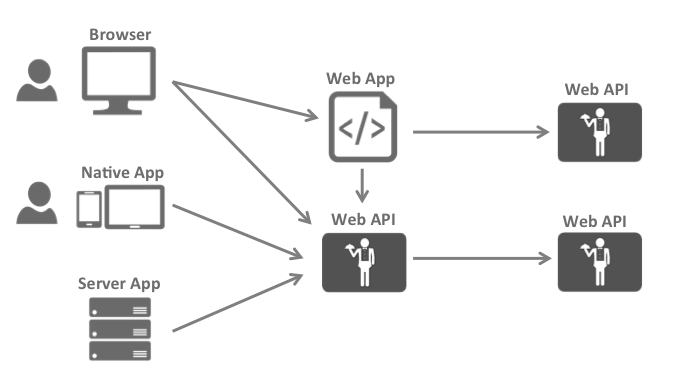
Identity Server Developer Guide

# Overview

Most modern applications look more or less like this:

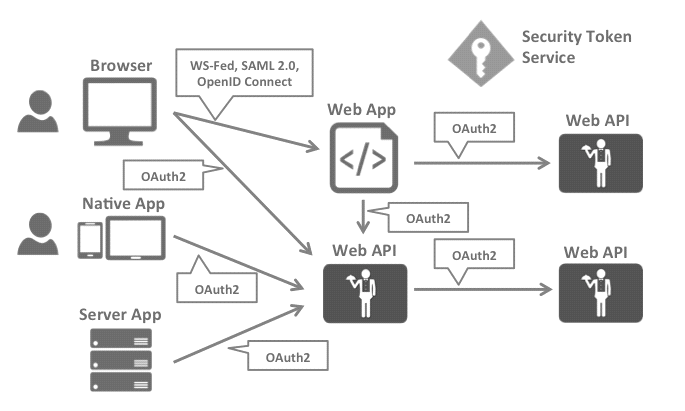
The typical interactions are:

* Browsers communicate with web applications
* Web applications communicate with web APIs (sometimes on their own, sometimes on behalf of a user)
* Browser-based applications communicate with web APIs
* Native applications communicate with web APIs
* Server-based applications communicate with web APIs
* Web APIs communicate with web APIs (sometimes on their own, sometimes on behalf of a user)

Typically, each and every layer (front-end, middle-tier and back-end) has to protect resources and implement authentication and/or authorization – and quite typically against the same user store.

This is why we don’t implement these fundamental security functions in the business applications/endpoints themselves, but rather outsource that critical functionality to a service - the security token service.

This leads to the following security architecture and usage of protocols:

This divides the security concerns into two parts.

## Authentication

Authentication is needed when an application needs to know about the identity of the current user. Typically, these applications manage data on behalf of that user and need to make sure that this user can only access the data he is allowed to. The most common example for that is (classic) web applications – but native and JS-based applications also have need for authentication.

The most common authentication protocols are SAML2p, WS-Federation and OpenID Connect – SAML2p being the most popular and the most widely deployed.

OpenID Connect is the newest of the three, but is generally considered to be the future because it has the most potential for modern applications. It was built for mobile application scenarios right from the start and is designed to be API friendly.

## API Access

Applications have two fundamental ways with which they communicate with APIs – using the application identity, or delegating the user’s identity. Sometimes both ways need to be combined.

OAuth2 is a protocol that allows applications to request access tokens from a security token service and use them to communicate with APIs. This reduces complexity on both the client applications as well as the APIs since authentication and authorization can be centralized.

## OpenID Connect and OAuth2 – better together

OpenID Connect and OAuth2 are very similar – in fact OpenID Connect is an extension on top of OAuth2. This means that you can combine the two fundamental security concerns – authentication and API access into a single protocol – and often a single round trip to the security token service.

This is why we believe that the combination of OpenID Connect and OAuth2 is the best approach to secure modern applications for the foreseeable future. IdentityServer3 is an implementation of these two protocols and is highly optimized to solve the typical security problems of today’s mobile, native and web applications.

# High Level Features

## Authentication as a Service

Centralized login logic and workflow at a single & well-secured place.

## Single Sign-on / Sign-out

Single sign-on (and out) over multiple application types like web or mobile.

## Access Control for APIs

Issue access tokens for APIs for various types of clients, e.g. server to server, web applications, SPAs and native/mobile apps.

### Federation

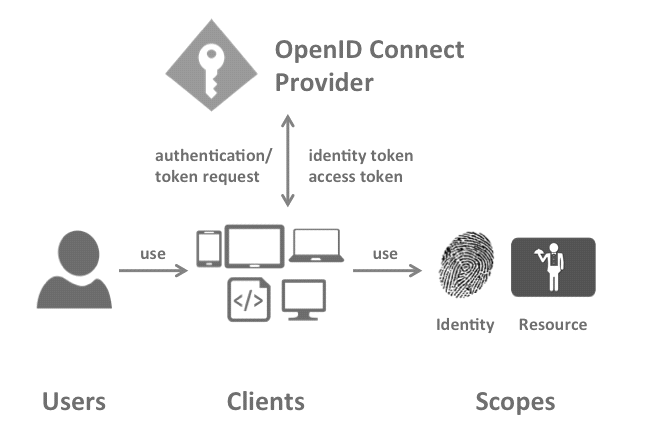
Support for external social identity providers like Google, Facebook etc. as well as integration for enterprise identity management systems via SAML and WS-Federation.

### Customization everywhere

The most important part - every aspect of Identity Server can be customized to fit **your** needs. Since Identity Server is a framework, you can write code to adapt the system in a way it makes sense for your scenarios.

# Terminology

The specs, documentation and object model use a certain terminology that you should be aware of.



## OpenID Connect Provider (OP)

Identity Server is an OpenID Connect provider - it implements the OpenID Connect protocol (and OAuth2 as well).

Different literature uses different terms for the same role - you probably also find security token service, identity provider, authorization server, IP-STS and more.

But they are in a nutshell all the same: a piece of software that issues security tokens to clients.

Identity Server has a number of jobs and features - including:

* authenticate users using a local account store or via an external identity provider
* provide session management and single sign-on
* manage and authenticate clients
* issue identity and access tokens to clients
* validate tokens

### Client

A client is a piece of software that requests tokens from Identity Server - either for authenticating a user or for accessing a resource (also often called a relying party or RP). A client must be registered with the OP.

Examples for clients are web applications, native mobile or desktop applications, SPAs, server processes etc.

### User

A user is a human that is using a registered client to access his or her data.

### Scope

Scopes are identifiers for resources that a client wants to access. This identifier is sent to the OP during an authentication or token request.

By default, every client is allowed to request tokens for every scope, but you can restrict that.

They come in two flavors.

#### Identity scopes

Requesting identity information (aka claims) about a user, e.g. his name or email address is modeled as a scope in OpenID Connect.

There is e.g. a scope called profile that includes first name, last name, preferred username, gender, profile picture and more. You can read about the standard scopes [here](http://openid.net/specs/openid-connect-core-1_0.html) and you can create your own scopes in Identity Server to model your own requirements.

#### Resource scopes

Resource scopes identify web APIs (also called resource servers) - you could have e.g. a scope named calendar that represents your calendar API.

### Authentication/Token Request

Clients request tokens from the OP. Depending on the scopes requested, the OP will return an identity token, an access token, or both.

#### Identity Token

An identity token represents the outcome of an authentication process. It contains at a bare minimum an identifier for the user (called the sub aka subject claim). It can contain additional information about the user and details on how the user authenticated at the OP.

#### Access Token

An access token allows access to a resource. Clients request access tokens and forward them to an API. Access tokens contain information about the client and the user (if present). APIs use that information to authorize access to their data.

# Configuration

IdentityServer3 is packaged as middleware and uses the typical “Options” patterns for configuration:

*public void Configuration (IAppBuilder appBuilder)*

*{*

*var options = new IdentityServerOptions*

*{*

*SigningCertificate = Certificate.Get(),*

*Factory = factory,*

*};*

*appBuilder.UseIdentityServer(options);*

*}*

The IdentityServerOptions class contains all configuration for Identity Server. One part consists of simple properties like the issuer name or site title which you can source from wherever you see fit (static in code, configuration file or database). The other part is the so called service factory which acts as a registry for certain aspects of Identity Server’s internal processing.

## Hosting in IIS and RAMMFAR

The files for the web pages are served up as embedded assets within the Identity Server assembly itself. When hosting in IIS or IIS Express to allow these files to be served RAMMFAR (*runAllManagedModulesForAllRequests*) needs to be enabled in web.config:

*<system.webServer>*

*<modules runAllManagedModulesForAllRequests="true">*

*</modules>*

*</system.webServer>*

# Identity Server Options

The IdentityServerOptions class is the top level container for all configuration settings of Identity Server.

* **IssuerUri**
* Unique name of this server instance, e.g. https://myissuer.com. Defaults to the base URL where Identity Server is installed
* **SiteName**
* Display name of the site used in standard views.
* **SigningCertificate**
* X.509 certificate (and corresponding private key) for signing security tokens.
* **SecondarySigningCertificate**
* A secondary certificate that will appear in the discovery document. Can be used to prepare clients for certificate rollover.
* **RequireSsl**
* Indicates if SSL is required for Identity Server. Defaults to true.
* **PublicOrigin**
* By default, Identity Server uses the host, protocol, and port from the HTTP request when creating links. This might not be accurate in reverse proxy or load-balancing situations. You can override the origin used for link generation using this property.
* **Endpoints**
* Allows enabling or disabling specific endpoints (by default all endpoints are enabled).
* **Factory (required)**
* Sets the IdentityServerFactory
* **DataProtector**
* Sets a custom data protector. By default, the Katana host data protector is used.
* **AuthenticationOptions**
* Configures AuthenticationOptions
* **PluginConfiguration**
* Allows adding protocol plugins like the WS-Federation support.
* **CspOptions**
* Configures CSP
* **ProtocolLogoutUrls**
* Configures callback URLs that should be called during logouts (mainly useful for protocol plugins).
* **LoggingOptions**
* Configures settings related to logging
* **EventsOptions**
* Configures settings related to events
* **EnableWelcomePage**
* Enables or disables the default welcome page. Defaults to true.

# Service Factory

IdentityServer3 contains many features for implementing OpenID Connect and OAuth2. Many of these features have been designed so they can be replaced. This would be useful for the scenarios where the default logic doesn’t match the hosting application’s requirements, or simply the application wishes to provide an entirely different implementation. And in fact, there are some extensibility points within IdentityServer3 that are required to be provided by the hosting application (such as the storage for configuration data or the identity management implementation for validating users’ credentials).

The IdentityServer3.Core.Configuration.IdentityServerServiceFactory holds all these building blocks and must be supplied at startup time using the IdentityServerOptions class (see here for more information on configuration options).

The extensibility points fall into three categories.

## Mandatory

* **UserService**
* Implements user authentication against the local user store, association of external users, claims retrieval and sign-out logic. There are two standard implementations for MembershipReboot and ASP.NET Identity
* **ScopeStore**
* Implements retrieval of scopes configuration data
* **ClientStore**
* Implements retrieval of client configuration data

The IdentityServerServiceFactory allows setting up a service factory by providing in-memory stores for users, clients and scopes (see here).

Mandatory for production scenarios (but with default in-memory implementations)

* **AuthorizationCodeStore**
* Implements storage and retrieval of authorization codes
* **TokenHandleStore**
* Implements storage and retrieval of handles for reference tokens
* **RefreshTokenStore**
* Implements storage and retrieval of refresh tokens
* **ConsentStore**
* Implements storage and retrieval of consent decisions
* **ViewService**
* Implements retrieval of UI assets. Defaults to using the embedded assets.

## Optional (can be replaced, but have default implementations)

* **TokenService**
* Implements creation of identity and access tokens
* **ClaimsProvider**
* Implements retrieval of claims for identity and access tokens
* **TokenSigningService**
* Implements creation and signing of security tokens
* **CustomGrantValidator**
* Implements validation of custom grant types
* **CustomRequestValidator**
* Implements custom additional validation of authorize and token requests
* **RefreshTokenService**
* Implements creation and updates of refresh tokens
* **ExternalClaimsFilter**
* Implements filtering and transformation of claims for external identity providers
* **CustomTokenValidator**
* Implements custom additional validation of tokens for the token validation endpoints
* **CustomTokenResponseGenerator**
* Allows adding additional data to a token response
* **ConsentService**
* Implements logic of consent decisions
* **ClientPermissionsService**
* Implements retrieval and revocation of consents, reference and refresh tokens
* **EventService**
* Implements forwarding events to some logging system (e.g. elastic search)
* **RedirectUriValidator**
* Implements validation of redirect and post logout URIs
* **LocalizationService**
* Implements localization of display strings
* **CorsPolicyService**
* Implements CORS policy

# Clients

The Client class models an OpenID Connect or OAuth2 client - e.g. a native application, a web application or a JS-based application.

* **Enabled**
* Specifies if client is enabled. Defaults to true.
* **ClientId**
* Unique ID of the client
* **ClientSecrets**
* List of client secrets - only relevant for flows that require a secret
* **ClientName**
* Client display name (used for logging and consent screen)
* **ClientUri**
* URI to further information about client (used on consent screen)
* **LogoUri**
* URI to client logo (used on consent screen)
* **RequireConsent**
* Specifies whether a consent screen is required. Defaults to true.
* **AllowRememberConsent**
* Specifies whether user can choose to store consent decisions. Defaults to true.
* **Flow**
* Specifies allowed flow for client (either AuthorizationCode, Implicit, Hybrid, ResourceOwner, ClientCredentials or Custom). Defaults to Implicit.
* **AllowClientCredentialsOnly**
* Gets or sets a value indicating whether this client is allowed to request token using client credentials only. This is useful when you want a client to be able to use both a user-centric flow like implicit and additionally client credentials flow. Defaults to false. Should only be used for confidential clients (e.g. not Implicit).
* **RedirectUris**
* Specifies the allowed URIs to return tokens or authorization codes to
* **PostLogoutRedirectUris**
* Specifies allowed URIs to redirect to after logout
* **LogoutUri (added in v2.2)**
* Specifies logout URI at client for HTTP based logout
* **LogoutSessionRequired (added in v2.2)**
* Specifies if the user’s session id should be sent to the LogoutUri. Defaults to true.
* **RequireSignOutPrompt (added in v2.4)**
* Specifies if the client will always show a confirmation page for sign-out. Defaults to false.
* **AllowedScopes**
* By default, a client has no access to any scopes - either specify the scopes explicitly here (recommended) - or set AllowAccessToAllScopes to true.
* **AllowAccessTokensViaBrowser (added in v2.5)**
* Specifies whether this client is allowed to request access tokens via the browser. This is useful to harden flows that allow multiple response types (e.g. by disallowing a hybrid flow client that is supposed to use code id\_token to add the token response type and thus leaking the token to the browser.
* **AllowedCustomGrantTypes**
* When Custom flow is used, you also need to specify which custom grant types this client can use. Explicitly specify the grant types here (recommended) or set AllowAccessToAllCustomGrantTypes to true.
* **IdentityTokenLifetime**
* Lifetime to identity token in seconds (defaults to 300 seconds / 5 minutes)
* **AccessTokenLifetime**
* Lifetime of access token in seconds (defaults to 3600 seconds / 1 hour)
* **AuthorizationCodeLifetime**
* Lifetime of authorization code in seconds (defaults to 300 seconds / 5 minutes)
* **AbsoluteRefreshTokenLifetime**
* Maximum lifetime of a refresh token in seconds. Defaults to 2592000 seconds / 30 days
* **SlidingRefreshTokenLifetime**
* Sliding lifetime of a refresh token in seconds. Defaults to 1296000 seconds / 15 days
* **RefreshTokenUsage**
* ReUse: the refresh token handle will stay the same when refreshing tokens
* OneTime: the refresh token handle will be updated when refreshing tokens
* **RefreshTokenExpiration**
* Absolute: the refresh token will expire on a fixed point in time (specified by the AbsoluteRefreshTokenLifetime)
* Sliding: when refreshing the token, the lifetime of the refresh token will be renewed (by the amount specified in SlidingRefreshTokenLifetime). The lifetime will not exceed AbsoluteRefreshTokenLifetime.
* **UpdateAccessTokenClaimsOnRefresh**
* Gets or sets a value indicating whether the access token (and its claims) should be updated on a refresh token request.
* **AccessTokenType**
* Specifies whether the access token is a reference token or a self-contained JWT token (defaults to Jwt).
* **EnableLocalLogin**
* Specifies if this client can use local accounts, or external IdPs only. Defaults to true.
* **IdentityProviderRestrictions**
* Specifies which external IdPs can be used with this client (if list is empty all IdPs are allowed). Defaults to empty.
* **IncludeJwtId**
* Specifies whether JWT access tokens should have an embedded unique ID (via the jticlaim).
* **AllowedCorsOrigins**
* If specified, will be used by the default CORS policy service implementations (In-Memory and EF) to build a CORS policy for JavaScript clients.
* **Claims**
* Allows settings claims for the client (will be included in the access token).
* **AlwaysSendClientClaims**
* If set, the client claims will be sent for every flow. If not, only for client credentials flow (default is false)
* **PrefixClientClaims**
* If set, all client claims will be prefixed with client\_ to make sure they don’t accidentally collide with user claims. Default is true.

In addition there are a number of settings controlling the behavior of refresh tokens - see [here](https://identityserver.github.io/Documentation/docsv2/advanced/refreshTokens.html)

Example: Configure a client for implicit flow

*var client = new Client*

*{*

*ClientName = "JS Client",*

*Enabled = true,*

*ClientId = "implicitclient",*

*Flow = Flows.Implicit,*

*RequireConsent = true,*

*AllowRememberConsent = true,*

*RedirectUris = new List<string>*

*{*

*"*[*https://myapp/callback.html*](https://myapp/callback.html)*",*

*},*

*PostLogoutRedirectUris = new List<string>*

*{*

*"*[*http://localhost:23453/index.html*](http://localhost:23453/index.html)*",*

*}*

*}*

# Scopes and Claims

The IdentityServer.Core.Models.Scope class models an OpenID Connect or OAuth2 scope.

* **Enabled**
* Indicates if scope is enabled and can be requested. Defaults to true.
* **Name**
* Name of the scope. This is the value a client will use to request the scope.
* **DisplayName**
* Display name for consent screen.
* **Description**
* Description for the consent screen.
* **Required**
* Specifies whether the user can de-select the scope on the consent screen. Defaults to false.
* **ScopeSecrets (added in v2.2)**
* Adds secret to scope (for the introspection endpoint)
* **AllowUnrestrictedIntrospection (added in v2.3)**
* Allows this scope to see all other scopes in the access token when using the introspection endpoint
* **Emphasize**
* Specifies whether the consent screen will emphasize this scope. Use this setting for sensitive or important scopes. Defaults to false.
* **Type**
* Either Identity (OpenID Connect related) or Resource (OAuth2 resources). Defaults to Resource.
* **Claims**
* List of user claims that should be included in the identity (identity scope) or access token (resource scope).
* **IncludeAllClaimsForUser**
* If enabled, all claims for the user will be included in the token. Defaults to false.
* **ClaimsRule**
* Rule for determining which claims should be included in the token (this is implementation specific)
* **ShowInDiscoveryDocument**
* Specifies whether this scope is shown in the discovery document. Defaults to true.

**Scope can also specify claims that go into the corresponding token - the ScopeClaim class has the following properties:**

* **Name**
* Name of the claim
* **Description**
* Description of the claim
* **AlwaysIncludeInIdToken**
* Specifies whether this claim should always be present in the identity token (even if an access token has been requested as well). Applies to identity scopes only. Defaults to false.

**Example of a role identity scope:**

*var roleScope = new Scope*

*{*

*Name = "roles",*

*DisplayName = "Roles",*

*Description = "Your organizational roles",*

*Type = ScopeType.Identity,*

*Claims = new List<ScopeClaim>*

*{*

*new ScopeClaim(Constants.ClaimTypes.Role, alwaysInclude: true)*

*}*

*};*

The ‘AlwaysIncludeInIdentityToken’ property specifies that a certain claim should always be part of the identity token, even when an access token for the userinfo endpoint is requested.

*Example of a scope for the IdentityManager API:*

*var idMgrScope = new Scope*

*{*

*Name = "idmgr",*

*DisplayName = "IdentityManager",*

*Type = ScopeType.Resource,*

*Emphasize = true,*

*Claims = new List<ScopeClaim>*

*{*

*new ScopeClaim(Constants.ClaimTypes.Name),*

*new ScopeClaim(Constants.ClaimTypes.Role)*

*}*

*};*

# Keys, Signatures and Cryptography

Identity Server depends on cryptography in various places. Here’s an overview.

## Secure Transport (HTTPS)

By default, Identity Server requires all incoming connections to come over HTTPS. It is absolutely mandatory that communication with Identity Server is done over secured transports only. There are certain deployment scenarios like SSL offloading where this requirement can be relaxed. See the deployment section for more information.

SSL/TLS configuration is done at the hosting level - e.g in IIS or with HTTP.SYS directly.

## Token Signing

Identity and JWT access tokens are signed with an X.509 certificate using the RSA algorithm (RS256). The signing certificate is set on the IdentityServerOptions using the SigningCertificate property. Setting this property is mandatory for identity tokens and JWT access tokens.

If you are using reference tokens only, you don’t need to set the signing certificate.

**Remark** The minimum supported key length is 2048 bytes.

## Signing Key Rollover

X.509 certificates have a finite lifetime. Renewing and rolling over the singing key without downtime of your application can be a challenge. Identity Server has a couple of features that make this process easier:

* The discovery document publishes the current (and a secondary) public key. This way token consumers can learn about the key material.
* All JWTs contain a key identifier that matches a key published in the discovery document
* The access token validation middleware periodically (every 24h) checks the discovery document to update its own key configuration.

A key rollover could work like this:

* Get the new certificate that should replace the old one
* Set this new certificate as the SecondarySigningCertificate on the options. Identity Server will now publish both certificates in the discovery document and the access token validation middleware will accept tokens signed with both keys
* Wait at least 24h to give every consumer a chance to update its configuration
* Set the new certificate as the primary SigningCertificate.
* Keep the old certificate as a secondary for as long as you need (maybe you have long running tokens out there that use the old certificate and need to be validated)

( Generating and Installing Signing-Certificates for Identity Server - <http://amilspage.com/signing-certificates-idsv4/>)

# Authentication Options

The AuthenticationOptions is a property on the IdentityServerOptions to customize the login and logout views and behavior.

* **EnableLocalLogin**
* Indicates if Identity Server will allow users to authenticate with a local account. Disabling this setting will not display the username/password form on the login page. This also will disable the resource owner password flow. Defaults to true.
* **EnableLoginHint**
* Indicates whether the login\_hint parameter is used to prepopulate the username field. Defaults to true.
* **LoginPageLinks**
* List of LoginPageLink objects. These allow the login view to provide the user custom links to other web pages that they might need to visit before they can login (such as a registration page, or a password reset page).
* LoginPageLink contains:
* Type: An identifier for the type of link.
* Text: The text to appear in the link.
* Href: The URL for the href of the link.
* The custom web page represented by the LoginPageLink would be provided by the hosting application. Once it has performed its task then it can resume the login workflow by redirecting the user back to the login view.
* When a user follows one of the LoginPageLinks, a signin query string parameter is passed to the page. This parameter should be echoed back as a signin query string parameter to the login page when the user wishes to resume their login. The login view is located at the path “~/login” relative to Identity Server’s application base.
* **RememberLastUsername**
* Indicates whether Identity Server will remember the last username entered on the login page. Defaults to false.
* **IdentityProviders**
* Allows configuring additional identity providers
* **CookieOptions**
* CookieOptions object that configures how cookies are managed by Identity Server.
* CookieOptions has these properties:
* **Prefix**: Allows setting a prefix on cookies to avoid potential conflicts with other cookies with the same names. By default, no prefix is used.
* **ExpireTimeSpan**: The expiration duration of the authentication cookie. Defaults to 10hours.
* **IsPersistent**: Indicates whether the authentication cookie is marked as persistent. Defaults to false.
* **SlidingExpiration**: Indicates if the authentication cookie is sliding, which means it auto renews as the user is active. Defaults to false.
* **Path**: Sets the cookie path. Defaults to the base path of Identity Server in the hosting application.
* **AllowRememberMe**: Indicates whether the “remember me” option is presented to users on the login page. If selected this option will issue a persistent authentication cookie. Defaults to true.
* If this setting is in use then the user’s decision (either yes or no) will override the IsPersistent setting. In other words, if both IsPersistent and AllowRememberMeis enabled and the user decides to not remember their login, then no persistent cookie will be issued.
* **RememberMeDuration**: Duration of the persistent cookie issued by the “remember me” option on the login page. Defaults to 30 days.
* **SecureMode**: Gets or sets the mode for issuing the secure flag on the cookies issued. Defaults to SameAsRequest.
* **RequireSignOutPrompt (added in v2.4)**
* Gets or sets a value indicating whether Identity Server will always show a confirmation page for sign-out. Defaults to false.
* **EnableSignOutPrompt**
* Indicates whether Identity Server will show a confirmation page for sign-out. When a client initiates a sign-out, by default Identity Server will ask the user for confirmation. This is a mitigation technique against “logout spam”. Defaults to true.
* **EnablePostSignOutAutoRedirect**
* Gets or sets a value indicating whether Identity Server automatically redirects back to a validated post\_logout\_redirect\_uri passed to the signout endpoint. Defaults to false.
* **PostSignOutAutoRedirectDelay**
* Gets or sets the delay (in seconds) before redirecting to a post\_logout\_redirect\_uri. Defaults to 0.
* **SignInMessageThreshold**
* Gets or sets the limit after which old signin messages (cookies) are purged. Defaults to 5.
* **InvalidSignInRedirectUrl**
* Gets or sets the invalid sign in redirect URL. If the user arrives at the login page without a valid sign-in request, then they will be redirected to this URL. The URL must be absolute or can relative URLs (starting with “~/”).

# Identity Providers

Identity Server supports authentication using external identity providers. The external authentication mechanism must be encapsulated in a Katana authentication middleware.

Katana itself ships with middleware for Google, Facebook, Twitter, Microsoft Accounts, WS-Federation and OpenID Connect - but there are also community developed middleware (including Yahoo, LinkedIn, and SAML2p).

To configure the middleware for the external providers, add a method to your project that accepts an IAppBuilder and a string as parameters.

*public static void ConfigureIdentityProviders(IAppBuilder app, string signInAsType)*

*{*

*var google = new GoogleOAuth2AuthenticationOptions*

*{*

*AuthenticationType = "Google",*

*Caption = "Google",*

*SignInAsAuthenticationType = signInAsType,*

*ClientId = "...",*

*ClientSecret = "..."*

*};*

*app.UseGoogleAuthentication(google);*

*var fb = new FacebookAuthenticationOptions*

*{*

*AuthenticationType = "Facebook",*

*Caption = "Facebook",*

*SignInAsAuthenticationType = signInAsType,*

*AppId = "...",*

*AppSecret = "..."*

*};*

*app.UseFacebookAuthentication(fb);*

*var twitter = new TwitterAuthenticationOptions*

*{*

*AuthenticationType = "Twitter",*

*Caption = "Twitter",*

*SignInAsAuthenticationType = signInAsType,*

*ConsumerKey = "...",*

*ConsumerSecret = "..."*

*};*

*app.UseTwitterAuthentication(twitter);*

*}*

## Notes

* AuthenticationType must be a unique value to identify the external identity provider. This value will also be used for the idp claim in the resulting tokens. Furthermore the same value can be used to pre-select identity providers during authorization/authentication requests using the acr\_values parameter (see [this](https://identityserver.github.io/Documentation/docsv2/endpoints/authorization.html) for more information). This value is also used to restrict the allowed identity providers on the Client configuration.
* Caption specifies the label of the button on the login page for the identity provider. If Caption is an empty string, the identity provider will not be shown on the login page. But can still be used via the login hint.
* SignInAsAuthenticationType must be set to the value we pass in via the signInAsTypeparameter

Assign the configuration method to the IdentityProviders property on the AuthenticationOptions:

*var idsrvOptions = new IdentityServerOptions*

*{*

*SiteName = "IdentityServer3",*

*Factory = factory,*

*SigningCertificate = Cert.Load(),*

*AuthenticationOptions = new AuthenticationOptions*

*{*

*IdentityProviders = ConfigureIdentityProviders*

*}*

*};*

*app.UseIdentityServer(idsrvOptions);*

# Authorization/Authentication Endpoint

The authorization endpoint can be used to request either access tokens or authorization codes (implicit and authorization code flow). You either use a web browser or a web view to start the process.

Supported Parameters

See [spec](http://openid.net/specs/openid-connect-core-1_0.html).

* **client\_id (required)**
* identifier of the client
* **scope (required)**
* one or more registered scopes
* **redirect\_uri (required)**
* must exactly match one of the allowed redirect URIs for that client
* **response\_type (required)**
* code requests an authorization code
* token requests an access token (only resource scopes are allowed)
* id\_token token requests an identity token and an access token (both resource and identity scopes are allowed)
* **response\_mode (optional)**
* form\_post sends the token response as a form post instead of a fragment encoded redirect
* **state (recommended)**
* idsrv will echo back the state value on the token response, this is for correlating request and response
* **nonce (required for identity tokens using implicit flow)**
* idsrv will echo back the nonce value in the identity token, this is for correlating the token to the request)
* **prompt (optional)**
* none no UI will be shown during the request. If this is not possible (e.g. because the user has to sign in or consent) an error is returned
* login the login UI will be shown, even if the user is already signed-in and has a valid session
* **code\_challenge (required when using proof keys - added in v2.5)**
* send the code challenge for proof key flows)
* **code\_challenge\_method (optional - default to plain when using proof keys - added in v2.5)**
* plain indicates that the challenge is using plain text (not recommended)
* S256 indicates the the challenge is hashed with SHA256
* **login\_hint (optional)**
* can be used to pre-fill the username field on the login page
* **ui\_locales (optional)**
* gives a hint about the desired display language of the login UI
* **max\_age (optional)**
* if the user’s logon session exceeds the max age (in seconds), the login UI will be shown
* **acr\_values (optional)**
* allows to pass additional authentication related information to the user service - there are also values with special meaning:
* **idp**:name\_of\_idp bypasses the login/home realm screen and forwards the user directly to the selected identity provider (if allowed per client configuration)
* **tenant**:name\_of\_tenant can be used to pass a tenant name to the user service

**Example**

(URL encoding removed for readability)

*GET /connect/authorize?client\_id=client1&scope=openid email api1&response\_type=id\_token token*

# Token Endpoint

The token endpoint can be used to programmatically request or refresh tokens (resource owner password credential flow, authorization code flow, client credentials flow and custom grant types).

Supported Parameters

See [spec](http://openid.net/specs/openid-connect-core-1_0.html).

* **grant\_type**(required)
* authorization\_code, client\_credentials, password, refresh\_token or custom
* **scope**(required for all grant types besides refresh\_token and code)
* **redirect\_uri** (required for code grant type)
* **code**(required for code grant)
* **code\_verifier**(required when using proof keys - added in v2.5)
* **username**(required for password grant type)
* **password**(required for password grant\_type)
* **acr\_values** (allowed for password grant type to pass additional information to user service)
* there are values with special meaning:
* idp:name\_of\_idp bypasses the login/home realm screen and forwards the user directly to the selected identity provider (if allowed per client configuration)
* tenant:name\_of\_tenant can be used to pass extra information to the user service
* **refresh\_token** (required for refresh token grant)
* **client\_id** (either in the post body, or as a basic authentication header)
* **client\_secret** (either in the post body, or as a basic authentication header)

Authentication

All requests to the token endpoint must be authenticated - either pass client id and secret via Basic Authentication or add client\_id and client\_secret fields to the POST body.

When providing the client\_id and client\_secret in the Authorization header it is expected to be:

* client\_id:client\_secret
* Base64 encoded

*var clientId = "...";*

*var clientSecret = "...";*

*var encoding = Encoding.UTF8;*

*var credentials = string.Format("{0}:{1}", clientId, clientSecret);*

*var headerValue = Convert.ToBase64String(encoding.GetBytes(credentials));*

**Example**

(Form-encoding removed and line breaks added for readability)

*POST /connect/token*

*Authorization: Basic abcxyz*

*grant\_type=authorization\_code&*

*code=hdh922&*

*redirect\_uri=https://myapp.com/callback*

# UserInfo Endpoint

The UserInfo endpoint can be used to retrieve identity information about a subject. It requires a valid access token with at least the ‘openid’ scope.

see [spec](http://openid.net/specs/openid-connect-core-1_0.html)

**Example**

*GET /connect/userinfo*

*Authorization: Bearer <access\_token>*

*HTTP/1.1 200 OK*

*Content-Type: application/json*

*{*

*"sub": "248289761001",*

*"name": "Bob Smith",*

*"given\_name": "Bob",*

*"family\_name": "Smith",*

*"role": [*

*"user",*

*"admin"*

*]*

*}*

# Discovery Endpoint

The discovery endpoint can be used to retrieve metadata about Identity Server - it returns information like the issuer name, key material, supported scopes etc.

see [spec](http://openid.net/specs/openid-connect-discovery-1_0.html)

**Example**

*GET /.well-known/openid-configuration*

# Logout Endpoint

Redirecting to the logout endpoint clears the authentication session and cookie.

You can pass the following optional parameters to the endpoint:

* **id\_token\_hint**
* The id\_token that the client acquired during authentication. This allows bypassing the logout confirmation screen as well as providing a post logout redirect URL
* **post\_logout\_redirect\_uri**
* A URI that Identity Server can redirect to after logout (by default a link is displayed). The URI must be in the list of allowed post logout URIs for the client.

*/connect/endsession?id\_token\_hint=...&post\_logout\_redirect\_uri=https://myapp.com*

# Access token validation endpoint

The access token validation endpoint can be used to validate reference tokens. It can be also used to validate self-contained JWTs if the consumer does not have support for appropriate JWT or cryptographic libraries.

You can either GET or POST to the validation endpoint. Due to query string size restrictions, POST is recommended.

**Example**

*POST /connect/accesstokenvalidation*

*token=<token>*

or

*GET /connect/accesstokenvalidation?token=<token>*

A successful response will return a status code of 200 and the associated claims for the token. An unsuccessful response will return a 400 with an error message.

It is also possible to pass a scope that is expected to be inside the token:

*POST /connect/accesstokenvalidation*

*token=<token>&*

*expectedScope=calendar*

**Remark** The access token validation endpoint does not enforce client authentication. Don’t use reference tokens for confidentiality purposes!

# Refresh Tokens

* Refresh tokens are supported for the following flows: authorization code, hybrid and resource owner password credential flow.
* The clients need to be allowed to request the offline\_access scope to get a refresh token.

**Settings on the Client class**

* **RefreshTokenUsage**
* ReUse: the refresh token handle will stay the same when refreshing tokens
* OneTime: the refresh token handle will be updated when refreshing tokens
* **RefreshTokenExpiration**
* Absolute: the refresh token will expire on a fixed point in time (specified by the AbsoluteRefreshTokenLifetime)
* Sliding: when refreshing the token, the lifetime of the refresh token will be renewed (by the amount specified in SlidingRefreshTokenLifetime). The lifetime will not exceed the absolute lifetime.
* **AbsoluteRefreshTokenLifetime**
* Maximum lifetime of a refresh token in seconds. Defaults to 2592000 seconds / 30 days
* **SlidingRefreshTokenLifetime**
* Sliding lifetime of a refresh token in seconds. Defaults to 1296000 seconds / 15 days

**Usage**

* Request the offline\_access scope (via code or resource owner flow)
* Refresh the token by using a refresh\_token grant

# Customizing Views

IdentityServer3 displays various “views” to the user. Identity Server requires views for login, logout prompt, logged out, consent, client permissions, and errors. These views are simply web pages displayed in the browser. To obtain the markup for these views, Identity Server defines the IViewService interface. Identity Server provides default views (via the DefaultViewServiceimplementation).

The views in Identity Server can be customized in one of two ways: 1) Customize the HTML templates provided by the DefaultViewService, or if more control is needed 2) define a custom IViewService.

# Entity Framework support for Clients, Scopes

## Clients and Scopes

### Client

The Client is referred to the IdentityServer3.EntityFramework.Entities’s *Class* *Client* It can be used independently from the Scope.

We have used XML to declare all the properties.

*var efConfig = new EntityFrameworkServiceOptions*

*{*

*ConnectionString = connString,*

*};*

*// these two calls just pre-populate the test DB from the in-memory config*

*ConfigureClients(efConfig);*

*ConfigureScopes(DefaultScopes.Get(), efConfig);*

*ConfigureUsers(DefaultUsers.Get(), efConfig.ConnectionString);*

*var factory = new IdentityServerServiceFactory();*

Where ConfigureClients(efConfig); is a method which is used to read the XML of Client

#### Progress Client XML

*<Client id="Progress" name="Progress" enabled="true">*

*<requireConsent>false</requireConsent>*

*<clientUri>https://localhost/veriqua/</clientUri>*

*<allowRememberConsent>true</allowRememberConsent>*

*<secrets>*

*<secret value="Progress"></secret>*

*</secrets>*

*<logoutSessionRequired>true</logoutSessionRequired>*

*<identityTokenLifetime>3600</identityTokenLifetime>*

*<allowAccessTokensViaBrowser>1</allowAccessTokensViaBrowser>*

*<accessTokenLifetime>3600</accessTokenLifetime>*

*<authorizationCodeLifetime>3600</authorizationCodeLifetime>*

*<absoluteRefreshTokenLifetime>2592000</absoluteRefreshTokenL ifetime>*

*<refreshTokenExpiration>1</refreshTokenExpiration>*

*<enableLocalLogin>1</enableLocalLogin>*

*<prefixClientClaims>1</prefixClientClaims>*

*<flow>Implicit</flow>*

*<allowedScopes>*

*<allowedScope>ProgressApi</allowedScope>*

*<allowedScope>openid</allowedScope>*

*</allowedScopes>*

*<redirectUris>*

*<redirectUri><![CDATA[https://localhost/veriqua/welcome]]> </redirectUri>*

*<redirectUri><![CDATA[https://localhost/veriqua/silentrene w.html]]></redirectUri>*

*</redirectUris>*

*<postLogoutRedirectUris>*

*<postLogoutRedirectUri><![CDATA[https://localhost/veriqua/ Login]]></postLogoutRedirectUri>*

*</postLogoutRedirectUris>*

*<identityProviderRestrictions>*

*<identityProviderRestriction>idsrv</identityProviderRestri ction>*

*</identityProviderRestrictions>*

*</Client>*

### Scope

The Scope is referred to the IdentityServer3.EntityFramework.Entities’s Class Scope It can be used independently from the Client.

Where ConfigureScopes(DefaultScopes.Get(), efConfig); is the method where Scope is being read from the XML.

#### Progress Scope XML

*<Scope name="ProgressApi" enabled="true" type="Resource">*

*<displayName><![CDATA[Progress API]]></displayName>*

*<description><![CDATA[Authorization For Progress API]]></des cription>*

*<showInDiscoveryDocument>true</showInDiscoveryDocument>*

*<emphasize>true</emphasize>*

*<ScopeClaims>*

*<ScopeClaim name="name"></ScopeClaim>*

*<ScopeClaim name="role"></ScopeClaim>*

*</ScopeClaims>*

*</Scope>*

### Registration

To use either of the stores, they need to be registered. There are extension methods on the IdentityServerServiceFactory that allow either or both of the stores to be configured. All of the extension methods accept an EntityFrameworkServiceOptions which contains these properties:

* ConnectionString: The name of the connection string as configured in the .config file.
* Schema: An optional database schema to use for the tables. If not provided, then the database default will be used.

To configure the stores independently, this code could be used:

*var efConfig = new EntityFrameworkServiceOptions {*

*ConnectionString = "SomeConnectionName",*

*//Schema = "someSchemaIfDesired"*

*};*

*var factory = new IdentityServerServiceFactory();*

*factory.RegisterClientStore(efConfig);*

*factory.RegisterScopeStore(efConfig);*

*If both stores will be used with the same EntityFrameworkServiceOptions, then a single convenient extension method is provided:*

*var efConfig = new EntityFrameworkServiceOptions {*

*ConnectionString = "SomeConnectionName",*

*//Schema = "someSchemaIfDesired"*

*};*

*var factory = new IdentityServerServiceFactory();*

*factory.RegisterConfigurationServices(efConfig);*

# Client application integration for Identity Server

## Veriqua Client

Veriqua application is a combination of webapp and web service, web app is used for front-end and web service is used as a back-end basically it is used for all the API calls that are going from front-end.

### Veriqua WebApp integration to Identity Server

In order to integrate Veriqua webapp with identity server we have used

"oidc-client" version "1.3.0"

oidc-client is a library to provide OpenID Connect (OIDC) and OAuth2 protocol support for client-side, browser-based JavaScript client applications. Also included support for user session and access token management.

as soon as a user is logged in, the library takes care of creating the hidden iframe from the OIDC IdP, and polls it at a regular and configurable interval to be aware of a potential change in the user session.

### Installation

**Node.js**

Node.js v4.4 or later required.

**NPM**

npm install oidc-client –save

**NOTE**: if you're not already using [babel-polyfill](https://www.npmjs.com/package/babel-polyfill) make sure you run npm install --save babel-polyfill as well. Then include it in your build.

### Configuration

In veriqua the identity server configuration setup is done in the account service js file as shown below

*var settings = {*

*// URL of your OpenID Connect server.*

*// The library uses it to access the metadata document*

*authority: '*[*https://localhost/IdentityServer/ids*](https://localhost/IdentityServer/ids)*',*

*client\_id: 'Progress',*

*redirect\_uri: '* [*https://localhost/veriqua/welcome'*](https://localhost/veriqua/welcome')*,*

*silent\_redirect\_uri: ' https:// HYPERLINK "https://localhost/veriqua/silentrenew.html',*

*//for eg.,"https://localhost/veriqua/silentrenew.html'"/veriqua/silentrenew.html*

*post\_logout\_redirect\_uri: '* [*https://localhost/veriqua/login'*](https://localhost/veriqua/login')*,*

*// What you expect back from The IdP.*

*// In that case, like for all JS-based applications, an identity token*

*// and an access token*

*response\_type: 'id\_token token',*

*// Scopes requested during the authorisation request*

*scope: 'ProgressApi openid',*

*// Number of seconds before the token expires to trigger*

*// the `tokenExpiring` event*

*accessTokenExpiringNotificationTime: 4,*

*// Do we want to renew the access token automatically when it's*

*// about to expire?*

*automaticSilentRenew: true,*

*// Do we want to filter OIDC protocal-specific claims from the response?*

*filterProtocolClaims: true*

*};*

*// `UserManager` is the main class exposed by the library*

*\_self.mgr = new Oidc.UserManager(settings);*

*// You can hook a logger to the library.*

*// Conveniently, the methods exposed by the logger match*

*// the `console` object*

*Oidc.Log.logger = console;*

*// After successful login, when an access token is expired this event is fired.*

*\_self.mgr.events.addAccessTokenExpired(function () {*

*console.log ("token expired");*

*});*

*// When the automatic session management feature detects a change in*

*// the user session state, the `userSignedOut` event is fired.*

*\_self.mgr.events.addUserSignedOut(function () {*

*\_self.mgr.signinRedirect();*

*});*

### Important options on Settings

* **authority** – Value is the Identity Server url.
* **client\_id** – Value is the id that is declared as client id in the identity server Client XML for particular application.
* **redirect\_uri** – Value is the url of the application route to which it has to be redirected after successful token grant.
* **post\_logout\_redirect\_uri** – Value is the url of the application route to which it has to be redirected after successful logout.
* **response\_type** – Value is the type of response that the application receives after log in, it has to be same what it has declared in the identity server client XML.
* **scope** – Value is the scope value of the application that is declared in the identity server.

Here Identity server manager that is “*\_self.mgr*” is accessed outside by injecting the account service and is accessed by “*accountservice.mgr*”

The below code snippet can be added from where the identity server call has to happen that is redirecting to the login page of identity server once the login is successful the identity server uses the redirect url to redirect to the application page and here it should contain a call back for redirect in order get the user token information.

### Redirect to Identity server

*accountService.mgr.signinRedirect({state: 'some data’}). then (function () {*

*console.log ("signin successful");*

*}). catch (function (err) {*

*console.log ("error in signin redirect");*

*});*

### Call back after Redirect

*accountService.mgr.signinRedirectCallback(). then (function (user) {*

*\_self.userDetails = user;*

*sessionStorage.AccessToken = user.access\_token;*

*sessionStorage.TokenType = user.token\_type || "Bearer";*

*sessionStorage.IdToken = user.id\_token;*

*if (sessionStorage.AccessToken != null && sessionStorage.AccessToken != undefined) {*

*}*

*}, function (error) {*

*console.log ("Error in signin redirect callback: " + error);*

*});*

## Evolve Client

Evolve application is a combination of webapp and web service, web app is used for front-end and web service is used as a back-end basically it is used for all the API calls that are going from front-end.

### Evolve WebApp integration to Identity Server

In order to integrate Evolve webapp with identity server we have used

"oidc-client" version "1.4.1"

oidc-client is a library to provide OpenID Connect (OIDC) and OAuth2 protocol support for client-side, browser-based JavaScript client applications. Also included support for user session and access token management.

as soon as a user is logged in, the library takes care of creating the hidden iframe from the OIDC IdP, and polls it at a regular and configurable interval to be aware of a potential change in the user session.

### Installation

**Node.js**

Node.js v4.4 or later required.

**NPM**

npm install oidc-client –save

### Configuration

In Evolve the identity server configuration setup is done in the account service js file as shown below

*settings: any = {*

*authority: "*[*https://localhost/IdentityServer/ids*](https://localhost/IdentityServer/ids)*",*

*client\_id: "Evolve",*

*redirect\_uri: "*[*http://localhost:4200/*](http://localhost:4200/)*",*

*post\_logout\_redirect\_uri: "*[*http://localhost:4200/*](http://localhost:4200/)*",*

*response\_type: 'id\_token token',*

*scope: 'openid EvolveApi ProgressApi',*

*silent\_redirect\_uri: "*[*http://localhost:4200/silentrenew*](http://localhost:4200/silentrenew)*",*

*automaticSilentRenew: true,*

*accessTokenExpiringNotificationTime: 1,*

*//silentRequestTimeout:1,*

*filterProtocolClaims: true,*

*loadUserInfo: true,*

*acr\_values: ""*

*};*

*// `UserManager` is the main class exposed by the library*

*this.mgr = new UserManager(this.settings);*

*//After successful login this event triggered with the user details*

*this.mgr.events.addUserLoaded((user) => {*

*this.currentUser = user;*

*this.loggedIn = !(user === undefined);*

*sessionStorage.setItem("userData", user);*

*});*

*//Triggered when user is logged out*

*this.mgr.events.addUserUnloaded((e) => {*

*this.loggedIn = false;*

*});*

*//Triggered when user is logged out*

*this.mgr.events.addUserSignedOut((e) => {*

*this.startSignoutMainWindow();*

*});*

*// Sign out function*

*startSignoutMainWindow() {*

*this.mgr = this.configure();*

*sessionStorage.clear();*

*return this.mgr.signoutRedirect({ id\_token\_hint: user.id\_token }).then(resp => {*

*});*

*}).catch(function (err) {*

*});*

*};*

*// Sign out callback function*

*endSignoutMainWindow() {*

*this.mgr.signoutRedirectCallback().then(function (resp) {*

*}).catch(function (err) {*

*});*

*};*

*// Silent renew function called when token is about to expire*

*silentCallBack() {*

*this.mgr = this.configure();*

*this.mgr.signinSilentCallback();*

*this.mgr.signinRedirectCallback().then(function (user) {*

*user = user;*

*sessionStorage.setItem("Access\_token", user.access\_token);*

*}).catch(function (err) {*

*});*

*}*

### Important options on Settings

* **authority** – Value is the Identity Server url.
* **client\_id** – Value is the id that is declared as client id in the identity server Client XML for particular application.
* **redirect\_uri** – Value is the url of the application route to which it has to be redirected after successful token grant.
* **post\_logout\_redirect\_uri** – Value is the url of the application route to which it has to be redirected after successful logout.
* **response\_type** – Value is the type of response that the application receives after log in, it has to be same what it has declared in the identity server client XML.
* **scope** – Value is the scope value of the application that is declared in the identity server.

The below code snippet can be called from where the identity server call has to happen that is redirecting to the login page of identity server once the login is successful the identity server uses the redirect url to redirect to the application page and here it should contain a call to call back function in order get the user token information.

### Redirect to Identity server

*startSigninMainWindow() {*

*this.mgr.signinRedirect({ data: 'some data' }).then(function () {*

*}). catch (function (err) {*

*});*

*}*

### Call back after Redirect

*endSigninMainWindow(): Observable<any> {*

*this.mgr = this.configure();*

*return Observable.fromPromise(this.mgr.signinRedirectCallback().then(function (user) {*

*user = user;*

*sessionStorage.setItem("Access\_token", user.access\_token);*

*return user;*

*}).catch(function (err) {*

*return err;*

*}));*

*}*

# Veriqua and Evolve WebService integration to Identity Server

The Web API's that need to be added before starting are as follows:

* Microsoft.Owin.Host.SystemWeb
* Microsoft.Aspnet.WebApi.Owin
* IdentityServer3.AccessTokenValidation
* IdentityModel

We want to secure our API using IdentityServer - two things are needed for that:

* accept only tokens issued by IdentityServer
* accept only tokens that are issued for our API - for that we’ll give the API a name of sampleApi (also called scope)

The typical startup.cs for our configuration lokks as below

*public partial class Startup*

*{*

*public void Configuration (IAppBuilder app)*

*{*

*ConfigureAuth(app);*

*}*

*}*

In order to set up identity server for our Web API we need to add a Startup.Auth.cs under App\_Start Folder of our application.

*public void Configuration (IAppBuilder app)*

*{*

*app.UseIdentityServerBearerTokenAuthentication(new IdentityServerBearerTokenAuthenticationOptions*

*{*

*Authority = ConfigurationManager.AppSettings["Authority\_URL"].ToString(),*

*// For access to the introspection endpoint*

*ClientId = "Client\_ID",*

*RequiredScopes = new[] { "Client\_Scope" },*

*NameClaimType = "name",*

*RoleClaimType = "role"*

*});*

*}*

Where,

"Authority\_URL" - is the url of the Identity Server here it is "<https://localhost/IdentityServer/ids>"

"Client\_ID” - is the ID for the Client, which is registered in the Identity Server for Progress it is "Progress" and for evolve it is "Evolve".

"Client\_Scope" - is the Scope for the Client Application which is registered in the Identity Server for Progress it is "ProgressApi" and for evolve it is "EvolveApi".

The final step for implementation requires us to create an Application OAuth Provider. This is a default implementation that outlines how to authenticate and login a user through the process.

*public class ApplicationOAuthProvider : OAuthAuthorizationServerProvider*

*{*

*private readonly string \_publicClientId;*

*public ApplicationOAuthProvider(string publicClientId)*

*{*

*if (publicClientId == null)*

*{*

*throw new ArgumentNullException(nameof(publicClientId));*

*}*

*\_publicClientId = publicClientId;*

*}*

*public override async Task*

*GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)*

*{*

*var userManager = context.OwinContext.GetUserManager<ApplicationUserManager>();*

*var user = await userManager.FindAsync(context.UserName, context.Password);*

*if (user == null)*

*{*

*context.SetError("invalid\_grant", "The user name or password is incorrect.");*

*return;*

*}*

*ClaimsIdentity oAuthIdentity = await user.GenerateUserIdentityAsync(userManager, OAuthDefaults.AuthenticationType);*

*ClaimsIdentity cookiesIdentity = await user.GenerateUserIdentityAsync(userManager, CookieAuthenticationDefaults.AuthenticationType);*

*AuthenticationProperties properties = CreateProperties(user.UserName);*

*AuthenticationTicket ticket = new AuthenticationTicket(oAuthIdentity, properties);*

*context.Validated(ticket);*

*context.Request.Context.Authentication.SignIn(cookiesIdentity);*

*}*

*public override Task TokenEndpoint(OAuthTokenEndpointContext context)*

*{*

*foreach (KeyValuePair<string, string> property in context.Properties.Dictionary)*

*{*

*context.AdditionalResponseParameters.Add(property.Key, property.Value);*

*}*

*return Task.FromResult<object>(null);*

*}*

*public override Task ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)*

*{*

*// Resource owner password credentials does not provide a client ID.*

*if (context.ClientId == null)*

*{*

*context.Validated();*

*}*

*return Task.FromResult<object>(null);*

*}*

*public override Task ValidateClientRedirectUri(OAuthValidateClientRedirectUriContext context)*

*{*

*if (context.ClientId == \_publicClientId)*

*{*

*Uri expectedRootUri = new Uri(context.Request.Uri, "/");*

*if (expectedRootUri.AbsoluteUri == context.RedirectUri)*

*{*

*context.Validated();*

*}*

*}*

*return Task.FromResult<object>(null);*

*}*

*public static AuthenticationProperties CreateProperties(string userName)*

*{*

*IDictionary<string, string> data = new Dictionary<string,>*

*{*

*{ "userName", userName }*

*};*

*return new AuthenticationProperties(data);*

*}*

*}*

# Identity Manager

Identity Manager is a tool for developers and/or administrators to manage the identity information for users of their applications. This includes creating users, editing user information (passwords, email, claims, etc.) and deleting users. It provides a modern replacement for the ASP.NET WebSite Administration tool that used to be built into Visual Studio.

Here it is included as one of the Client for Identity Server.

Identity Manager has its own scope that allows it to directly manage the user info on the identity server.

## Operations that can be performed on Identity Manager

* Registering and managing user information
* Creating roles which can to added to the user
* adding or deleting roles for the user

# Setting Up Identity Server and Identity Manager on Development environment

## IIS Configuration

To start with, we have to create a self-signed certificate for the website hosting all the applications. We can use the open source tool [itiverba](http://www.itiverba.com/en/software/itisscg.php) to generate a self-signed certificate. Please follow the instruction provided in the link to generate certificate.

To Import certificate to IIS,

1. Open **Internet Information Services Manager (IIS)**
2. Select the name of Server
3. Under IIS section, double-click on **Server Certificates**
4. **Click on Import**
5. **Browse the certificate file (\*.pfx) and enter the password**
6. **Click OK**

**To add the https binding to the site,**

1. **Select Site in IIS**
2. **Click on Bindings**
3. **Add Binding**
4. **Select https as Type and IP Address**
5. Select the Server certificate imported
6. Click **OK**

Restart the IIS Website to take the new changes

## Identity Server Setup

Get the latest IdentityServer git repository and publish the HSOIdentityServer project from Visual Studio IDE

1. Choose Server localhost
2. Choose Site Name as *Default Web Site/IdentityServer*
3. Choose Build Configuration *Debug*
4. Click on *Publish*

## Identity Manager Setup (Optional)

This is Backend admin tool for maintaining Users and Roles. To setup IdentityManager, Get the latest HSOIdentityServer git repository and publish the HSOIdentityManager.Host project from Visual Studio IDE

1. Choose Server localhost
2. Choose Site Name as *Default Web Site/IdentityManager*
3. Choose Build Configuration *Debug*
4. Click on *Publish*

## Application Web.config configuration

To setup local Evolve or Progress with IDS, few web.config configuration has to be checked, it should sync with the content stored client tables of IdentityServer.

WebApp/Portal

<add key="AuthorityURL" value="https://localhost/IdentityServer/ids" />

<add key="ClientID" value="Progress" />

<add key="Redirect\_URL" value="https://localhost/veriqua/welcome" />

<add key="Silent\_Redirect\_URL" value="https://localhost/veriqua/silentrenew.html" />

<add key="Post\_Logout\_Redirect\_URL" value="https://localhost/veriqua/Login" />

<add key="Identity\_Server\_Call\_URL" value="https://localhost/IdentityServer/" />

Webservice

<add key="Progress\_AuthorityURL" value="https://localhost/IdentityServer/ids" />

<add key="Identity\_Service\_Call" value="https://localhost/IdentityServer" />