ASP.NET Core Identity Series – OAuth 2.0, OpenID Connect & IdentityServer

BY CHRISTOS S. *on* MARCH 11, 2019 • (23)

As the web evolved over the years it proved that the traditional security options and mechanics such as client-server authentication, had several limitations and couldn't cover *(at least properly)* the cases introduced by the evolution. Take for example the case where a third-party application requires access to your profile data in a different web application such as Facebook. Years ago this would require to provide your Facebook credentials to the third-party so it can access your account information. This of course, raised several problems such as:

- Third-party applications must be able to store the user's credentials
- Servers that host the protected resources must support password authentication
- Third-party applications gain access probably to all of the owner's protected resources
- In case the owner decides to revoke access to the third-party application, password change is required something that will cause the revocation to all other third-party apps
- The owner's credentials are way too much vulnerable and any compromise of a third-party application would result in compromise of all the user's data

OAuth 2.0 & OpenID Connect to the rescue

Fortunately **OAuth** protocol introduced and along with **OpenID Connect** provided a wide range of options for properly securing applications in the cloud. In the world of .NET applications this was quickly connected with an open source framework named IdentityServer which allows you to integrate all the protocol implementations in your apps. IdentityServer made **Token-based authentication**, **Single-Sign-On**, centralized and **restricted API access** a matter of a few lines of code. What this post is all about is to learn the basic concepts of `OAuth 2.0` & `OpenID Connect` so that when using IdentityServer in your .NET Core applications you are totally aware of what's happening behind the scenes. The post is a continuation of the **ASP.NET Core Identity Series** where the main goal is to understand ASP.NET Core Identity in depth. More specifically here's what's we gonna cover:

- Explain what **OAuth 2.0** is and what problems it solves
- Learn about OAuth 2.0 basic concepts such as *Roles*, *Tokens* and *Grants*
- Introduce **OpenID Connect** and explain its relation with OAuth 2.0
- Learn about OpenID Connect **Flows**
- Understand how to choose the correct authorization/authentication flow for securing your apps
- Learn how to integrate IdentityServer to your ASP.NET Core application

It won't be a walk in the park though so make sure to bring all your focus from now on.

The source code for the series is available here. Each part has a related branch on the repository. To follow along with this part clone the repository and checkout the ***identity-server*** *branch as follow:*

```
git clone https://github.com/chsakell/aspnet-core-identity.git
cd .\aspnet-core-identity
git fetch
git checkout identity-server
```

Keep in mind that master branch has been updated with .NET Core 3 & Angular 8!

This post is a continuation of the **ASP.NET Core Identity Series**:

- Part 1: Getting Started
- Part 2: Integrate Entity Framework
- Part 3: Deep Dive in authorization
- Part 4: OAuth 2.0, OpenID Connect & IdentityServer
- Part 5: External Provider authentication & registration strategy
- Part 6: Two-Factor Authentication

It is recommended *(but not required)* that you read the first 3 posts of the series before continue. This will help you understand better the project we have built so far.

The theory for OAuth 2.0 and OpenID Connect is also available in the following presentation.

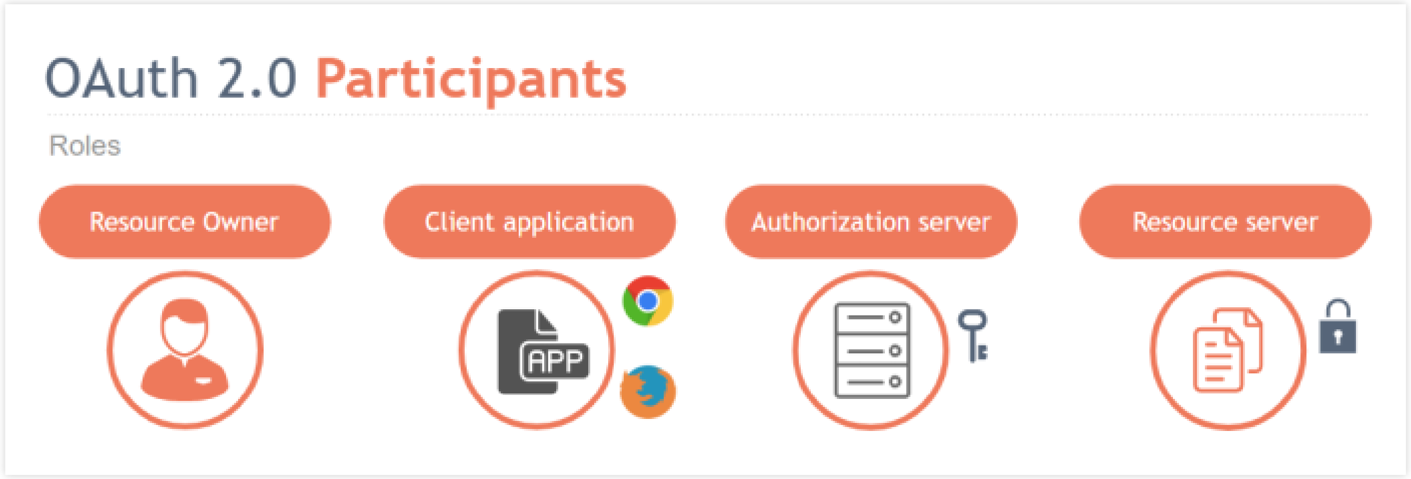
OAuth 2.0 Framework

OAuth 2.0 is an open standard authorization framework that can securely issue access tokens so that third-party applications gain **limited** access to protected resources. This access may be on behalf of the **resource owner** in which case the resource owner's approval is required or on its own behalf. You have probably used OAuth many times but haven't realized it yet. Have you ever been asked by a website to login with your Facebook or Gmail account in order to proceed? Well.. that's pretty much OAuth where you are being redirected to the *authorization server's* authorization endpoint and you give your **consent** that you allow the third-party application to access specific **scopes** of your main account *(e.g., profile info in Facebook, Gmail or read repositories in GitHub)*. We mentioned some strange words such as *resource owner* or *authorization server* but we haven't defined what exactly they represent yet so let's do it now.

OAuth 2.0 Participants

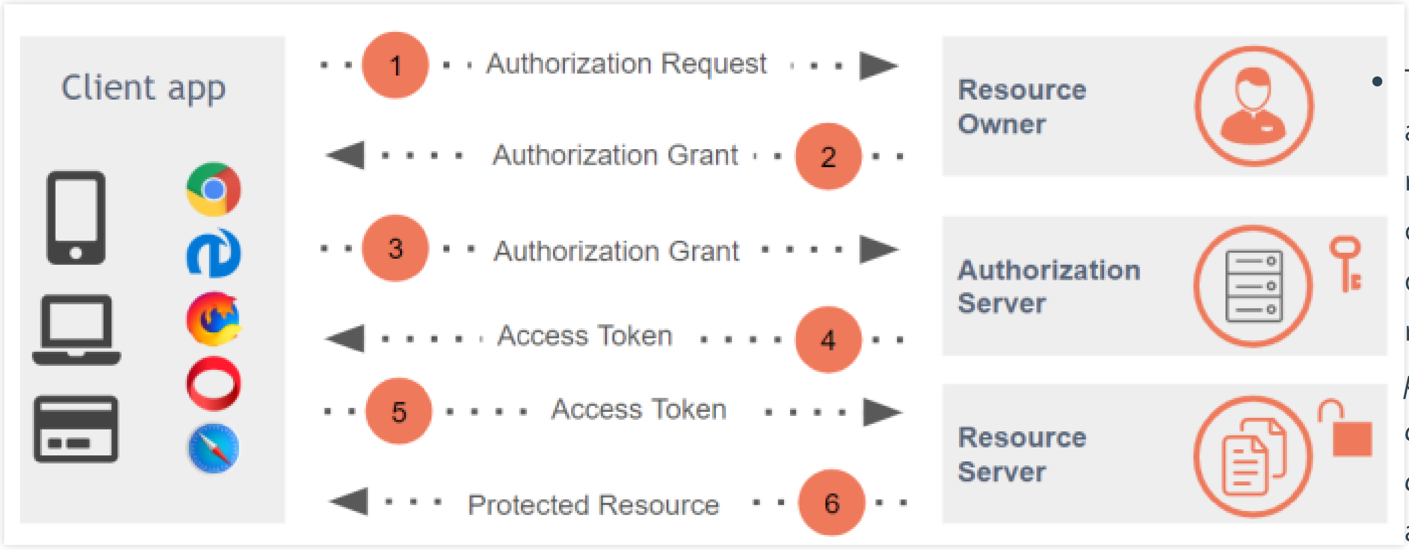
The following are the participants or the so-called **Roles** that evolved and interact with each other in OAuth 2.0.

- **Resource Owner**: It's the entity that owns the data, capable of granting access to its protected resources. When this entity is a person then is referred as the `End-User`
- **Authorization Server**: The server that issues access tokens to the client. It is also the entity that authenticates the resource owner and obtains authorization
- **Client**: The application that wants to access the resource owner's data. The client obtains an access token before start sending protected resource requests
- **Resource Server**: The server that hosts the protected resources. The server is able to accept and respond to protected resource requests that contain access tokens



OAuth 2.0 Abstraction Flow

The abstract flow illustrated in the following image describes the basic interaction between the roles in OAuth 2.0.



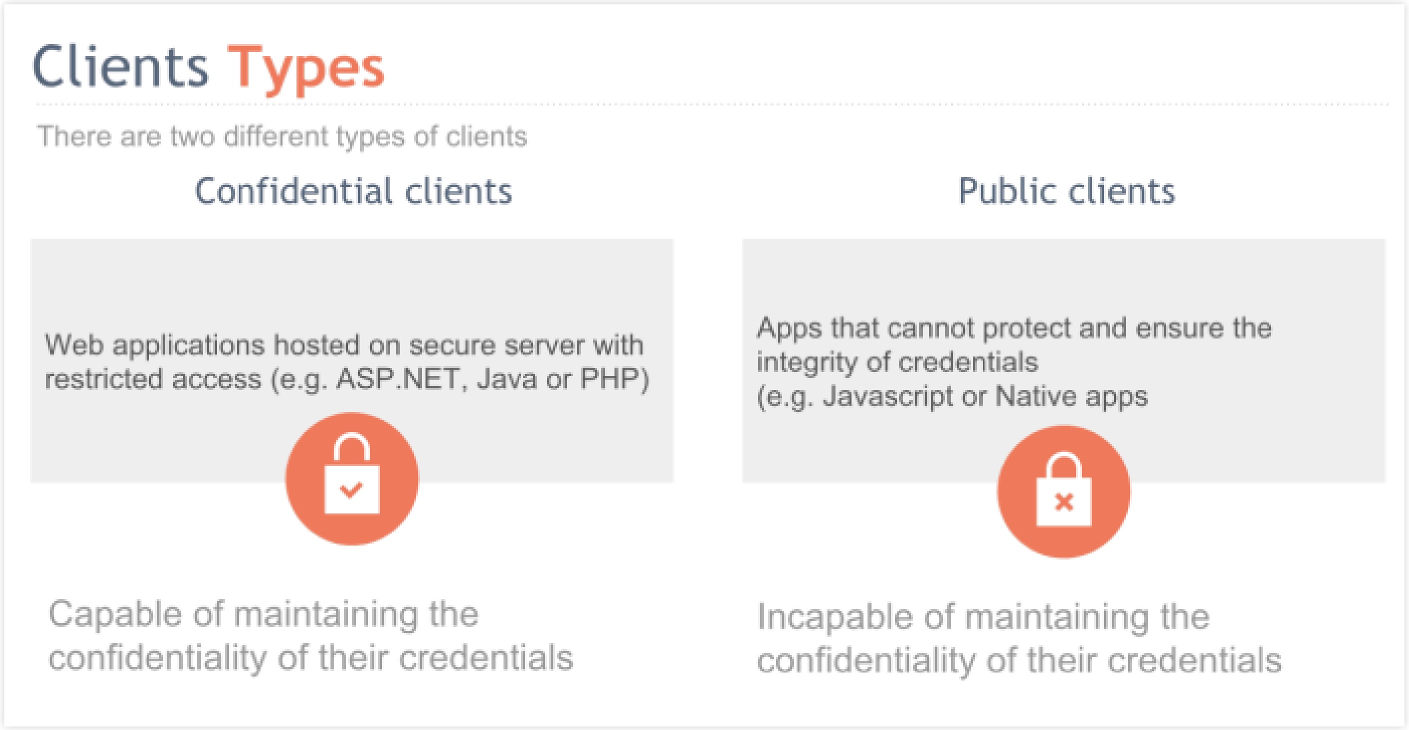
- The client requests authorization from the resource owner. This can be made either directly with the resource owner *(user provides directly the credentials to the client)* or via the authorization server using a redirection URL
- The client receives an **authorization grant** representing the resource owner's authorization. OAuth 2.0 provides 4 different types of grants but can also be extended. The grand type depends on the method used by the client to request

authorization and the types supported by the authorization server

- The client uses the authorization grant received and requests an access token by the authorization server's token endpoint
- Authorization server authenticates the client, validates the authorization grant and if valid issues an **access token**
- The client uses the access token and makes a protected resource request
- The resource server validates the access token and if valid serves the request

Before explain the 4 different grants in OAuth 2.0 let's see the types of clients in OAuth:

- **Confidential clients**: Clients that are capable to protect their credentials – **client_key** & **client_secret**. Web applications *(ASP.NET, PHP, Java)* hosted on secure servers are examples of this type of clients
- **Public clients**: Clients that are incapable of maintaining the confidentiality of their credentials. Examples of this type of clients are mobile devices or browser-based web applications *(angular, vue.js, etc..)*


Clients Types
There are two different types of clients
Confidential clients
Public clients
Web applications hosted on secure server with restricted access (e.g. ASP.NET, Java or PHP)
Apps that cannot protect and ensure the integrity of credentials (e.g. Javascript or Native apps
Capable of maintaining the confidentiality of their credentials
Incapable of maintaining the confidentiality of their credentials

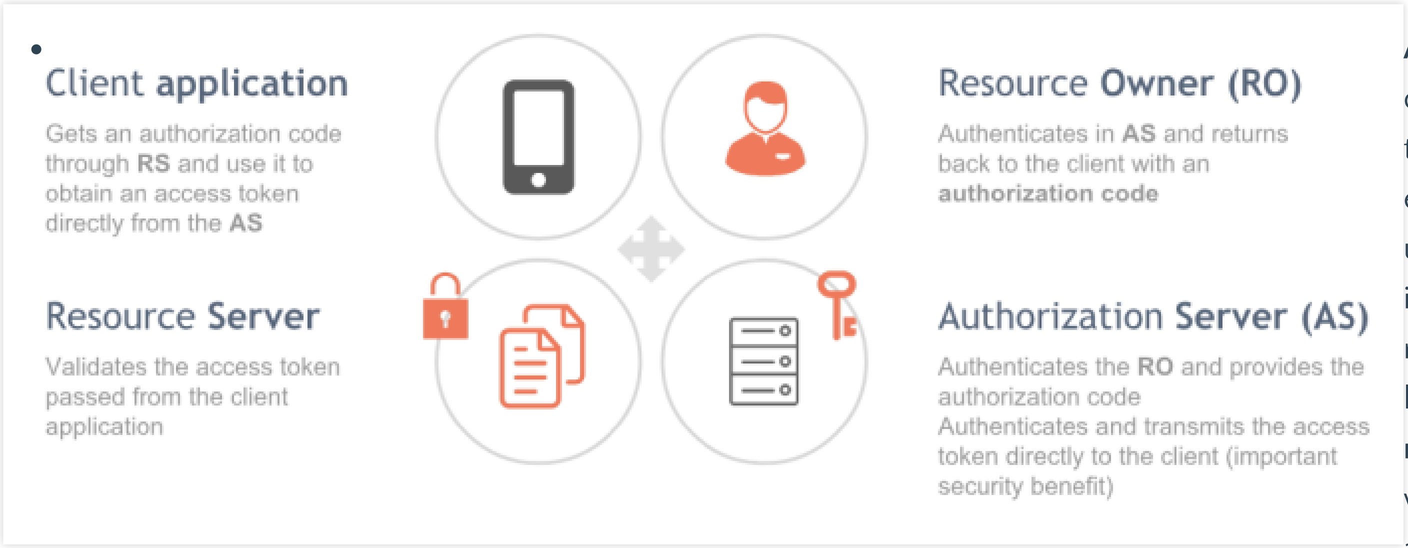
Authorization Grants

There are 4 basic grants that clients may use in OAuth 2.0 in order to get an access token, the `Authorization Code`, the `Implicit`, `Client Credentials` and the `Resource Owner Password Credentials` grant.

Authorization Code

The authorization code grant is a **redirection based flow**, meaning an authorization server is used as an intermediary between the client and the resource owner. In this flow the client directs the resource owner to an authorization server via the user-agent. After the resource owner's consent, the owner directs back to the client with an **authorization code**. Let's see the main responsibilities for each role on this grant

And here's the entire Flow

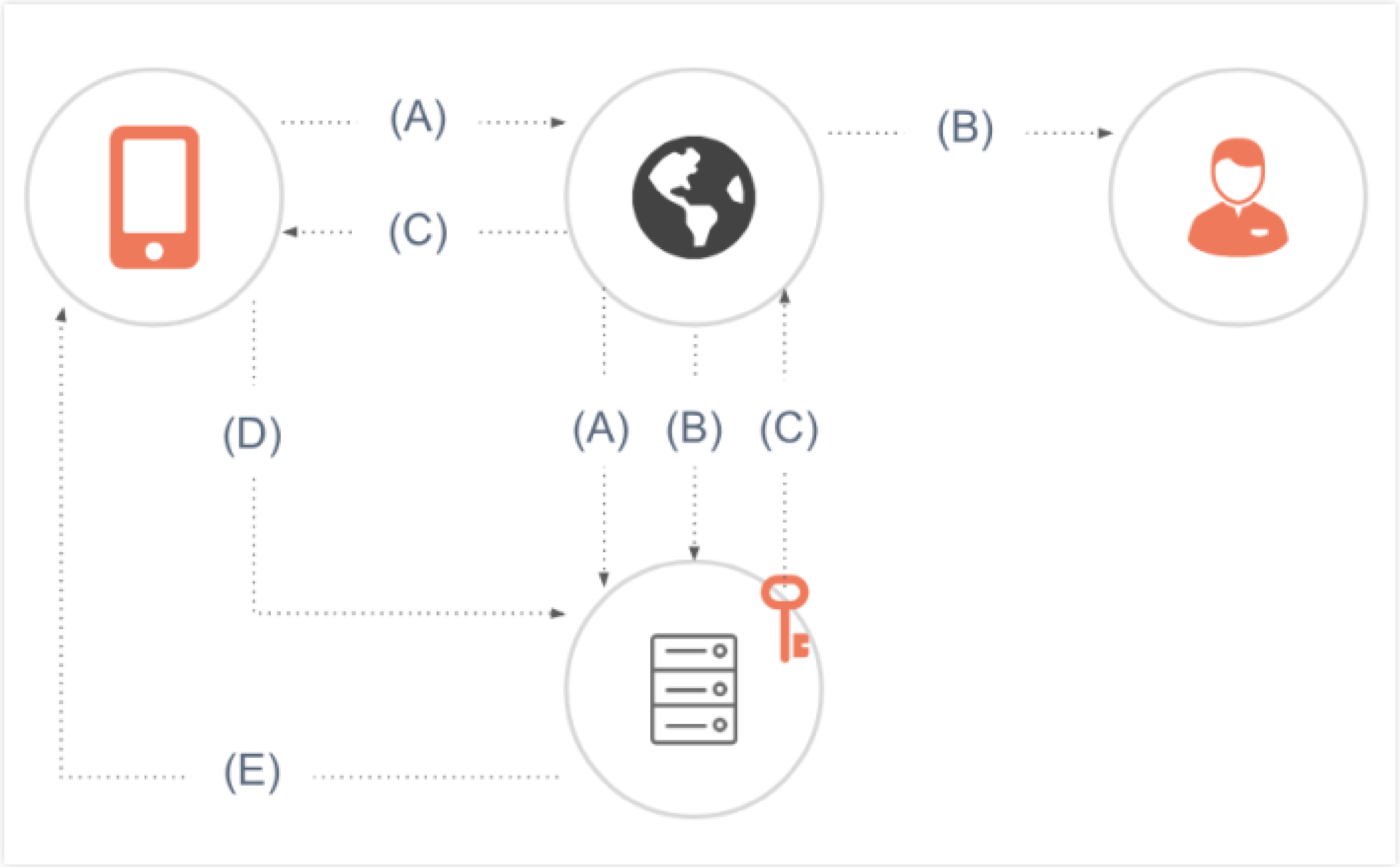

Client application
Gets an authorization code through RS and use it to obtain an access token directly from the AS
Resource Owner (RO)
Authenticates in AS and returns back to the client with an authorization code
Resource Server
Validates the access token passed from the client application
Authorization Server (AS)
Authenticates the RO and provides the authorization code
Authenticates and transmits the access token directly to the client (important security benefit)

- **A**: The Resource owner is directed to the authorization endpoint through the user-agent. The Client includes its identifier, requested scope, local state, and a **redirection URI** to which the authorization server will send the user-agent back once access is granted (or denied). The client's request looks like this:

```
GET /authorize?
    response_type=code&
    client_id=<clientId>&
    scope=email+api_access&
    state=xyz&
    redirect_uri=https://example.com/callback
```

The **response_type** which is equal to *code* means that the *authorization code* grant will be used. The *client_id* is the client's identifier and the *scope* defines what the client ask access for

- **B**: The authorization server authenticates the resource owner via the user-agent. The resource owner then grants or denies the client's access request usually via a **consent page**
- **C:** In case the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier in the *query parameter: **redirect_uri***. The redirection URI includes the authorization code in a **code** query string parameter and any state provided by the client on the first step. A redirection URI along with an authorization code looks like this:



```
GET /https://example.com/callback?
    code=SplxlOBeZQQYbYS6WxSbIA&
    state=xyz
```

- **D**: The client requests an **access token** from the authorization server's token endpoint by including the authorization code received in the previous step. The client also authenticates with the authorization server. For verification reason, the request also includes the redirection URI used to obtain the authorization code
  The request looks like this:

```
POST /token HTTP/1.1
Host: auth-server.example.com
Authorization: Basic F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

```

```
grant_type=authorization_code&
code=SplxlOBeZQQYbYS6WxSbIA&
redirect_uri=https://example.com/callback
```

- **E**: The authorization server authenticates the client, validates the authorization code, and ensures that the redirection URI received matches the URI used to redirect the client in the third step. If valid, the authorization server responds back with an access token and optionally, a refresh token. The response looks like this:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "access_token":"2YotnFZFEjr1zCsipAA",
  "token_type":"bearer",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0TlKWIA"
}
```

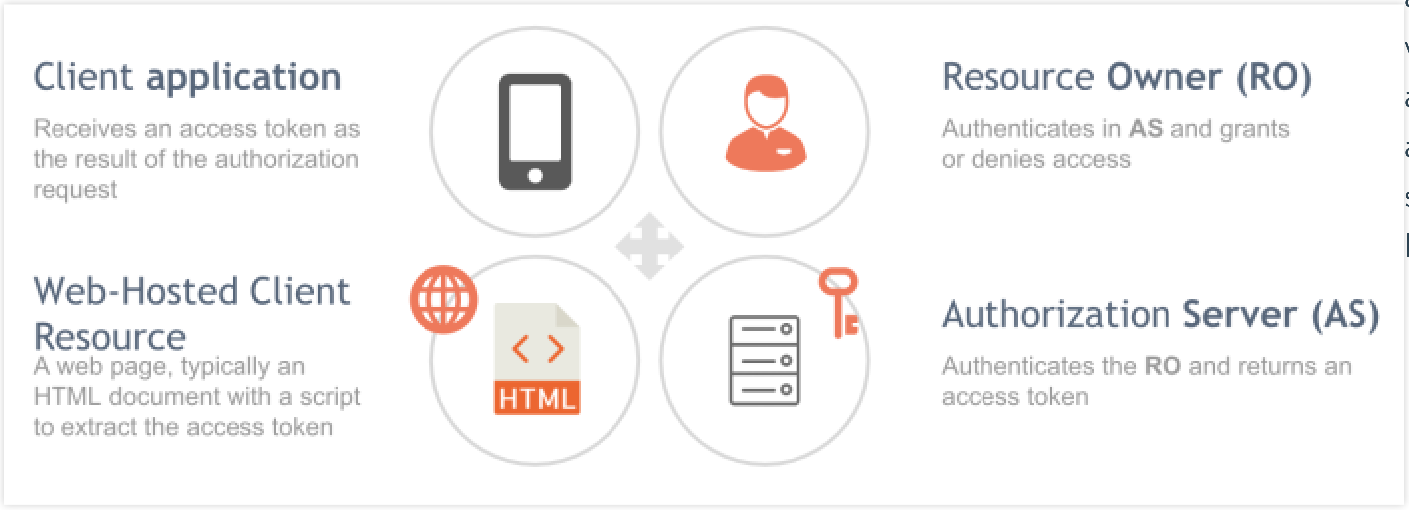
The `Authorization Code` grant is the one that provides the greater level of security since **a)** resource owner's credentials are never exposed to the client, **b)** it's a redirection based flow, **c)** client authenticates with the resource server and **d)** the access token is transmitted directly to the client without exposing it through the resource owner's user-agent *(implicit grant case)*

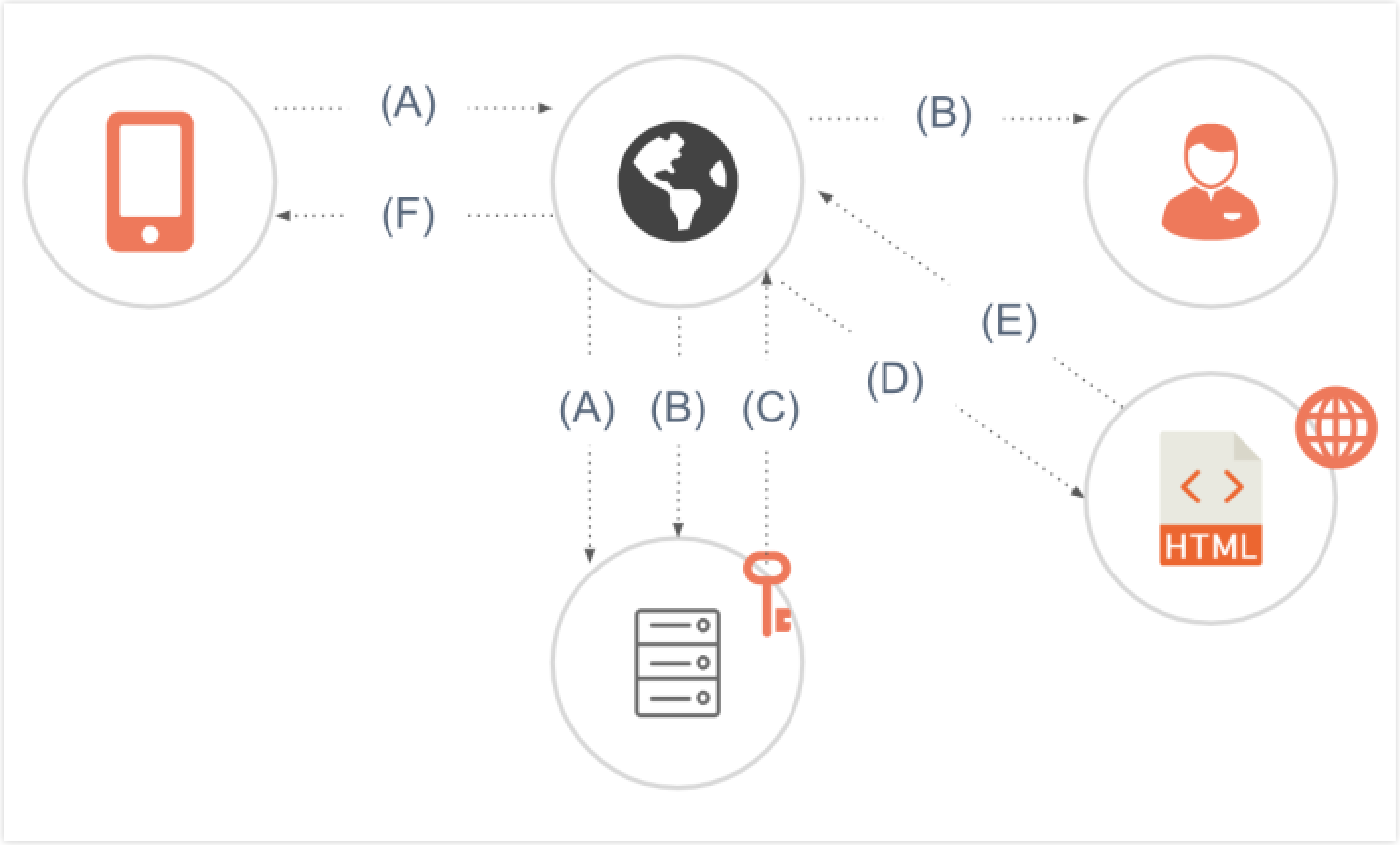
Implicit Grant

Implicit grant type is a simplified version of the *authorization code* where the client is issued an access token directly through the owner's authorization rather than issuing a new request using an authorization code.

Following are the steps for the implicit grant type.

- **A**: Client initiates the flow and directs the resource owner's user-agent to the authorization endpoint. The request includes the client's identifier, requested scope, any local state to be preserved and a redirection URI to which the


Client application
Receives an access token as the result of the authorization request
Resource Owner (RO)
Authenticates in AS and grants or denies access
Web-Hosted Client Resource
A web page, typically an HTML document with a script to extract the access token
HTML
Authorization Server (AS)
Authenticates the RO and returns an access token



authorization server will send the user-agent back once access is granted. A sample request looks like this:

```
GET /authorize?
    response_type=token&
    client_id=<clientId>&
    scope=email+api_access&
    state=xyz&
    redirect_uri=https://example.com/callback
```

Note that this time the **response_type** parameter has the value *token* instead of *code*, indicating that implicit grant is used

- **B**: The authorization server authenticates the resource owner via the user-agent. The resource owner then grants or denies the client's access request, usually via a **consent page**
- **C**: In case the resource owner grants access, the authorization server directs the owner back to the client using the **redirection URI**. The access token is now included in the URI fragment. The response looks like this:

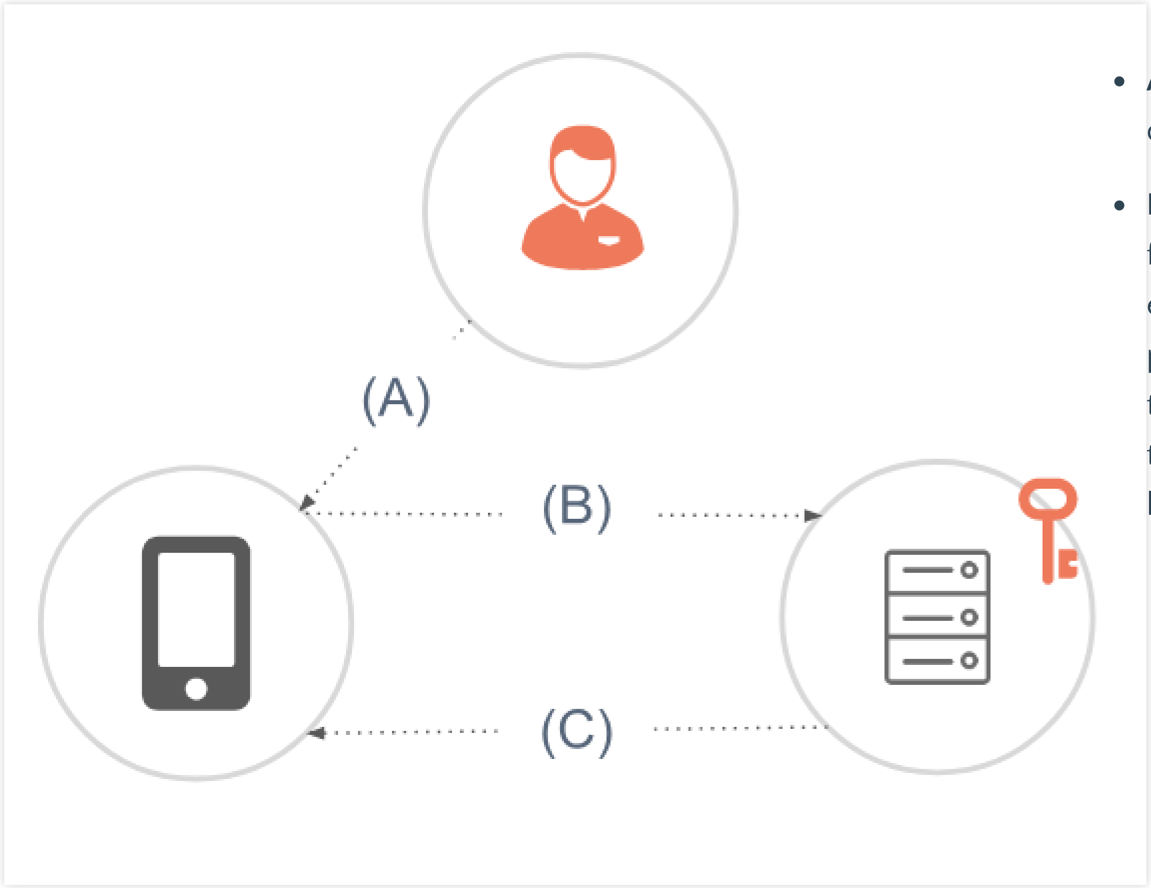
```
GET /https://example.com/callback?
    access_token=SpBeZQWxSbIA&
    expires_in=3600&
    token_type=bearer&
    state=xyz

```

- **D**: The user-agent follows the redirection instructions and makes a request to the web-hosted client resource. This is typically an HTML page with a script to extract the token from the URI
- **E:** The web page executes the script and extracts the access token from the URI fragment
- **F:** The user-agent finally passes the access token to the client

Implicit grant is optimized for **public** clients that typically run in a browser such as full Javascript web apps. There isn't a separate request for receiving the access token which makes it a little bit more responsive and efficient for that kind of clients. On the other hand, **it doesn't include client authentication** and the access token is exposed directly in the user-agent.

Resource Owner Password Credentials

The `Resource Owner Password Credentials` grant is a very simplified, non-directional flow where the Resource Owner provides the client with its username and password and the client itself use them to ask directly for an access token from the authorization server.



- **A**: The resource owner provides the client with its username and password
- **B**: The client requests an access token from the authorization server's token endpoint by including the credentials provided by the resource owner. During the request the client authenticates with the authorization server. The request looks like this:

```
POST /token HTTP/1.1
Host: auth-server.example.com:443
Authorization: Basic F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=password&
username=chsakell&
password=random_password
```

Notice that the **grant_type** is equal to *password* for this type of grant

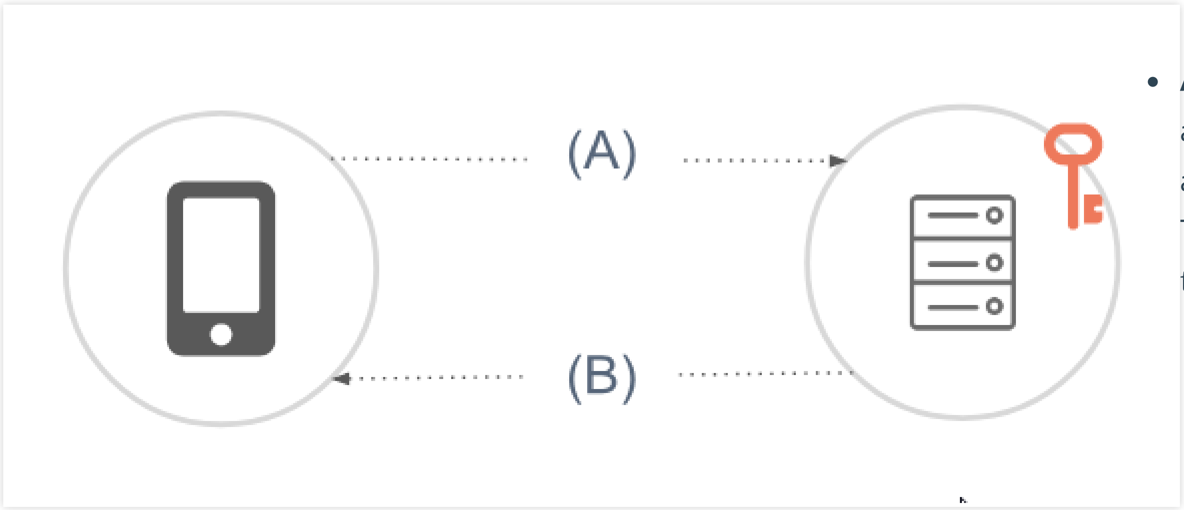
- **C**: The authorization server authenticates the client and validates the resource owner credentials. If all are valid issues an access token.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
"access_token":"2YotnFZFEjr1zCsipAA",
"token_type":"bearer",
"expires_in":3600,
"refresh_token":"tGzv3JOkF0TlKWIA"
}
```

This grant type is suitable for **trusted clients only** and when the other grant types are not available *(e.g. not a browser based client and user-agent cannot be used)*

Client Credentials Grant

The `Client Credentials` grant is again a simplified grant type that works entirely without a resource owner *(you can say that the client IS the resource owner)*.



- **A**: The client authenticates with the authorization server and requests an access token from the token endpoint. The authorization request looks like this:

```
POST /token HTTP/1.1
```

```
Host: auth-server.example.com:443
Authorization: Basic F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&
scope=email&api_access
```

Notice that the **grant_type** parameter is equal to *client_credentials*

- **B**: The authorization server authenticates the client and if valid, issues an access token

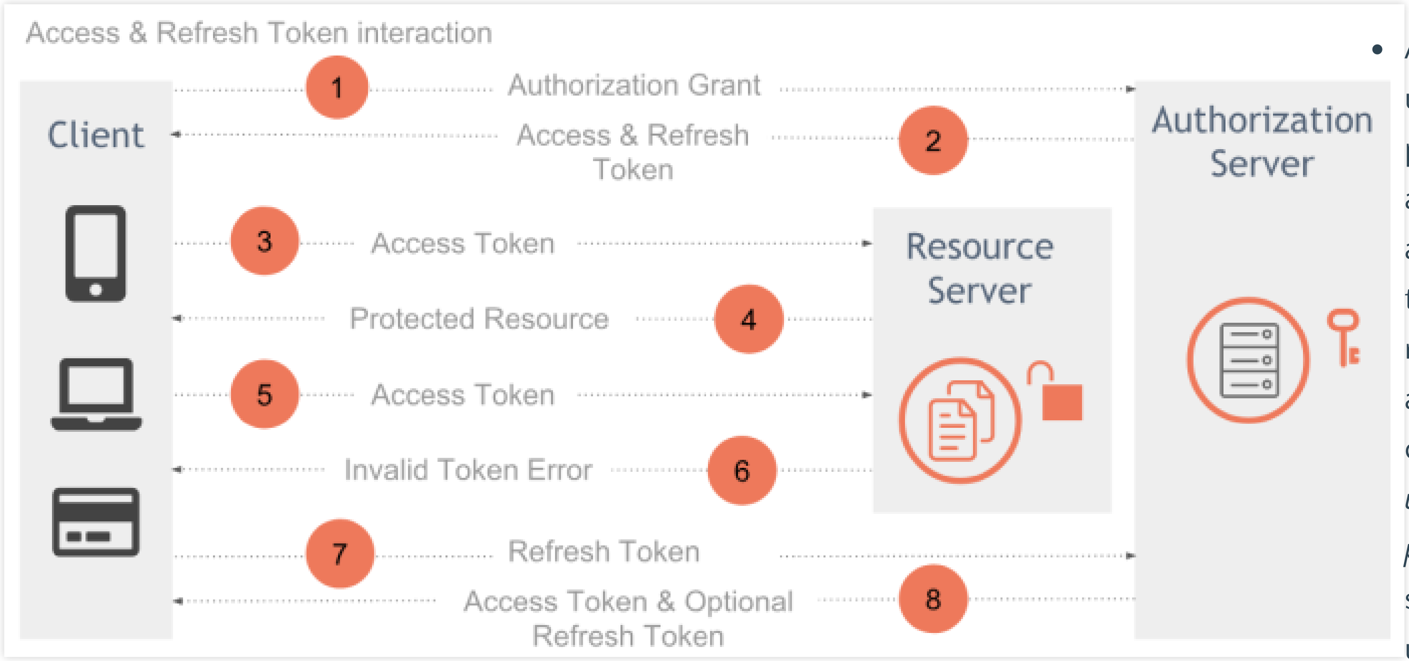
```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "access_token":"2YotnFZFEjr1zCsipAA",
  "token_type":"bearer",
  "expires_in":3600
}
```

This grant type is commonly used when the client acts on its own behalf. A very common case is when internal micro-services communicate with each other. The client also MUST be a confidential client.

Token Types

During the description of each Grant type you may have noticed that apart of the **access_token** an additional **refresh_token** may be returned by the authorization server. A refresh token may be returned only for the `Authorization Code` and the `Resource Owner Password Credentials` grants. `Implicit` grant doesn't support refresh tokens and **shouldn't be included** in the access token response of the `Client Credentials` grant. But what is the different between an access and a refresh token anyway?

The image illustrates the different between the two token types:



- An **access token** is used to access protected resources and represents authorization issued to the client. It replaces different authorization constructs *(e.g., username and password)* with a single token understood by the resource server

- A **refresh token** on the other hand which is also issued to the client by the Authorization server, is used to **obtain new access token** when current token becomes invalid or expires. If authorization server issues a refresh token, it is included when issuing an access token. The refresh token can only be used by the authorization server

OpenID Connect

When describing OAuth 2.0 we said that its purpose is to issue access tokens in order to provide limited **access** to protected resources, in other words OAuth 2.0 provides **authorization** but it doesn't provide authentication. The actual user is never authenticate directly with the client application itself. Access tokens provide a level of `pseudo-authentication` with no identity implication at all. This pseudo-authentication doesn't provide information about when, where or how the authentication occurred. This is where `OpenID Connect` enters and fills the authentication gap or limitations in OAuth 2.0. `OpenID Connect` is a simple identity layer on top of the OAuth 2.0 protocol. It enables clients to verify the identity of the End-User based on the authentication performed by an authorization server. It obtains basic profile information about the

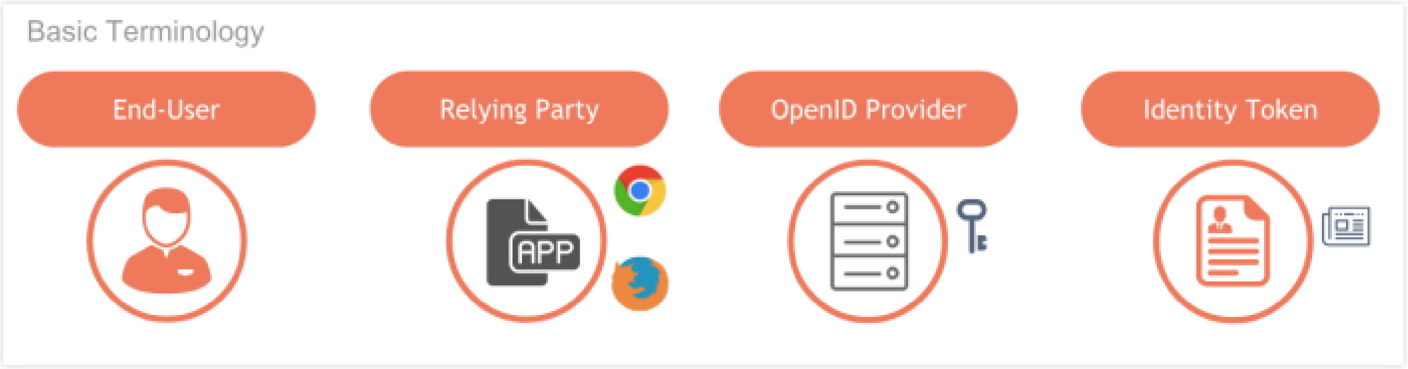
End-User in an interoperable and REST-like manner *(introduction of new REST endpoints)*. It uses **Claims** to communicate information about the End-User and extends OAuth in a way that cloud based applications can:

- Get identity information
- Retrieve details about the authentication event
- Allow federated Single Sign On

Let's see the basic terminology used in OpenID Connect.

1. **End-User**: Human participant – in OAuth this refers to the resource owner having their own identity as one of their protected resources
2. **Relying Party**: OAuth 2.0 client application. Requires End-User authentication and Claims from an OpenID Provider
3. **Identity Provider**: An OAuth 2.0 Authorization Server that authenticates the End-User and provides Claims to the Relying Party about the authentication event and the End-User
4. **Identity Token**: A **JSON Web Token (JWT)** containing claims about the authentication event. It may contain other claims as well

As OpenID Connect sits on top of OAuth 2.0, it makes sense if we say that it uses some of the OAuth 2.0 flows. In fact, OpenID Connect can follow the `Authorization Code` flow, the `Implicit` and the `Hybrid` which is a combination of the previous two. The flows are exactly the same with the only difference that an **id_token** is issued along with the **access_token**. Whether the flow is a pure OAuth 2.0 or an OpenID Connect is determined by the presence if the **openid** scope in the authorization request.



OAuth 2.0 & OpenID Connect Terminology

Don't get confused by the different terminology that OpenID Connect uses, they are just different names for the same entities

- End User *(OpenID Connect)* – Resource Owner *(OAuth 2.0)*
- Relying Party *(OpenID Connect)* – Client *(OAuth 2.0)*
- OpenID Provider *(OpenID Connect)* – Authorization Server *(OAuth 2.0)*

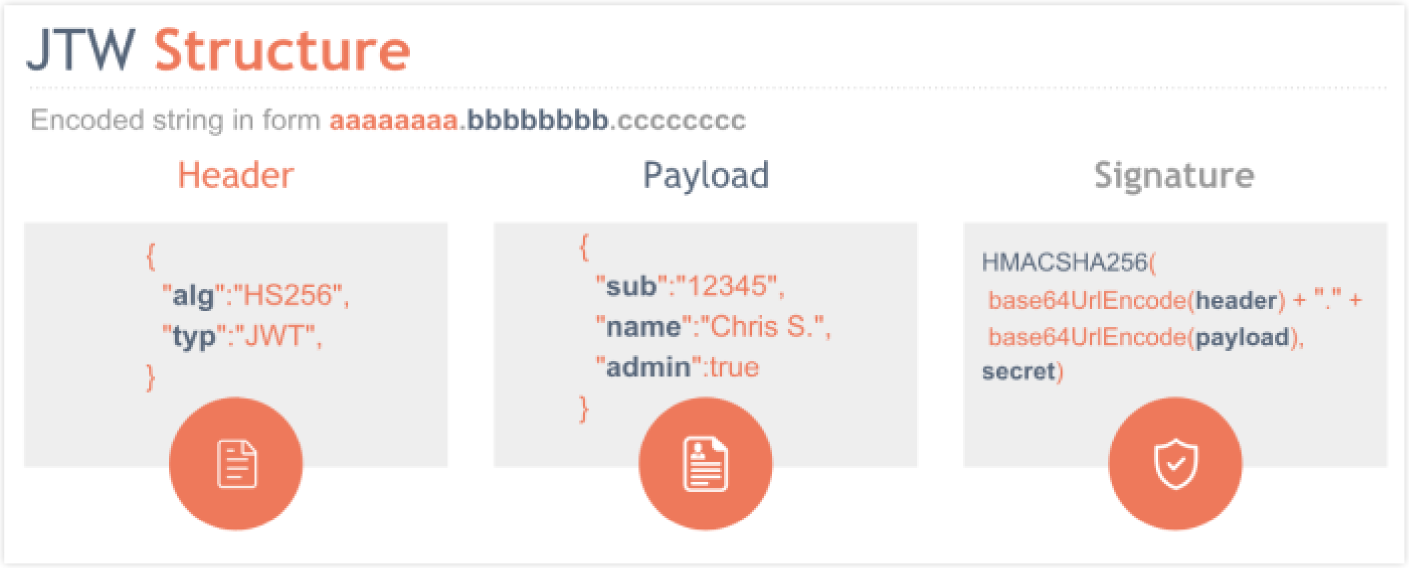
Identity Token & JWT

The identity token contains the information about the authentication performed and is returned as a `JSON Web Token`. But what is a JSON Web Token anyway? JSON Web Tokens is an **open standard** method for representing claims that can be securely transferred between two parties. They are **digitally signed** meaning the information is verified and trusted that there is no alteration of data during the transfer. They are **compact** and can be send via URL, POST request or HTTP header. They are **Self-Contained** meaning they are validated locally by resource servers using the Authorization Server signing key. This is very important to remember and understand it – the token is issued from the authorization server and normally, when sent to the resource server would require to send it back to the authorization server for validation!

JWT Structure

A JWT is a encoded string that has 3 distinct parts: the *header*, the *payload* and the *signature*:

- **Header**: A *Base64Url* encoded JSON that has two properties: a) **alg** – the algorithm like HMAC SHA256 or RSA used to generate the signature and b) **typ** the type of the JWT token
- **Payload**: A *Base64Url* encoded JSON that contains the **claims** which are user details or additional metadata
- **Signature**: It ensures that data haven't changed during the transfer by combining the base64 header and payload with a secret



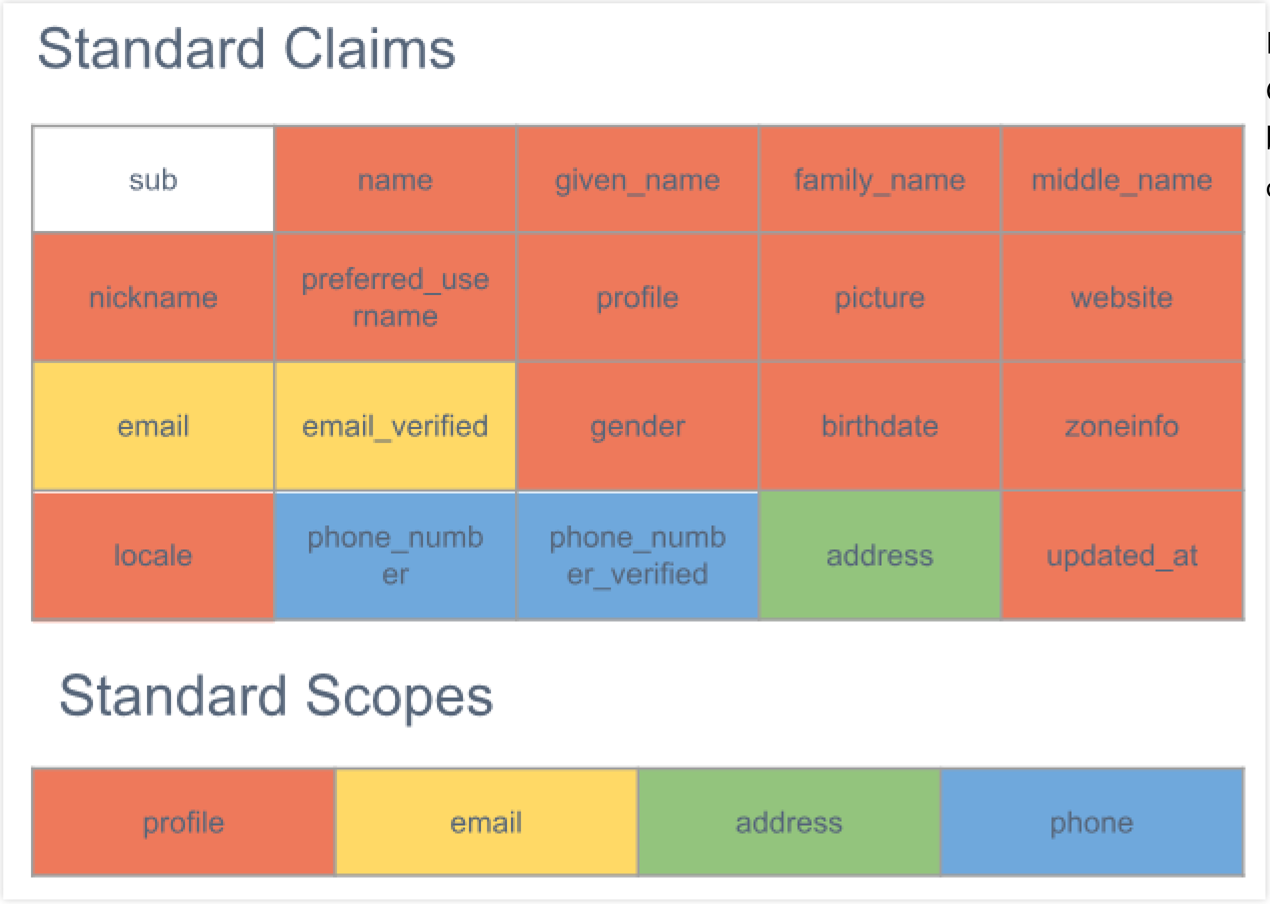
```
{
  "alg":"HS256",
  "typ":"JWT",
}
```

```
{
  "sub":"12345",
  "name":"Chris S.",
  "admin":true
}
```

```
HMACSHA256(
 base64UrlEncode(header) + "." +
 base64UrlEncode(payload),
secret)
```

Claims and Scopes

Claim is an individual piece of information in a key-value pair. Scopes are used to request specific sets of claims. **OpenId** scope is mandatory scope to specify that OpenID Connect should be used. You will see later on when describing the OpenID Connect flows, that all scopes will contain the *openid* word, meaning this is an OpenID Connect authorization request. OpenID Connect defines a standard set of basic profile claims. Pre-defined sets of claims can be requested using specific scope values. Individual claims can be requested using the claims request parameter. Standard claims can be requested to be returned either in the **UserInfo response** or in the ID Token. The following table shows the association between standard scopes with the claims provided.



Standard Claims

sub	name	given_name	family_name	middle_name
nickname	preferred_use rname	profile	picture	website
email	email_verified	gender	birthdate	zoneinfo
locale	phone_numb er	phone_numb er_verified	address	updated_at

Standard Scopes

profile	email	address	phone

If you add the *email* scope in an OpenID Connect request, then both *email* and *email_verified* claims will be returned.

OAuth 2.0 & OpenID Connect Endpoints

OAuth 2.0 provides endpoints to support the entire authorization process. Obviously, these endpoints are also used by OpenID Connect which in turn adds a new one named **UserInfo Endpoint**.

- **Authorization endpoint**: Used by the client to obtain authorization from the resource owner via user-agent redirection. Performs Authentication of the End-User which is directed through User-Agent. This is the endpoint where you directed when you click the *Login with some-provider* button

- **Token endpoint**: Used by the client to exchange an authorization grant for an access token. It returns an **access token**, an **id token** in case it's an OpenID Connect request and optionally a **refresh token**
- **UserInfo endpoint**: This is an addition to OAuth 2.0 by the OpenID Connect and its purpose is to return claims about the authenticated end-user. The request to this endpoint requires an access token retrieved by an authorization request
- **Client endpoint**: This is actually an endpoint that belongs to the client, not to the authorization server. It is used though by the authorization server to return responses back to the client via the resource owner's user-agent

OpenID Connect Flows

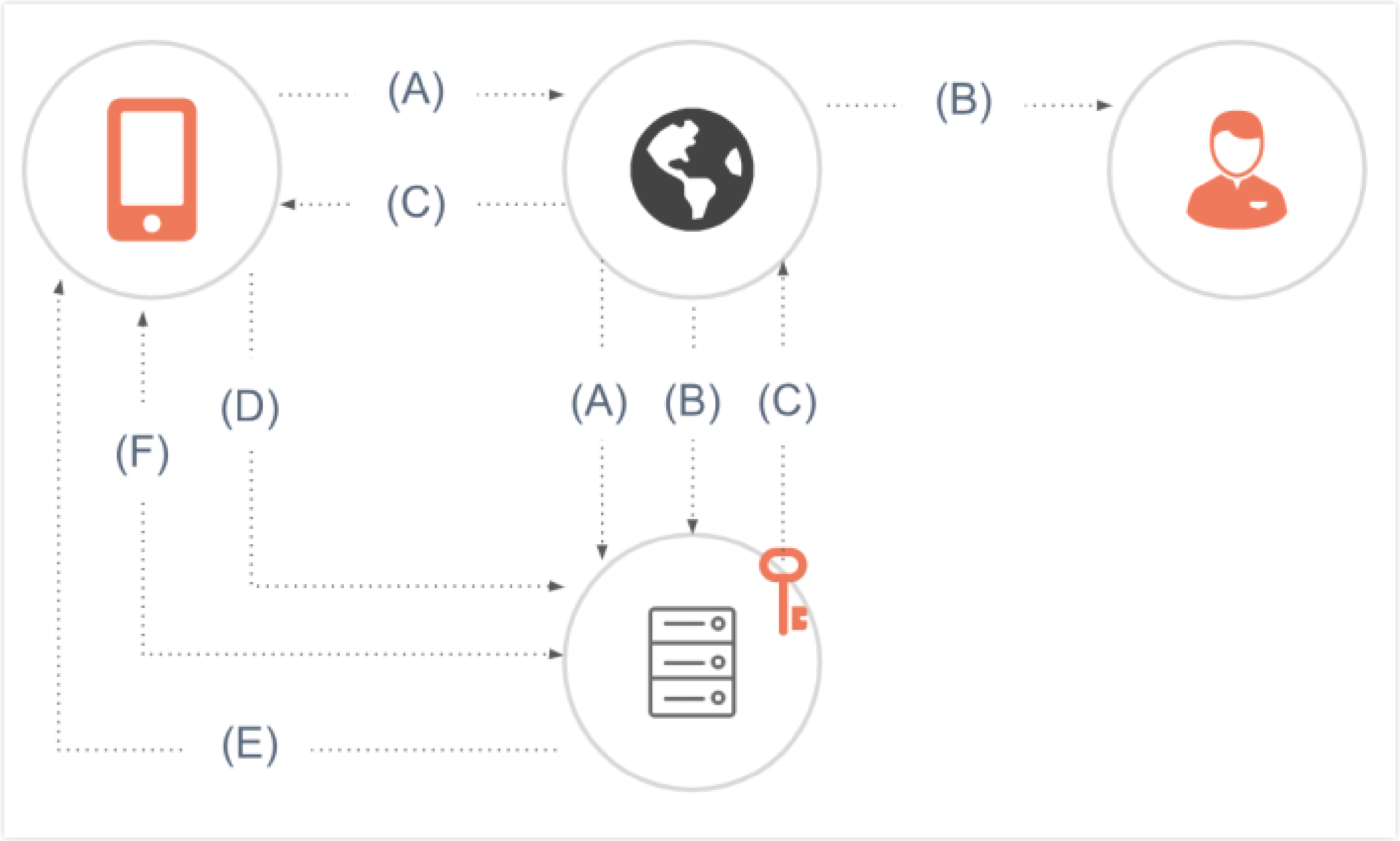
Let's see how `Authorization Code` and `Implicit` flows work with OpenID Connect. We 'll leave the `Hybrid` flow out of the scope of this post.

Authorization Code

Generally speaking the flow is exactly the same as described in the OAuth 2.0 authorization code grant. The first difference is that since we need to initiate an OpenID Connect flow instead of a pure OAuth flow, we add the **openid** scope in the authorization request *(which is sent to the authorization endpoint..)*. The **response_type** parameter remains the same, *code*

```
GET /authorize?
    response_type=code&
    client_id=<clientId>&
    scope=openid profile email&
    state=xyz&
        redirect_uri=https://example.com/callback
```

The response is again a redirection to the client's redirection URI with a *code* fragment.



```
GET /https://example.com/callback?
    code=Splxl0BeZQQYbYS6WxSbIA&
    state=xyz
```

Following is the request to the **token endpoint**, same as described in the OAuth 2.0.

```
POST /token HTTP/1.1
Host: auth-server.example.com
Authorization: Basic F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=Splxl0BeZQQYbYS6WxSbIA&
redirect_uri=https://example.com/callback
```

The difference though is that now we don’t expect only an **access_token** and optionally a **refresh_token** but also an **id_token**.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
    "access_token":"2YotnFZFEjr1zCsipAA",
    "id_token":"2YotnFZFEjr1zCsipAA",
    "token_type":"bearer",
    "expires_in":3600,
    "refresh_token":"tGzv3JOkF0TlKWIA"
}
```

The id_token itself contains basic information about the authentication event along with a **subject** identifier such as the user’s id or name. For any additional claims or scopes that are added in the initial authorization request *(e.g. email, profile)* the client sends an extra request to the authorization endpoint. This request requires the access token retrieved in the previous step.

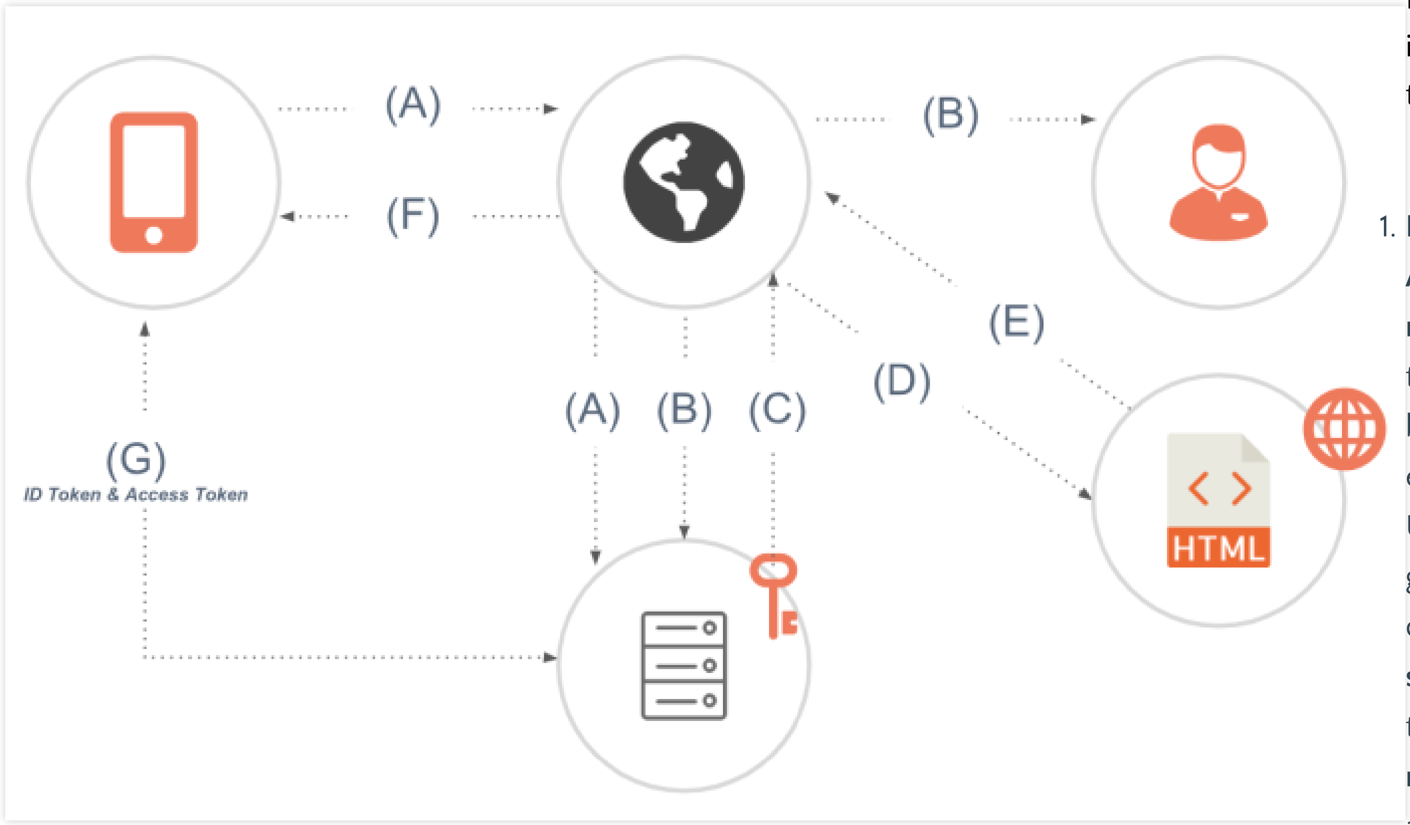
```
GET /userinfo HTTP/1.1
Host: auth-server.example.com
Authorization: Bearer F0MzpnWDFmQmF0M2JW
```

Notice that the access token is sent as a bearer token. The UserInfo response contains the claims asked on the initial request

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
    "sub":"12345”,
    "name":"Christos Sakellarios",
    "given_name":”Christos”,
    "picture":"http://example.com/chsakell/me.jpg"
}
```

Implicit Flow

Recall from the implicit flow described in the OAuth 2.0 that this is a simplified version of *authentication flow* where the access token is returned directly as the result of the resource owner’s authorization.



In the OpenID Connect implicit flow there are two cases:

1. **Both ID Token and Access Token are returned:** In this case the access token will be used to send an extra request to the UserInfo endpoint and get the additional claims defined on the **scope** parameter. In this case you set the **response_type** authorization's request parameter to **id_token token** meaning you expect both an *id_token* & an *access_token* The authorization's request in this case looks like this:

```
GET /authorize?
    response_type=id_token token&
    client_id=<clientId>&
    scope=openid profile&
    state=xyz&
    redirect_uri=https://example.com/callback
```

2. **Only ID Token is returned**: In this case you have no intentions to make an extra call to the UserInfo endpoint for getting additional claims but you want them directly on the **id token**. To do this you set the **response_type** equal to *id_token*

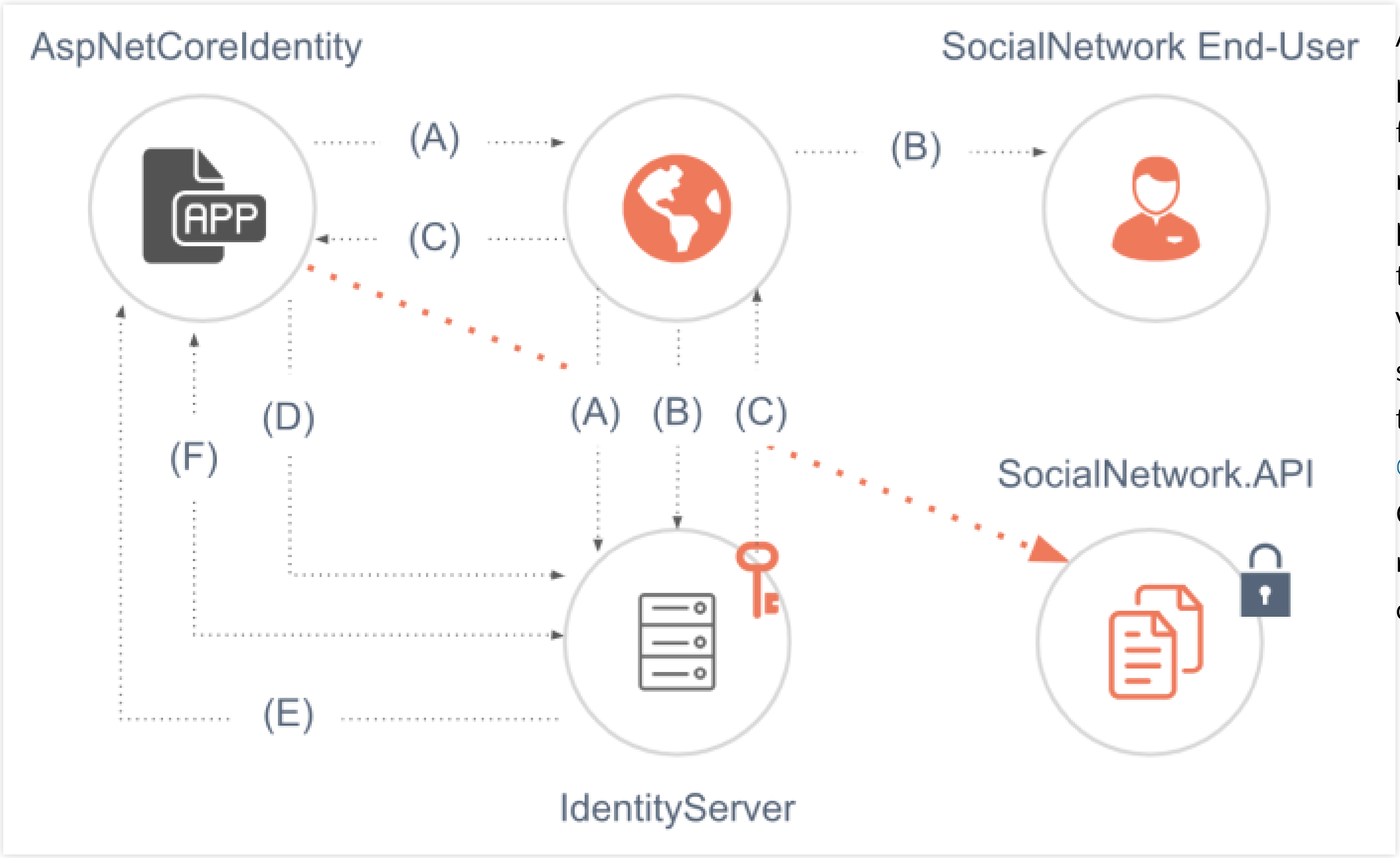
```
GET /authorize?
    response_type=id_token&
    client_id=<clientId>&
    scope=openid profile&
    state=xyz&
    redirect_uri=https://example.com/callback
```

ID Token will contain the standard claims along with those asked in the scope

IdentityServer 4

It would take a lot of effort to implement all the specs defined by OAuth 2.0 and OpenID Connect by yourself, luckily though, you don't have to because there is IdentityServer. All that IdentityServer does is adds the spec compliant OpenID Connect and OAuth 2.0 endpoints to an ASP.NET Core application through **middleware**. This means that by adding its middleware to your application's pipeline you get the **authorization** and **token** endpoints we have talked about and all the core functionality needed *(redirecting, granting access, token validation, etc..)* for implementing the spec. All you have to do is provide some basic pages such as the Login, Logout and Logout views. It the IdentityServer4 you will find lots of samples which I recommend you to spend some time and study them. In this post we will use the project we have built so far during the series and cover the following scenario:

- **AspNetCoreIdentity** web application will play the role of a third-party application or a `Relying party` if you prefer
- There will be a hypothetical Social Network where you have an account. This account of course is an entire different account from the one you have in the *AspNetCoreIdentity* web application
- There will be a **SocialNetwork.API** which exposes your contacts on the Social Network
- The *SocialNetwork.API* will be protected through an **IdentityServer** for which will be a relevant project in the solution
- The idea is to share something with your SocialNetwork contacts through the *AspNetCoreIdentity* web app. To achieve this, the *AspNetCoreIdentity* web app needs to receive an **access token** from *IdentityServer* app and use it to access the protected resource which is the *SocialNetwork.API*

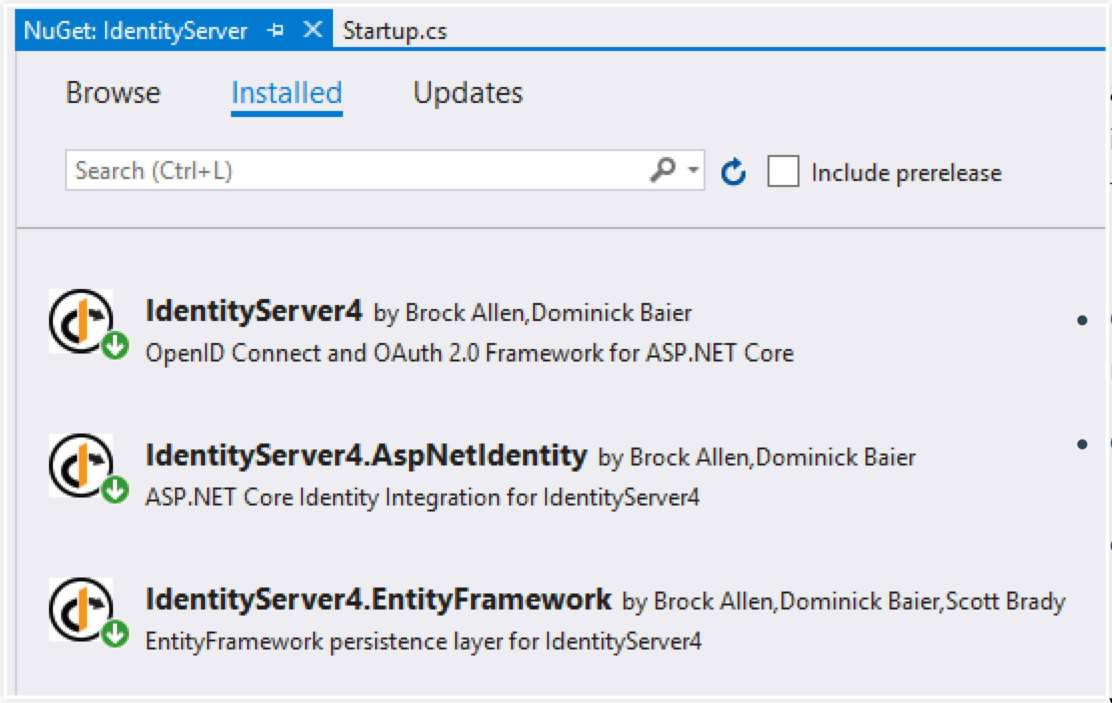


As illustrated on the previous image, our final goal is to send a request to the protected resource in the *SocialNetwork.API*. We will use the most secure flow which is the `Authorization Code` with OpenID Connect. Are you ready? Let's see some code!

Authorization Server Setup

The `IdentityServer` project in the solution was created as an empty .NET Core Web Application. Its role is to act as the **Identity Provider** *(or as the **Authorization Server** if you prefer – from now on we will use Identity Provider when we refer to this project)*. The first thing you need to do to integrate IdentityServer in your app is to install the `IdentityServer4` NuGet package. This will provide the core middleware to be plugged in your pipeline. Since this series are related to `ASP.NET Core Identity` we will also use the IdentityServer4.AspNetIdentity and the IdentityServer4.EntityFramework integration packages.

`IdentityServer4.AspNetIdentity` provides a configuration API to use the ASP.NET Identity management library for IdentityServer users. `IdentityServer4.EntityFramework` package provides an EntityFramework implementation for



IdentityServer4 by Brock Allen,Dominick Baier
OpenID Connect and OAuth 2.0 Framework for ASP.NET Core

IdentityServer4.AspNetIdentity by Brock Allen,Dominick Baier
ASP.NET Core Identity Integration for IdentityServer4

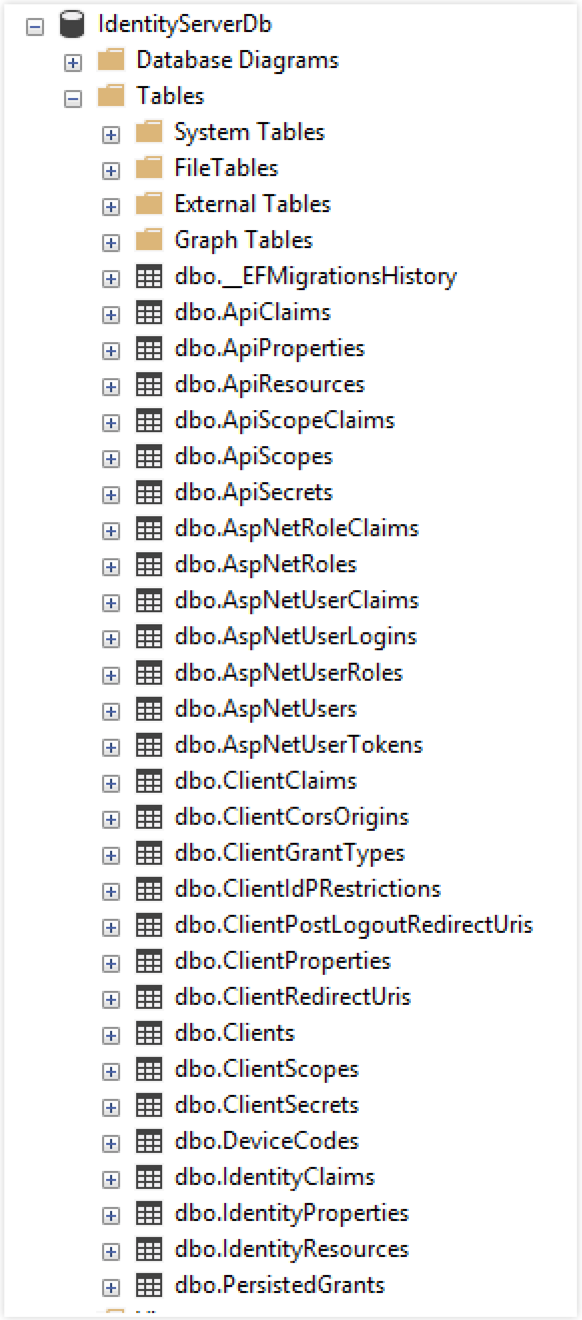
IdentityServer4.EntityFramework by Brock Allen,Dominick Baier,Scott Brady
EntityFramework persistence layer for IdentityServer4

the configuration and operational stores in IdentityServer. But what does this mean anyway? IdentityServer uses some type of infrastructure in order to provide its functionality and more specifically:

- **Configuration data**: Data for defining resources and clients
- **Operational data**: Data produced by the IdentityServer, such as **tokens**, **codes** and **consents**

When you integrate EntityFramework it means that the database will contain all the required tables for IdentityServer to work. Let's see how this looks like.

Keep in mind that they are handled by two different `DbContext` classes, `PersistedGrantDbContext` and `ConfigurationDbContext`. Now let's switch to the `Startup` class and see how we plug IdentityServer into the pipeline. First we add the services for `ASP.NET Identity` in the way we have learned through the series, nothing new yet..



IdentityServerDb
Database Diagrams
Tables
System Tables
FileTables
External Tables
Graph Tables
dbo.__EFMigrationsHistory
dbo.ApiClaims
dbo.ApiProperties
dbo.ApiResources
dbo.ApiScopeClaims
dbo.ApiScopes
dbo.ApiSecrets
dbo.AspNetRoleClaims
dbo.AspNetRoles
dbo.AspNetUserClaims
dbo.AspNetUserLogins
dbo.AspNetUserRoles
dbo.AspNetUsers
dbo.AspNetUserTokens
dbo.ClientClaims
dbo.ClientCorsOrigins
dbo.ClientGrantTypes
dbo.ClientIdPRestrictions
dbo.ClientPostLogoutRedirectUris
dbo.ClientProperties
dbo.ClientRedirectUris
dbo.Clients
dbo.ClientScopes
dbo.ClientSecrets
dbo.DeviceCodes
dbo.IdentityClaims
dbo.IdentityProperties
dbo.IdentityResources
dbo.PersistedGrants

```
services.AddDbContext<ApplicationDbContext>(options =>
{
    if (useInMemoryStores)
    {
        options.UseInMemoryDatabase("IdentityServerDb");
    }
    else
    {
        options.UseSqlServer(connectionString);
    }
});

services.AddIdentity<IdentityUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

Next thing we need to do is to register the required IdentityServer services and `DbContext` stores.

```
var builder = services.AddIdentityServer(options =>
{
    options.Events.RaiseErrorEvents = true;
    options.Events.RaiseInformationEvents = true;
    options.Events.RaiseFailureEvents = true;
    options.Events.RaiseSuccessEvents = true;
})
// this adds the config data from DB (clients, resources)
.AddConfigurationStore(options =>
{
    options.ConfigureDbContext = opt =>
    {
        if (useInMemoryStores)
        {
            opt.UseInMemoryDatabase("IdentityServerDb");
        }
        else
        {
            opt.UseSqlServer(connectionString);
        }
    };
})
// this adds the operational data from DB (codes, tokens, consents)
.AddOperationalStore(options =>
{
    options.ConfigureDbContext = opt =>
    {
        if (useInMemoryStores)
        {
            opt.UseInMemoryDatabase("IdentityServerDb");
        }
        else
        {
            opt.UseSqlServer(connectionString);
        }
    };

    // this enables automatic token cleanup. this is optional.
    options.EnableTokenCleanup = true;
})
.AddAspNetIdentity<IdentityUser>();
```

`AddAspNetIdentity` may take a custom `IdentityUser` of your choice, for example a class `ApplicationUser` that extends `IdentityUser`. `ASP.NET Identity` services needs to be registered before integrating IdentityServer because the latter needs to override some configuration from ASP.NET Identity. In the `ConfigureServices` function you will also find a call to `builder.AddDeveloperSigningCredential()` which creates a temporary key for signing tokens. It's OK for development but you need to be replace it with a valid persistent key when moving to production environment.

We use a *useInMemoryStores* variable read from the **appsettings.json** file to indicate whether we want to use an actual SQL Server database or not. If this variable is *false* then we make use of the EntityFramework's `UseInMemoryDatabase` functionality, otherwise we hit an actual database which of course needs to be setup first. IdentityServer also provides the option to keep store data in memory as shown below:

```
var builder = services.AddIdentityServer()
    .AddInMemoryIdentityResources(Config.GetIdentityResources())
    .AddInMemoryApiResources(Config.GetApis())
    .AddInMemoryClients(Config.GetClients());
```

But since we use EntityFramework integration we can use its `UseInMemoryDatabase` in-memory option

Next we need to register 3 things: **a)** Which are the API resources needs to be protected, **b)** which are the clients and how they can get access tokens, meaning what flows they are allowed to use and last but not least **c)** what are the OpenID Connect scopes allowed. This configuration exists in the `Config` class as shown below.

OpenID Connect allowed scopes

```
public static IEnumerable<IdentityResource> GetIdentityResources()
{
    return new List<IdentityResource>
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Profile(),
    };
}
```

Scopes represent something you want to protect and that clients want to access. In OpenID Connect though, scopes represent **identity data** like user id, name or email address and they need to be registered.

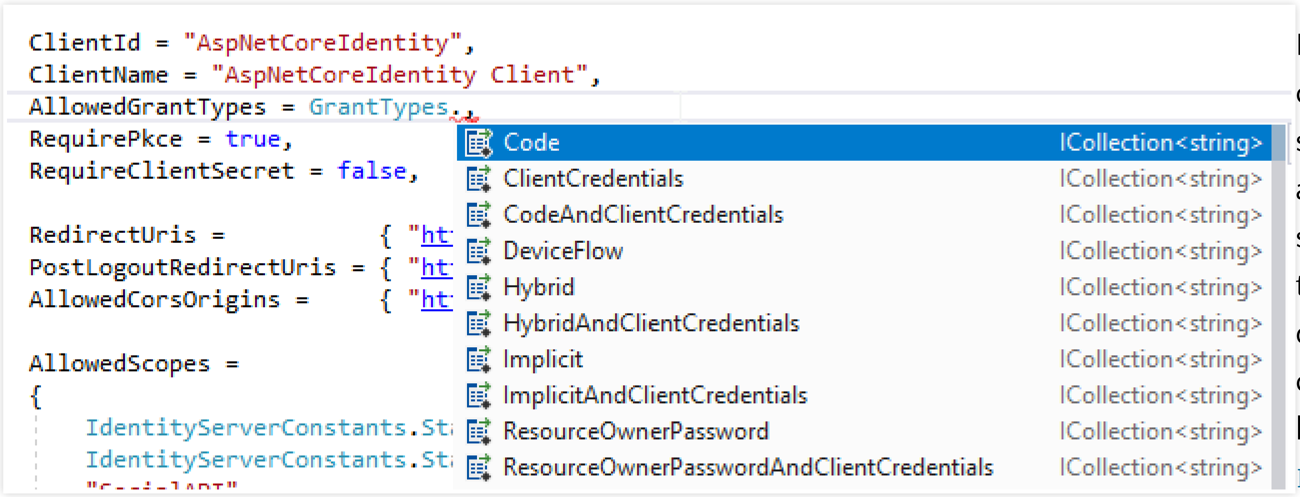
APIs to be protected

```
public static IEnumerable<ApiResource> GetApis()
{
    return new List<ApiResource>
    {
        new ApiResource("SocialAPI", "Social Network API")
    };
}
```

Clients allowed to request for tokens

```
public static IEnumerable<Client> GetClients()
{
    return new List<Client>
    {
        new Client
        {
            ClientId = "AspNetCoreIdentity",
            ClientName = "AspNetCoreIdentity Client",
            AllowedGrantTypes = GrantTypes.Code,
            RequirePkce = true,
            RequireClientSecret = false,

            RedirectUris =           { "http://localhost:5000" },
            PostLogoutRedirectUris = { "http://localhost:5000" },
            AllowedCorsOrigins =     { "http://localhost:5000" },

            AllowedScopes =
            {
                IdentityServerConstants.StandardScopes.OpenId,
                IdentityServerConstants.StandardScopes.Profile,
                "SocialAPI"
            }
        }
    };
}
```

We register the *AspNetCoreIdentity* client and we defined that it can use the `authorization code` flow to receive tokens. The redirect URIs needs to be registered as it has to match the authorization's request redirect URI parameter. We have also defined that this client is allowed to request the *openid*, *profile* OpenID Connect scopes plus the *SocialAPI* for accessing the SocialNetwork.API resources. Client will be hosted in ***http://localhost:5000***. The `AllowedGrantTypes` property is where you define how clients get access to the protected resources. Intellisense shows that there are several options to pick.



```
ClientId = "AspNetCoreIdentity",
ClientName = "AspNetCoreIdentity Client",
AllowedGrantTypes = GrantTypes.,
RequirePkce = true,
RequireClientSecret = false,

RedirectUris =           { "ht
PostLogoutRedirectUris = { "ht
AllowedCorsOrigins =     { "ht

AllowedScopes =
{
    IdentityServerConstants.St
    IdentityServerConstants.St
```

Code ICollection<string>
ClientCredentials ICollection<string>
CodeAndClientCredentials ICollection<string>
DeviceFlow ICollection<string>
Hybrid ICollection<string>
HybridAndClientCredentials ICollection<string>
Implicit ICollection<string>
ImplicitAndClientCredentials ICollection<string>
ResourceOwnerPassword ICollection<string>
ResourceOwnerPasswordAndClientCredentials ICollection<string>

Each option will require the client to act respectively and send the appropriate authorization request to the server for getting access and id tokens. Now that we have defined IdentityServer configuration data we have to load them. You will find a `DatabaseInitializer` class that does this.

```
DB initializer - register IdentityServer configs
```

```
private static void InitializeIdentityServer(IServiceProvider provider)
{
    var context = provider.GetRequiredService<ConfigurationDbContext>();
    if (!context.Clients.Any())
    {
        foreach (var client in Config.GetClients())
```

```
            {
                context.Clients.Add(client.ToEntity());
            }
            context.SaveChanges();
        }

        if (!context.IdentityResources.Any())
        {
            foreach (var resource in Config.GetIdentityResources())
            {
                context.IdentityResources.Add(resource.ToEntity());
            }
            context.SaveChanges();
        }

        if (!context.ApiResources.Any())
        {
            foreach (var resource in Config.GetApis())
            {
                context.ApiResources.Add(resource.ToEntity());
            }
            context.SaveChanges();
        }
    }
```

This class also registers a default `IdentityUser` so that you can login when you fire up the application. You will also find a register link in case you want to create your own user.

```
var userManager = provider.GetRequiredService<UserManager<IdentityUser>>();
var chsakell = userManager.FindByNameAsync("chsakell").Result;
if (chsakell == null)
{
    chsakell = new IdentityUser
    {
        UserName = "chsakell"
    };
    var result = userManager.CreateAsync(chsakell, "$AspNetIdentity10$").Result;
    if (!result.Succeeded)
    {
        throw new Exception(result.Errors.First().Description);
    }

    chsakell = userManager.FindByNameAsync("chsakell").Result;

    result = userManager.AddClaimsAsync(chsakell, new Claim[]{
        new Claim(JwtClaimTypes.Name, "Chris Sakellarios"),
        new Claim(JwtClaimTypes.GivenName, "Christos"),
        new Claim(JwtClaimTypes.FamilyName, "Sakellarios"),
        new Claim(JwtClaimTypes.Email, "chsakellsblog@blog.com"),
        new Claim(JwtClaimTypes.EmailVerified, "true", ClaimValueTypes.Boolean),
        new Claim(JwtClaimTypes.WebSite, "https://chsakell.com"),
        new Claim(JwtClaimTypes.Address, @"{ 'street_address': 'localhost 10', 'postal_code
            IdentityServer4.IdentityServerConstants.ClaimValueTypes.Json)
    }).Result;
    // code omitted
```

Notice that we assigned several claims for this user but only a few belongs to the open id **profile** scope that the *AspNetCoreIdentity* client can get access to. We ‘ll see in action what this means.

SocialNetwork.API

SocialNetwork.API is a simple .NET Core Web application exposing the *api/contacts* protected endpoint.

```
[HttpGet]
[Authorize]
public ActionResult<IEnumerable<Contact>> Get()
{
    return new List<Contact>
    {
        new Contact
        {
            Name = "Francesca Fenton",
            Username = "Fenton25",
            Email = "francesca@example.com"
        },
        new Contact {
            Name = "Pierce North",
            Username = "Pierce",
            Email = "pierce@example.com"
        },
        new Contact {
            Name = "Marta Grimes",
            Username = "GrimesX",
            Email = "marta@example.com"
        },
        new Contact{
            Name = "Margie Kearney",
            Username = "Kearney20",
            Email = "margie@example.com"
        }
    };
}
```

All you have to do to protect this API using the OpenID Provider we described, is define how authorization and authentication works for this project in the `Startup` class.

```
services.AddAuthorization();

services.AddAuthentication("Bearer")
    .AddJwtBearer("Bearer", options =>
    {
        options.Authority = "http://localhost:5005";
        options.RequireHttpsMetadata = false;

        options.Audience = "SocialAPI";
    });
```

Here we define that **Bearer** scheme will be the default authentication scheme and that we trust the OpenID Provider hosted in port *5005*. The *Audience* must match the API resource name we defined before.

Client setup

The client uses a javascript library named *oidc-client* which you can find **here**. You can find the same functionality for interacting with OpenID Connect flows written in popular client side frameworks *(angular, vue.js, etc..)*. The client needs to setup its own configuration which must match the Identity Provider's setup. There is an `openid-connect.service.ts` file that does this.

OpenIdConnectService

```
declare var Oidc : any;

```

```
@Injectable()
export class OpenIdConnectService {

    config = {
        authority: "http://localhost:5005",
        client_id: "AspNetCoreIdentity",
        redirect_uri: "http://localhost:5000",
        response_type: "code",
        scope: "openid profile SocialAPI",
        post_logout_redirect_uri: "http://localhost:5000",
    };
    userManager : any;

    constructor() {
        this.userManager = new Oidc.UserManager(this.config);
    }

    public getUser() {
        return this.userManager.getUser();
    }

    public login() {
        return this.userManager.signinRedirect();;
    }

    public signinRedirectCallback() {
        return new Oidc.UserManager({ response_mode: "query" }).signinRedirectCallback();
    }

    public logout() {
        this.userManager.signoutRedirect();
    }
}
```

The library exposes an `Oidc` object that provides all the OpenID Connect features. Notice that the config object matches exactly the configuration expected by the authorization server. The **response_type** is equal to *code* and along with the **openid** scope means that the authorization response result is expected to have both an **access token** and an **id token**. Since this is an *authorization code* flow, the access token retrieved will be used to send an extra request to the *UserInfo* endpoint and get the user claims for the **profile** scope. The `share.component` angular component checks if you are logged in with your Social Network account and if so sends a request to the SocialNetwork.API by adding the access token in an *Authorization* header.

```
export class SocialApiShareComponent {

    public socialLoggedIn: any;
    public contacts: IContact[] = [];
    public socialApiAccessDenied : boolean = false;

    constructor(public http: Http,
        public openConnectIdService: OpenIdConnectService,
        public router: Router, public stateService: StateService) {
        openConnectIdService.getUser().then((user: any) => {
            if (user) {
                console.log("User logged in", user.profile);
                console.log(user);
                this.socialLoggedIn = true;

                const headers = new Headers();
                headers.append("Authorization", `Bearer ${user.access_token}`);

                const options = new RequestOptions({ headers: headers });

                const socialApiContactsURI = "http://localhost:5010/api/contacts";

                this.http.get(socialApiContactsURI, options).subscribe(result => {
                    this.contacts = result.json() as IContact[];

                }, error => {
                    if (error.status === 401) {
                        this.socialApiAccessDenied = true;
                    }
                });
            }

        });
    }

    login() {
        this.openConnectIdService.login();
    }

    logout() {
        this.openConnectIdService.logout();
    }
}
```

Now let's see in action the entire flow. In case you want to use SQL Server database for the IdentityServer make sure you run through the following steps:

Using Visual Studio

1. Open the **Package Manager Console** and *cd* to the *IdentityServer* project path
2. Migrations have already run for you so the only thing you need to do is update the database for the 3 db contexts. To do so, change the connection string in the appsettings.json file to reflect your SQL Server environment and run the following commands:

   ```
   Update-Database -Context ApplicationDbContext
   ```

   ```
   Update-Database -Context PersistedGrantDbContext
   ```

   ```
   Update-Database -Context ConfigurationDbContext
   ```

Without Visual Studio

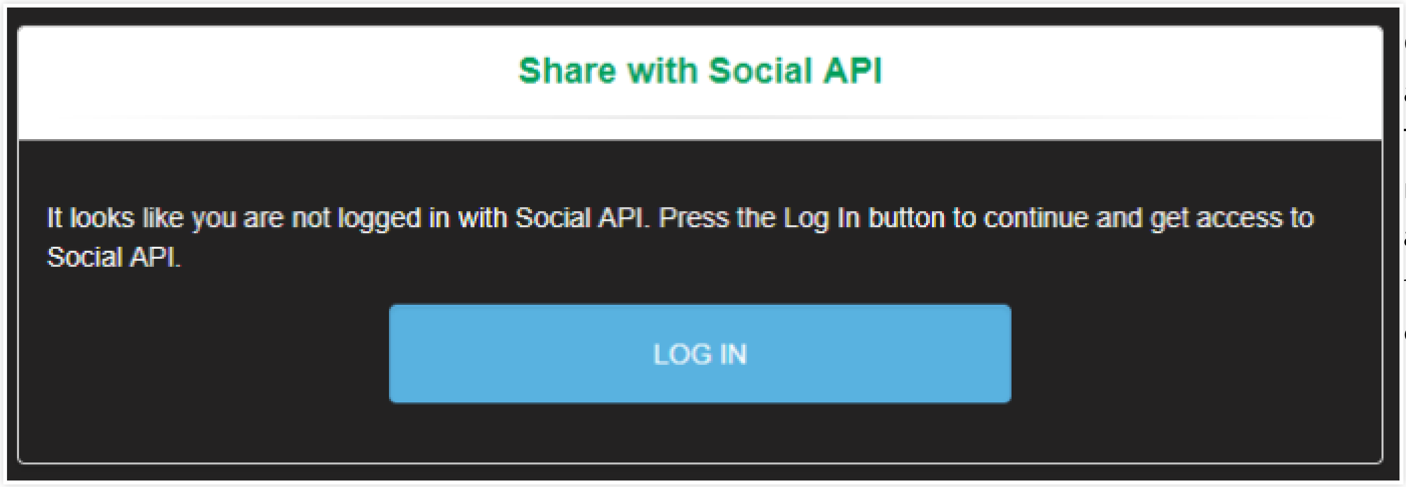
1. Open a **terminal** and *cd* to the *IdentityServer* project path
2. Migrations have already run for you so the only thing you need to do is update the database for the 3 db contexts. To do so, change the connection string in the appsettings.json file to reflect your SQL Server environment and run the following commands:

   ```
   dotnet ef database update -Context ApplicationDbContext
   ```

   ```
   dotnet ef database update -Context PersistedGrantDbContext
   ```

   ```
   dotnet ef database update -Context ConfigurationDbContext
   ```

Fire up all the projects and in the AspNetCoreIdentity web application click the *Share* from the menu. The oidc library will detect that you are not logged in with your Social Network account and present you with the following screen.



Share with Social API

It looks like you are not logged in with Social API. Press the Log In button to continue and get access to Social API.

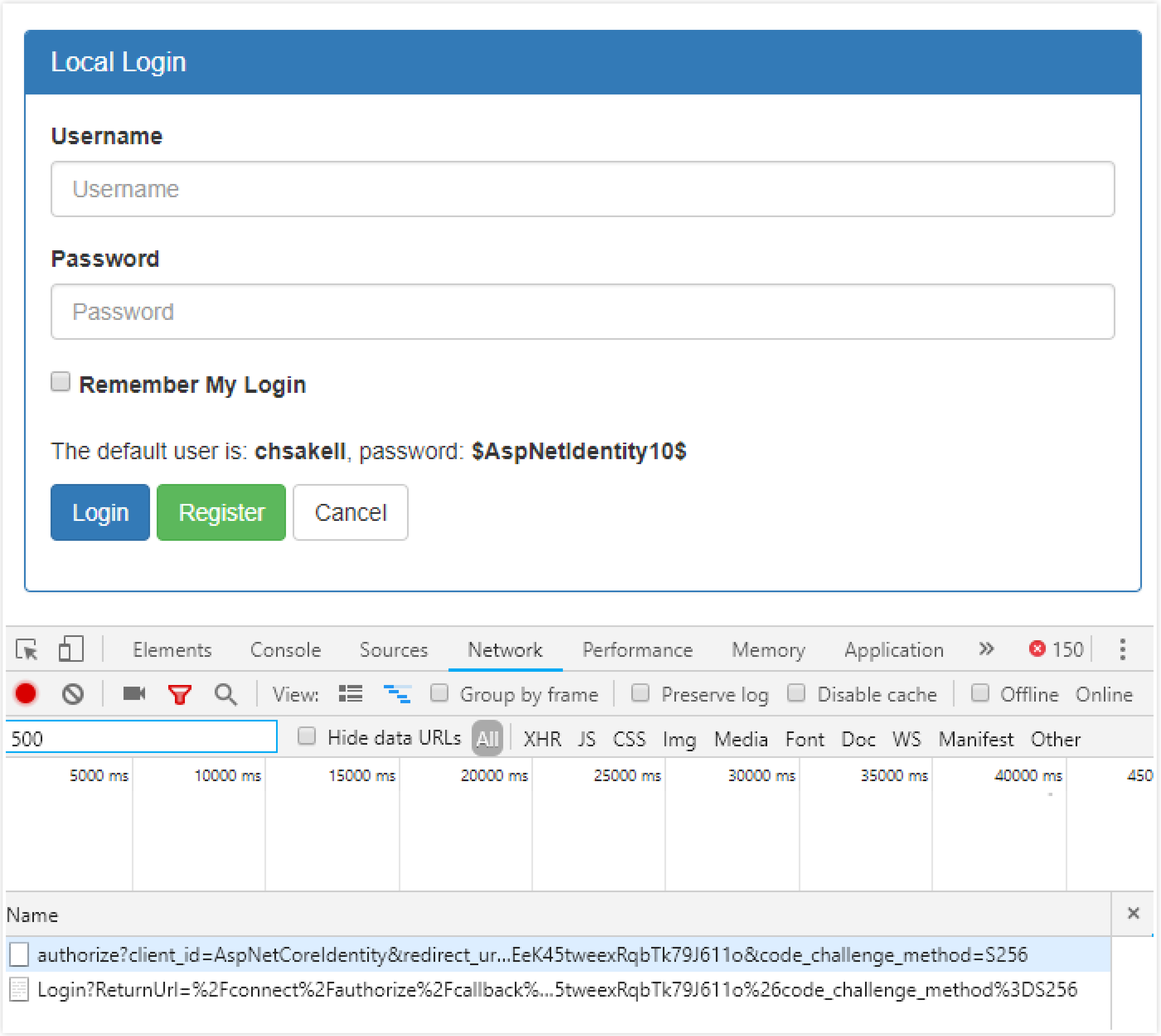
Click the login button and see what happens. The first network request is the **authorization request** to the authorization endpoint:

```
http://localhost:5005/connect/authorize?
    client_id=AspNetCoreIdentity&
    redirect_uri=http://localhost:5000&
    response_type=code&
    scope=openid profile SocialAPI&
    state=be1916720a2e4585998ae504d43a3c7c&
    code_challenge=pxUY7Dldu3UtT1BM4YGNLEeK45tweexRqbTk79J611o&
    code_challenge_method=S256
```

You need to be logged in to access this endpoint and thus you are being redirected to login with your Social Network account.

Use the default user credentials created for you **chsakell** – **$AspNetIdentity10$** and press login. After a successful login and only if you haven't already grant access to the *AspNetCoreIdentity* client you will be directed to the **Consent** page.

There are two sections for granting access, one for your personal information which asked because of the **openid** and **profile** OpenID Connect scopes and another one coming from the **Social.API** scope. Grant access to all of them to continue. After granting access you will be directed to the initial request to the authorization endpoint. IdentityServer created a **code** for you and directed the user-agent back to the client's redirection URI by appending the code in the fragment.



Local Login

Username

Password

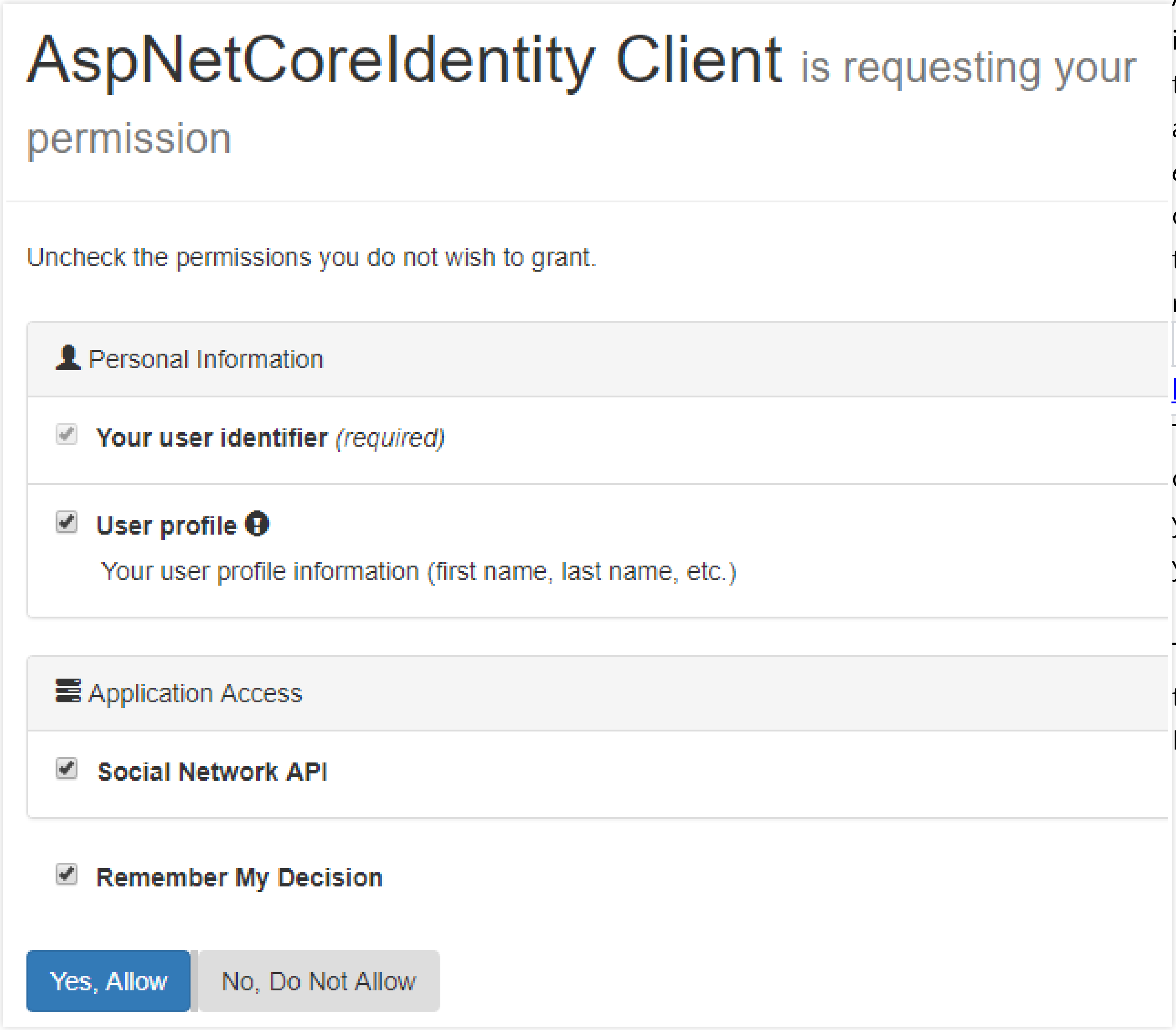
Remember My Login

The default user is: **chsakell**, password: **$AspNetIdentity10$**

authorize?client_id=AspNetCoreIdentity&redirect_ur...EeK45tweexRqbTk79J611o&code_challenge_method=S256

Login?ReturnUrl=%2Fconnect%2Fauthorize%2Fcallback%...5tweexRqbTk79J611o%26code_challenge_method%3DS256

```
http://localhost:5000/?
    code=090c6f68783c5b5fc267073990417c82ebfa01c1b70bc6107002ab0ae919dd8a
    &scope=openid profile SocialAPI&state=be1916720a2e4585998ae504d43a3c7c
    &session_state=7wBKoHgC7ld3_oO9e9wx-v_BfUa_mz9y6YDfwLKBhIQ.d0c4ee7f77d5da232806e05613067
```



AspNetCoreIdentity Client is requesting your permission

Uncheck the permissions you do not wish to grant.

Personal Information

✓ **Your user identifier** *(required)*

✓ **User profile**

Your user profile information (first name, last name, etc.)

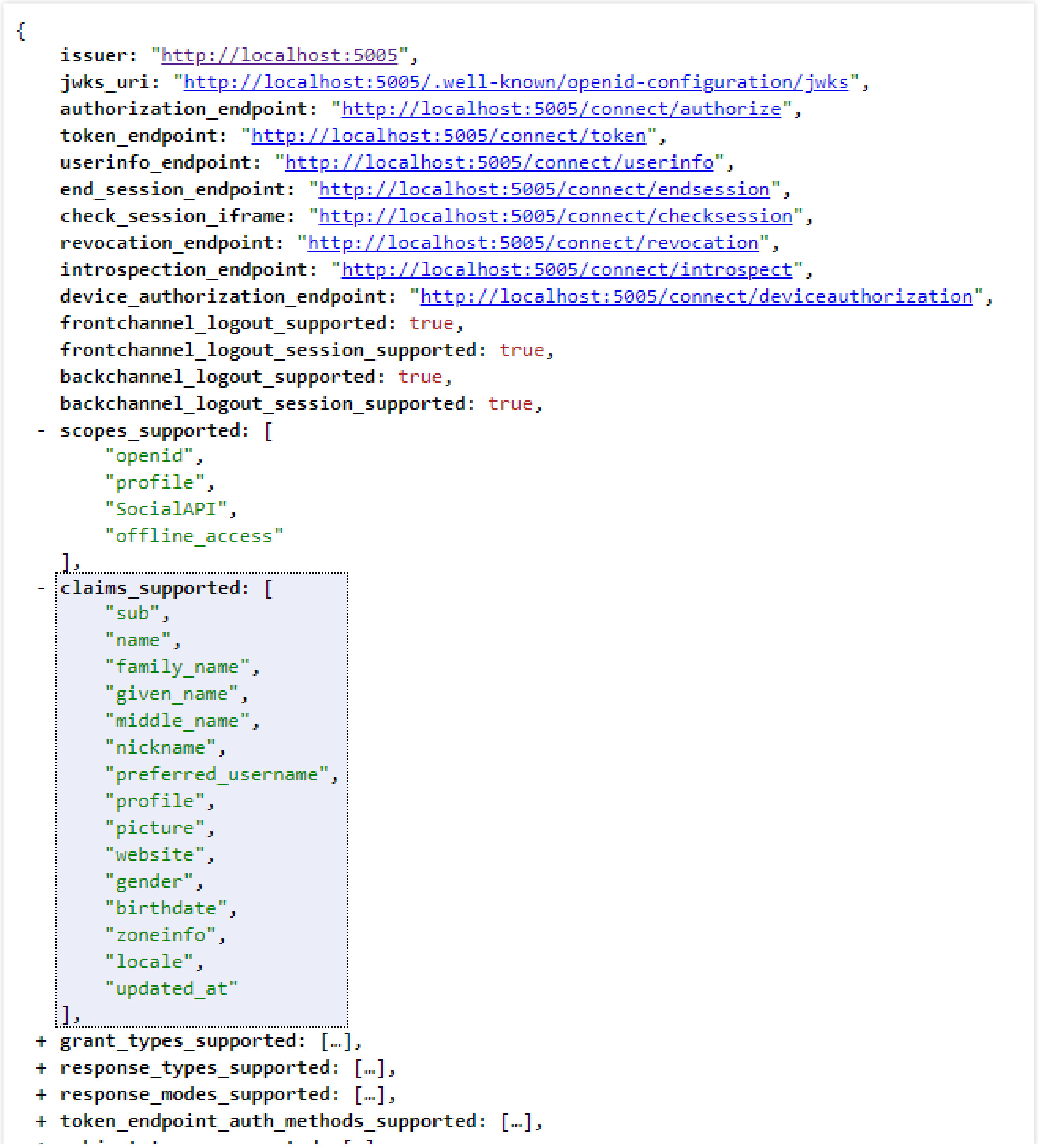
Application Access

✓ **Social Network API**

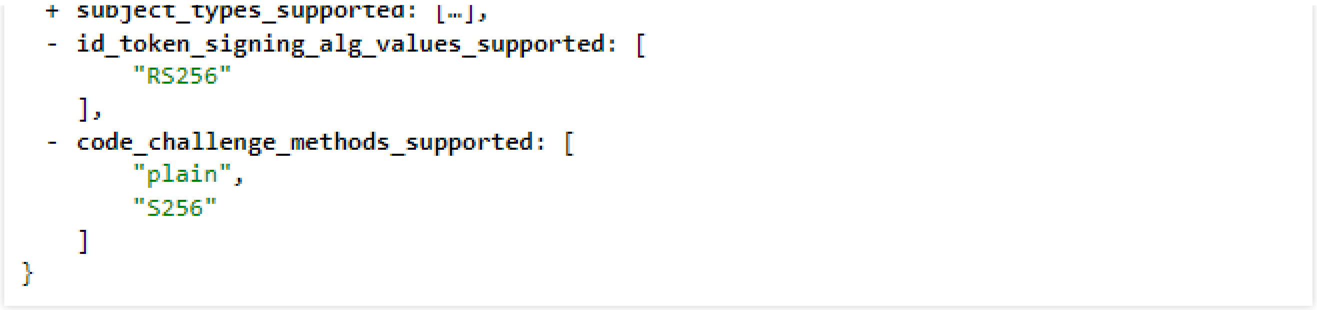
✓ **Remember My Decision**

As we described the next step in the *authorization code* flow is to use this code and request for an access token from the *token endpoint*. The client though doesn't know exactly where that endpoint resides so it makes a request to the http://localhost:5005/.well-known/openid-configuration. This is an IdentityServer's configuration endpoint where you can find information about your Identity Provider setup.

The client reads the URI for the token endpoint and sends a POST the request:



```
{
    issuer: "http://localhost:5005",
    jwks_uri: "http://localhost:5005/.well-known/openid-configuration/jwks",
    authorization_endpoint: "http://localhost:5005/connect/authorize",
    token_endpoint: "http://localhost:5005/connect/token",
    userinfo_endpoint: "http://localhost:5005/connect/userinfo",
    end_session_endpoint: "http://localhost:5005/connect/endsession",
    check_session_iframe: "http://localhost:5005/connect/checksession",
    revocation_endpoint: "http://localhost:5005/connect/revocation",
    introspection_endpoint: "http://localhost:5005/connect/introspect",
    device_authorization_endpoint: "http://localhost:5005/connect/deviceauthorization",
    frontchannel_logout_supported: true,
    frontchannel_logout_session_supported: true,
    backchannel_logout_supported: true,
    backchannel_logout_session_supported: true,
  - scopes_supported: [
        "openid",
        "profile",
        "SocialAPI",
        "offline_access"
    ],
  - claims_supported: [
        "sub",
        "name",
        "family_name",
        "given_name",
        "middle_name",
        "nickname",
        "preferred_username",
        "profile",
        "picture",
        "website",
        "gender",
        "birthdate",
        "zoneinfo",
        "locale",
        "updated_at"
    ],
  + grant_types_supported: […],
  + response_types_supported: […],
  + response_modes_supported: […],
  + token_endpoint_auth_methods_supported: […],
```

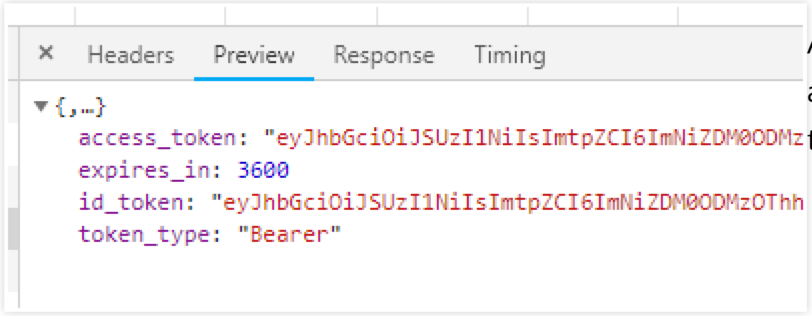


```
  + subject_types_supported: […],
  - id_token_signing_alg_values_supported: [
        "RS256"
    ],
  - code_challenge_methods_supported: [
        "plain",
        "S256"
    ]
}
```

```
Request URL: http://localhost:5005/connect/token
Request Method: POST

client_id: AspNetCoreIdentity
code: 090c6f68783c5b5fc267073990417c82ebfa01c1b70bc6107002ab0ae919dd8a
redirect_uri: http://localhost:5000
code_verifier: ad55ea0f077249ac99e190f576babb7bb9d14dcb229f4c1bb2fe1d0f87dc93d601374a833e464
grant_type: authorization_code
```

Identity provider returns both an **access_token** and a **id_token**

```
{
    "id_token":"<value-stripped-for-displaying-purposes>",
    "access_token":"<value-stripped-for-displaying-purposes>",
    "expires_in":3600,
    "token_type":"Bearer"
 }
```



```
▼{,…}
   access_token: "eyJhbGciOiJSUzI1NiIsImtpZCI6ImNiZDM0ODMz
   expires_in: 3600
   id_token: "eyJhbGciOiJSUzI1NiIsImtpZCI6ImNiZDM0ODMzOThh
   token_type: "Bearer"
```

Are you curious to find out what those JWT token say? Copy them and paste to jwt.io debugger. Here's the header and payload for the access token.

```
access token
```

```
// HEADER
{
    "alg": "RS256",
    "kid": "cbd3483398a40cf777e490cd2244deb3",
    "typ": "JWT"
}

// PAYLOAD
{
    "nbf": 1552313271,
    "exp": 1552316871,
    "iss": "http://localhost:5005",
    "aud": [
      "http://localhost:5005/resources",
      "SocialAPI"
    ],
    "client_id": "AspNetCoreIdentity",
    "sub": "09277cac-422d-43ee-b099-f99ff76bceda",
    "auth_time": 1552312960,
    "idp": "local",
    "scope": [
      "openid",
      "profile",
      "SocialAPI"
    ],
    "amr": [
      "pwd"
    ]
}
```

id token

```
// HEADER
{
    "alg": "RS256",
    "kid": "cbd3483398a40cf777e490cd2244deb3",
    "typ": "JWT"
}

// PAYLOAD
{
    "nbf": 1552313271,
    "exp": 1552313571,
    "iss": "http://localhost:5005",
    "aud": "AspNetCoreIdentity",
    "iat": 1552313271,
    "at_hash": "AM-fvLMnrmHCFu9nGDmY3Q",
    "sid": "aa8df27adf631604d855533b67c307ea",
    "sub": "09277cac-422d-43ee-b099-f99ff76bceda",
    "auth_time": 1552312960,
    "idp": "local",
    "amr": [
      "pwd"
    ]
  }
```

What's interesting is that the **id token** doesn't contain the claims that belongs to the *profile* scope asked in the authorization request and this is of course the expected behavior. By default you will find a **sub** claim which matches the user's id and some other information about the authentication event occurred. As described in the theory, the client in this flow uses the access token and sends an extra request to the **UserInfo** and point to get the user's claims.

```
Request URL: http://localhost:5005/connect/userinfo
Request Method: GET

Authorization: Bearer <access-token>
```

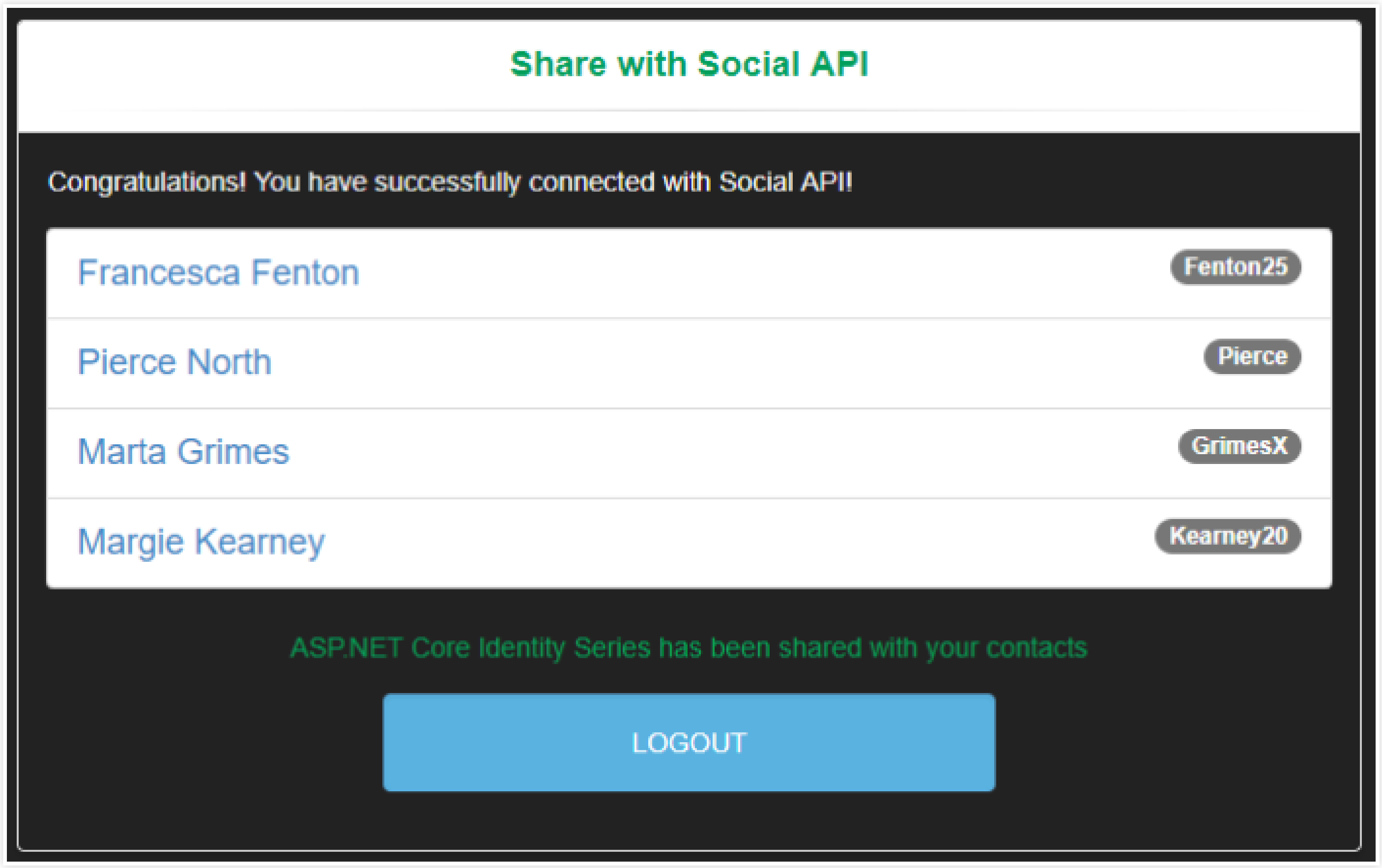
And here's the response..

```
{
    "sub":"09277cac-422d-43ee-b099-f99ff76bceda",
    "name":"Chris Sakellarios",
    "given_name":"Christos",
    "family_name":"Sakellarios",
    "website":"https://chsakell.com",
    "preferred_username":"chsakell"
 }
```

Let me remind you that we have added a claim for address for this user but we don't see it on the response since *address* doesn't belong to the *profile* scope nor is supported by our IdentityServer's configuration. Last but not least you will see the request to the SocialNetwork.API protected resource.

```
Request URL: http://localhost:5010/api/contacts
Request Method: GET

Accept: application/json, text/plain, */*
Authorization: Bearer </access-token>
```

If all work as intended you will see the following view.


Share with Social API

Congratulations! You have successfully connected with Social API!

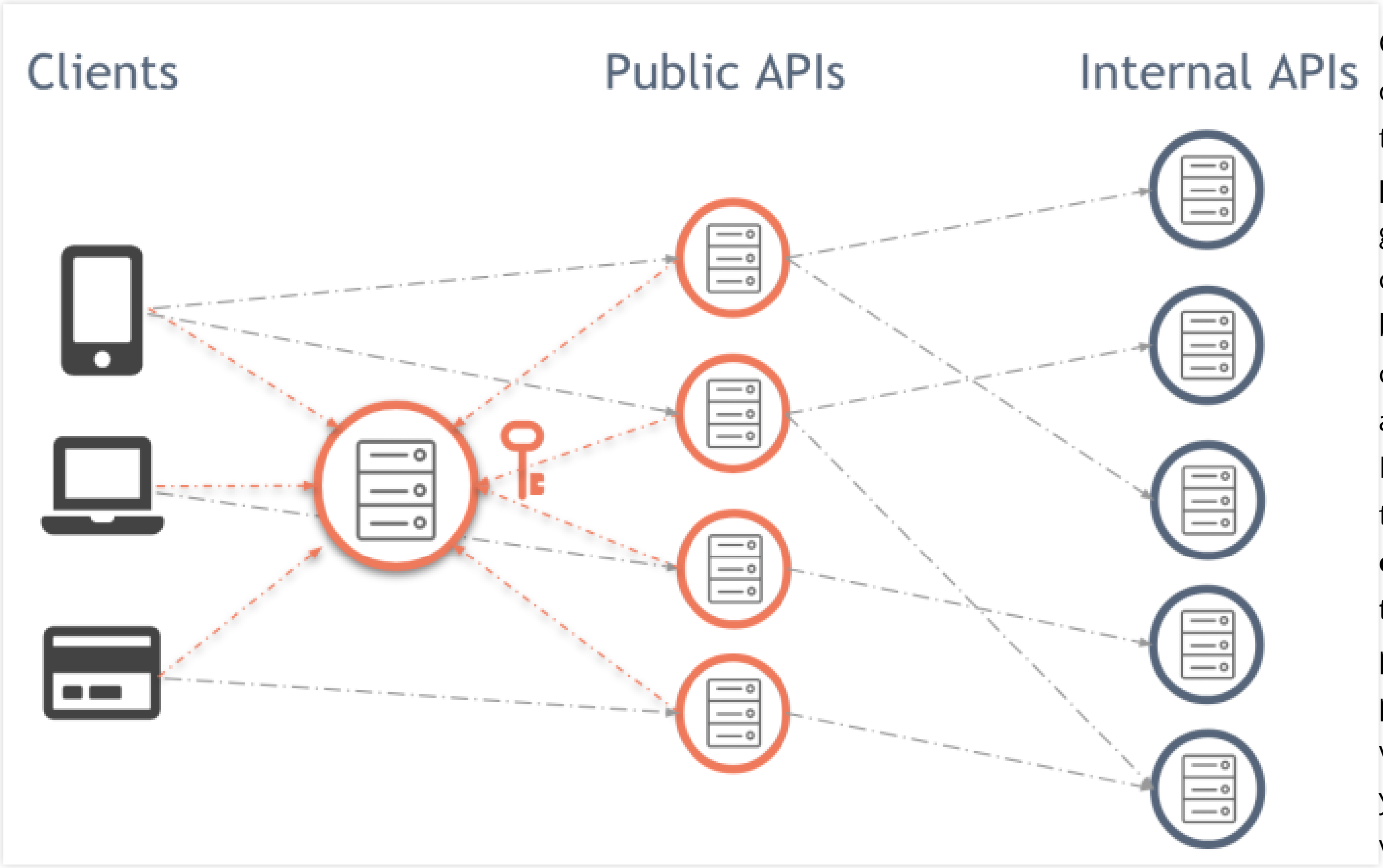
Francesca Fenton Fenton25
Pierce North Pierce
Marta Grimes GrimesX
Margie Kearney Kearney20

ASP.NET Core Identity Series has been shared with your contacts

Discussion

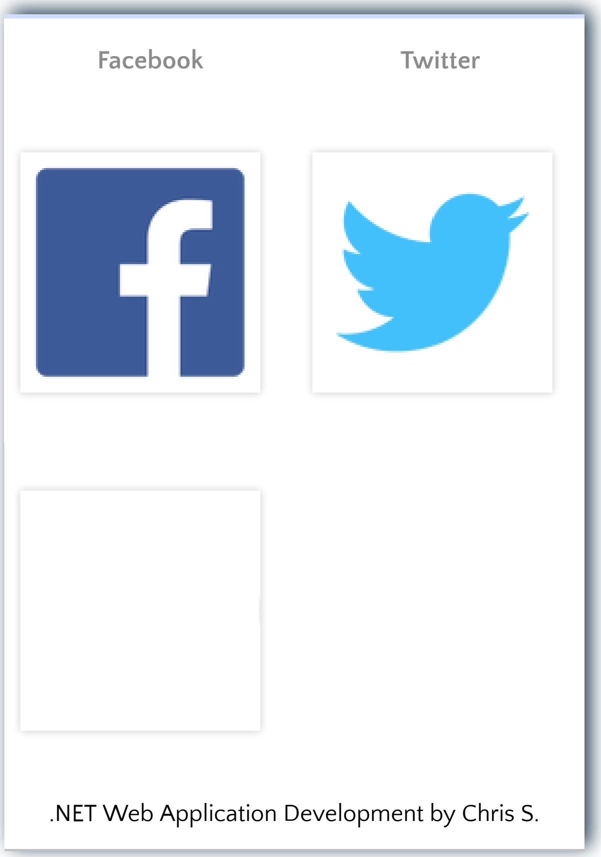
I believe that's more than enough for a single post so we 'll stop here. The idea was to understand the basic concepts of OAuth 2.0 and OpenID Connect so that you are aware what's going on when you use IdentityServer to secure your applications. No one expects from you to know by the book all the protocol specifications but now that you have seen a complete flow in action you will be able to handle any similar case for your projects. Any time you need to implement a flow,

read the specs and make the appropriate changes in your apps.
Now take a step back and think outside of the box. What does OAuth 2.0, OpenID Connect and IdentityServer provide us eventually? If you have a single web app, *(server side or not, it doesn't matter..)* and the only thing required is a simple sign in, then all these you 've learnt might not be a good fit for you. On the other hand, in case you *go big* and find yourself having a bunch of different clients, accessing different APIs which in turn accessing other internal APIs *(micro-services)* then you must be smart and **act big** as well. Instead of implementing a different authentication method for each type of your clients or reinventing the wheal to support limited access, use **IdentityServer**.



Orange arrows describe getting access tokens for accessing **protected APIs** while gray arrows illustrate communication between different components in the architecture. The IdentityServer will play the role of the **centralized security token service** which provides limited access per client type. This is where you define all of your clients and the way they are authorized via flows while each client requires a minimum configuration to start an authorization flow. Protected resources and APIs, regardless their type all they need is to handle bearer tokens and that's all.

In case you find my blog's content interesting, register your email to receive notifications of new posts and follow *chsakell's Blog* on its Facebook or Twitter accounts.



23 replies

Bipin Paul (@iAmBipinPaul)

March 11, 2019 · 7:40 pm

```
services.AddAuthentication("Bearer")
    .AddJwtBearer("Bearer", options =>
    {
        options.Authority = "http://localhost:5005";
```

```
            options.RequireHttpsMetadata = false;

            options.Audience = "SocialAPI";
        });
```

Here we are not providing secret key how it is being validated?

Christos S.

March 11, 2019 · 8:05 pm

Hello Paul,

Very good question. There are two types of access tokens, **reference tokens** and **self-contained** tokens which is our case because we use **JWT**.
If we used reference tokens, indeed the SocialAPI *(the resource server)* would have to validate the access token by sending an extra introspection request to the **introspection endpoint**, a request that requires the API's secret.
JWT on the other hand, are self-contained and **validated locally** by checking the signature, the expected issuer name *(authority)* and the expected audience.
I hope I helped.

Joel Palmer

April 26, 2019 · 10:34 pm

Thanks for this. You've helped me move forward. However, it is unclear where in your code you'll call the **InitializeIdentityServer** method and what the parameter value is. Can you help me understand this?

Thanks.

Christos S.

May 4, 2019 · 9:58 am

Hi Joel,

In the `Program.cs` file we call the `DatabaseInitializer.Init` method:

```
using (var scope = host.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var config = services.GetService<IConfiguration>(); // the key/fix!
    var useInMemoryStores = config.GetValue<bool>("UseInMemoryStores");
    DatabaseInitializer.Init(services, useInMemoryStores);
}
```

… which in turn calls the `InitializeIdentityServer` method:

```
public static void Init(IServiceProvider provider, bool useInMemoryStores)
{
    if (!useInMemoryStores)
    {
        provider.GetRequiredService<ApplicationDbContext>().Database.Migrate();
        provider.GetRequiredService<PersistedGrantDbContext>().Database.Migrate();
        provider.GetRequiredService<ConfigurationDbContext>().Database.Migrate();
    }
    InitializeIdentityServer(provider);
```

The **UseInMemoryStores** parameter comes from the *appsettings.json* file and if it's equal to **false** means that we use a database, so we need to ensure all migrations have been applied

---

Típ Đi Nắng

April 28, 2019 · 5:44 pm

```
Update-Database -Context ApplicationDbContext
```

Could not load assembly 'IdentityServer'. Ensure it is referenced by the startup project 'AspNetCoreIdentity'.
help me? thanks.

> **Christos S.**
>
> May 4, 2019 · 9:46 am
>
> This means that you have the wrong project selected from the drop-down list, in case you run the command through **Package Manager Console**.
> You can set the project by adding the -Project parameter as follow:
>
> ```
> Update-Database -Context ApplicationDbContext -Project "IdentityServer"
> ```
>
> By the way, I have pushed some changes that configure the migration assembly properly. Just in case, pull the latest changes, remove and re-run the migrations as described in the **Identity Server instructions**

Steven Carleton

May 9, 2019 · 5:24 pm

Excellent work! Brock and Dominick should put a link to your blog on the IS site. Anyone considering using IS should read your blog FIRST!

Have you considered adding claims for things like roles or apps/apis? We would like to setup our IS to support user roles and application/api authorization. So a given user may only be allowed a subset of protected apps/apis and would have a single assigned role. Authorization would be checked using the standard ASP.NET [Authorize] attribute. We have the usual web and api servers seen in the IS examples. This was fairly easy in the classic ASP.NET Identity framework using Owin/Kitana, but we would like to use ASP.NET Core Identity and IS4

Sacha

June 12, 2019 · 4:57 pm

Thanks for this great article! Do you have any pointers regarding SaaS (multi-tenancy) SPA application?

Shibu

June 16, 2019 · 7:44 pm

How can we integrate identityserver and api both in the same project(port)?

> **haipk**
>
> June 20, 2019 · 7:40 am
>
> You can use API Gateway (Ocelot) to use on one port for 2 projects.

Marian

August 2, 2019 · 6:05 pm

That's an amazing article!
Really helped to me to begin to understand the concept
Thanks a lot!

---

olek

October 14, 2019 · 6:48 am

Hi,thanks for your tutorial,it is really wonderful.
Here i have an error,i create the IdeneityServerDb database follow your instructions(**https://github.com/chsakell/aspnet-core-identity/blob/master/IdentityServer/Data/instructions.md**) successfully. Then i configured the UseInMemoryStores to false, and i meet errror "HTTP Error 500.30 – ANCM In-Process Start Failure" when i run the IdentityServer project. It can work when UseInMemoryStores set to true.
So is there any wrong i did here? Looking forward to your response,thanks.

> **Christos S.**
>
> **October 15, 2019 · 8:16 pm**
>
> Hi, make sure to set *UseInMemoryStores* to false before running the migrations and the database update commands.
> I have just tried them again and they all work fine. Mind that the solution *(master branch)* has been updated to .NET Core 3 and Angular 8.
> I would suggest you start from scratch and run the **instructions** again. I usually run them through the Package Manager Console.
>
> > **olek**
> >
> > **October 16, 2019 · 6:52 am**
> >
> > It worked,thanks a lot.

---

olek

October 14, 2019 · 7:02 am

By the way, are there any sso projects or tutorials on using IdentityServer4? Thanks.

Christos S.

October 15, 2019 · 8:17 pm

I will try to create an SSO related post as soon as possible.

olek

October 16, 2019 · 6:57 am

I'm looking forward to it.You are an amazing guy.I learned a lot from you.

Clinton B

October 21, 2019 · 10:34 pm

Very nice post, I have an scenario where I am stuck,Any help will be much appreciated.

I have an Asp.Net Core 3.0 based SPA application (Angular) with Identity Server configured and is working fine especially login, refresh token, logout etc.

The login function of my angular app access the /connect/token endpoint with resource owner password flow and obtain the id_token, refresh_token, claims etc. Now I need to implement captcha on the login page of angular app where I do not have any action method / Authorize controller where i could validate the captcha before authenticating the user.

Is there a way to override or intercept the connect/token method in the asp.net core before authentication ?

Kamran Shahid

December 10, 2019 · 9:12 am

Nice article. Love to see if you can update the post with .net core 3.1 where ever it needs

---

Domingo Vazquez

December 11, 2019 · 9:08 pm

Hi I am trying to implement this example with Ocelot API Gateway but I have always 401. I am missing something? There are a lot of examples but only with JWT or IdentityServer ... Thank you.

---

kattouchthis

December 25, 2019 · 7:47 pm

Hi Christos – thanks very much for the great series. It has taken me from near ground zero to almost reaching my goal. I am trying to implement SSO with ID4 and ASP.Net Core Identity as the authentication back end. However, there is very little info on implementing SSO with ID4 – do you have any suggestions? Thanks – Don

---

Hafiz

March 8, 2020 · 3:20 am

Great article. It explained lots of thing in details. Thanks putting this out there.

Let's say you have another endpoint on your socialNetwork.API like /post new contacts and based on the clients ROLE you want to guard the /get and /post endpoints.

Where should be that role info be tied to?
Should it be part of the identity Server?
Or should reside in Social network.API application?

Thanks in advance

JinLong

May 13, 2020 · 12:31 pm

please help me. Thanks
after i got token and call to https://localhost:44325/connect/userinfo, it returned this error

```
System.InvalidOperationException: Sequence contains more than one matching element
at System.Linq.ThrowHelper.ThrowMoreThanOneMatchException()
at System.Linq.Enumerable.SingleOrDefault[TSource](IEnumerable`1 source, Func`2 predicate)
at IdentityServer4.ResponseHandling.UserInfoResponseGenerator.ProcessAsync(UserInfoRequestValidationResult validationResult)
at IdentityServer4.Endpoints.UserInfoEndpoint.ProcessUserInfoRequestAsync(HttpContext context)
at IdentityServer4.Endpoints.UserInfoEndpoint.ProcessAsync(HttpContext context)
at IdentityServer4.Hosting.IdentityServerMiddleware.Invoke(HttpContext context, IEndpointRouter router, IUserSession session, IEventService events)
at IdentityServer4.Hosting.IdentityServerMiddleware.Invoke(HttpContext context, IEndpointRouter router, IUserSession session, IEventService events)
at IdentityServer4.Hosting.MutualTlsTokenEndpointMiddleware.Invoke(HttpContext context, IAuthenticationSchemeProvider schemes)
at Microsoft.AspNetCore.Authentication.AuthenticationMiddleware.Invoke(HttpContext context)
at IdentityServer4.Hosting.BaseUrlMiddleware.Invoke(HttpContext context)
at Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware.Invoke(HttpContext context)
```