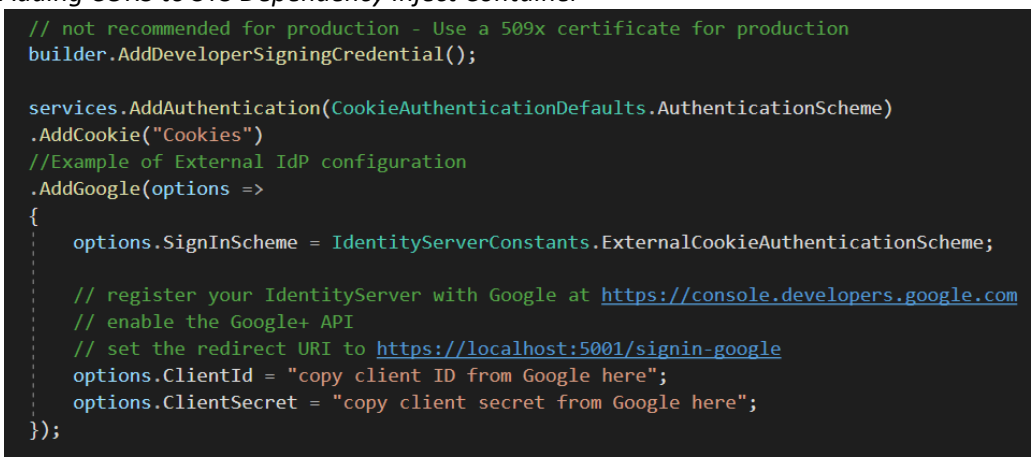
```
    options.Authentication.RequireCspFrameSrcForSignout = false;
    options.MutualTls.Enabled = true;
    options.Cors.CorsPolicyName = "STSCORS";
})
.AddAspNetIdentity<ApplicationUser>()
.AddProfileService<ProfileService>()
// this adds the config data from DB (clients, resources, CORS)
.AddConfigurationStore(options =>
{
    options.ConfigureDbContext = builder =>
    builder.UseSqlite(Configuration.GetConnectionString("STSConfigurationConnection"));
})
// this adds the operational data from DB (codes, tokens, consents)
.AddOperationalStore(options =>
{
    options.ConfigureDbContext = builder =>
    builder.UseSqlite(Configuration.GetConnectionString("STSOperationalConnection"));

    // this enables automatic token cleanup. this is optional.
    options.EnableTokenCleanup = true;
});
services.AddTransient<DefaultProfileService, ProfileService>();
services.AddAntiforgery(options =>
{
    options.Cookie.Name = "STSAnitForgeryCookie";
    options.SuppressXFrameOptionsHeader = true;
    options.Cookie.HttpOnly = true;
    options.Cookie.SameSite = SameSiteMode.Strict;
    options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
});
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Startup.cs

Figure 26 shows an example of how to register an external IdP and letting the STS know that it should be creating cookies. Also, it touches on token signing and signing credentials – in and of itself a large topic to write about.

Figure 26

Adding CORS to STS Dependency Inject Container



```
// not recommended for production - Use a 509x certificate for production
builder.AddDeveloperSigningCredential();

services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
.AddCookie("Cookies")
//Example of External IdP configuration
.AddGoogle(options =>
{
    options.SignInScheme = IdentityServerConstants.ExternalCookieAuthenticationScheme;

    // register your IdentityServer with Google at https://console.developers.google.com
    // enable the Google+ API
    // set the redirect URI to https://localhost:5001/signin-google
    options.ClientId = "copy client ID from Google here";
    options.ClientSecret = "copy client secret from Google here";
});
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Startup.cs

Figure 27

Adding CORS to STS Dependency Inject Container

```
services.ConfigureApplicationCookie(options => {
    options.Cookie.Name = "STSCookie";

    options.Cookie.HttpOnly = true;

    options.Cookie.Path = "/";
    options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
    //options.Cookie.HttpOnly = HttpOnlyPolicy.Always;
    options.Cookie.IsEssential = true;
    options.Cookie.SameSite = SameSiteMode.Strict;
    //Configures the ticket lifetime inside the cookie; not the cookie lifetime
    //This is separate from the value of , which specifies how long the browser will keep the cookie,
    //which should be controlled and set in STS Options
    options.ExpireTimeSpan = TimeSpan.FromSeconds(3600);
    //This is for session lifetimes....not token lifetime
    options.SlidingExpiration = true;
});
```

Note: Screenshot taken from directory: GJM_BAIS_4992_Research_Report > src > STS > Startup.cs

Figure 27 is the other part of cookie and token lifetime configuration. This was one of the bigger confusion points for me when I first started with Identity Server 4. There are a few areas where one has to configure tokens and the cookies they are held in.

End Points

For server side rendered application in the .NET environment, the IdentityModel, client library can be used to programmatically create requests to the STS's endpoints for .NET code. More information is available on the IdentityModel docs. For Javascript clients, the IdentityModel – Oidc-Client-js, client library can be used to programmatically create requests to the STS's endpoints for JavaScript client applications. More information is available on the IdentityModel wiki.

Discovery Endpoint

The discovery endpoint can be used to retrieve metadata about the STS and it returns information like the issuer name, key material, supported scopes etc. For any OpenID Connect complaint STS, the Discovery Document endpoint is available by navigating to https://domain.TLD/.well-

known/openid-configuration. For the sample code in this paper, https://localhost:5001/.well-known/openid-configuration

Authorization Endpoint

The authorize endpoint can be used to request tokens or authorization codes via the browser. This process typically happens during user authentication, but requests can be made to this endpoint anytime a token is needed.

Token Endpoint

The token endpoint can be used to programmatically request tokens. It supports the password authentication, the Authorization Code, Client Credentials, Refresh Token and Device Code grant types. When a refresh token is fetched silently (without explicit user request) using front or back channels, this is where it comes from.

UserInfo Endpoint

The UserInfo endpoint can be used to fetch identity information about a user. Claims that are included in the access token and id token will be returned, as will any user claims registered with the STS but not included in the tokens. The calling application needs to send a valid access token in the request's Authorization Header (Authorization: Bearer <access_token>) representing the logged in user. Depending on the granted scopes requested by the client application, the UserInfo endpoint will return the mapped claims.

Introspection Endpoint

The introspection endpoint is a way for an API registered with the STS to validate requests made to it from client applications. It's used to validate reference tokens (or JWTs if the consuming API does not have support for appropriate JWT or cryptographic libraries). Due to length constraints of this paper,

Reference Tokens were not discussed much but deserve a brief introduction. Reference Tokens cause twice the traffic to and from the STS, but are considerably more secure than self-contained JWT's. To use the introspection endpoint, an API secret must be passed to the STS on every validation request, and can be configured with the APIResouce object of the STS's configuration. If the design decision to use Reference Tokens is made, one should also learn about the Revocation Endpoint of the STS. It allows for Reference Tokens and Refresh Tokens to be revoked immediately and before they expire.

Note Worthy

This paper is becoming larger than the requested length but there are a few other Identity Server 4 features I should briefly mention.

Back-Channel and Front-Channel

Interactive applications (web applications or native desktop/mobile) use the authorization code flow. This flow gives the best security because the access tokens are transmitted via back-channel calls only (and gives you access to refresh tokens). Front-Channel is used for JavaScript client application but is less secure. One should take all precautions such as setting a Content Security Policy, making sure all cookies are Same-Site and HttpOnly.

Profile Service

Identity Server 4 middleware has a default Profile Service and it's that service that put scopes and claims into the tokens. When building the STS, the Profile Service can be overwritten and the access and id tokens can have certain scopes and claims either included, or excluded, from cookies. The Profile Service can be configured so the UserInfo Endpoint can return any Identity Resources that are excluded from the id token. See the STS project code for an example implementation.

Multi-Tenant

For this STS to serve as a Multi-Tenant STS, all that is needed is to add the appropriate UI and endpoints to the STS and the IdP application. Add a role claim of “Organization” and assign it to an individual account. Add a role of “Organization_Member” and assign it to an individual user account with the claim of “Administrator”. The user can then login and configure the claims, roles and STS configuration options for their organization’s client applications and APIs.

Conclusion

There are Software-as-a-Service (SaaS) providers whose whole business model revolves around managing and validating user identity and authorization tokens. Some of these SaaS providers offer an Identity Provider Service (IdP) using the OpenID Connect Protocol, while others use the older Security Assertion Markup Language (SAML) Protocol to provide the user authentication. Whether these SaaS organizations offer an IdP service as part of the subscription, or allow their customers to use an external IdP - such as Google Accounts, Microsoft Accounts or Facebook - token creation and validation is no longer obscured behind a curtain. You dear reader, are now better equipped to use services from organizations such as Okta or Auth0, or you can build your own Secure Token Service and Identity Management system.

Bibliography

Baier & Brock, A. &. (2020). *Resources.html.* Retrieved Novemeber 16, 2021, from http://docs.identityserver.io/en/latest: http://docs.identityserver.io/en/latest/topics/resources.html#apis

References

https://damienbod.com/tag/identityserver4/

https://identityserver4.readthedocs.io/en/latest/

https://leastprivilege.com/

https://www.scottbrady91.com/identity-server

Report Validation or Verification

Unfortunately, none of my friends have graduated high school let alone know anything about software security. Over and above those in my personal life, my supervisor at AltaML has told me that he doesn't understand the topic either. In fact, AltaML is looking for a SaaS authentication & authorization provider. I have dropped a few hints mentioning that I can help out there, but those hints seem to be ignored. As a result, I have no way to have this report validated other than to say, RUN THE CODE. You'll see that it works.

Personal Research Comments

I know Authentication and Authorization is an extensive topic, but I had no clue as to how long this report would grow to (too?). I attempted to trim it down a few times, but every time I tried to delete a section, it left a huge hole in the report, how Identity Server 4 is configured and works. I was able to cut out a lot by including screenshots of the code; rather than a typed explanation of everything the screenshots capture.