**IdentityServer and OpenID Connect**

For a typical client application nowadays you want to be able to **log in** and have something for **resource authorization** (typical web application talks to an API).

Need two kinds of tokens: **identity** token (for authentication – log in to your application) and **access** token (authorization – secure access to your API)

**Identity token –** contains information about who you are (the user)

**Access token –** allow the application to access resources on your behalf so it will contain “scopes” that allow or disallow access to an API

**IdentityServer** is the middleware that we use to generate these tokens and handle the tasks relating to the tokens

Depending on the type of application, you have to choose what kind of flow you’re going to use

**Flow** is the means to achieve those tokens. (Authorize, Implicit, Hybrid)

**Refresh token** can be used to exchange for an access token

**Tokens are stored** in either cookies, local storage or session storage

Grab the token from a URI, save it, and pass it on each call to the API as a bearer token

**Two types of clients:**

**Confidential client –** Capable of maintaining confidentiality of their credentials (“secrets”)

(Example: server-sided web apps)

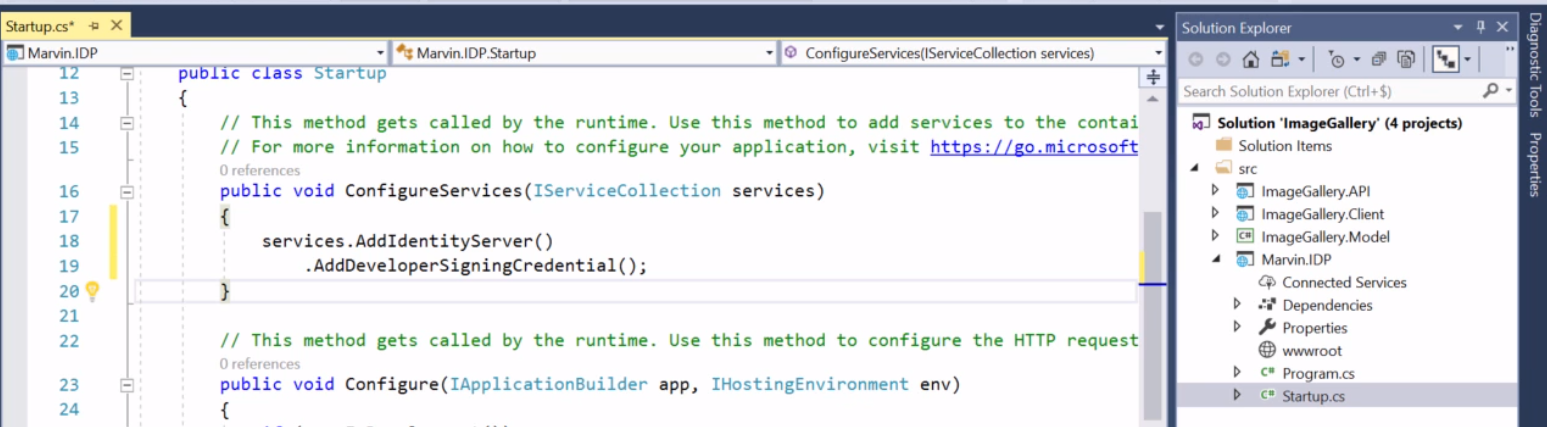
**Public clients – NOT** capable of maintaining confidentiality of their credentials can’t keep “secrets”)

(Example: JavaScript apps, mobile apps)

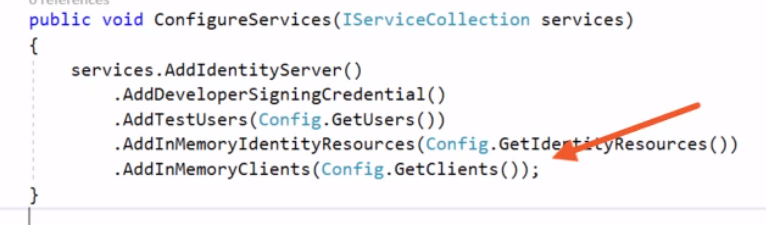
Setting up IdentityServer4

* “IdentityServer4” from npm

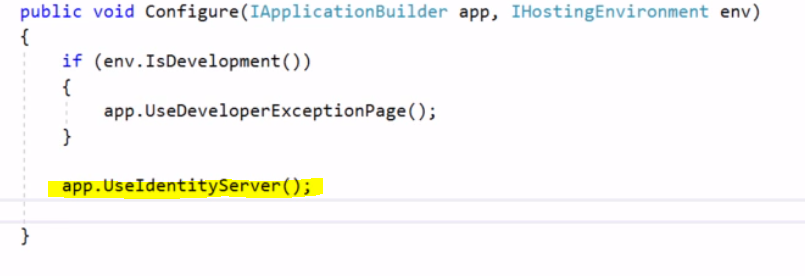
1. **Register Identity Server**



1. **Add (Test) Users, Resources and Clients**
2. Located in **Config.cs** of IDP
3. Create **GetUsers()** method to retrieve list of Test Users
4. Set up **scopes** (GetIdentityResources() method)
5. **Back to Startup.cs**

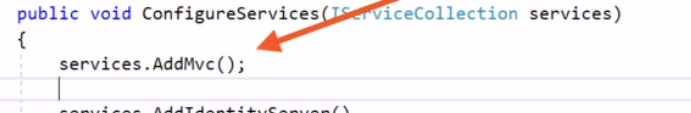


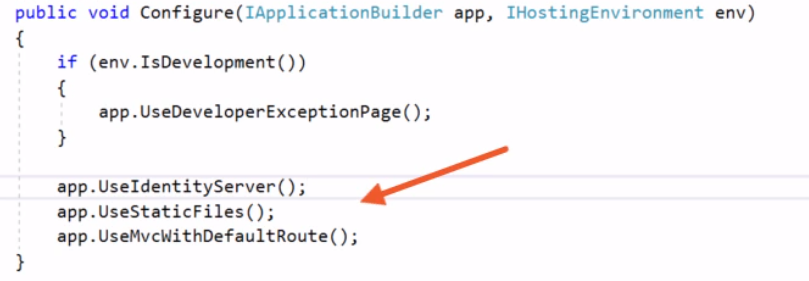
1. Add Identity Server to the request pipeline:



Adding a User Interface

1. **Assume you have an MVC ASP.Net client application already.**
2. In **Startup.cs** in the IDP, Add MVC

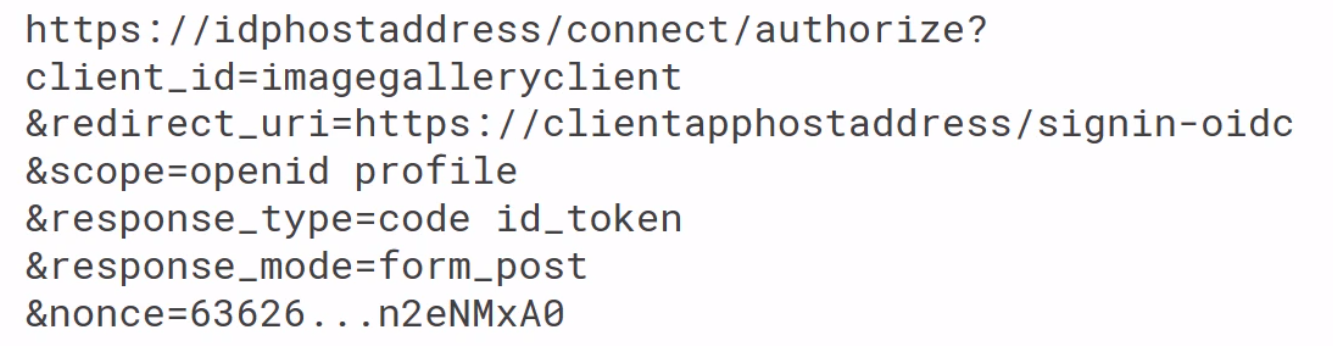


1. 

Ensuring Traffic is encrypted

1. Go to IDP project, enable SSL.
2. Copy and paste the “https://...” to the “App URL”
3. (If necessary) Add localhost certificate

The Hybrid Flow



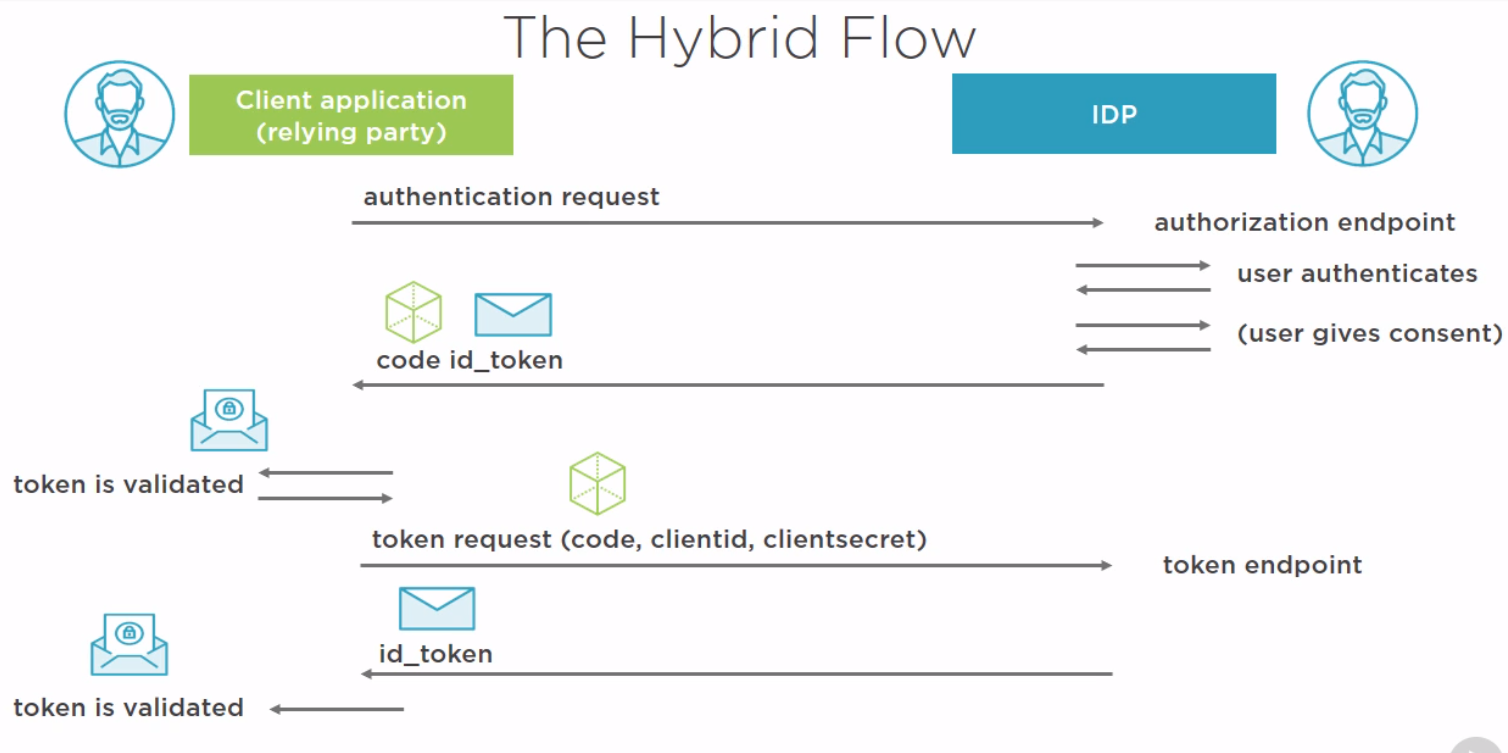
**authorize?:** the flow starts with an authentication request to the **authorization endpoint**

**client\_id:** identifier of the client

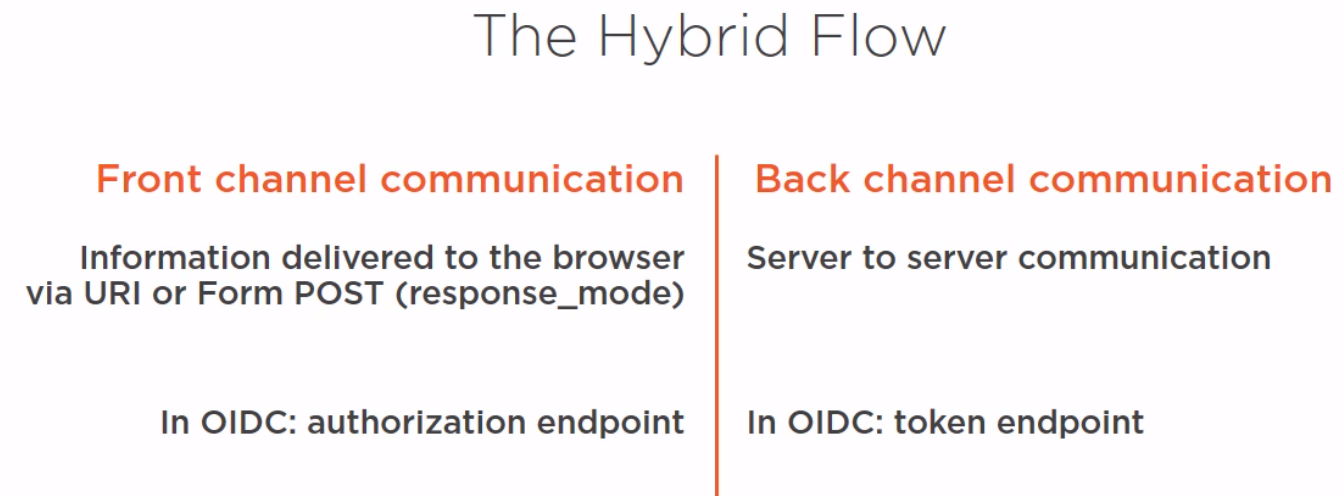
**redirect\_uri: Redirection endpoint** at client level

**scope:** requested scopes by the client application (‘openid’ – wants access to the user’s identifier; ‘profile’ – wants access to the user’s claims like family\_name, given\_name)

**response\_type:** tells us the type of flow we are using (code id\_token implies we’re using Hybrid)



* **Token request -> token endpoint is back channel**
* In the token endpoint, access and refresh tokens are also returned, so it’s not just a matter of returning a second id\_token
* Id\_token is able to be used to make a **ClaimsIdentity,** and be used to sign into the ASP.Net MVC application



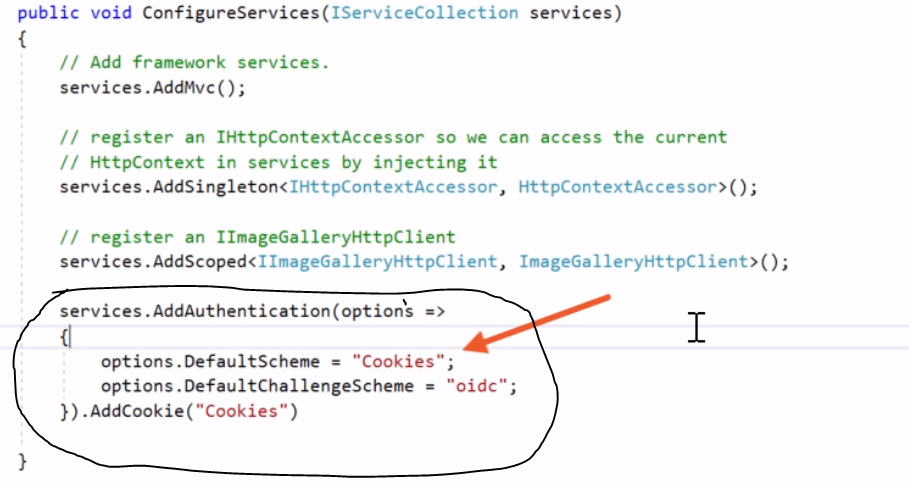
Configure IdentityServer to Log In with the Hybrid Flow

1. Enable SSL on the **MVC client** also (Ensure Traffic is encrypted)
2. **Config.cs** in IDP

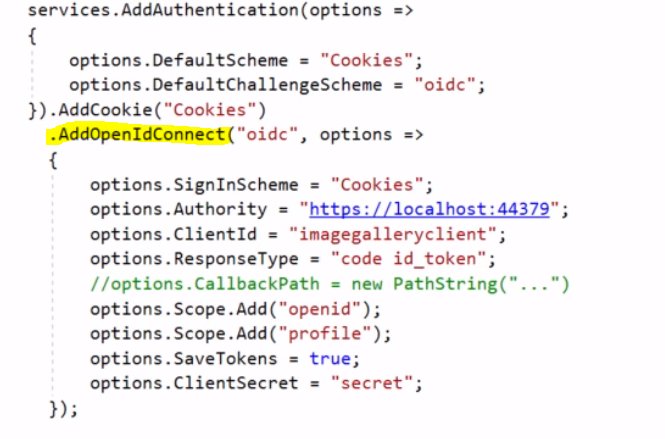


Logging in with the Hybrid Flow

1. **Startup.cs** in Client application – **configure the Authentication middleware**

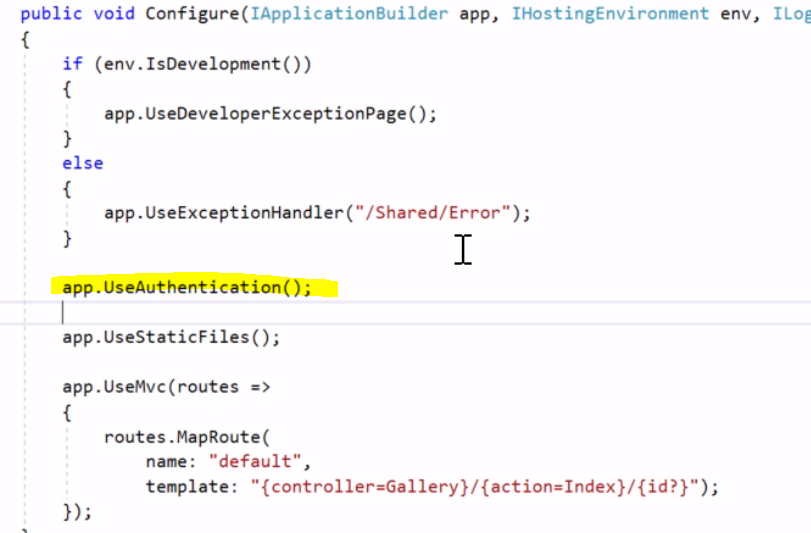


1. Register and configure the **OIDC handler** – handle creating authorization requests, token and other requests and handle identity validation

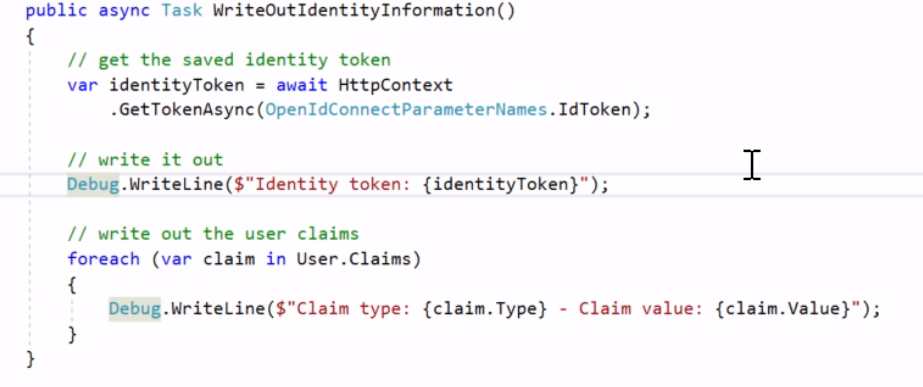


* **Authority** set to IDP url
* **ClientId** is from the specified client in the IDP
* **ResponseType** indicates *Hybrid flow*
* **‘openid’ and ‘profile’ scopes** to grab the user’s identity and also the claims related to the ‘profile’ scope such as given\_name, family\_name
* **SaveTokens –** save tokens from the IDP so we can use them afterwards
* **ClientSecret –** same name in the IDP level

1. UseAuthentication() inside **Configure** method



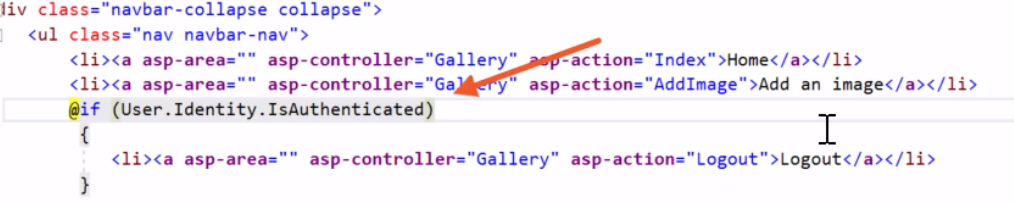
1. **GalleryController** at Client level – add the [Authorize] decorator to the Controller to make sure it is inaccessible to someone who isn’t authenticated
2. Write this function if you want to see the ID\_token and call it from Index()



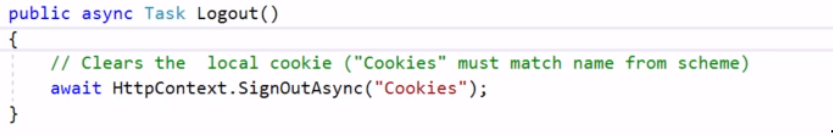
1. You can view token endpoint from the Debug window after signing in to the application to see **the claims returned, and the value of the id\_token**

Log out of our Application

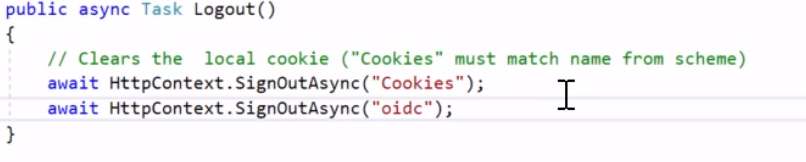
1. In \_Layout.cshtml in the Client level



1. **GalleryController** create Logout() Action



1. We are **logged out** of our application, but still **logged in** to the IDP

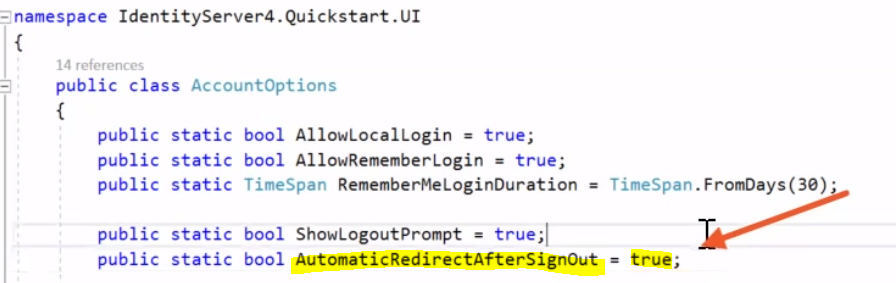


1. **Config.cs** in IDP level else you will be Invalid Post Logout URI – **Log out** needs a redirect URI just like **log in (**Logout = **End-session endpoint**)



Redirect after Logging out

1. At MVC app level, go to **AccountOptions.cs**



Return Additional Claims

* IdentityServer doesn’t include identity claims (except **sub)** in the identity token, **unless we specifically ask for this**
* Not including the claims in the id\_token keeps it small, which helps avoid URI length restrictions
* We can use the **UserInfo Endpoint** by the client application to request additional **user claims** 
  + Requires an **access** token with scopes related to the **claims** to have to be returned

1. **Startup.cs** in Client Level



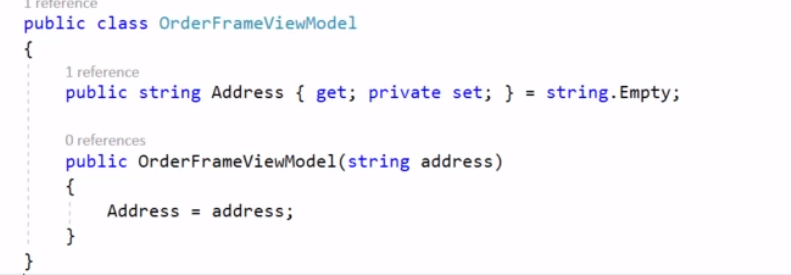
1. “given\_name”, and “family\_name” claim values should now output from the **token endpoint** in the Debug output window

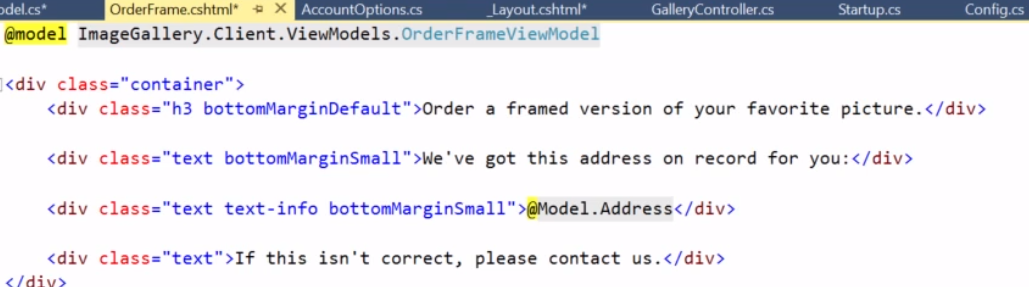
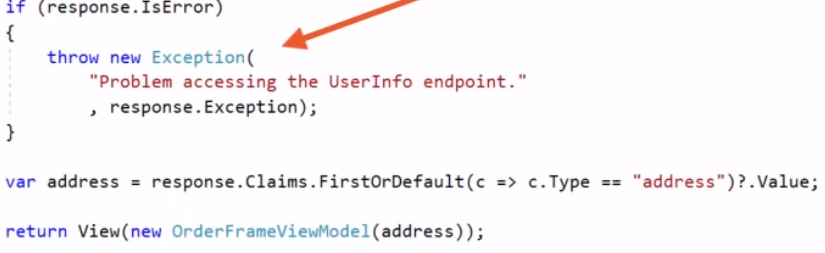
**Using the Hybrid flow…**

* Ensures the id\_token and authorization code are returned via the front channel
* Allows verifying the id\_token first
* **Front channel** communication goes via the browser
* **Back channel** communication is server to server **(token endpoint)**
* **ClaimsIdentity** is created from a validated id\_token (can be used to return claims by appending the claims to the id\_token)
* **HOWEVER,** claims should be returned from the **UserInfo** endpoint instead in order to avoid URI length issues
* When logging out, remember to log out of the IDP if required (call the **end-session endpoint)**

Calling the UserInfo Endpoint to Get More Claims

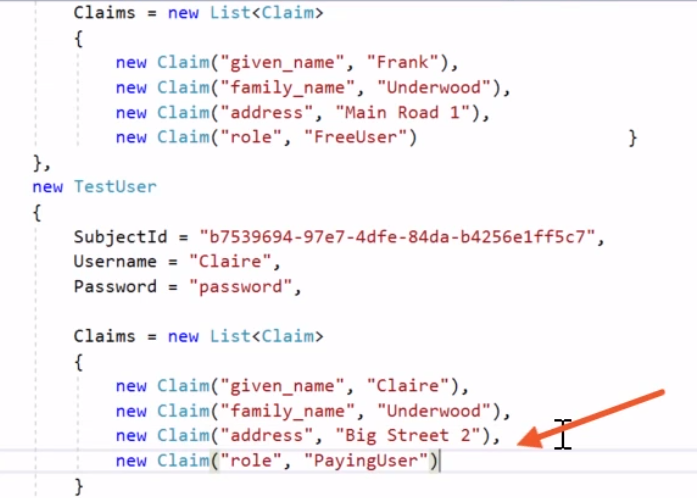
1. Create **OrderFrame.cshtml** in Client Views/Gallery
2. Create ViewModel/OrderFrameViewModel



1. 
2. Install **IdentityModel** from npm
3. **GalleryController ->** add OrderFrame() action
4. Need to get an **access token** in order to call the UserInfo endpoint in order to retrieve additional claims
5. 
6. 

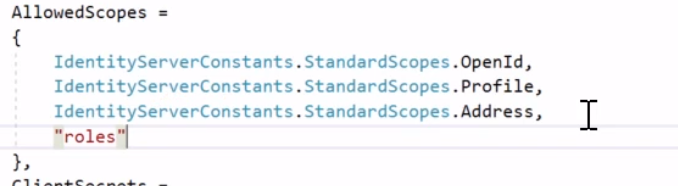
Role-Based Authorization

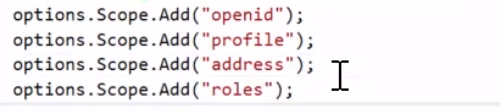
1. Add “role” claim for users in the **Config.cs** at IDP level

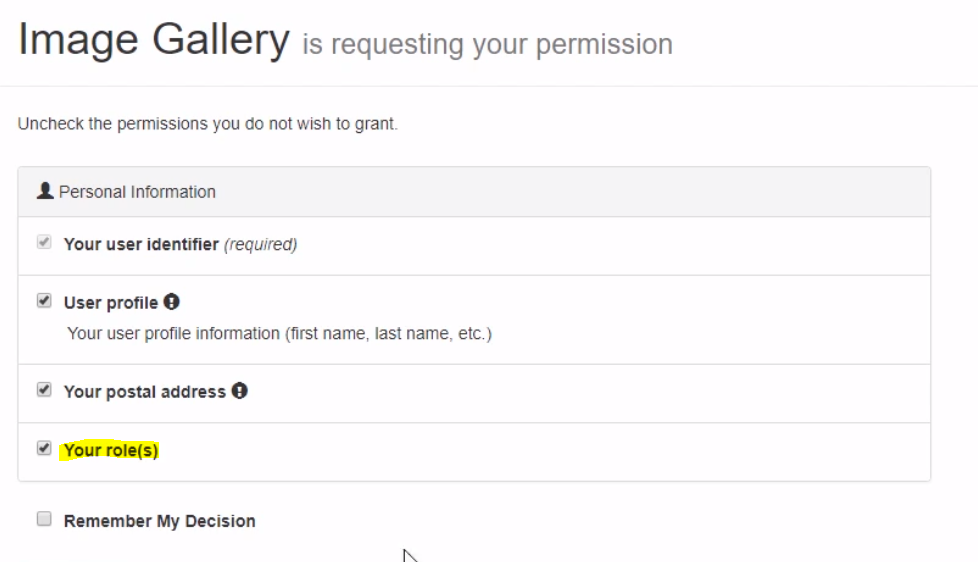


1. Create a new **identity scope** in Config.cs

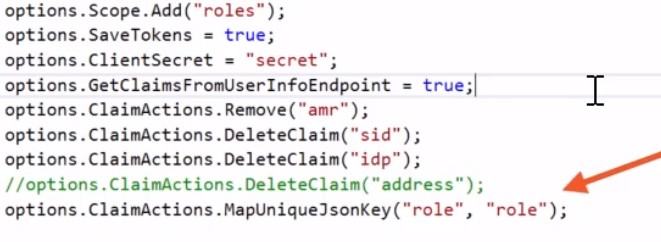


1. 
2. **Startup.cs** at client level





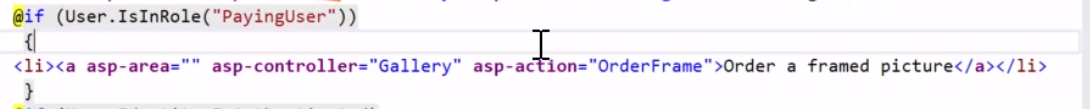
1. Need to add mapping for custom claims



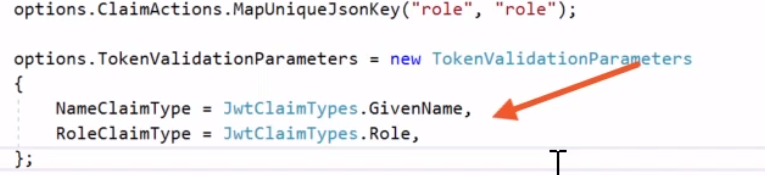
1. Role claim value should now appear in **Debug window**

Using Role-based Authorization in Views

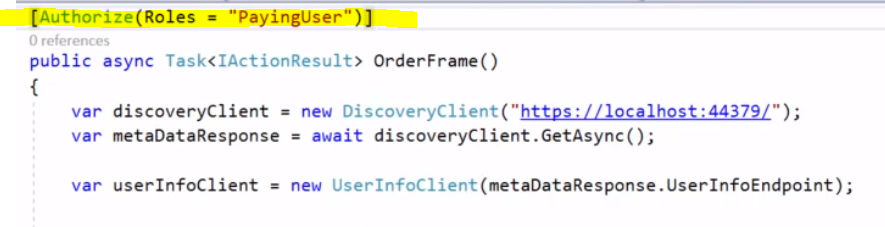
1. Change “Order a framed picture” to only a **PayingUser** option (\_Layout.cshtml in Client level)



1. Have to tell the framework where it can find the role
2. Go back to the Authentication middleware in **Startup.cs**



1. **This isn’t really secure yet** because you can still copy the URL for Order Frame and have access to that page if you log in with Frank (who isn’t a Paying User)
2. Decorate the OrderFrame() action with the [Authorize] and specify role



1. Now Frank will not have access to the page, however it will just crash the page if Frank tries to access it. We need a graceful solution: an **Access Denied** page

How OAuth2 works

* Client application sends authorization request to Authorization server
* Authorization server creates an access token and signs it
* Redirects user to the client application, passing in the access token
* Client application stores the access token and sends it as a bearer token on each request to the API
* Access token is validated by the API and resources are granted to the client application

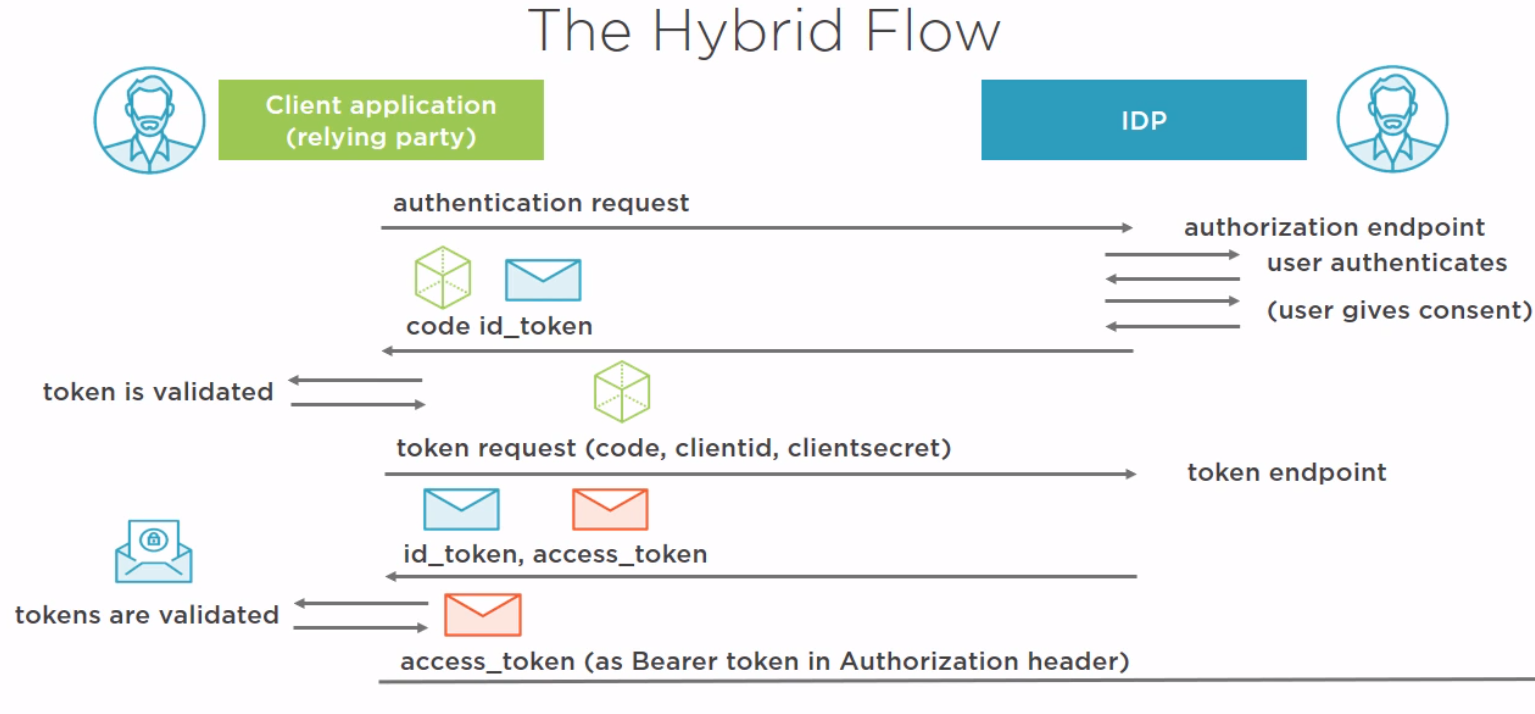
Why Open ID Connect is Preferred over OAuth2

* Identity token can be linked to the access token (at\_hash)
* Identity token can be verified at client level first, which is tighter security

Using OpenID Connect for Authentication and Authorization

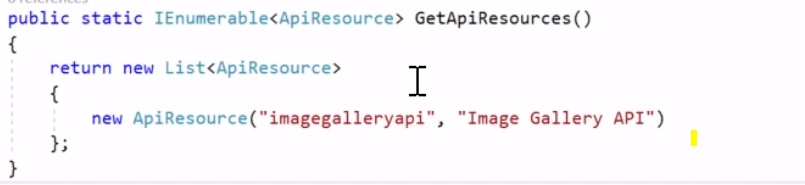
* Client application sends authorization request to IDP
* IDP creates an **ID Token AND** an **access token** and signs both
* Passed back to client; ID token is validated and used as proof of identity
* With this, we can sign into the client application, and the **access token is stored by the client application** in an **encrypted cookie**
* On each request to the API, it’s sent as a bearer token and is validated at the API level

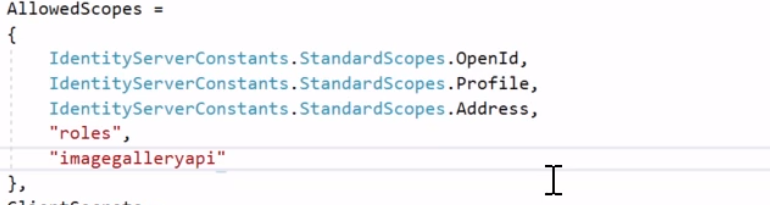
The Hybrid Flow (API)

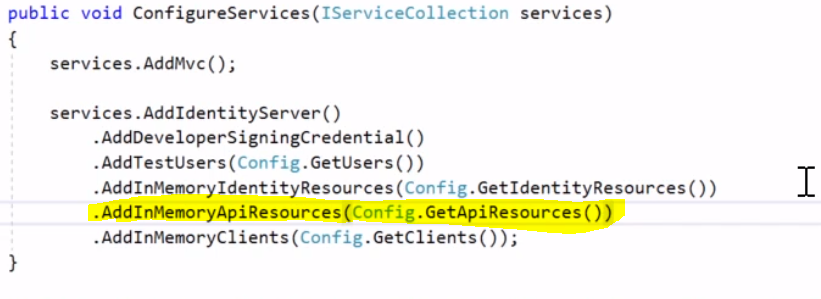
 API IS HERE

SECURING ACCESS TO OUR API

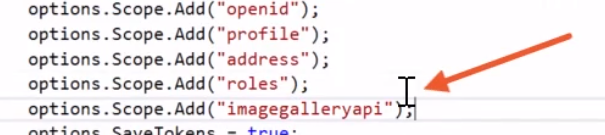
1. **Config.cs** at IDP level
2. We require an access token before we have access to the API
3. **Get API resource scopes**



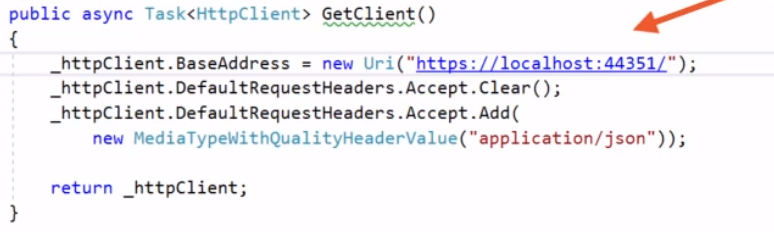
1. 
2. **Startup.cs** at IDP level – when configuring IdentityServer, we added In Memory identity resources. We need to add In Memory API resources now for the API



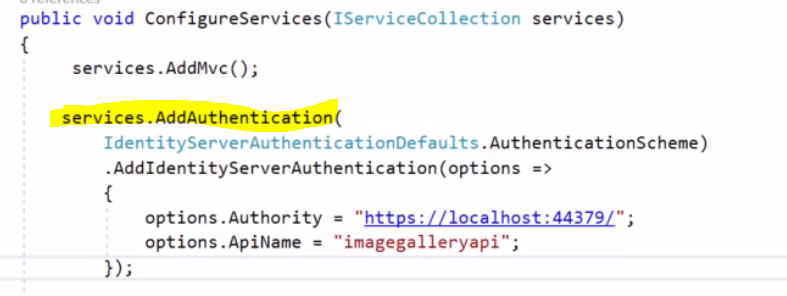
1. **Startup.cs** at Client level



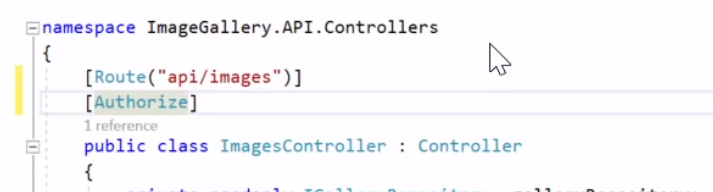
1. Set API to **enable SSL**
2. Change URL in Gallery/Index.cshtml and ImageGalleryHttpClient.cs



1. Install **AccessTokenValidation** from npm
2. **Startup.cs** at API level



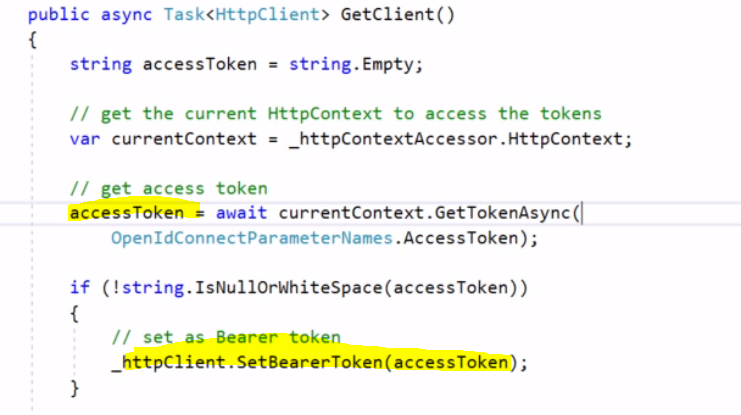
1.  **before** app.UseMvc() is called.
2. Put Authorization in ImagesController at API level



1. 
2. For now, blocked access to API

Passing Access Token to API

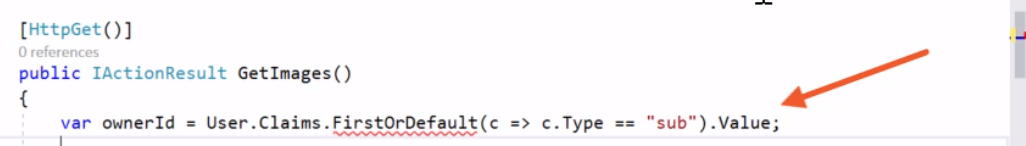
1. ImageGalleryHttpClient.cs – get **access token** and add it as a **bearer token**



1. Now the API should be accessible to authorized users

Using Access Token Claims when getting Resource Collections

1. We want to ensure that only images for the **logged in user** are accessible
2. ImagesController > GetImages() is called when the app grabs images
3. Find out who the current user is (“sub” claim identifies the current user):

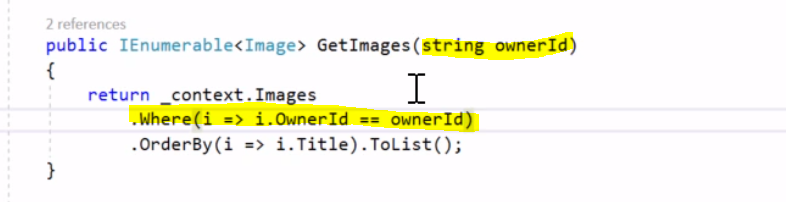


1. Change GetImages() method to accept Id as a **mandatory** input





1. In **GalleryRepository.cs**

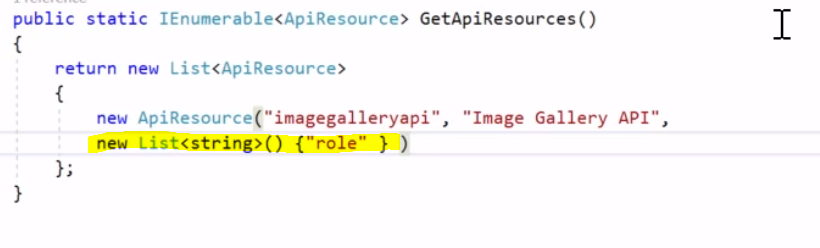


1. Now an authenticated user will **only** see their own images, and not other’s

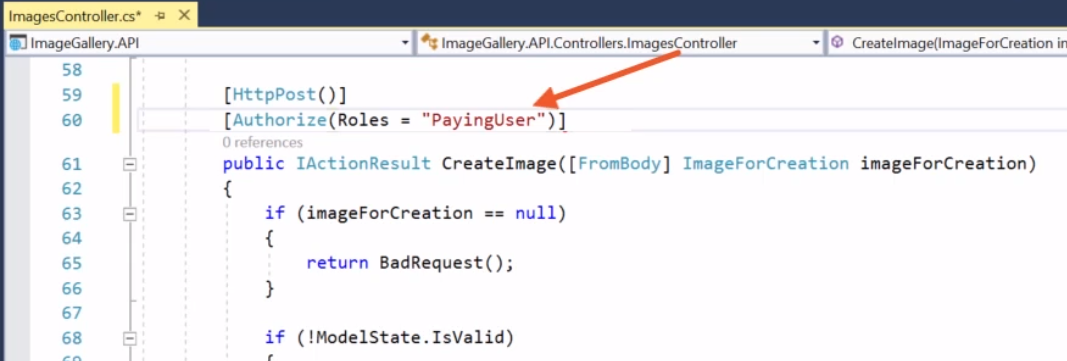
Including Identity Claims in an Access Token

* Sometimes an API needs to access identity claims
* When defining a resource scope (API resource), include the required claims in the claims list

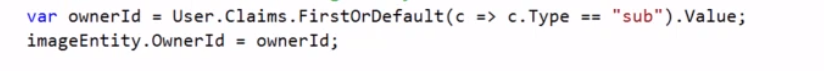
1. **Config.cs** in IDP level



1. Authorize PayingUser role for all CRUD operations on Images (so put Authorize decorator on **ImagesController**)



1. The API has to fill out the **OwnerId** field for an image based on the verified access token in the Action for creating the image
2. Inside **CreateImage()…** get OwnerId, and assign the OwnerId to the ImageEntity



* Access tokens are passed to the API as bearer tokens
* AccessTokenValidation middleware can be used to validate an access token at API level
* Configure ApiResource include additional identity claims in the access token
* Role-based authorization is achievable through the Authorize decorator