
IdentityServer4 Documentation

Release 1.0.0

Brock Allen, Dominick Baier

September 03, 2016

Introduction

1 Authentication as a Service 3

2 Single Sign-on / Sign-out 5

3 Access Control for APIs 7

4 Federation Gateway 9

5 Focus on Customization 11
  5.1 The big Picture 11
  5.2 Terminology 13
  5.3 Supported Specifications 15
  5.4 Packaging and Builds 15
  5.5 Defining Scopes 16
  5.6 Defining Clients 17
  5.7 Scope 19
  5.8 Client 20

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core.

It enables the following features in your applications:

CHAPTER 1

Authentication as a Service

Centralized login logic and workflow for all of your applications (web, native, mobile, services).

CHAPTER 2

Single Sign-on / Sign-out

Single sign-on (and out) over multiple application types.

CHAPTER 3

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.

CHAPTER 4

Federation Gateway

Support for external identity providers like Azure Active Directoy, Google, Facebook etc. This shields your applicatios from the details of how to connect to these external providers.

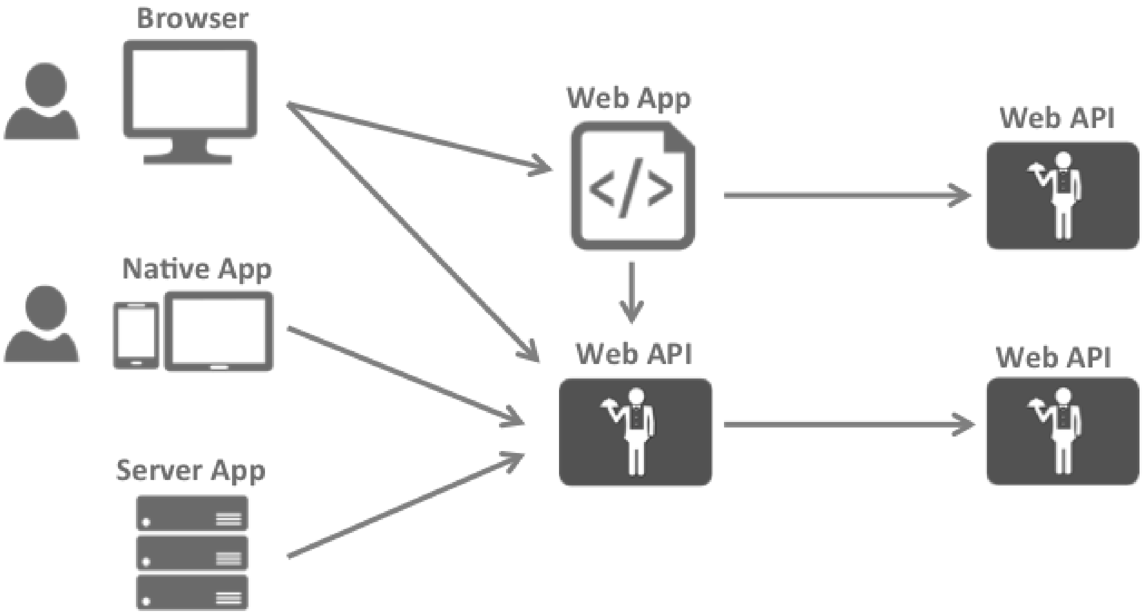
CHAPTER 5

Focus on Customization

The most important part - many aspect of IdentityServer can be customized to fit **your** needs. Since IdentityServer is a framework and not a boxed product or a SaaS, you can write code to adapt the system the way it makes sense for your scenarios.

5.1 The big Picture

Most modern applications look more or less like this:



The typical interactions are:

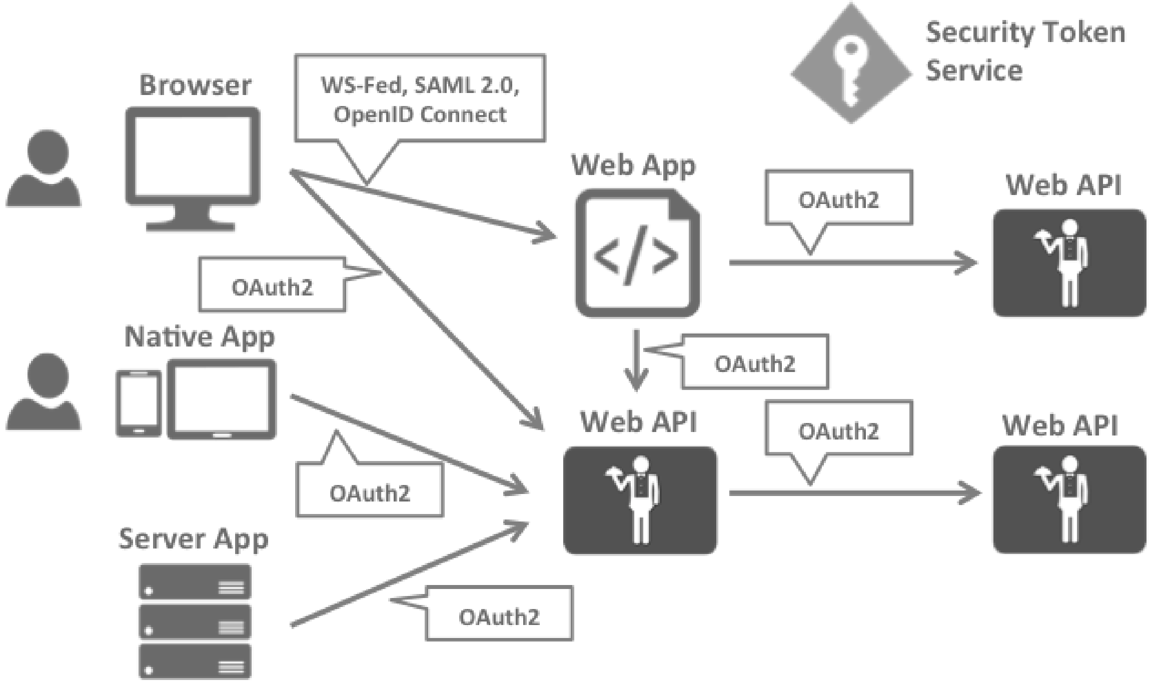
- Browsers communicate with web applications
- Web applications communicate with web APIs (sometimes on their own, sometimes on behalf of a user)
- Browser-based applications communicate with web APIs
- Native applications communicate with web APIs
- Server-based applications communicate with web APIs

- 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.

5.1.1 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.

5.1.2 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.

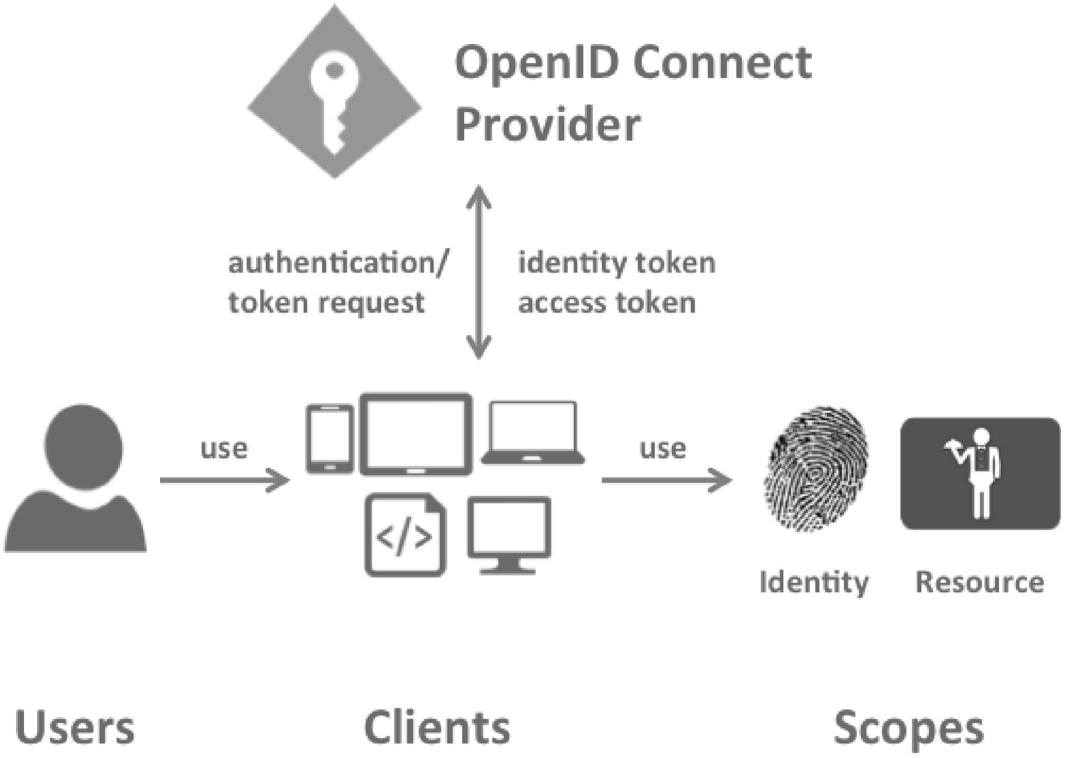
5.1.3 OpenID Connect and OAuth2 – better together

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.

5.2 Terminology

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



5.2.1 OpenID Connect Provider (OP)

IdentityServer 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.

IdentityServer 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

5.2.2 Client

A client is a piece of software that requests tokens from IdentityServer - 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.

5.2.3 User

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

5.2.4 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 flavours.

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 and you can create your own scopes in IdentityServer 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.

5.2.5 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.

5.2.6 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.

5.2.7 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.

5.3 Supported Specifications

IdentityServer implements the following specifications:

- OpenID Connect Core 1.0 (spec)
- OpenID Connect Discovery 1.0 (spec)
- OpenID Connect Session Management 1.0 - draft 22 (spec)
- OpenID Connect HTTP-based Logout 1.0 - draft 03 (spec)
- OAuth 2.0 (RFC 6749)
- OAuth 2.0 Bearer Token Usage (RFC 6750)
- OAuth 2.0 Multiple Response Types (spec)
- OAuth 2.0 Form Post Response Mode (spec)
- OAuth 2.0 Token Revocation (RFC 7009)
- OAuth 2.0 Token Introspection (RFC 7662)

5.4 Packaging and Builds

IdentityServer consists of a number of nuget packages.

5.4.1 IdentityServer4

nuget | github

Contains the core IdentityServer object model, services and middleware. Only contains support for in-memory configuration and user stores - but you can plug-in support for other stores via the configuration. This is what the other repos and packages are about.

5.4.2 Access token validation middleware

nuget | github

ASP.NET Core middleware for validating tokens in APIs. Provides an easy way to validate access tokens (both JWT and reference) and enforce scope requirements.

5.4.3 Dev builds

In addition we publish dev/interim builds to MyGet. Add the following feed to your Visual Studio if you want to give them a try:

https://www.myget.org/F/identity/

5.5 Defining Scopes

The first thing you typically define in your system are the resources that you want to protect. That could be identity information of your users like profile data or email addresses or access to APIs.

Note: At runtime, scopes are retrieved via an implementation of the `IScopeStore`. This allows loading them from arbitrary data sources like config files or databases. For this document we gonna use the in-memory version of the scope store. You can wire up the in-memory store in `ConfigureServices` via the `AddInMemoryScopes` extensions method.

5.5.1 Defining the minimal scope for OpenID Connect

OpenID Connect requires a scope with a name of *openid*. Since this scope is defined in the OIDC specification, we have built-in support for it via the *StandardScopes* class.

Alls our samples define a class called *Scopes* with a method called *Get*. In this method you simply return a list of scopes you want to support in your identityserver. This list will be later used to configure the identityserver service:

```
public class Scopes
{
    public static IEnumerable<Scope> Get()
    {
        return new List<Scope>
        {
            StandardScopes.OpenId
        };
    }
}
```

The *StandardScopes* class supports all scopes defined in the specification (openid, email, profile, address and offline_access). If you want to support them all, you can add them to your list of supported scopes:

```
public class Scopes
{
    public static IEnumerable<Scope> Get()
    {
        return new List<Scope>
        {
            StandardScopes.OpenId,
            StandardScopes.Profile,
            StandardScopes.Email,
            StandardScopes.Address,
            StandardScopes.OfflineAccess
        };
    }
}
```

5.5.2 Defining custom identity scopes

You can also define custom identity scopes. Create a new *Scope* class, give it a name and a display name and define which user claims should be included in the identity token when this scope gets requested:

```
new Scope
{
    Name = "tenant.info",
    DisplayName = "Tenant Information",
    Type = ScopeType.Identity,

    Claims = new List<ScopeClaim>
    {
        new ScopeClaim("tenantid"),
        new ScopeClaim("subscriptionid")
    }
}
```

Add that scope to your list of supported scopes.

5.5.3 Defining scopes for APIs

To get access tokens for APIs, you also need to register them as a scope. This time the scope type is of type *Resource*:

```
new Scope
{
    Name = "api1",
    DisplayName = "API #1",
    Description = "API 1",
    Type = ScopeType.Resource
}
```

If you don't define any scope claims, the access token will contain the subject ID of the user (if present), the client ID and the scope name.

You can also add additional user claims to the token by defining scope claims as shown above.

5.6 Defining Clients

Clients represent applications that can request tokens from your identityserver.

The details vary, but you typically define the following common settings for a client:

- a unique client ID
- a secret if needed
- the allowed interactions with the token service (called a grant type)
- a network location where identity and/or access token gets sent to (called a redirect URI)
- a list of scopes (aka resources) the client is allowed to access

Note: At runtime, clients are retrieved via an implementation of the `IClientStore`. This allows loading them from arbitrary data sources like config files or databases. For this document we gonna use the in-memory version of the client store. You can wire up the in-memory store in `ConfigureServices` via the `AddInMemoryClients` extensions method.

5.6.1 Defining a client for server to server communication

In this scenario no interactive user is present - a service (aka client) wants to communicate with an API (aka scope):

```
public class Clients
{
    public static IEnumerable<Client> Get()
    {
        return new List<Client>
        {
            new Client
            {
                ClientId = "service.client",
                ClientSecrets = new List<Secret>
                {
                    new Secret("secret".Sha256())
                },

                AllowedGrantTypes = GrantTypes.ClientCredentials,

                AllowedScopes = new List<string>
                {
                    "api1", "api2"
                }
            };
        }
    }
}
```

5.6.2 Defining browser-based JavaScript client (e.g. SPA) for user authentication and delegated access and API

This client uses the so called implicit flow to request an identity and access token from JavaScript:

```
var jsClient = new Client
{
    ClientId = "js",
    ClientName = "JavaScript Client",
    ClientUri = "http://identityserver.io",

    AllowedGrantTypes = GrantTypes.Implicit,
    AllowAccessTokensViaBrowser = true,

    RedirectUris = new List<string>
    {
        "http://localhost:7017/index.html",
    },
    PostLogoutRedirectUris = new List<string>
    {
        "http://localhost:7017/index.html",
    },
    AllowedCorsOrigins = new List<string>
    {
        "http://localhost:7017"
    },

    AllowedScopes = new List<string>
    {
        StandardScopes.OpenId.Name,
        StandardScopes.Profile.Name,
        StandardScopes.Email.Name,
        "api1", "api2"
    },
};
```

5.6.3 Defining a server-side web application (e.g. MVC) for use authentication and delegated API access

Interactive server side (or native desktop/mobile) applications use the hybrid flow. This flow gives you the best security because the access tokens are transmitted via back-channel calls only (and gives you access to refresh tokens):

```
var mvcClient = new Client
{
    ClientId = "mvc",
    ClientName = "MVC Client",
    ClientSecrets = new List<Secret>
    {
        new Secret("secret".Sha256())
    },
    ClientUri = "http://identityserver.io",

    AllowedGrantTypes = GrantTypes.Hybrid,

    RedirectUris = new List<string>
    {
        "http://localhost:21402/signin-oidc"
    },
    PostLogoutRedirectUris = new List<string>
    {
        "http://localhost:21402/"
    },
    LogoutUri = "http://localhost:21402/signout-oidc",

    AllowedScopes = new List<string>
    {
        StandardScopes.OpenId.Name,
        StandardScopes.Profile.Name,
        StandardScopes.OfflineAccess.Name,

        "api1", "api2",
    },
};
```

5.7 Scope

The *Scope* class models a resource in your system.

- ***Enabled*** Indicates if scope is enabled and can be requested. Defaults to *true*.
- ***Name*** The unique 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*** Adds a secret to scope for accessing the the introspection endpoint - see also [here](secrets.html).
- ***AllowUnrestrictedIntrospection*** 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*.

5.8 Client

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 - credentials to access the token endpoint.
- ***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*.
- ***AllowedGrantTypes*** Specifies the grant types the client is allowed to use. Use the *GrantTypes* class for common combinations.
- ***RedirectUris*** Specifies the allowed URIs to return tokens or authorization codes to
- ***PostLogoutRedirectUris*** Specifies allowed URIs to redirect to after logout
- ***LogoutUri*** Specifies logout URI at client for HTTP based logout
- ***LogoutSessionRequired*** Specifies if the user's session id should be sent to the LogoutUri. Defaults to true.

- ***RequireSignOutPrompt*** 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*** 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.
- ***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 *jti* claim).
- ***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*.