# IdentityServer4 Documentation

***Release 1.0.0***

**Brock Allen, Dominick Baier**

February 01, 2017

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 Directory, Google, Facebook etc. This shields your applications 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.

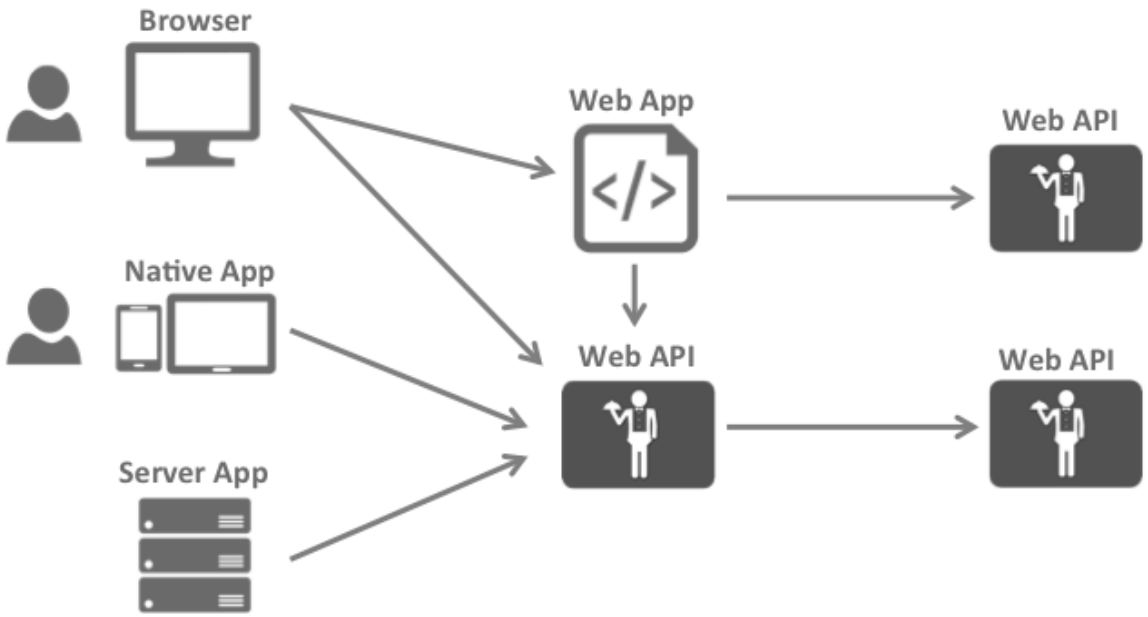
CHAPTER 6

# Free and Commercial Support

If you need help building or running your identity platform, *let us know*. There are several way we can help you out. IdentityServer is officially certified by the OpenID Foundation and part of the .NET Foundation.

## 6.1 The big Picture

Most modern applications look more or less like this:




The typical interactions are:

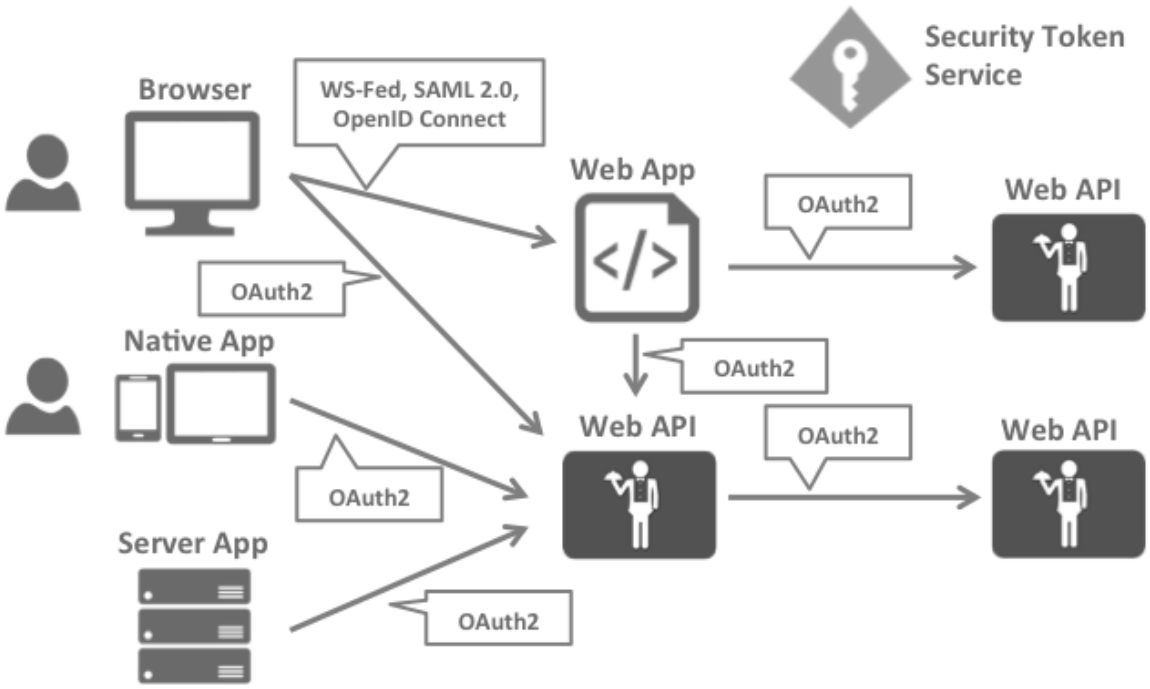
- Browsers communicate with web applications
- Web applications communicate with web APIs (sometimes on their own, sometimes on behalf of a user)
- Browser-based applications communicate with web APIs
- Native applications communicate with web APIs
- Server-based applications communicate with web APIs

- 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.

### 6.1.1 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.

### 6.1.2 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.

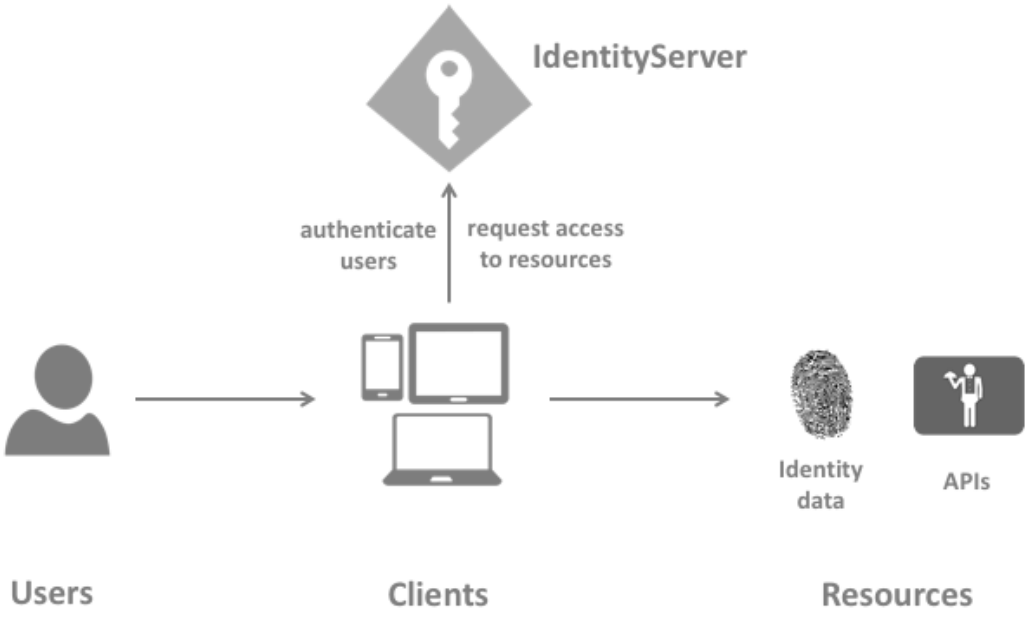
### 6.1.3 OpenID Connect and OAuth 2.0 – better together

OpenID Connect and OAuth 2.0 are very similar – in fact OpenID Connect is an extension on top of OAuth 2.0. 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 OAuth 2.0 is the best approach to secure modern applications for the foreseeable future. IdentityServer4 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.

## 6.2 Terminology

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




### 6.2.1 IdentityServer

IdentityServer is an OpenID Connect provider - it implements the OpenID Connect and OAuth 2.0 protocol.

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:

- protect your resources

- 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

### 6.2.2 User

A user is a human that is using a registered client to access resources.

### 6.2.3 Client

A client is a piece of software that requests tokens from IdentityServer - either for authenticating a user (requesting an identity token) or for accessing a resource (requesting an access token). A client must be first registered with IdentityServer before it can request tokens.

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

### 6.2.4 Resources

Resources are something you want to protect with IdentityServer - either identity data of your users, or APIs.

Every resource has a unique name - and clients use this name to specify to which resources they want to get access to.

**Identity data** Identity information (aka claims) about a user, e.g. name or email address.

**APIs** APIs resources represent functionality a client wants to invoke - typically modelled as Web APIs, but not necessarily.

### 6.2.5 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) and information about how and when the user authenticated. It can contain additional identity data.

### 6.2.6 Access Token

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

## 6.3 Supported Specifications

IdentityServer implements the following specifications:

### 6.3.1 OpenID Connect

- 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)

### 6.3.2 OAuth 2.0

- 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)
- Proof Key for Code Exchange (RFC 7636)

## 6.4 Packaging and Builds

IdentityServer consists of a number of nuget packages.

### 6.4.1 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.

### 6.4.2 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.

### 6.4.3 ASP.NET Core Identity

nuget | github

ASP.NET Core Identity integration package for IdentityServer. This package provides a simple configuration API to use the ASP.NET Identity managament library for your IdentityServer users.

### 6.4.4 EntityFramework Core

nuget | github

EntityFramework Core storage implementation for IdentityServer. This package provides an EntityFramework implementation for the configuration and operational stores in IdentityServer.

### 6.4.5 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/

## 6.5 Support and Consulting Options

We have several free and commercial support and consulting options for IdentityServer.

### 6.5.1 Free support

Free support is community-based and uses public forums

**StackOverflow**

There's an ever growing community of people using IdentityServer that monitor questions on StackOverflow. If time permits, we also try to answer as many questions as possible

You can subscribe to all IdentityServer4 related questions using this feed:

https://stackoverflow.com/questions/tagged/?tagnames=identityserver4&sort=newest

Please use the `IdentityServer4` tag when asking new questions

**Reporting a bug**

If you think you have found a bug or unexpected behavior, please open an issue on the Github issue tracker. We try to get back to you ASAP. Please understand that we also have day jobs, and might be to busy to reply immediately.

Also check the contribution guidelines before posting.

### 6.5.2 Commercial support

Both Brock and I do consulting around identity & access control architecture in general, and IdentityServer in particular. Please get in touch with us to discuss possible options.

**Training**

Brock and Dominick are regularly doing workshops around identity & acess control for modern applications. Check the agenda and upcoming dates here.

**Production support in North America**

If you are looking for production support in North America - write us an email.

**Production support in Europe**

If you are looking for production support please visit http://identityserver.com

**Admin UI and appliance**

if you are interested in commercial products using IdentityServer - e.g. the new Admin API/UI or an appliance - check https://www.identityserver.com/upcoming-products.

## 6.6 Demo Server and Tests

You can try IdentityServer4 with your favourite client library. We have a test instance at demo.identityserver.io. On the main page you can find instructions on how to configure your client and how to call an API.

Furthermore we have a repo that exercises a variety of IdentityServer and Web API combinations (IdentityServer 3 and 4, ASP.NET Core and Katana). We use this test harness to make sure all permutations work. You can test it yourself by cloning this repo.

## 6.7 Setup and Overview

There are two fundamental ways to start a new IdentityServer project:

- start from scratch
- start with the ASP.NET Identity template in Visual Studio

If you start from scratch, we provide a couple of helpers and in-memory stores, so you don't have to worry about persistence right from the start.

If you start with ASP.NET Identity, we provide an easy way to integrate with that as well.

The quickstarts provide step by step instructions for various common identityserver scenarios. They start with the absolute basics and become more complex - it is recommended you do them in in order.

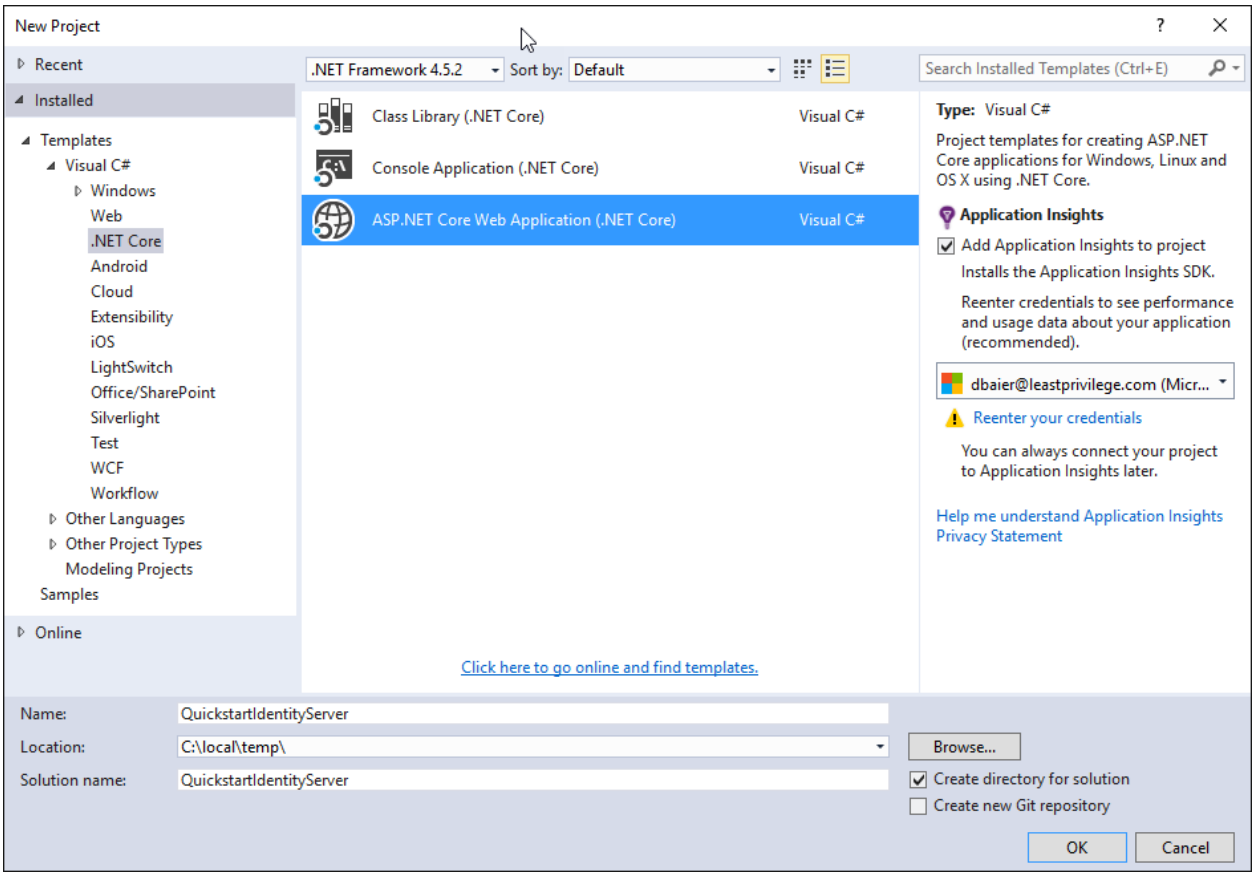
Every quickstart has a reference solution - you can find the code in the IdentityServer4.Samples repo in the quickstarts folder.

### 6.7.1 Basic setup

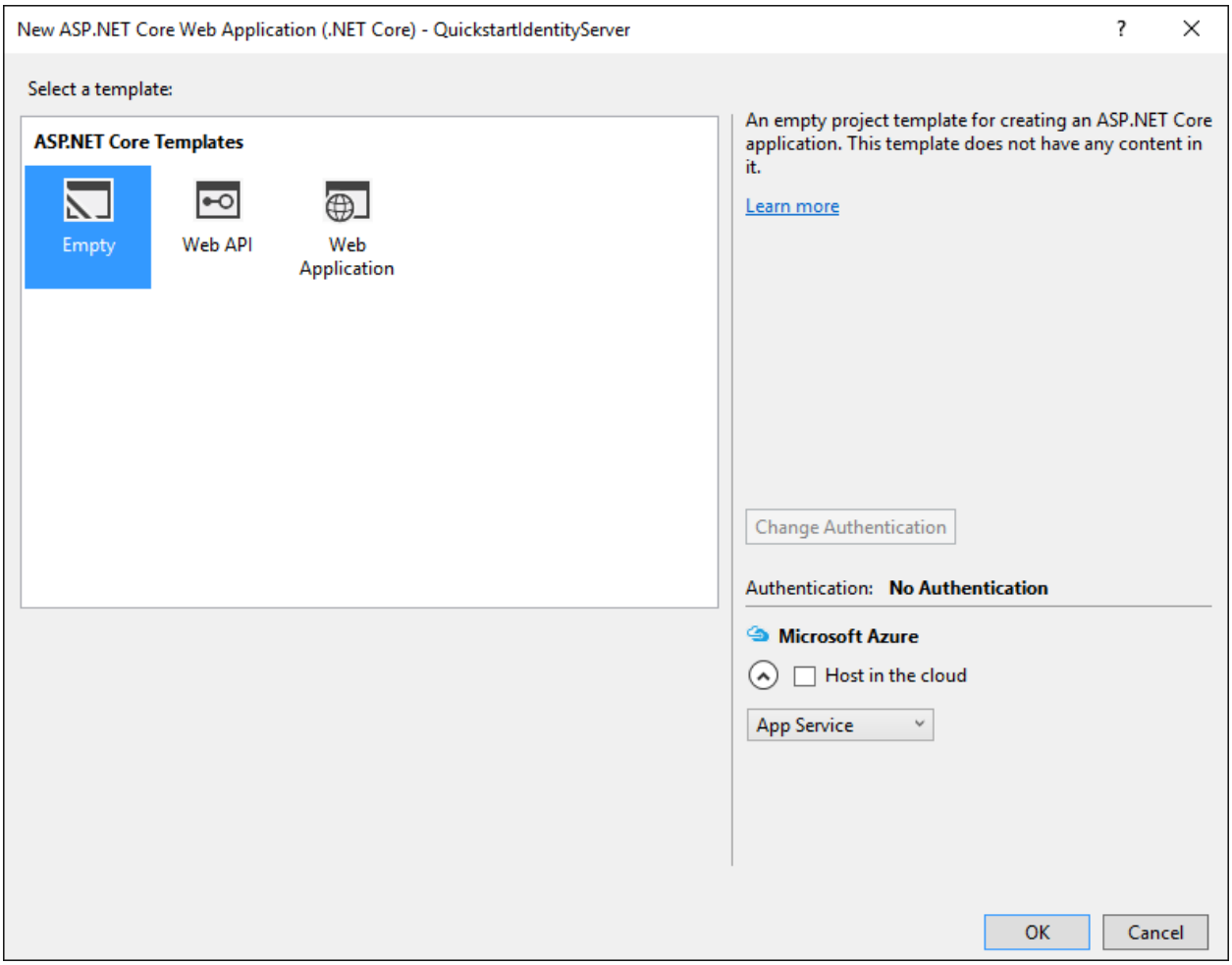
The screen shots show Visual Studio - but this is not a requirement.

**Creating the quickstart IdentityServer**

Start by creating a new ASP.NET Core project.




Then select the “Empty Web” option.




**Note:** IdentityServer currently only targets ASP.NET Core 1.1, so if you are starting with an ASP.NET Core 1.0 project then you can follow this guide for updating to ASP.NET Core 1.1.

Next, add the IdentityServer4 nuget package by adding the following line to your project.json under the ´dependencies´ property:

```
"IdentityServer4": "1.0.0"
```

Alternatively you can use Package Manager Console to add the dependency by running the following command:

> "Install-Package IdentityServer4"

IdentityServer uses the usual pattern to configure and add services to an ASP.NET Core host. In `ConfigureServices` the required services are configured and added to the DI system. In `Configure` the middleware is added to the HTTP pipeline.

Modify your `Startup.cs` file to look like this:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddIdentityServer()
            .AddTemporarySigningCredential();
```

```
    }

    public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
    {
        loggerFactory.AddConsole(LogLevel.Debug);
        app.UseDeveloperExceptionPage();

        app.UseIdentityServer();
    }
}
```

`AddIdentityServer` registers the IdentityServer services in DI. It also registers an in-memory store for runtime state. This is useful for development scenarios. For production scenarios you need a persistent or shared store like a database or cache for that. See the *EntityFramework* quickstart for more information.

The `AddTemporarySigningCredential` extension creates temporary key material for signing tokens on every start. Again this might be useful to get started, but needs to be replaced by some persistent key material for production scenarios. See the *cryptography docs* for more information.

---

**Note:** IdentityServer is not yet ready to be launched. In fact, when you try it, you should see an exception at startup time stating that services are missing. We will add those services in the following quickstarts.

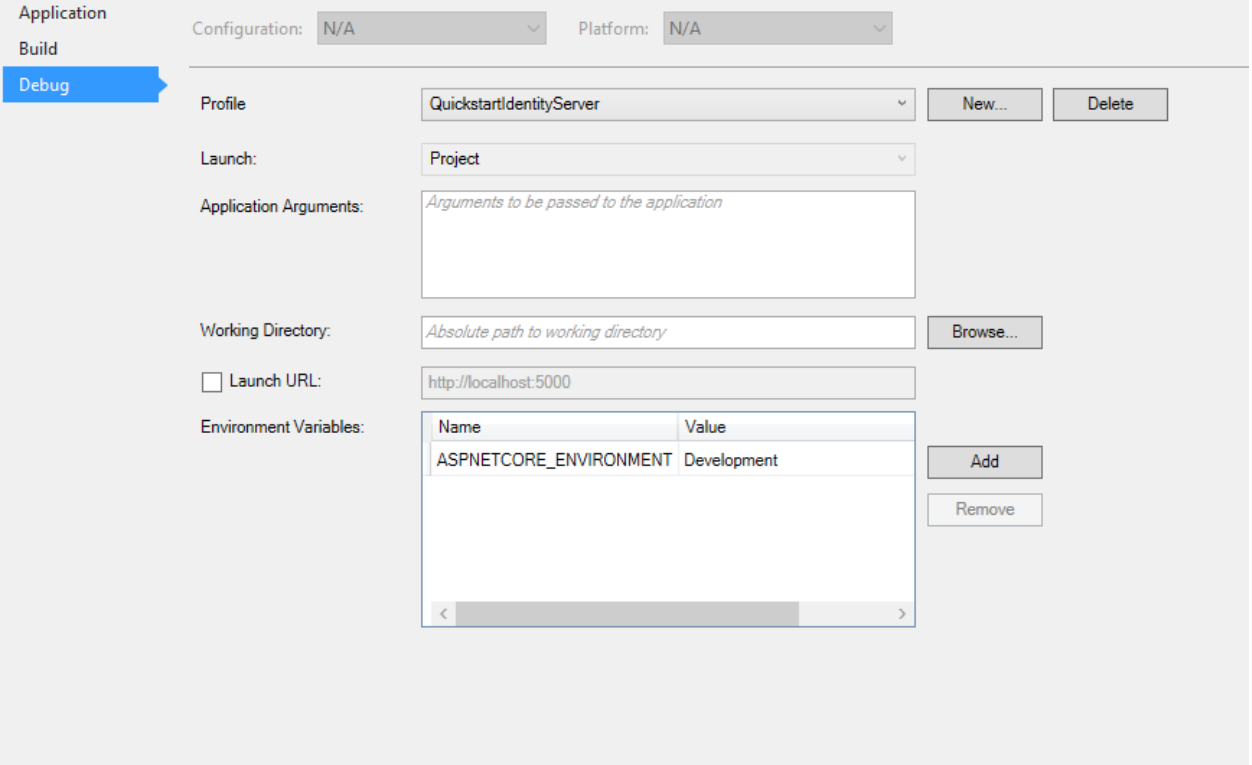
---

### 6.7.2 Modify hosting

By default Visual Studio uses IIS Express to host your web project. This is totally fine, besides that you won't be able to see the real time log output to the console.

IdentityServer makes extensive use of logging whereas the "visible" error message in the UI or returned to clients are deliberately vague.

We recommend to run IdentityServer in the console host. You can do this by switching the launch profile in Visual Studio. You also don't need to launch a browser every time you start IdentityServer - you can turn that off as well:




When you switch to self-hosting, the web server port defaults to 5000. You can configure this in `Program.cs` - we use the following configuration for the IdentityServer host in the quickstarts:

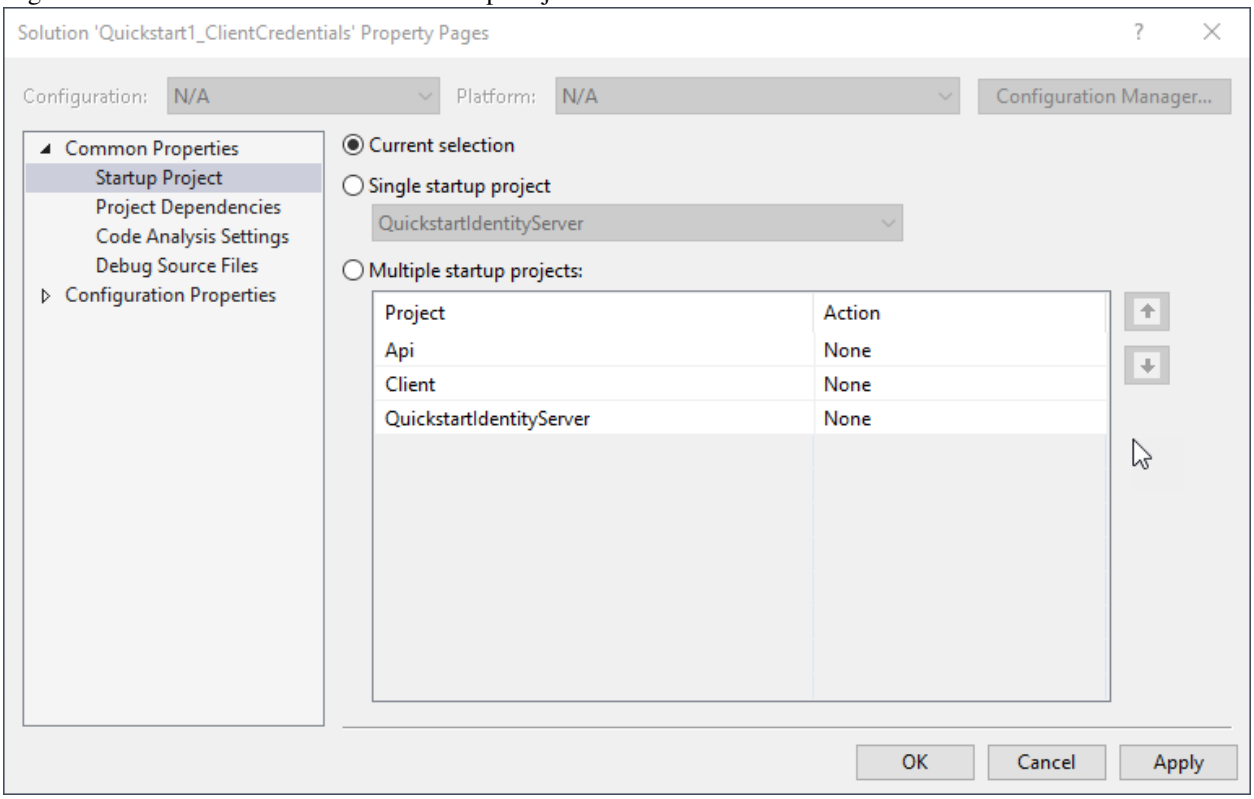
```
public class Program
{
    public static void Main(string[] args)
    {
        Console.Title = "IdentityServer";

        var host = new WebHostBuilder()
            .UseKestrel()
            .UseUrls("http://localhost:5000")
            .UseContentRoot(Directory.GetCurrentDirectory())
            .UseIISIntegration()
            .UseStartup<Startup>()
            .Build();

        host.Run();
    }
}
```

**Note:** We recommend to configure the same port for IIS Express and self-hosting. This way you can switch between the two without having to modify any configuration in your clients.

### 6.7.3 How to run the quickstart

As mentioned above every quickstart has a reference solution - you can find the code in the IdentityServer4.Samples repo in the quickstarts folder.

The easiest way to run the individual parts of a quickstart solution is to set the startup mode to "current selection". Right click the solution and select "Set Startup Projects":




Typically you start IdentityServer first, then the API, and then the client. Only run in the debugger if you actually want to debug. Otherwise Ctrl+F5 is the best way to run the projects.

## 6.8 Protecting an API using Client Credentials

This quickstart presents the most basic scenario for protecting APIs using IdentityServer.

In this scenario we will define an API and a client that wants to access it. The client will request an access token at IdentityServer and use it to gain access to the API.

### 6.8.1 Defining the API

Scopes define the resources in your system that you want to protect, e.g. APIs.

Since we are using the in-memory configuration for this walkthrough - all you need to do to add an API, is to create an object of type `ApiResource` and set the appropriate properties.

Add a file (e.g. `Config.cs`) into your project and add the following code:

```
public static IEnumerable<ApiResource> GetApiResources()
{
    return new List<ApiResource>
    {
        new ApiResource("api1", "My API")
    };
}
```

### 6.8.2 Defining the client

The next step is to define a client that can access this API.

For this scenario, the client will not have an interactive user, and will authenticate using the so called client secret with IdentityServer. Add the following code to your configuration:

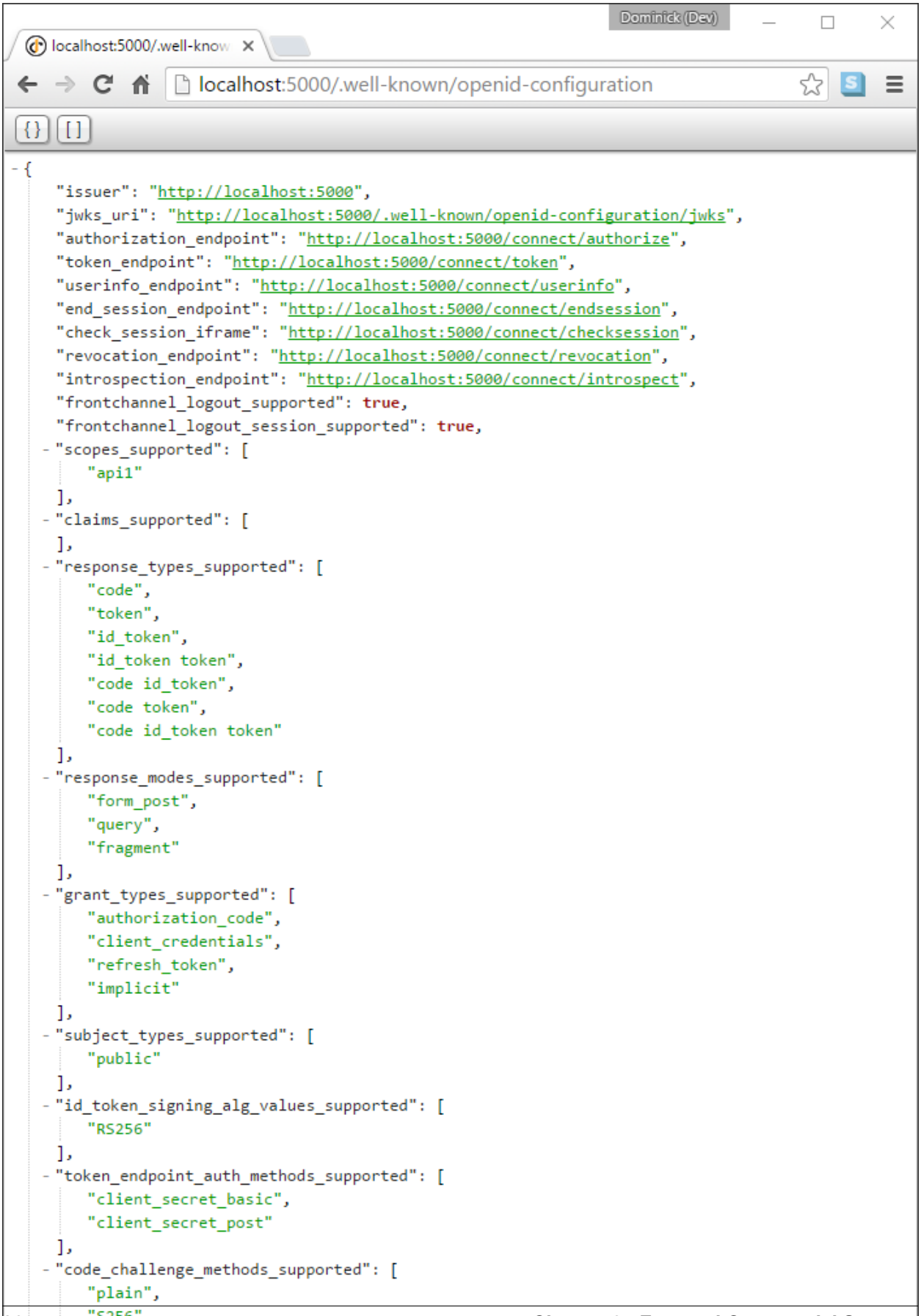
```
public static IEnumerable<Client> GetClients()
{
    return new List<Client>
    {
        new Client
        {
            ClientId = "client",

            // no interactive user, use the clientid/secret for authentication
            AllowedGrantTypes = GrantTypes.ClientCredentials,

            // secret for authentication
            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },

            // scopes that client has access to
            AllowedScopes = { "api1" }
        }
    };
}
```

### 6.8.3 Configure IdentityServer

To configure IdentityServer to use your scopes and client definition, you need to add code to the `ConfigureServices` method. You can use convenient extensions methods for that - under the covers these add the relevant stores and data into the DI system:

```
public void ConfigureServices(IServiceCollection services)
{
    // configure identity server with in-memory stores, keys, clients and resources
    services.AddIdentityServer()
        .AddTemporarySigningCredential()
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients());
}
```

That's it - if you run the server and navigate the browers to `http://localhost:5000/.well-known/openid-configurati` you should see the so-called discovery document. This will be used by your clients and APIs to download the necessary configuration data.


Dominick (Dev)
localhost:5000/.well-know
localhost:5000/.well-known/openid-configuration
{}
[]
{
"issuer": "http://localhost:5000",
"jwks_uri": "http://localhost:5000/.well-known/openid-configuration/jwks",
"authorization_endpoint": "http://localhost:5000/connect/authorize",
"token_endpoint": "http://localhost:5000/connect/token",
"userinfo_endpoint": "http://localhost:5000/connect/userinfo",
"end_session_endpoint": "http://localhost:5000/connect/endsession",
"check_session_iframe": "http://localhost:5000/connect/checksession",
"revocation_endpoint": "http://localhost:5000/connect/revocation",
"introspection_endpoint": "http://localhost:5000/connect/introspect",
"frontchannel_logout_supported": true,
"frontchannel_logout_session_supported": true,
"scopes_supported": [
"api1"
],
"claims_supported": [
],
"response_types_supported": [
"code",
"token",
"id_token",
"id_token token",
"code id_token",
"code token",
"code id_token token"
],
"response_modes_supported": [
"form_post",
"query",
"fragment"
],
"grant_types_supported": [
"authorization_code",
"client_credentials",
"refresh_token",
"implicit"
],
"subject_types_supported": [
"public"
],
"id_token_signing_alg_values_supported": [
"RS256"
],
"token_endpoint_auth_methods_supported": [
"client_secret_basic",
"client_secret_post"
],
"code_challenge_methods_supported": [
"plain",
"S256"
]
}

### 6.8.4 Adding an API

Next, add an API to your solution.

You can use the ASP.NET Core Web API template for that. Again, we recommend you take control over the ports and use the same technique as you used to configure Kestrel and the launch profile as before. This walkthrough assumes you have configured your API to run on `http://localhost:5001`.

**The controller**

Add a new controller to your API project:

```
[Route("identity")]
[Authorize]
public class IdentityController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        return new JsonResult(from c in User.Claims select new { c.Type, c.Value });
    }
}
```

This controller will be used later to test the authorization requirement, as well as visualize the claims identity through the eyes of the API.

**Configuration**

The last step is to add authentication middleware to your API host. The job of that middleware is:

- validate the incoming token to make sure it is coming from a trusted issuer
- validate that the token is valid to be used with this api (aka scope)

Add the following package to your project.json:

```
"IdentityServer4.AccessTokenValidation": "1.0.1"
```

You also need to add the middleware to your pipeline. It must be added **before** MVC, e.g.:

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    app.UseIdentityServerAuthentication(new IdentityServerAuthenticationOptions
    {
        Authority = "http://localhost:5000",
        RequireHttpsMetadata = false,

        ApiName = "api1"
    });

    app.UseMvc();
}
```

If you use the browser to navigate to the controller (`http://localhost:5001/identity`), you should get a 401 status code in return. This means your API requires a credential.

That's it, the API is now protected by IdentityServer.

### 6.8.5 Creating the client

The last step is to write a client that requests an access token, and then uses this token to access the API. For that, add a console project to your solution.

The token endpoint at IdentityServer implements the OAuth 2.0 protocol, and you could use raw HTTP to access it. However, we have a client library called IdentityModel, that encapsulates the protocol interaction in an easy to use API.

Add IdentityModel to your project.json:

```
"IdentityModel": "2.0.0"
```

IdentityModel includes a client library to use with the discovery endpoint. This way you only need to know the base-address of IdentityServer - the actual endpoint addresses can be read from the metadata:

```
// discover endpoints from metadata
var disco = await DiscoveryClient.GetAsync("http://localhost:5000");
```

Next you can use the `TokenClient` class to request the token. To create an instance you need to pass in the token endpoint address, client id and secret.

Next you can use the `RequestClientCredentialsAsync` method to request a token for your API:

```
// request token
var tokenClient = new TokenClient(disco.TokenEndpoint, "client", "secret");
var tokenResponse = await tokenClient.RequestClientCredentialsAsync("api1");

if (tokenResponse.IsError)
{
    Console.WriteLine(tokenResponse.Error);
    return;
}

Console.WriteLine(tokenResponse.Json);
```

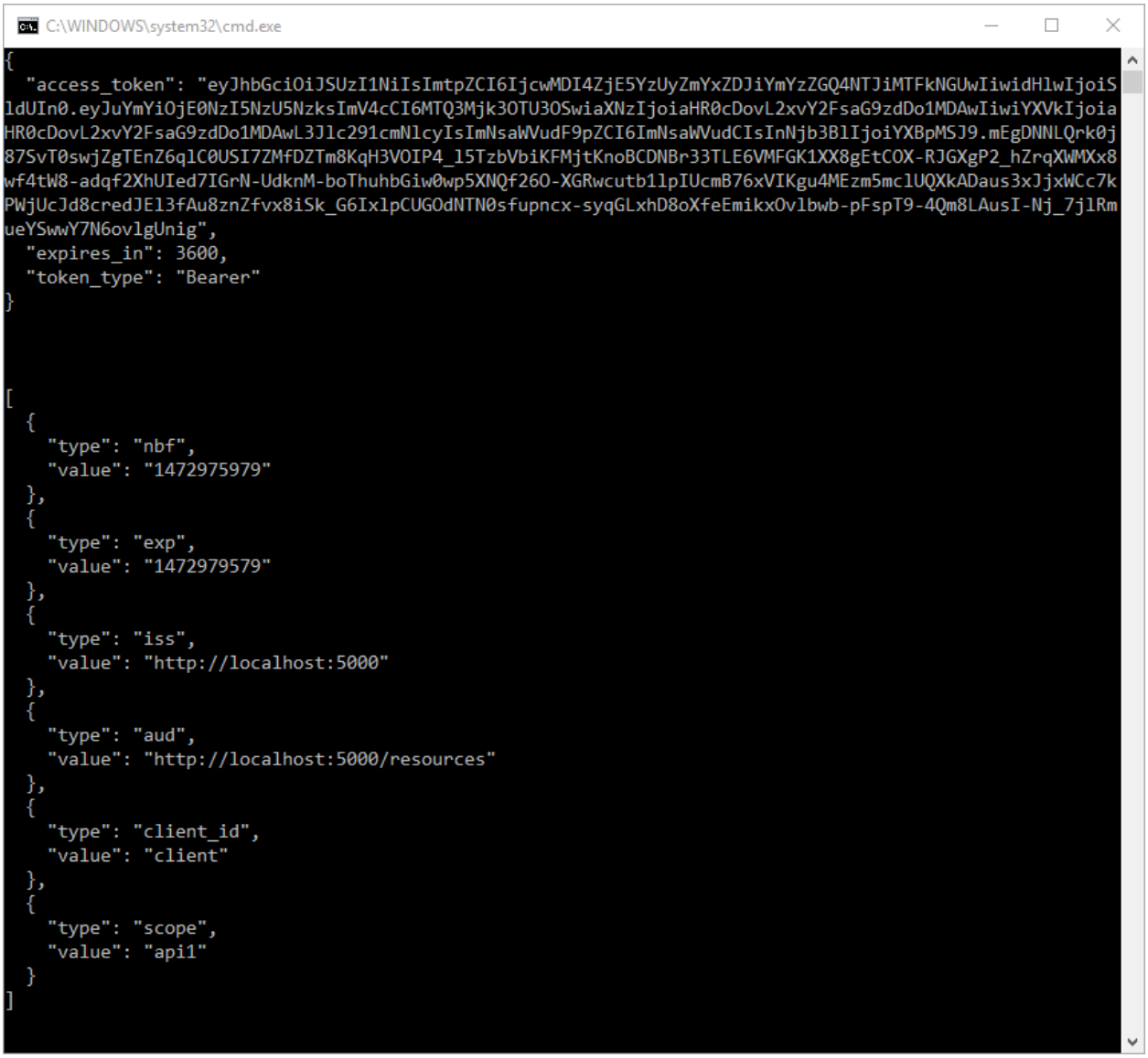
**Note:** Copy and paste the access token from the console to jwt.io to inspect the raw token.

The last step is now to call the API.

To send the access token to the API you typically use the HTTP Authorization header. This is done using the `SetBearerToken` extension method:

```
// call api
var client = new HttpClient();
client.SetBearerToken(tokenResponse.AccessToken);

var response = await client.GetAsync("http://localhost:5001/identity");
if (!response.IsSuccessStatusCode)
{
    Console.WriteLine(response.StatusCode);
}
else
{
    var content = await response.Content.ReadAsStringAsync();
    Console.WriteLine(JArray.Parse(content));
}
```

The output should look like this:




**Note:** By default an access token will contain claims about the scope, lifetime (nbf and exp), the client ID (client_id) and the issuer name (iss).

### 6.8.6 Further experiments

This walkthrough focused on the success path so far

- client was able to request token
- client could use the token to access the API

You can now try to provoke errors to learn how the system behaves, e.g.

- try to use an invalid client id or secret to request the token
- try to ask for an invalid scope during the token request

- don't send the token to the API
- configure the API to require a different scope than the one in the token

## 6.9 Protecting an API using Passwords

The OAuth 2.0 resource owner password grant allows a client to send username and password to the token service and get an access token back that represents that user.

The spec recommends using the resource owner password grant only for "trusted" (or legacy) applications. Generally speaking you are typically far better off using one of the interactive OpenID Connect flows when you want to authenticate a user and request access tokens.

Nevertheless, this grant type allows us to introduce the concept of users to our quickstart IdentityServer, and that's why we show it.

### 6.9.1 Adding users

Just like there are in-memory stores for resources (aka scopes) and clients, there is also one for users.

---

**Note:** Check the ASP.NET Identity based quickstarts for more information on how to properly store and manage user accounts.

---

The class `TestUser` represents a test user and its claims. Let's create a couple of users by adding the following code to our config class: First add the following using statement to the config.cs file: using IdentityServer4.Test;

```
public static List<TestUser> GetUsers() {
    return new List<TestUser> {
        new TestUser {
            SubjectId = "1", Username = "alice", Password = "password"
        }, new TestUser {
            SubjectId = "2", Username = "bob", Password = "password"
        }
    };
}
```

Then register the test users with IdentityServer:

```
public void ConfigureServices(IServiceCollection services)
{
    // configure identity server with in-memory stores, keys, clients and scopes
    services.AddIdentityServer()
        .AddTemporarySigningCredential()
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients())
        .AddTestUsers(Config.GetUsers());
}
```

The `AddTestUsers` extension method does a couple of things under the hood

- adds support for the resource owner password grant

- adds support to user related services typically used by a login UI (we'll use that in the next quickstart)
- adds support for a profile service based on the test users (you'll learn more about that in the next quickstart)

### 6.9.2 Adding a client for the resource owner password grant

You could simply add support for the grant type to our existing client by changing the `AllowedGrantTypes` property. If you need your client to be able to use both grant types that is absolutely supported.

Typically you want to create a separate client for the resource owner use case, add the following to your clients configuration:

```
public static IEnumerable<Client> GetClients()
{
    return new List<Client>
    {
        // other clients omitted...

        // resource owner password grant client
        new Client
        {
            ClientId = "ro.client",
            AllowedGrantTypes = GrantTypes.ResourceOwnerPassword,

            ClientSecrets =
            {
                new Secret("secret".Sha256())
            },
            AllowedScopes = { "api1" }
        }
    };
}
```

### 6.9.3 Requesting a token using the password grant

The client looks very similar to what we did for the client credentials grant. The main difference is now that the client would collect the user's password somehow, and send it to the token service during the token request.

Again IdentityModel's `TokenClient` can help out here:

```
// request token
var tokenClient = new TokenClient(disco.TokenEndpoint, "ro.client", "secret");
var tokenResponse = await tokenClient.RequestResourceOwnerPasswordAsync("alice", "password", "api1");

if (tokenResponse.IsError)
{
    Console.WriteLine(tokenResponse.Error);
    return;
}

Console.WriteLine(tokenResponse.Json);
Console.WriteLine("\n\n");
```

When you send the token to the identity API endpoint, you will notice one small but important difference compared to the client credentials grant. The access token will now contain a `sub` claim which uniquely identifies the user. This "sub" claim can be seen by examining the content variable after the call to the API and also will be displayed on the screen by the console application.

The presence (or absence) of the `sub` claim let's the API distinguish between calls on behalf of clients and calls on behalf of users.

## 6.10 Adding User Authentication with OpenID Connect

In this quickstart we want to add support for interactive user authentication via the OpenID Connect protocol to our IdentityServer.

Once that is in place, we will create an MVC application that will use IdentityServer for authentication.

### 6.10.1 Adding the UI

All the protocol support needed for OpenID Connect is already built into IdentityServer. You need to provide the necessary UI parts for login, logout, consent and error.

While the look & feel as well as the exact workflows will probably always differ in every IdentityServer implementation, we provide an MVC-based sample UI that you can use as a starting point.

This UI can be found in the Quickstart UI repo. You can either clone or download this repo and drop the controllers, views, models and CSS into your web application.

Alternatively you can run this command from the command line in your web application to automate the download:

```
iex ((New-Object System.Net.WebClient).DownloadString('https://raw.githubusercontent.com/IdentityServ
```

See the readme for the quickstart UI for more information.

---

**Note:** The `release` branch of the UI repo has the UI that matches the latest stable release. The `dev` branch goes along with the current dev build of IdentityServer4. If you are looking for a specific version of the UI - check the tags.

---

Spend some time inspecting the controllers and models, the better you understand them, the easier it will be to make future modifications. Most of the code lives in the "Quickstart" folder using a "feature folder" style. If this style doesn't suit you, feel free to organize the code in any way you want.

### 6.10.2 Creating an MVC client

Next you will add an MVC application to your solution. Use the ASP.NET Core "Web Application" template for that. Configure the application to use port 5002 (see the overview part for instructions on how to do that).

To add support for OpenID Connect authentication to the MVC application, add the following packages to project.json:

```
"Microsoft.AspNetCore.Authentication.Cookies": "1.0.0",
"Microsoft.AspNetCore.Authentication.OpenIdConnect": "1.0.0"
```

Next add both middlewares to your pipeline - the cookies one is simple:

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationScheme = "Cookies"
});
```

The OpenID Connect middleware needs slightly more configuration. You point it to your IdentityServer, specify a client ID and tell it which middleware will do the local signin (namely the cookies middleware). As well, we've turned

off the JWT claim type mapping to allow well-known claims (e.g. 'sub' and 'idp') to flow through unmolested. This "clearing" of the claim type mappings must be done before the call to *UseOpenIdConnectAuthentication()*:

```
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

app.UseOpenIdConnectAuthentication(new OpenIdConnectOptions
{
    AuthenticationScheme = "oidc",
    SignInScheme = "Cookies",

    Authority = "http://localhost:5000",
    RequireHttpsMetadata = false,

    ClientId = "mvc",
    SaveTokens = true
});
```

Both middlewares should be added before the MVC in the pipeline.

The last step is to trigger the authentication handshake. For that go to the home controller and add the `[Authorize]` on one of the actions. Also modify the view of that action to display the claims of the user, e.g.:

```
<dl>
    @foreach (var claim in User.Claims)
    {
        <dt>@claim.Type</dt>
        <dd>@claim.Value</dd>
    }
</dl>
```

If you now navigate to that controller using the browser, a redirect attempt will be made to IdentityServer - this will result in an error because the MVC client is not registered yet.

### 6.10.3 Adding support for OpenID Connect Identity Scopes

Similar to OAuth 2.0, OpenID Connect also uses the scopes concept. Again, scopes represent something you want to protect and that clients want to access. In contrast to OAuth, scopes in OIDC don't represent APIs, but identity data like user id, name or email address.

Add support for the standard `openid` (subject id) and `profile` (first name, last name etc..) scopes by adding a new helper (in `config.cs`) to create a collection of `IdentityResource` objects:

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

**Note:** All standard scopes and their corresponding claims can be found in the OpenID Connect specification

You will then need to add these identity resources to your IdentityServer configuration in `Startup.cs`. Use the `AddInMemoryIdentityResources` extension method where you call `AddIdentityServer()`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    // configure identity server with in-memory stores, keys, clients and scopes
    services.AddIdentityServer()
        .AddTemporarySigningCredential()
        .AddInMemoryIdentityResources(Config.GetIdentityResources())
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients())
        .AddTestUsers(Config.GetUsers());
}
```

### 6.10.4 Adding a client for OpenID Connect implicit flow

The last step is to add a new client to IdentityServer.

OpenID Connect-based clients are very similar to the OAuth 2.0 clients we added so far. But since the flows in OIDC are always interactive, we need to add some redirect URLs to our configuration.

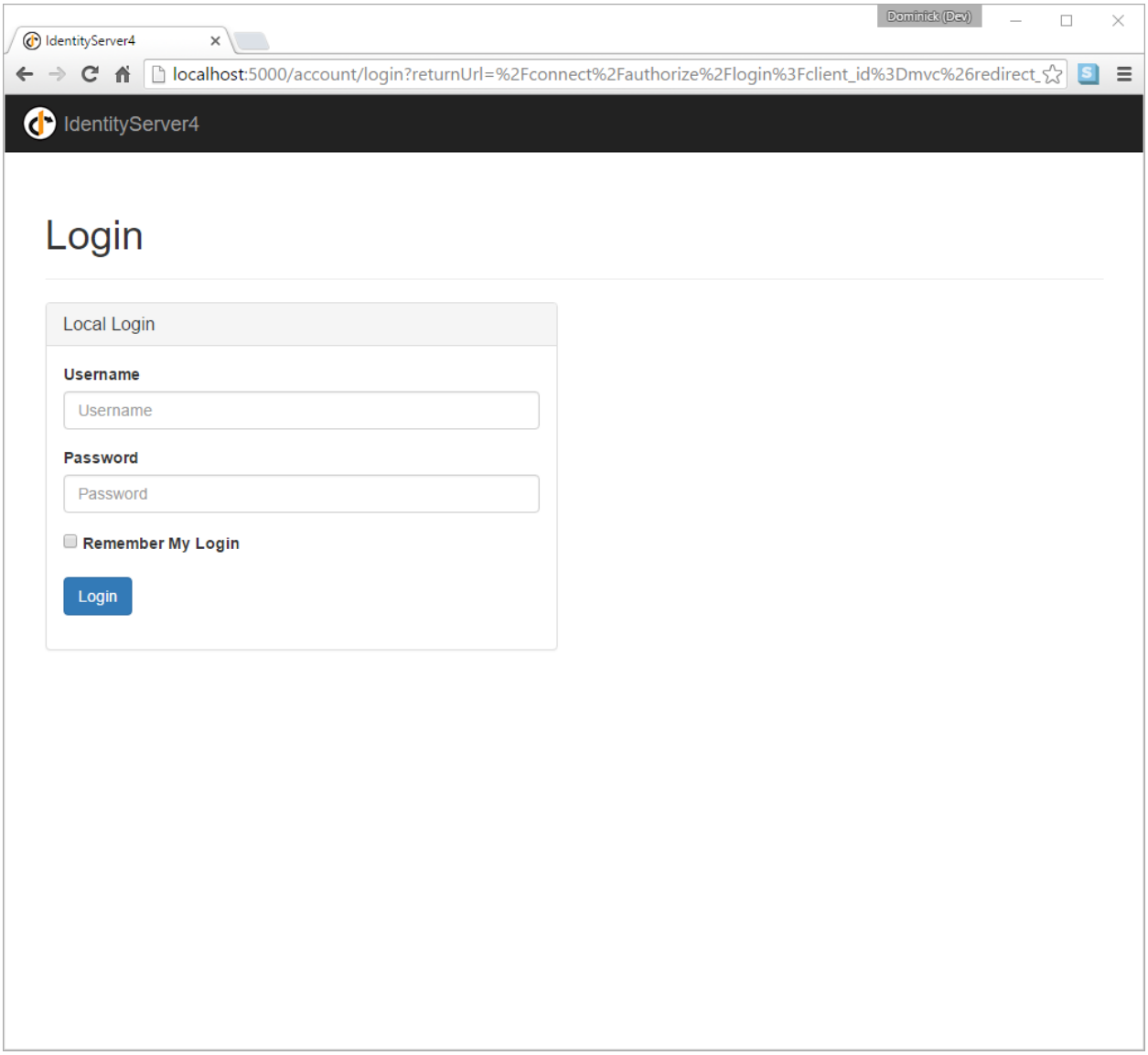
Add the following to your clients configuration:

```
public static IEnumerable<Client> GetClients()
{
    return new List<Client>
    {
        // other clients omitted...

        // OpenID Connect implicit flow client (MVC)
        new Client
        {
            ClientId = "mvc",
            ClientName = "MVC Client",
            AllowedGrantTypes = GrantTypes.Implicit,

            // where to redirect to after login
            RedirectUris = { "http://localhost:5002/signin-oidc" },

            // where to redirect to after logout
            PostLogoutRedirectUris = { "http://localhost:5002" },

            AllowedScopes = new List<string>
            {
                IdentityServerConstants.StandardScopes.OpenId,
                IdentityServerConstants.StandardScopes.Profile
            }
        }
    };
}
```

### 6.10.5 Testing the client

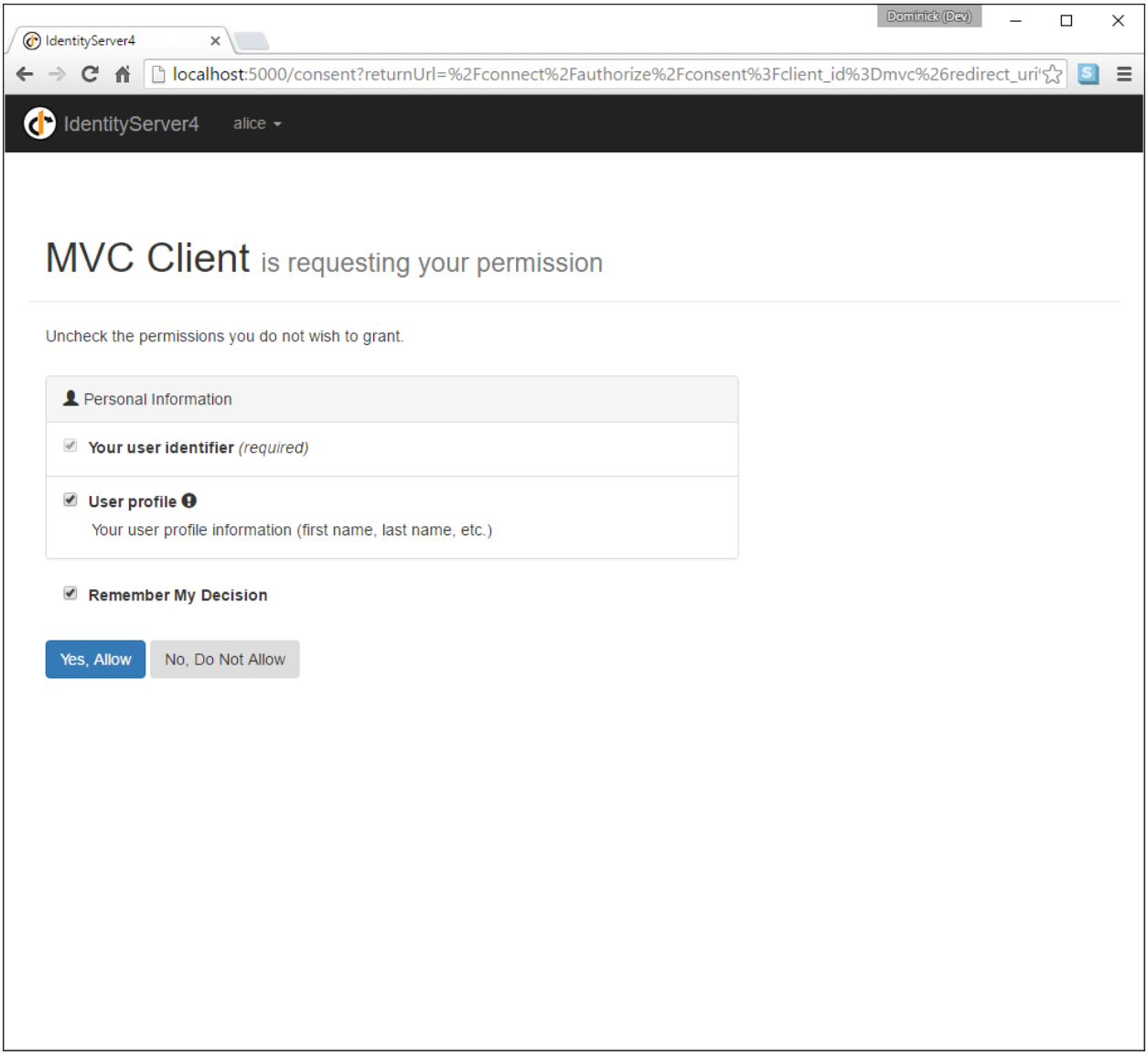
Now finally everything should be in place for the new MVC client.

Trigger the authentication handshake by navigating to the protected controller action. You should see a redirect to the login page at IdentityServer.

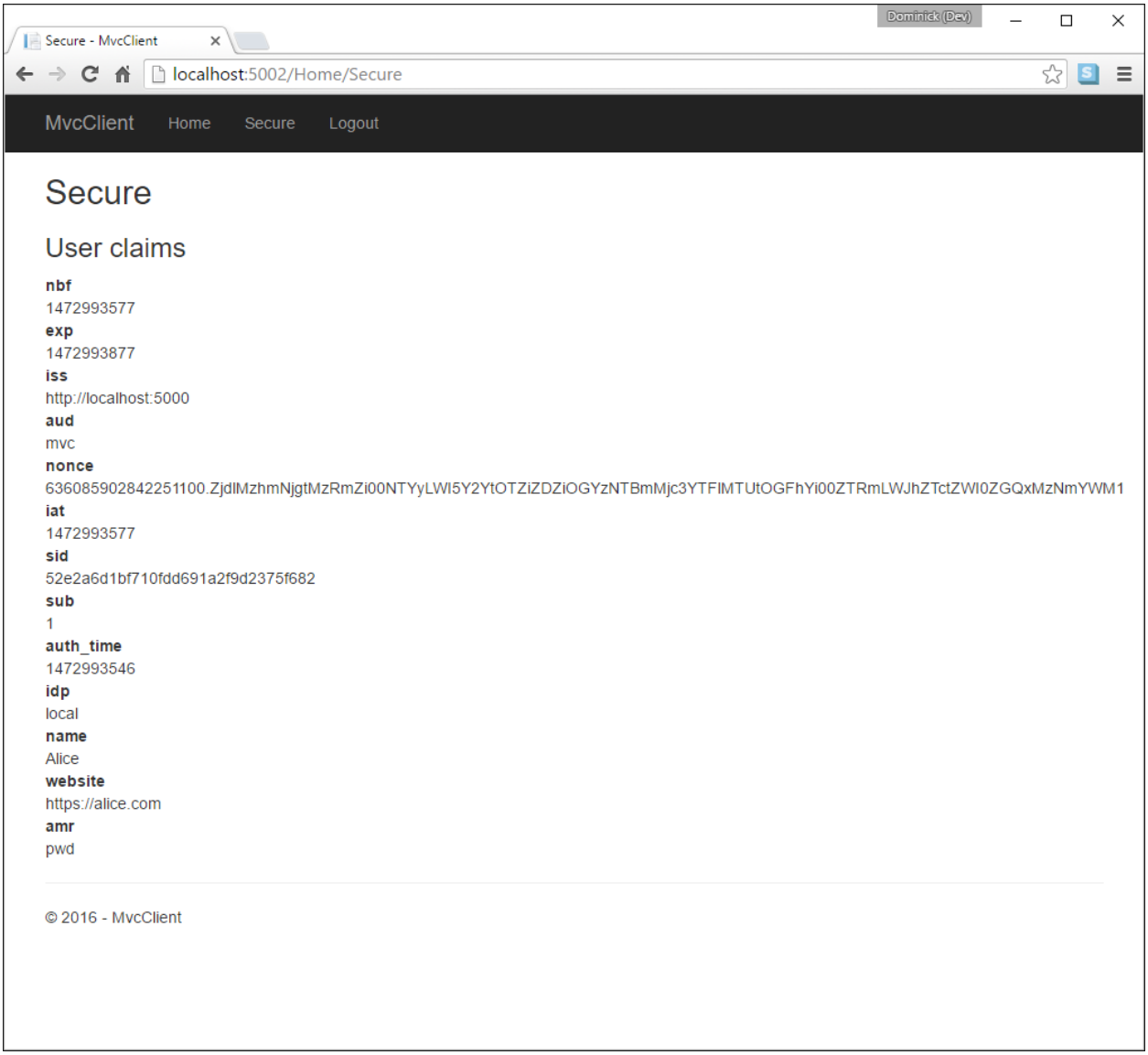



After successful login, the user is presented with the consent screen. Here the user can decide if he wants to release his identity information to the client application.

**Note:** Consent can be turned off on a per client basis using the `RequireConsent` property on the client object.




..and finally the browser redirects back to the client application, which shows the claims of the user.




**Note:** During development you might sometimes see an exception stating that the token could not be validated. This is due to the fact that the signing key material is created on the fly and kept in-memory only. This exception happens when the client and IdentityServer get out of sync. Simply repeat the operation at the client, the next time the metadata has caught up, and everything should work normal again.

### 6.10.6 Adding sign-out

The very last step is to add sign-out to the MVC client.

With an authentication service like IdentityServer, it is not enough to clear the local application cookies. In addition you also need to make a roundtrip to IdentityServer to clear the central single sign-on session.

The exact protocol steps are implemented inside the OpenID Connect middleware, simply add the following code to some controller to trigger the sign-out:

```
public async Task Logout()
{
    await HttpContext.Authentication.SignOutAsync("Cookies");
    await HttpContext.Authentication.SignOutAsync("oidc");
}
```

This will clear the local cookie and then redirect to IdentityServer. IdentityServer will clear its cookies and then give the user a link to return back to the MVC application.

### 6.10.7 Further experiments

As mentioned above, the OpenID Connect middleware asks for the *profile* scope by default. This scope also includes claims like *name* or *website*.

Let's add these claims to the user, so IdentityServer can put them into the identity token:

```
public static List<TestUser> GetUsers()
{
    return new List<TestUser>
    {
        new TestUser
        {
            SubjectId = "1",
            Username = "alice",
            Password = "password",

            Claims = new []
            {
                new Claim("name", "Alice"),
                new Claim("website", "https://alice.com")
            }
        },
        new TestUser
        {
            SubjectId = "2",
            Username = "bob",
            Password = "password",

            Claims = new []
            {
                new Claim("name", "Bob"),
                new Claim("website", "https://bob.com")
            }
        }
    };
}
```

Next time you authenticate, your claims page will now show the additional claims.

Feel free to add more claims - and also more scopes. The `Scope` property on the OpenID Connect middleware is where you configure which scopes will be sent to IdentityServer during authentication.

It is also noteworthy, that the retrieval of claims for tokens is an extensibility point - `IProfileService`. Since we are using `AddTestUsers`, the `TestUserProfileService` is used by default. You can inspect the source code here to see how it works.

## 6.11 Adding Support for External Authentication

Next we will add support for external authentication. This is really easy, because all you really need is an ASP.NET Core compatible authentication middleware.

ASP.NET Core itself ships with support for Google, Facebook, Twitter, Microsoft Account and OpenID Connect. In Addition you can find providers for many other authentication provider here.

### 6.11.1 Adding Google support

To be able to use Google for authentication, you first need to register with them. This is done at their developer console. Create a new project, enable the Google+ API and configure the callback address of your local IdentityServer by adding the */signin-google* path to your base-address (e.g. http://localhost:5000/signin-google).

If you are running on port 5000 - you can simply use the client id/secret from the code snippet below, since this is pre-registered by us.

Add the Google and cookie authentication package to your project.json:

```
"Microsoft.AspNetCore.Authentication.Cookies": "1.0.0",
"Microsoft.AspNetCore.Authentication.Google": "1.0.0"
```

Next we need to add the middleware to the pipeline. Order is important, the additional authentication middleware must run **after** IdentityServer and **before** MVC.

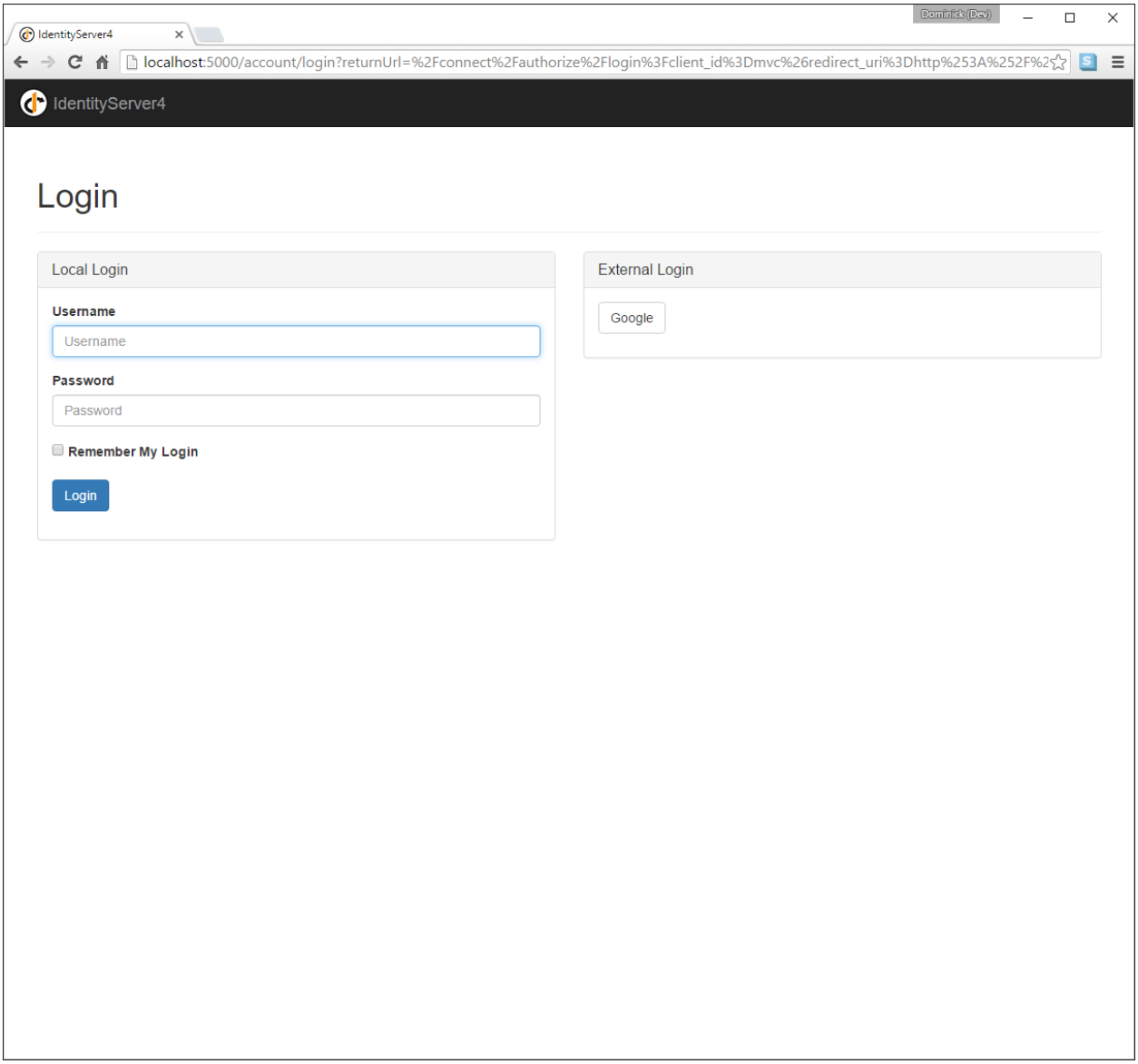
The cookie middleware is used to temporarily store the outcome of the external authentication in a temporary cookie - register it like this:

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationScheme = IdentityServerConstants.ExternalCookieAuthenticationScheme,

    AutomaticAuthenticate = false,
    AutomaticChallenge = false
});
```

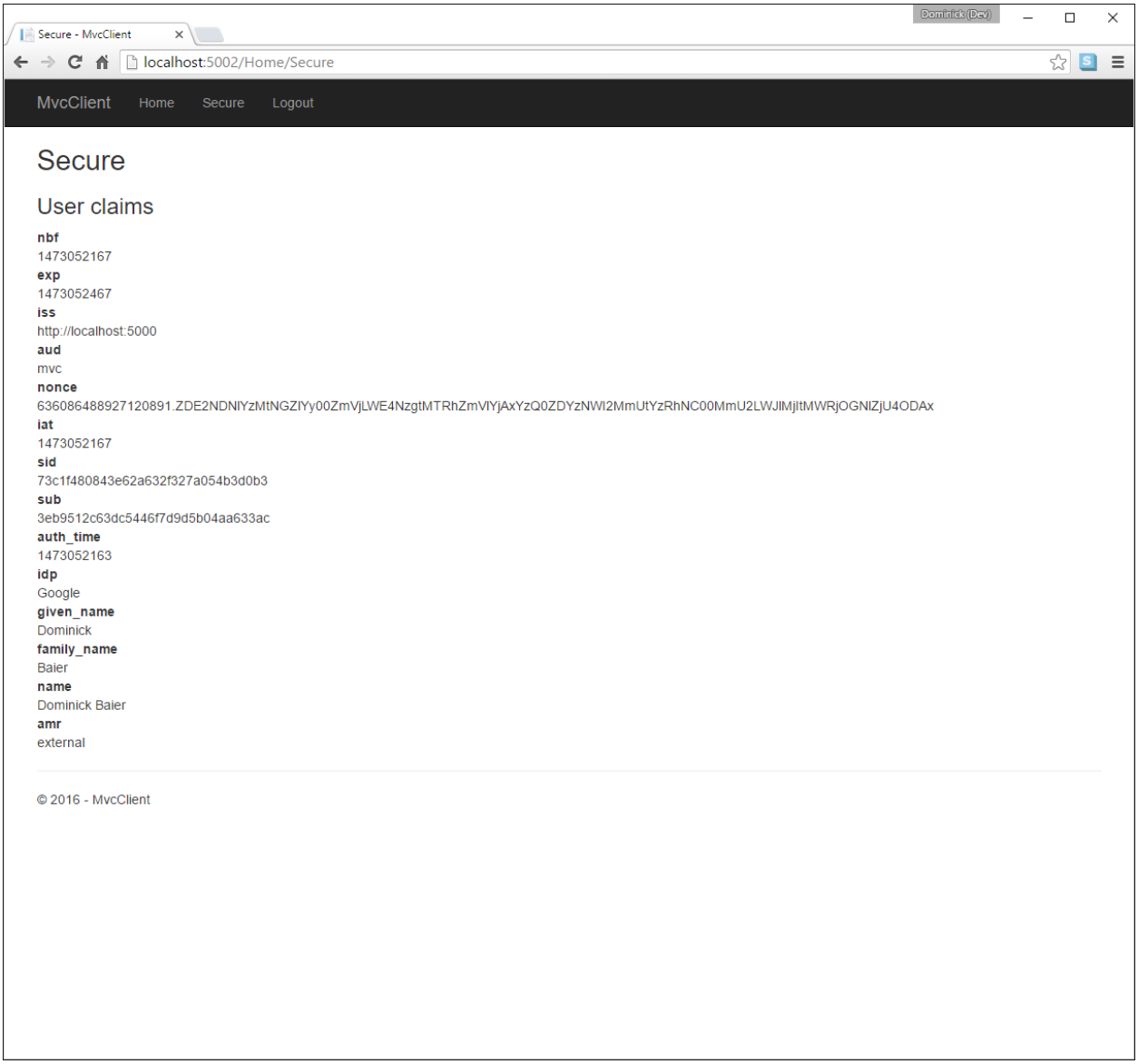
After that add the Google middleware:

```
app.UseGoogleAuthentication(new GoogleOptions
{
    AuthenticationScheme = "Google",
    DisplayName = "Google",
    SignInScheme = IdentityServerConstants.ExternalCookieAuthenticationScheme,

    ClientId = "434483408261-55tc8n0cs4ff1fe21ea8df2o443v2iuc.apps.googleusercontent.com",
    ClientSecret = "3gcoTrEDPPJ0ukn_aYYT6PWo"
});
```

Now run the MVC client and try to authenticate - you will see a Google button on the login page:




After authentication, you can see that the claims are now being sourced from Google data.




### 6.11.2 Further experiments

You can add an additional external provider. We have a cloud-hosted demo version of IdentityServer3 which you can integrate using OpenID Connect.

First add the OpenID Connect package to project.json:

```
"Microsoft.AspNetCore.Authentication.OpenIdConnect": "1.0.0"
```

Next add the middleware:

```
// middleware for external openid connect authentication
app.UseOpenIdConnectAuthentication(new OpenIdConnectOptions
{
    SignInScheme = IdentityServerConstants.ExternalCookieAuthenticationScheme,
    SignOutScheme = IdentityServerConstants.SignoutScheme,
```

```
    DisplayName = "OpenID Connect",
    Authority = "https://demo.identityserver.io/",
    ClientId = "implicit",

    TokenValidationParameters = new TokenValidationParameters
    {
        NameClaimType = "name",
        RoleClaimType = "role"
    }
});
```

**Note:** The quickstart UI auto-provisions external users. IOW - if an external user logs in for the first time, a new local user is created, all the external claims are copied over and associated with the new user. The way you deal with such a situation is completely up to you though. Maybe you want to show some sort of registration UI first. The source code for the default quickstart can be found here.

## 6.12 Switching to Hybrid Flow and adding API Access back

In the previous quickstarts we explored both API access and user authentication. Now we want to bring the two parts together.

The beauty of the OpenID Connect & OAuth 2.0 combination is, that you can achieve both with a single protocol and a single round-trip to the token service.

In the previous quickstart we used the OpenID Connect implicit flow. In the implicit flow all tokens are transmitted via the browser, which is totally fine for the identity token. Now we also want to request an access token.

Access tokens are a bit more sensitive than identity tokens, and we don't want to expose them to the "outside" world if not needed. OpenID Connect includes a flow called "Hybrid Flow" which gives us the best of both worlds, the identity token is transmitted via the browser channel, so the client can validate it before doing any more work. And if validation is successful, the client opens a back-channel to the token service to retrieve the access token.

### 6.12.1 Modifying the client configuration

There are not many modifications necessary. First we want to allow the client to use the hybrid flow, in addition we also want the client to allow doing server to server API calls which are not in the context of a user (this is very similar to our client credentials quickstart). This is expressed using the `AllowedGrantTypes` property.

Next we need to add a client secret. This will be used to retrieve the access token on the back channel.

And finally, we also give the client access to the `offline_access` scope - this allows requesting refresh tokens for long lived API access:

```
new Client
{
    ClientId = "mvc",
    ClientName = "MVC Client",
    AllowedGrantTypes = GrantTypes.HybridAndClientCredentials,

    ClientSecrets =
    {
        new Secret("secret".Sha256())
    },
```

```
    RedirectUris           = { "http://localhost:5002/signin-oidc" },
    PostLogoutRedirectUris = { "http://localhost:5002" },

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        "api1"
    },
    AllowOfflineAccess = true
}
```

};

### 6.12.2 Modifying the MVC client

The modifications at the MVC client are also minimal - the ASP.NET Core OpenID Connect middleware has built-in support for the hybrid flow, so we only need to change some configuration values.

We configure the `ClientSecret` to match the secret at IdentityServer. Add the `offline_access` scopes, and set the `ResponseType` to `code id_token` (which basically means "use hybrid flow")

```
app.UseOpenIdConnectAuthentication(new OpenIdConnectOptions
{
    AuthenticationScheme = "oidc",
    SignInScheme = "Cookies",

    Authority = "http://localhost:5000",
    RequireHttpsMetadata = false,

    ClientId = "mvc",
    ClientSecret = "secret",

    ResponseType = "code id_token",
    Scope = { "api1", "offline_access" },

    GetClaimsFromUserInfoEndpoint = true,
    SaveTokens = true
});
```

When you run the MVC client, there will be no big differences, besides that the consent screen now asks you for the additional API and offline access scope.

### 6.12.3 Using the access token

The OpenID Connect middleware saves the tokens (identity, access and refresh in our case) automatically for you. That's what the `SaveTokens` setting does.

Technically the tokens are stored inside the properties section of the cookie. The easiest way to access them is by using extension methods from the `Microsoft.AspNetCore.Authentication` namespace.

For example on your claims view:

```
<dt>access token</dt>
<dd>@await ViewContext.HttpContext.Authentication.GetTokenAsync("access_token")</dd>
```

```
<dt>refresh token</dt>
<dd>@await ViewContext.HttpContext.Authentication.GetTokenAsync("refresh_token")</dd>
```

For accessing the API using the user token, all you need to do is retrieve the token, and set it on your *HttpClient*:

```
public async Task<IActionResult> CallApiUsingUserAccessToken()
{
    var accessToken = await HttpContext.Authentication.GetTokenAsync("access_token");

    var client = new HttpClient();
    client.SetBearerToken(accessToken);
    var content = await client.GetStringAsync("http://localhost:5001/identity");

    ViewBag.Json = JArray.Parse(content).ToString();
    return View("json");
}
```

## 6.13 Using ASP.NET Core Identity

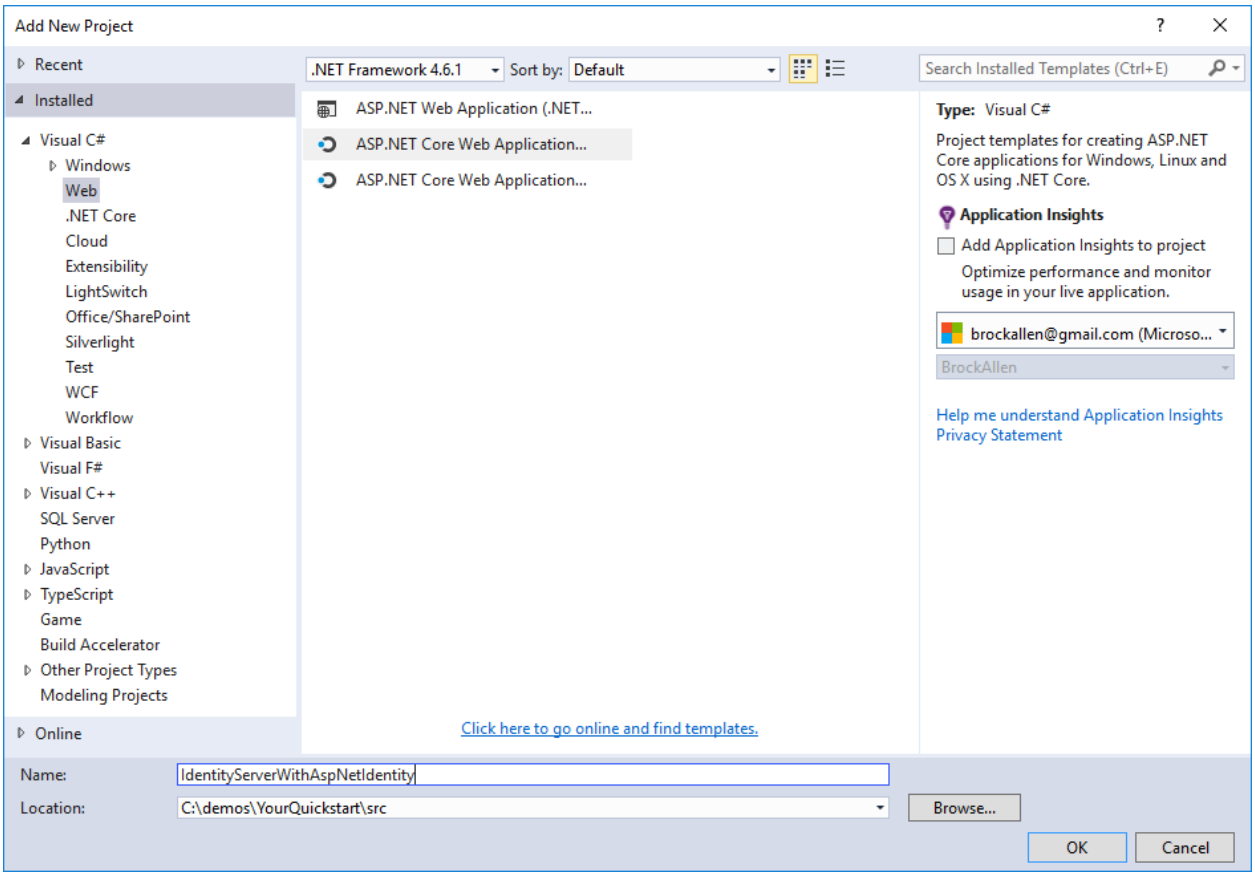
IdentityServer is designed for flexibility and part of that is allowing you to use any database you want for your users and their data (including passwords). If you are starting with a new user database, then ASP.NET Identity is one option you could choose. This quickstart shows how to use ASP.NET Identity with IdentityServer.

This quickstart assumes you've been through all of the prior quickstarts. The approach this quickstart takes to using ASP.NET Identity is to create a new project from the ASP.NET Identity template in Visual Studio. This new project will replace the prior IdentityServer project we built up from scratch in the previous quickstarts. All the other projects in this solution (for the clients and the API) will remain the same.

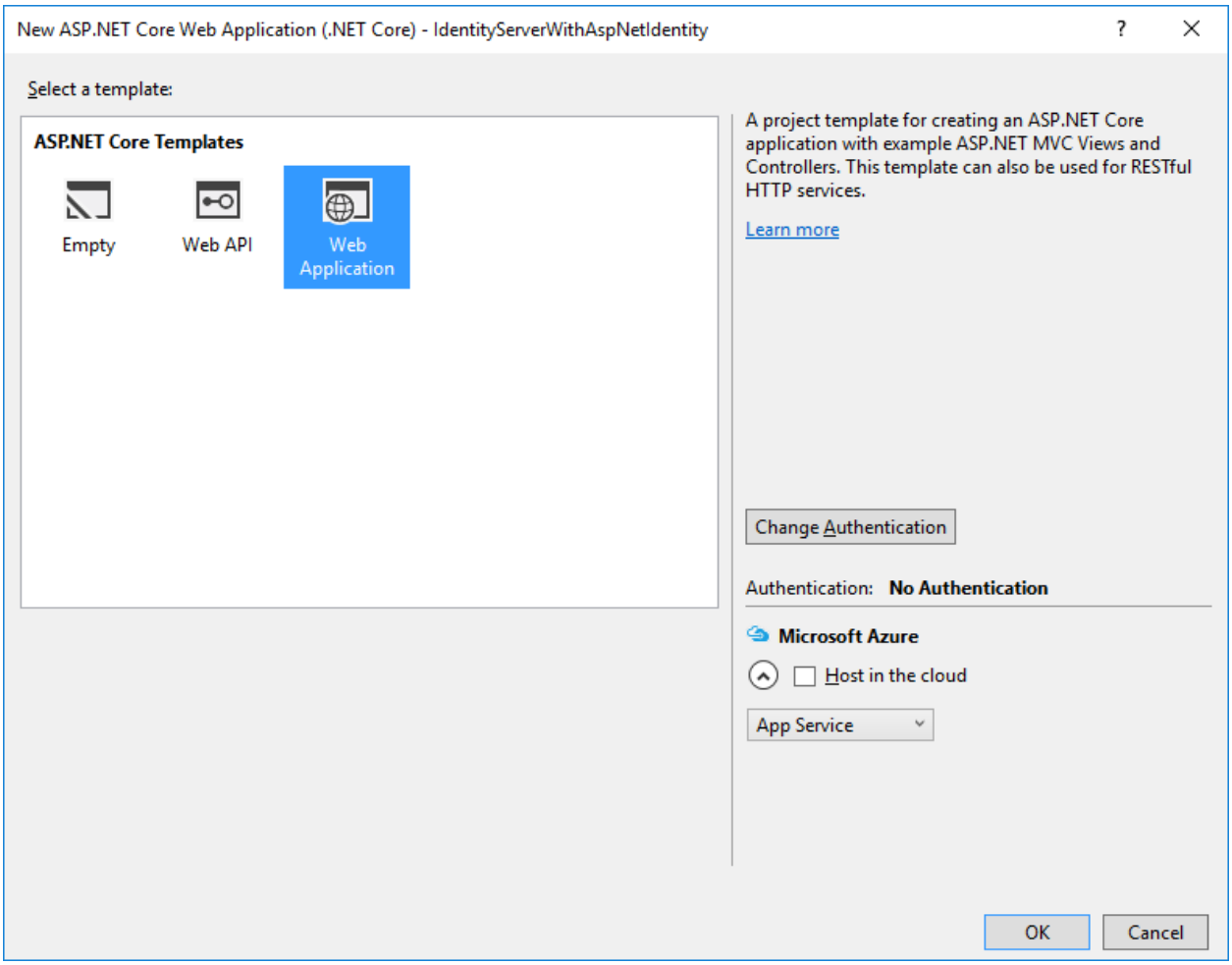
### 6.13.1 New Project for ASP.NET Identity

The first step is to add a new project for ASP.NET Identity to your solution. Given that a lot of code is required for ASP.NET Identity, it makes sense to use the template from Visual Studio. You will eventually delete the old project for IdentityServer (assuming you were following the other quickstarts), but there are several items that you will need to migrate over (or rewrite from scratch as decribed in the prior quickstarts).

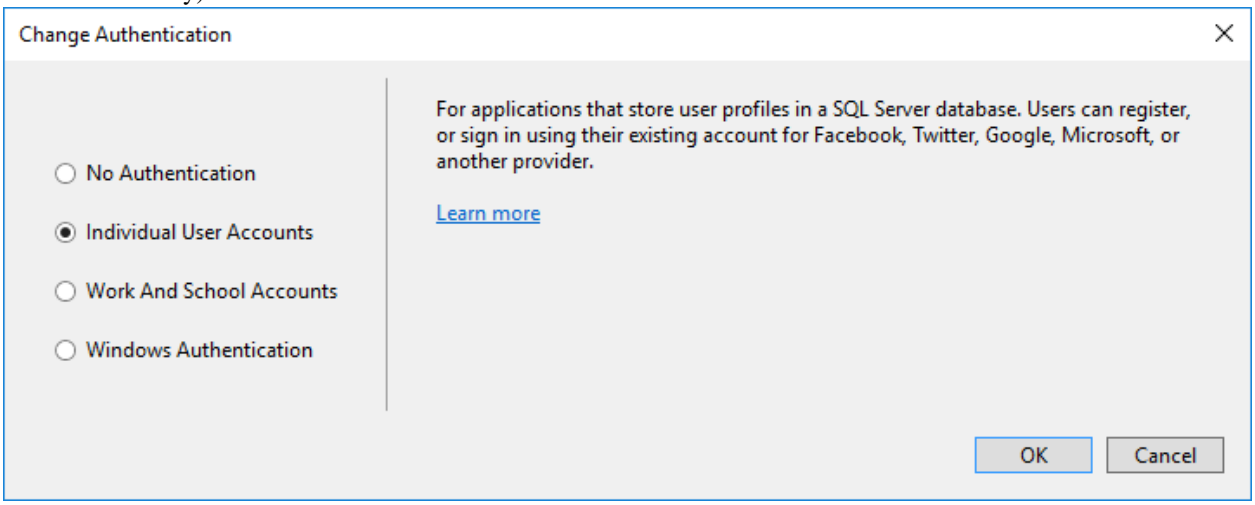
Start by creating a new "ASP.NET Core Web Application" project.



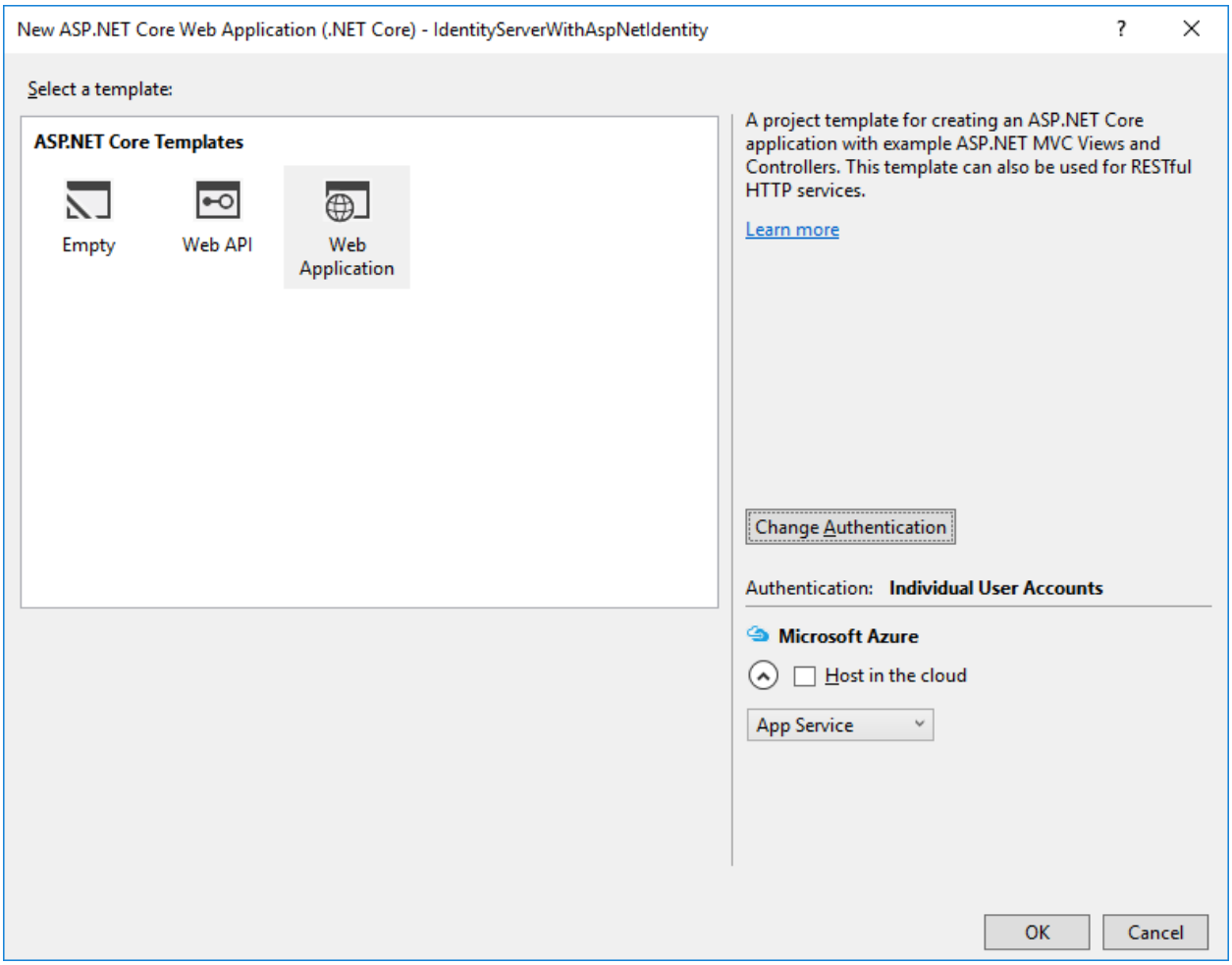

Then select the "Web Application Template" option.




Then click the "Change Authentication" button, and choose "Individual User Accounts" (which means to use ASP.NET Identity):




Finally, your new project dialog should look something like this. Once it does, click "OK" to create the project.




**Note:** Don't forget to update your project to ASP.NET Core 1.1. You can follow this guide for updating to ASP.NET Core 1.1.

### 6.13.2 Modify hosting

Don't forget to modify the hosting (as described here) to run on port 5000. This is important so the existing clients and api projects will continue to work.

### 6.13.3 Add IdentityServer packages

Add the `IdentityServer4.AspNetIdentity` package to *project.json*:

```
"IdentityServer4.AspNetIdentity": "1.0.0"
```

### 6.13.4 Scopes and Clients Configuration

Despite this being a new project for IdentityServer, we still need the same scope and client configuration as the prior quickstarts. Copy the configuration class (in Config.cs) you used for the previous quickstarts into this new project.

One change to the configuration that is necessary (for now) is to disable consent for the MVC client. We've not yet copied over the consent code from the prior IdentityServer project, so for now make this one modification to the MVC client and set `RequireConsent=false`:

```
new Client
{
    ClientId = "mvc",
    ClientName = "MVC Client",
    AllowedGrantTypes = GrantTypes.HybridAndClientCredentials,

    RequireConsent = false,

    ClientSecrets =
    {
        new Secret("secret".Sha256())
    },

    RedirectUris           = { "http://localhost:5002/signin-oidc" },
    PostLogoutRedirectUris = { "http://localhost:5002" },

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        "api1"
    },
    AllowOfflineAccess = true
}
```

### 6.13.5 Configure IdentityServer

As before, IdentityServer needs to be configured in both `ConfigureServices` and in `Configure` in *Startup.cs*.

**ConfigureServices**

This shows both the template code generated for ASP.NET Identity, plus the additions needed for IdentityServer (at the end). In the previous quickstarts, the `AddInMemoryUsers` extension method was used to register the users, but in this situation we replace that extension method with `AddAspNetIdentity` to use the ASP.NET Identity users instead. The `AddAspNetIdentity` extension method requires a generic parameter which is your ASP.NET Identity user type (the same one needed in the `AddIdentity` method from the template).

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

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

    services.AddMvc();

    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();

    // Adds IdentityServer
    services.AddIdentityServer()
        .AddTemporarySigningCredential()
```

```
        .AddInMemoryIdentityResources(Config.GetIdentityResources())
        .AddInMemoryApiResources(Config.GetApiResources())
        .AddInMemoryClients(Config.GetClients())
        .AddAspNetIdentity<ApplicationUser>();
}
```

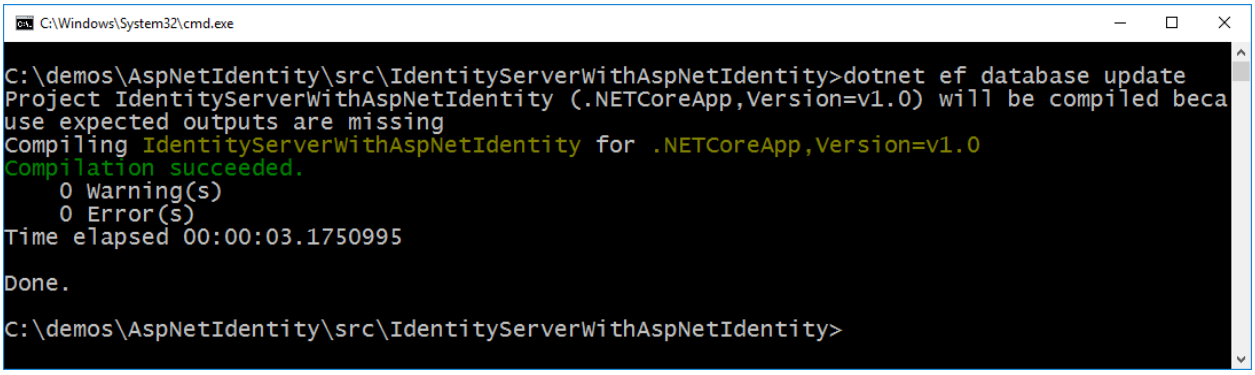
**Configure**

This shows both the template code generated for ASP.NET Identity, plus the additions needed for IdentityServer (just after `UseIdentity`). It's important when using ASP.NET Identity that IdentityServer be registered *after* ASP.NET Identity in the pipeline because IdentityServer is relying upon the authentication cookie that ASP.NET Identity creates and manages.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseIdentity();

    // Adds IdentityServer
    app.UseIdentityServer();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

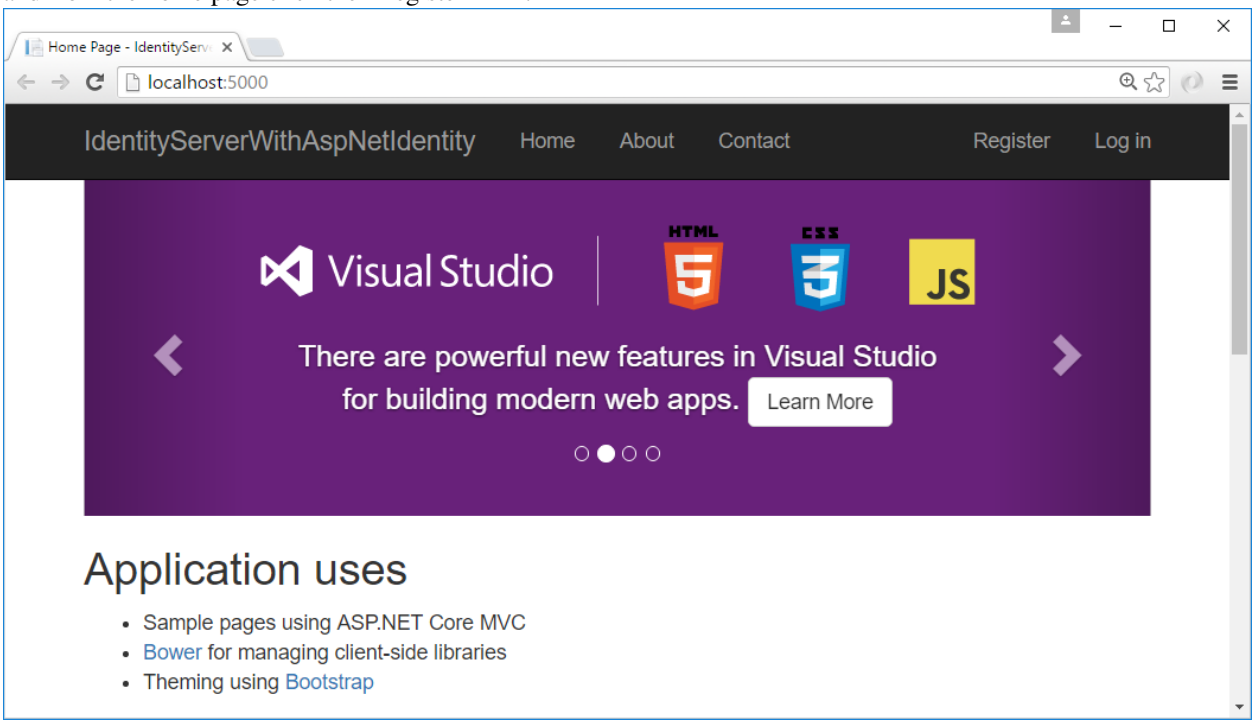
### 6.13.6 Creating the user database

Given that this is a new ASP.NET Identity project, you will need to create the database. You can do this by running a command prompt from the project directory and running `dotnet ef database update`, like this:

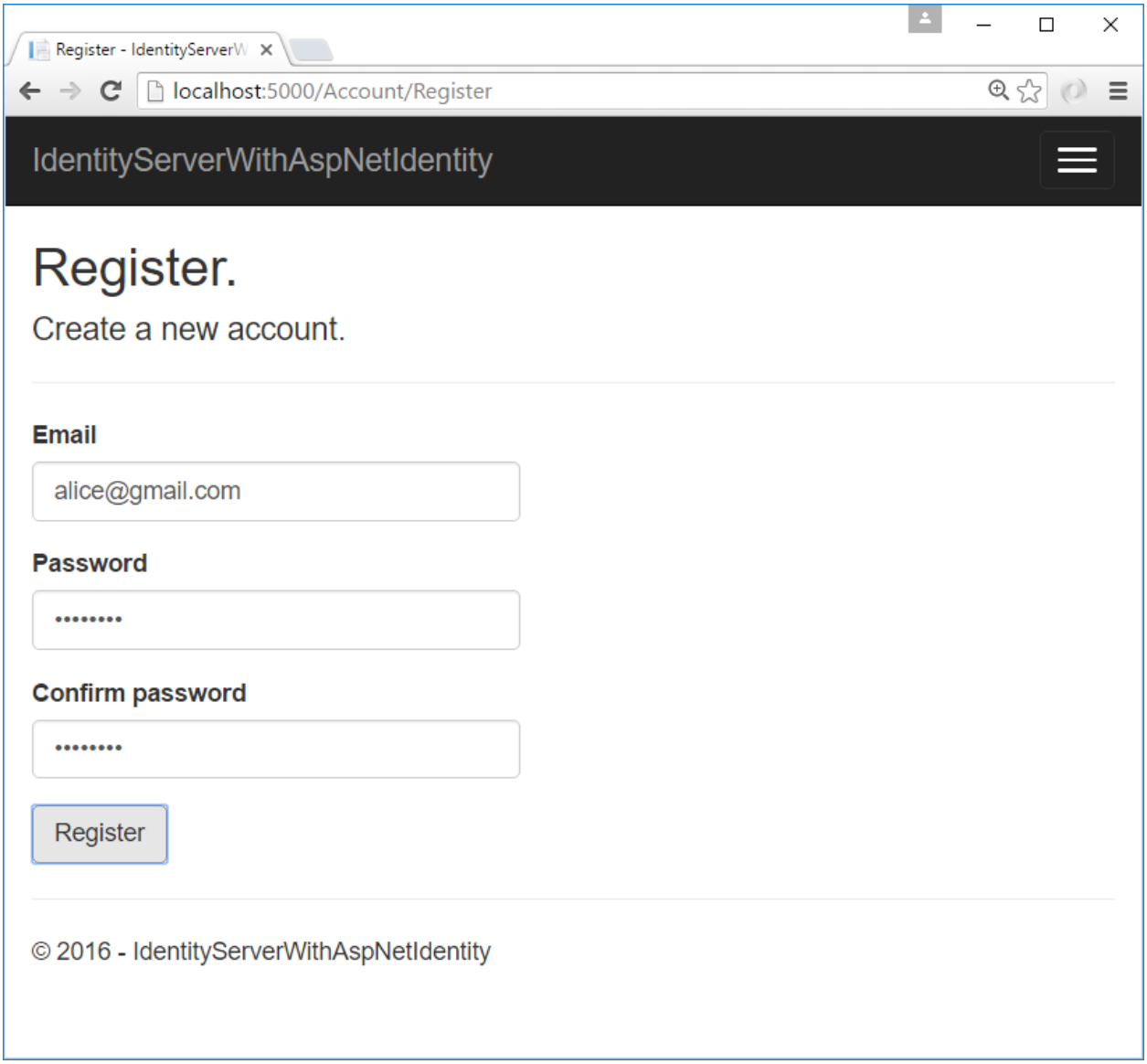



### 6.13.7 Creating a user

At this point, you should be able to run the project and create/register a user in the database. Launch the application, and from the home page click the "Register" link:



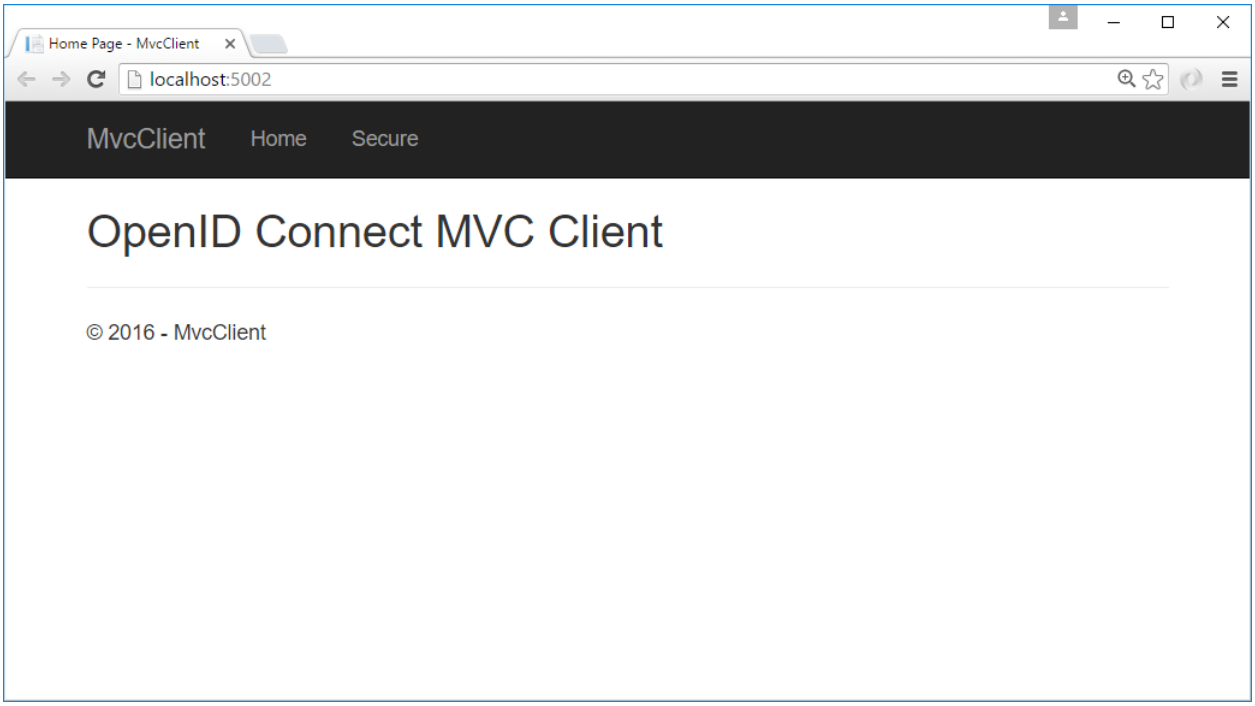

And on the register page create a new user account:



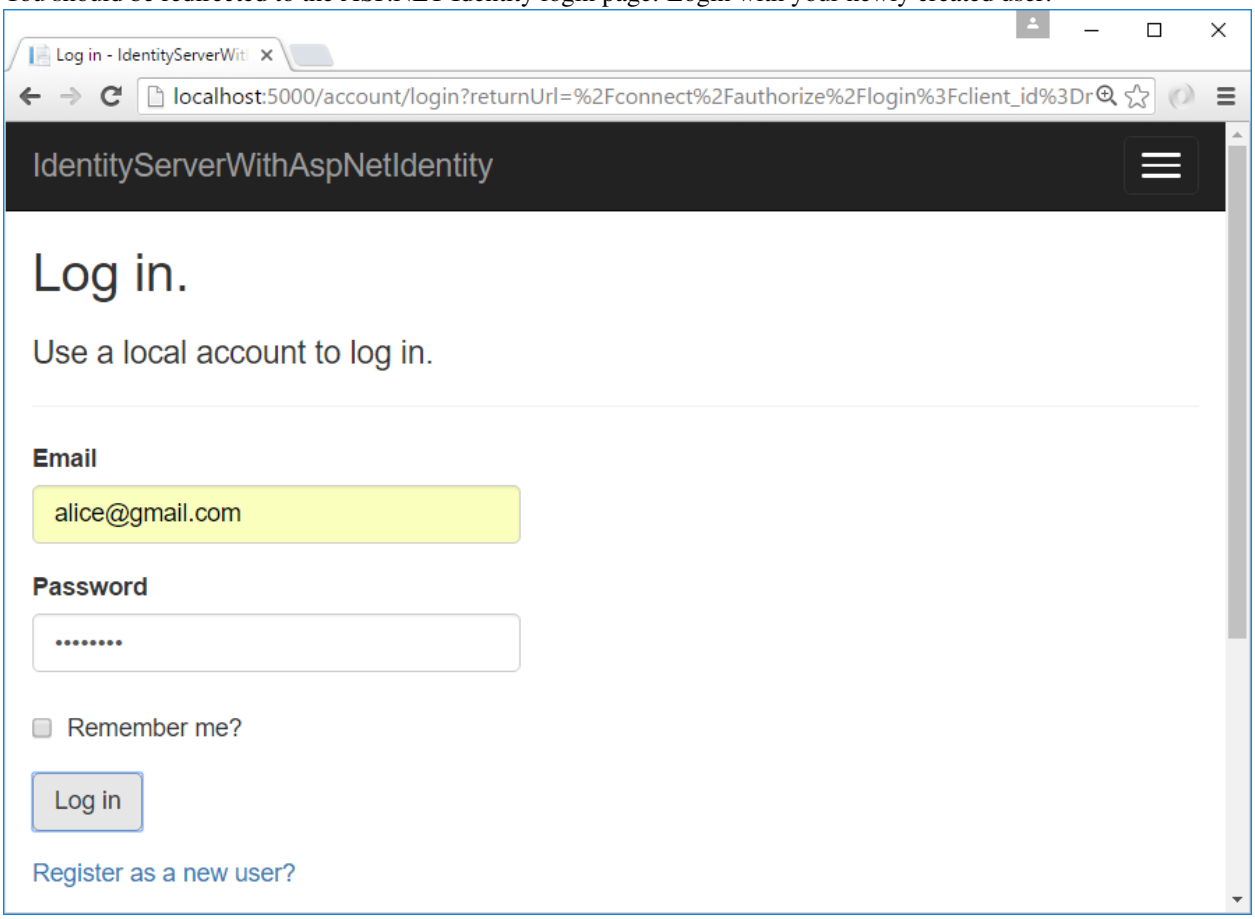

Now that you have a user account, you should be able to login, use the clients, and invoke the APIs.

### 6.13.8 Logging in with the MVC client

Launch the MVC client application, and you should be able to click the "Secure" link to get logged in.

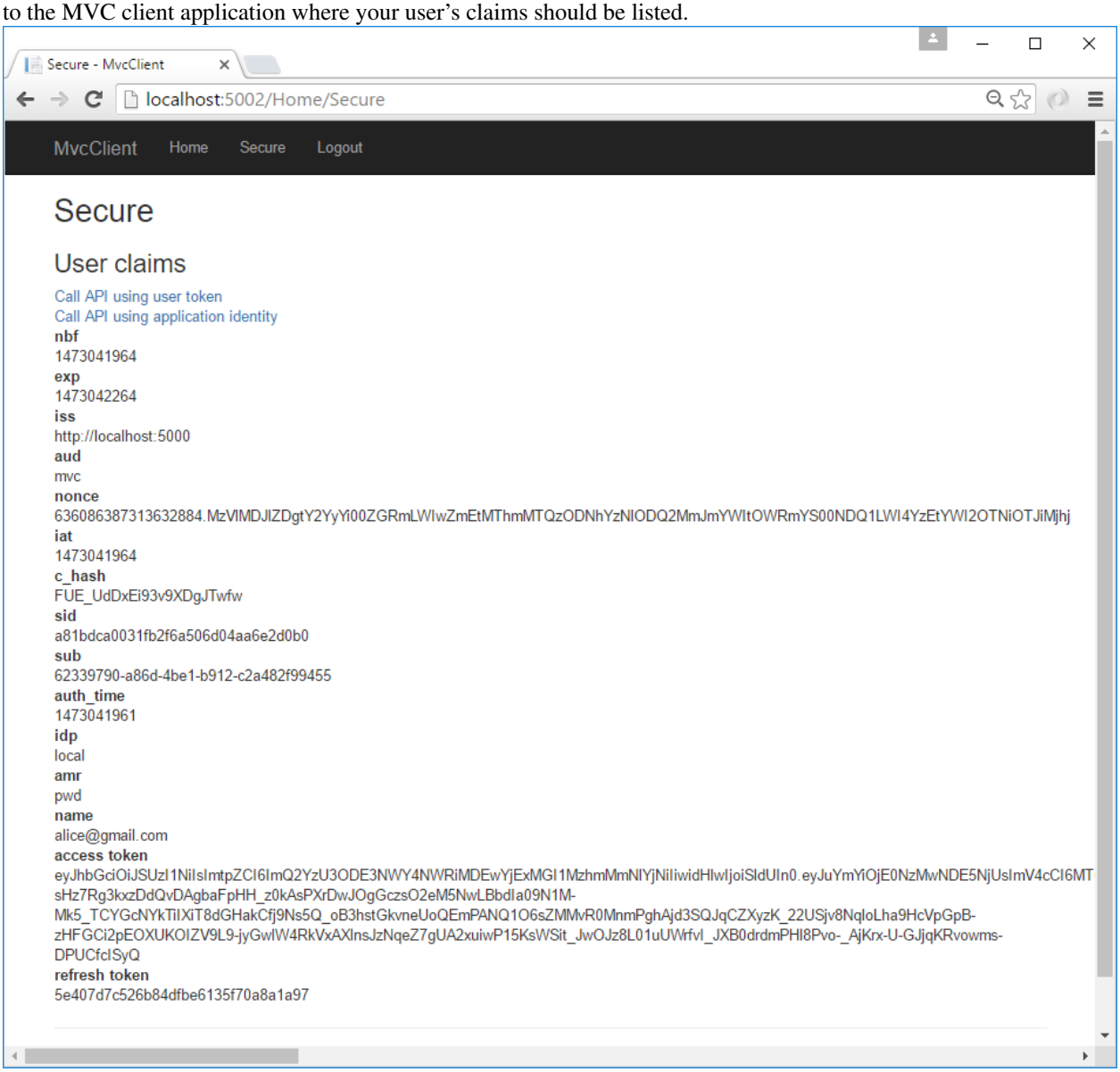



You should be redirected to the ASP.NET Identity login page. Login with your newly created user:

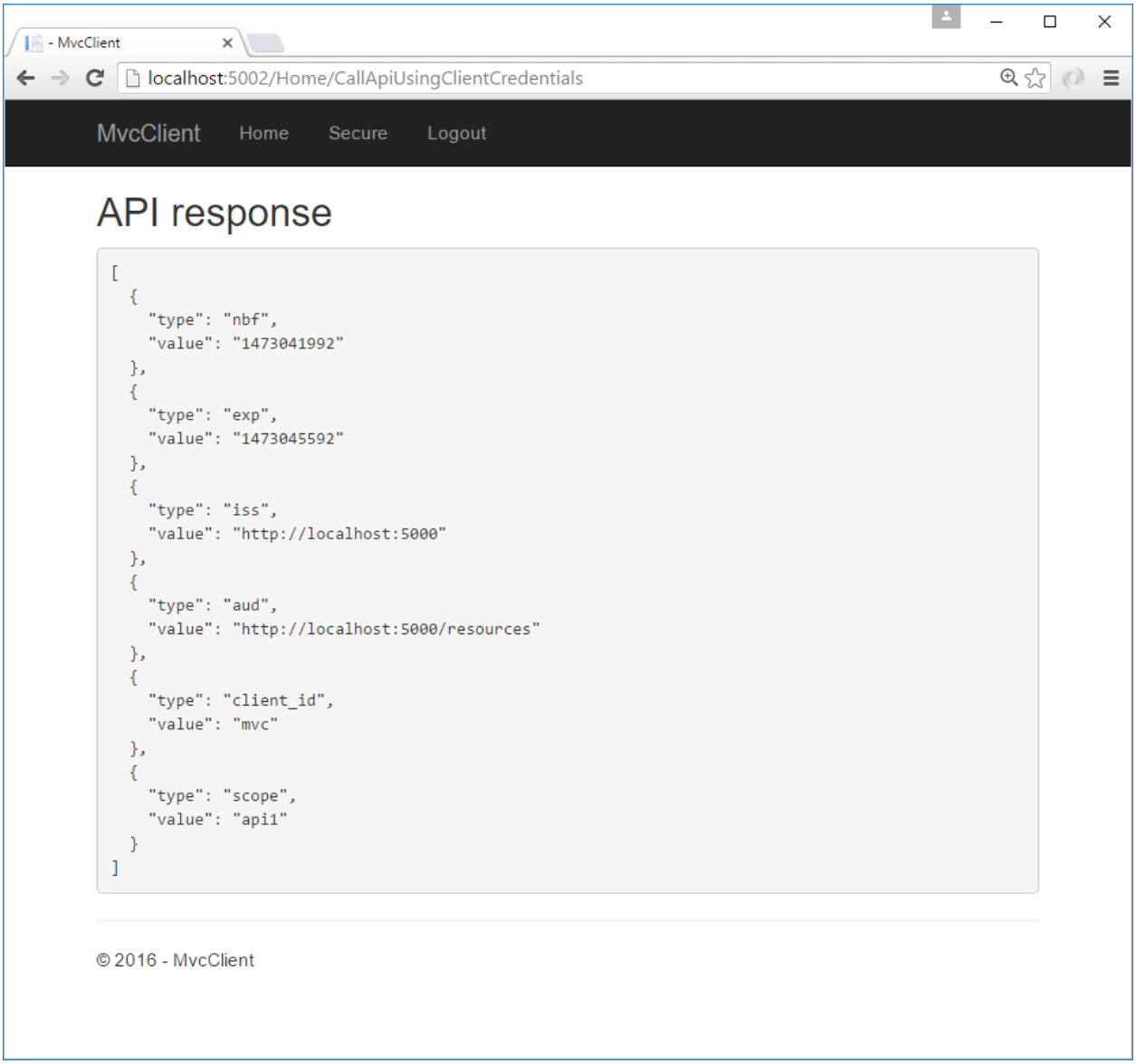



After login you should skip the consent page (given the change we made above), and be immediately redirected back

to the MVC client application where your user's claims should be listed.




You should also be able to click "Call API using application identity" to invoke the API on behalf of the user:




And now you've logged in with a user from ASP.NET Identity.

### 6.13.9 What's Next?

The prior quickstart project for IdentityServer provided a consent page, an error page, and a logout page. The code for these missing pieces can simply be copied over from the prior quickstart project into this one. Once you've done that, then you can finally delete/remove the old IdentityServer project. Also, once you've done this don't forget to re-enable the `RequireConsent=true` flag on the MVC client configuration.

The sample code for this quickstart has already done these steps for you, so you can get started quickly with all of these features. Enjoy!

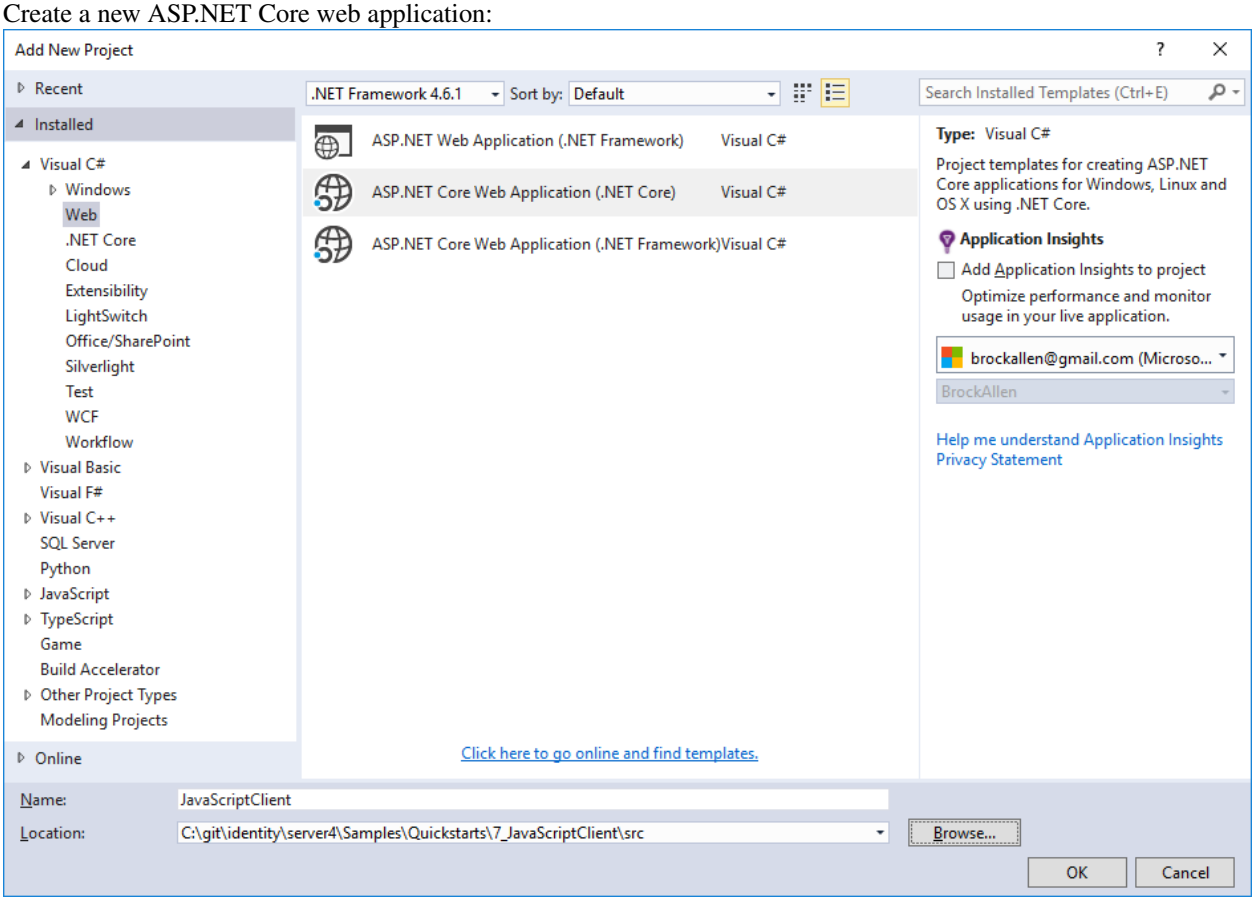
## 6.14 Adding a JavaScript client

This quickstart will show how to build a JavaScript client application. The user will login to IdentityServer, invoke the web API with an access token issued by IdentityServer, and logout of IdentityServer.

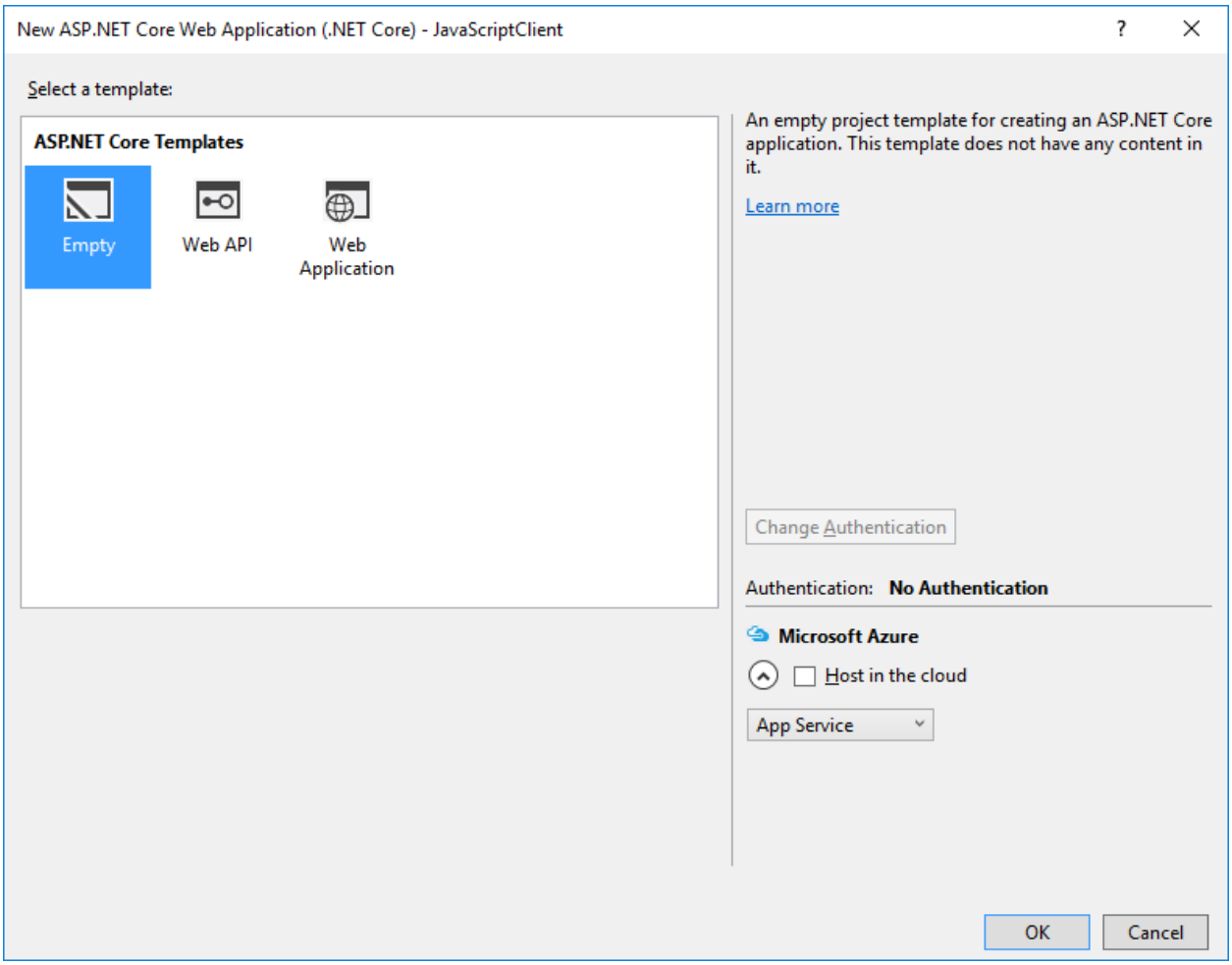
### 6.14.1 New Project for the JavaScript client

Create a new project for the JavaScript application. It can simply be an empty web project, or an empty ASP.NET Core application. This quickstart will use an empty ASP.NET Core application.

Create a new ASP.NET Core web application:




Choose the "Empty" template:




Click the "OK" button to create the project.

### 6.14.2 Modify hosting

Modify the hosting (as described here) to run on port 5003.

### 6.14.3 Add the static file middleware

Given that this project is designed to mainly run client-side, we need ASP.NET Core to serve up the static HTML and JavaScript files that will make up our application. The static file middleware is designed to do this. Add the NuGet package `Microsoft.AspNetCore.StaticFiles` to *project.json*:

```
"Microsoft.AspNetCore.StaticFiles": "1.0.0"
```

### 6.14.4 Register the static file middleware

Next, register the static file middleware in *Startup.cs* in the `Configure` method:

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
```

```
    app.UseStaticFiles();
}
```

This middleware will now serve up static files from the application's *~/wwwroot* folder. This is where we will put our HTML and JavaScript files.

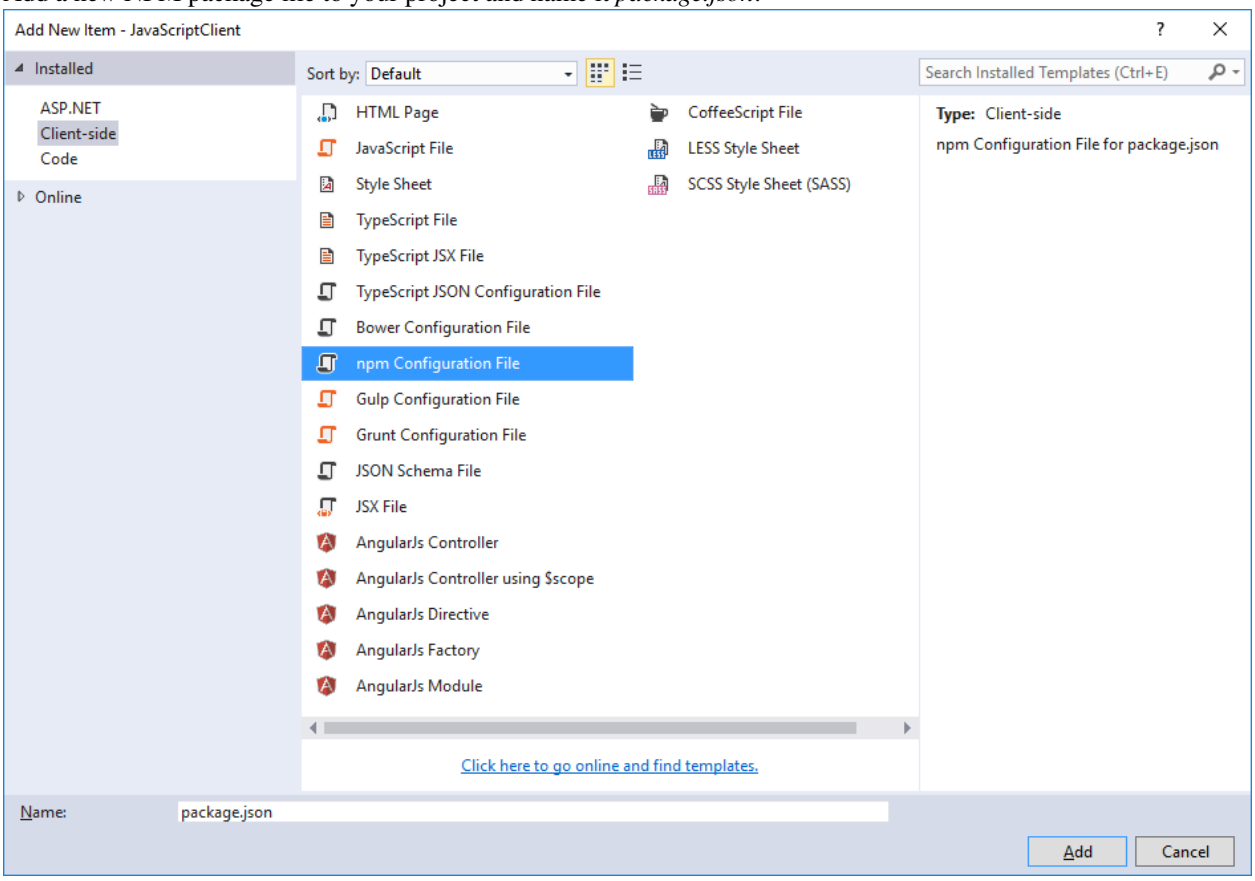
### 6.14.5 Reference oidc-client

In the MVC project, we used a library to handle the OpenID Connect protocol. In this project we need a similar library, except one that works in JavaScript and is designed to run in the browser. The oidc-client library is one such library. It is available via NPM, Bower, as well as a direct download from github.

**NPM**

If you want to use NPM to download *oidc-client*, then follow these steps:

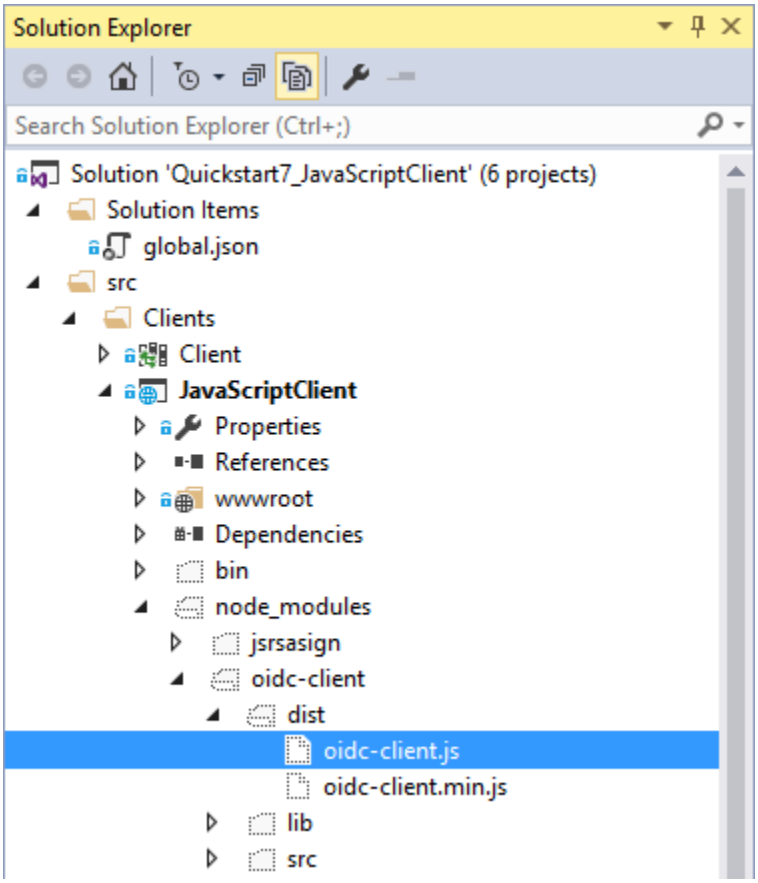
Add a new NPM package file to your project and name it *package.json*:




In *package.json* add a `devDependency` to `oidc-client`:

```
"devDependencies": {
  "oidc-client": "1.2.2"
}
```

Once you have saved this file, Visual Studio should automatically restore these packages into a folder called *node_modules*:




Locate the file called *oidc-client.js* in the *~/node_modules/oidc-client/dist* folder and copy it into your application's *~/wwwroot* folder. There are more sophisticated ways of copying your NPM packages into *~/wwwroot*, but those techniques are beyond the scope of this quickstart.

### 6.14.6 Add your HTML and JavaScript files

Next is to add your HTML and JavaScript files to *~/wwwroot*. We will have two HTML files and one application-specific JavaScript file (in addition to the *oidc-client.js* library). In *~/wwwroot*, add a HTML file named *index.html* and *callback.html*, and add a JavaScript file called *app.js*.

**index.html**

This will be the main page in our application. It will simply contain the HTML for the buttons for the user to login, logout, and call the web API. It will also contain the `<script>` tags to include our two JavaScript files. It will also contain a `<pre>` used for showing messages to the user.

It should look like this:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <button id="login">Login</button>
    <button id="api">Call API</button>
```

```
    <button id="logout">Logout</button>

    <pre id="results"></pre>

    <script src="oidc-client.js"></script>
    <script src="app.js"></script>
</body>
</html>
```

**app.js**

This will contain the main code for our application. The first thing is to add a helper function to log messages to the `<pre>`:

```
function log() {
    document.getElementById('results').innerText = '';

    Array.prototype.forEach.call(arguments, function (msg) {
        if (msg instanceof Error) {
            msg = "Error: " + msg.message;
        }
        else if (typeof msg !== 'string') {
            msg = JSON.stringify(msg, null, 2);
        }
        document.getElementById('results').innerHTML += msg + '\r\n';
    });
}
```

Next, add code to register "click" event handlers to the three buttons:

```
document.getElementById("login").addEventListener("click", login, false);
document.getElementById("api").addEventListener("click", api, false);
document.getElementById("logout").addEventListener("click", logout, false);
```

Next, we can use the `UserManager` class in the *oidc-client* library to manage the OpenID Connect protocol. It requires similar configuration that was necessary in the MVC Client (albeit with different values). Add this code to configure and instantiate the `UserManager`:

```
var config = {
    authority: "http://localhost:5000",
    client_id: "js",
    redirect_uri: "http://localhost:5003/callback.html",
    response_type: "id_token token",
    scope:"openid profile api1",
    post_logout_redirect_uri : "http://localhost:5003/index.html",
};
var mgr = new Oidc.UserManager(config);
```

Next, the `UserManager` provides a `getUser` API to know if the user is logged into the JavaScript application. It uses a JavaScript `Promise` to return the results asynchronously. The returned `User` object has a `profile` property which contains the claims for the user. Add this code to detect if the user is logged into the JavaScript application:

```
mgr.getUser().then(function (user) {
    if (user) {
        log("User logged in", user.profile);
    }
    else {
        log("User not logged in");
    }
});
```

Next, we want to implement the `login`, `api`, and `logout` functions. The `UserManager` provides a `signinRedirect` to log the user in, and a `signoutRedirect` to log the user out. The `User` object that we obtained in the above code also has an `access_token` property which can be used to authenticate with a web API. The `access_token` will be passed to the web API via the *Authorization* header with the *Bearer* scheme. Add this code to implement those three functions in our application:

```
function login() {
    mgr.signinRedirect();
}

function api() {
    mgr.getUser().then(function (user) {
        var url = "http://localhost:5001/identity";

        var xhr = new XMLHttpRequest();
        xhr.open("GET", url);
        xhr.onload = function () {
            log(xhr.status, JSON.parse(xhr.responseText));
        }
        xhr.setRequestHeader("Authorization", "Bearer " + user.access_token);
        xhr.send();
    });
}

function logout() {
    mgr.signoutRedirect();
}
```

**callback.html**

This HTML file is the designated `redirect_uri` page once the user has logged into IdentityServer. It will complete the OpenID Connect protocol sign-in handshake with IdentityServer. The code for this is all provided by the `UserManager` class we used earlier. Once the sign-in is complete, we can then redirect the user back to the main *index.html* page. Add this code to complete the signin process:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <script src="oidc-client.js"></script>
    <script>
        new Oidc.UserManager().signinRedirectCallback().then(function () {
            window.location = "index.html";
        }).catch(function (e) {
            console.error(e);
        });
    </script>
</body>
</html>
```

### 6.14.7 Add a client registration to IdentityServer for the JavaScript client

Now that the client application is ready to go, we need to define a configuration entry in IdentityServer for this new JavaScript client. In the IdentityServer project locate the client configuration (in *Config.cs*). Add a new *Client* to the list for our new JavaScript application. It should have the configuration listed below:

```
// JavaScript Client
new Client
{
    ClientId = "js",
    ClientName = "JavaScript Client",
    AllowedGrantTypes = GrantTypes.Implicit,
    AllowAccessTokensViaBrowser = true,

    RedirectUris =           { "http://localhost:5003/callback.html" },
    PostLogoutRedirectUris = { "http://localhost:5003/index.html" },
    AllowedCorsOrigins =     { "http://localhost:5003" },

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        "api1"
    }
}
```

### 6.14.8 Allowing Ajax calls to the Web API with CORS

One last bit of configuration that is necessary is to configure CORS in the web API project. This will allow Ajax calls to be made from *http://localhost:5003* to *http://localhost:5001*.

**CORS NuGet Package**

Add the `Microsoft.AspNetCore.Cors` NuGet package to *project.json* in the web API project:

```
"Microsoft.AspNetCore.Cors": "1.0.0"
```

**Configure CORS**

Next, add the CORS services to the dependency injection system in `ConfigureServices` in *Startup.cs*:

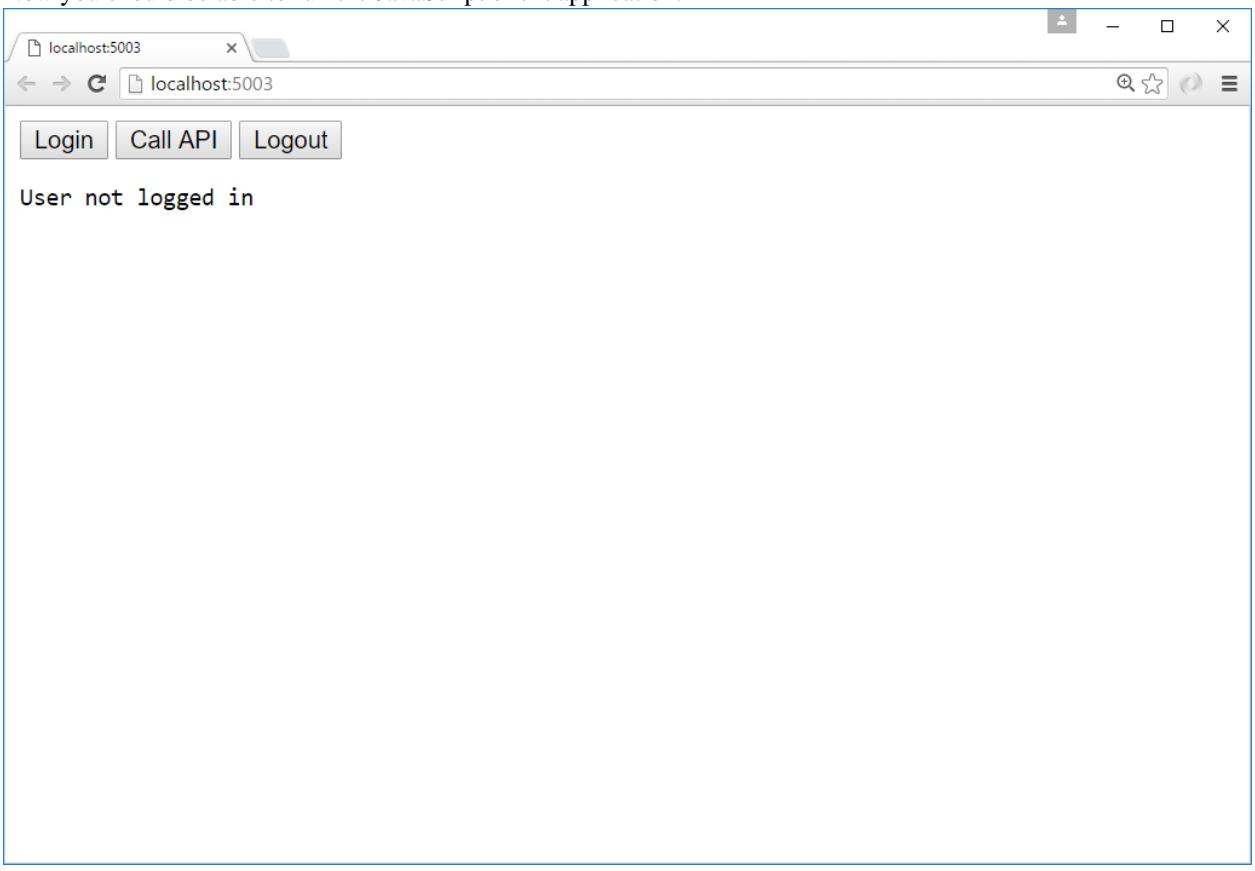
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options=>
    {
        // this defines a CORS policy called "default"
        options.AddPolicy("default", policy =>
        {
            policy.WithOrigins("http://localhost:5003")
                .AllowAnyHeader()
                .AllowAnyMethod();
        });
    });

    services.AddMvcCore()
        .AddAuthorization()
        .AddJsonFormatters();
}
```

Finally, add the CORS middleware to the pipeline in `Configure`:

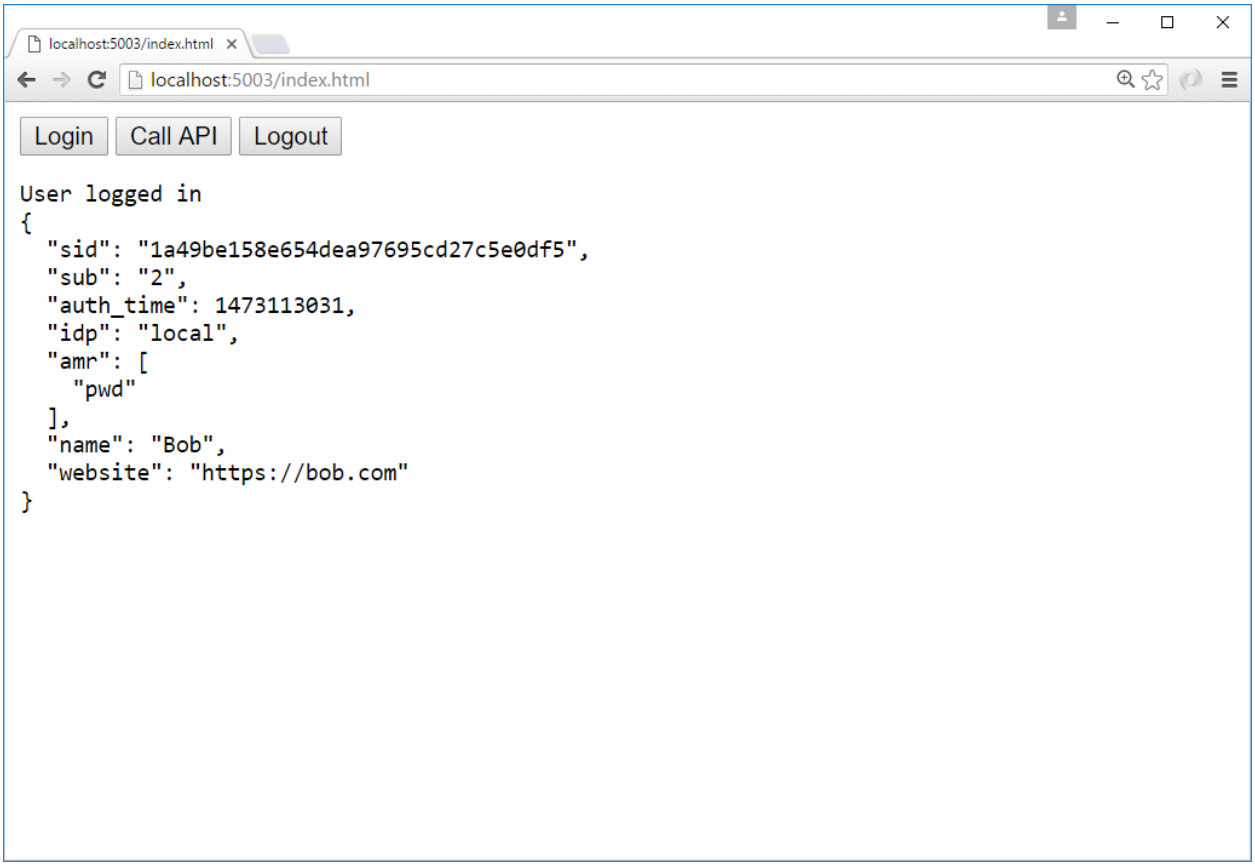
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    // this uses the policy called "default"
    app.UseCors("default");

    app.UseIdentityServerAuthentication(new IdentityServerAuthenticationOptions
    {
        Authority = "http://localhost:5000",
        AllowedScopes = { "api1" },

        RequireHttpsMetadata = false
    });

    app.UseMvc();
}
```

### 6.14.9 Run the JavaScript application

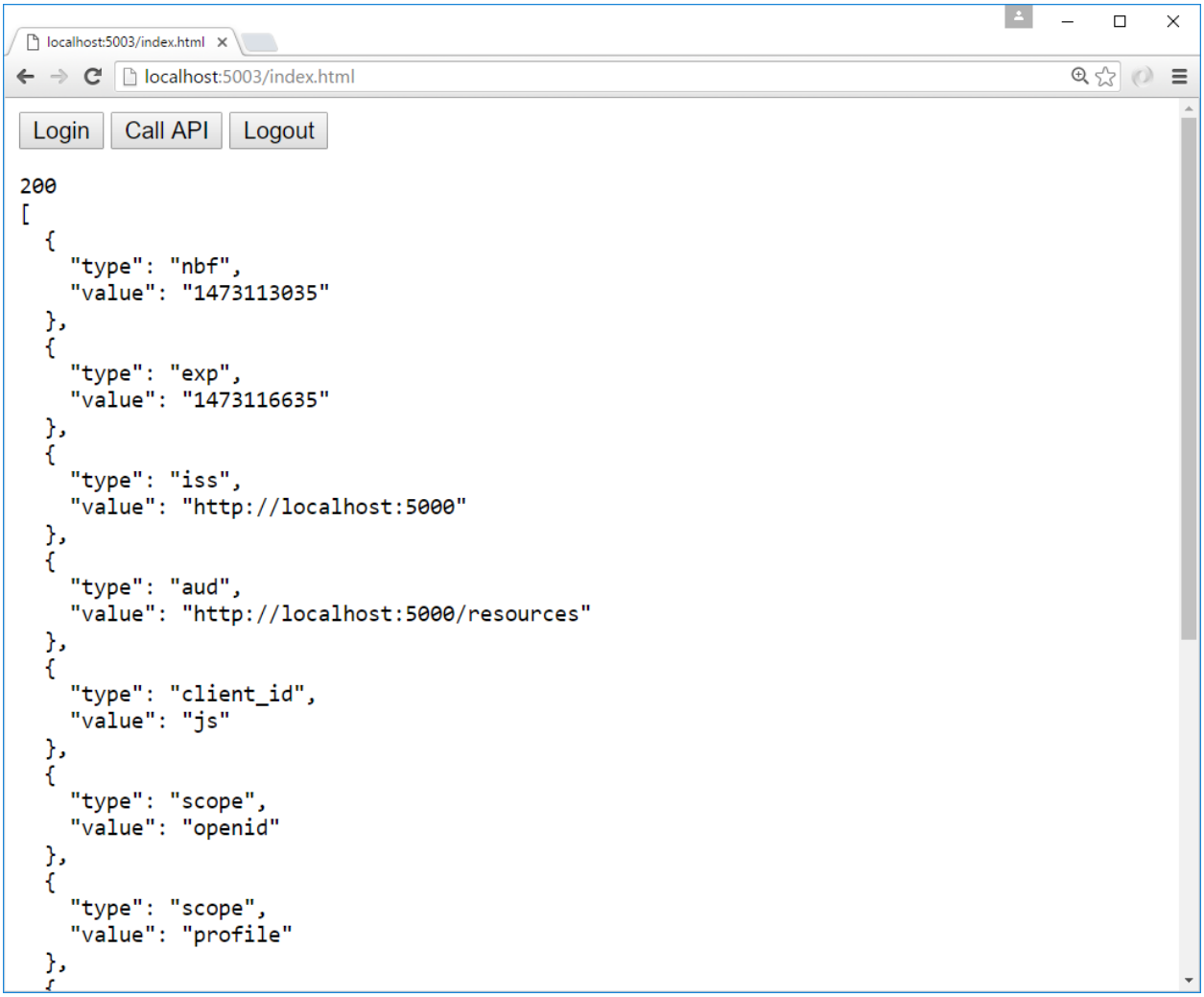
Now you should be able to run the JavaScript client application:



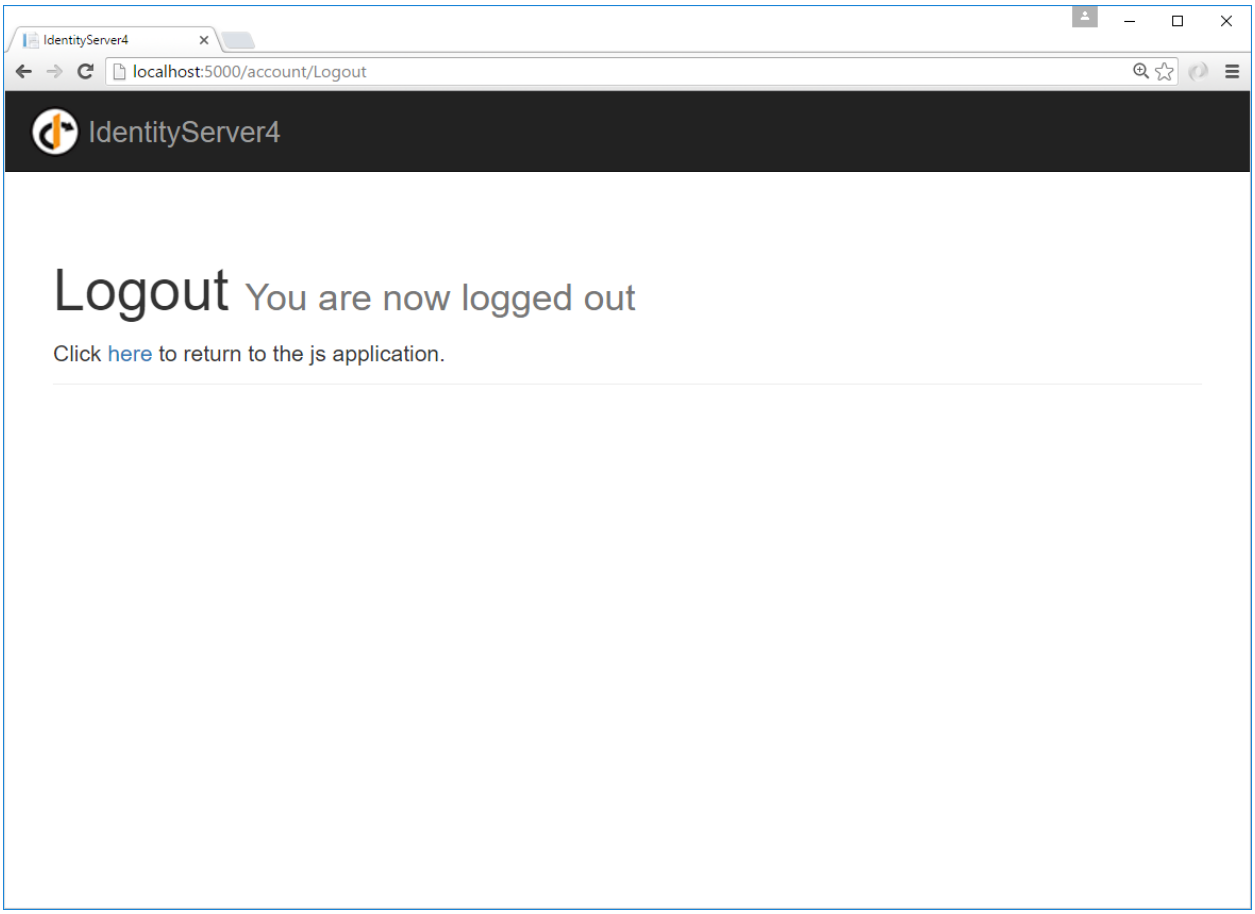

Click the "Login" button to sign the user in. Once the user is returned back to the JavaScript application, you should see their profile information:




And click the "API" button to invoke the web API:




And finally click “Logout” to sign the user out.




You now have the start of a JavaScript client application that uses IdentityServer for sign-in, sign-out, and authenticating calls to web APIs.

## 6.15 Using EntityFramework Core for configuration data

IdentityServer is designed for extensibility, and one of the extensibility points is the storage mechanism used for data that IdentityServer needs. This quickstart shows to how configure IdentityServer to use EntityFramework (EF) as the storage mechanism for this data (rather than using the in-memory implementations we had been using up until now).

### 6.15.1 IdentityServer4.EntityFramework

There are two types of data that we are moving to the database. The first is the configuration data (resources and clients). The second is operational data that IdentityServer produces as it's being used. These stores are modeled with interfaces, and we provide an EF implementation of these interfaces in the *IdentityServer4.EntityFramework* Nuget package.

Get started by adding a reference to the *IdentityServer4.EntityFramework* Nuget package in *project.json* in the IdentityServer project:

```
"IdentityServer4.EntityFramework": "1.0.0"
```

### 6.15.2 Adding SqlServer

Given EF's flexibility, you can then use any EF-supported database. For this quickstart we will use the LocalDb version of SqlServer that comes with Visual Studio.

To add SqlServer, we need several more dependencies. In the *"dependencies"* section in *project.json* add these packages:

```
"Microsoft.EntityFrameworkCore.SqlServer": "1.1.0",
"Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final"
```

And then in the *"tools"* section add this configuration:

```
"Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final"
```

### 6.15.3 Configuring the stores

The next step is to replace the current calls to `AddInMemoryClients`, `AddInMemoryIdentityResources`, and `AddInMemoryApiResources` in the `Configure` method in *Startup.cs*. We will replace them with this code:

```
using Microsoft.EntityFrameworkCore;
using System.Reflection;

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    var connectionString = @"server=(localdb)\mssqllocaldb;database=IdentityServer4.Quickstart;truste
    var migrationsAssembly = typeof(Startup).GetTypeInfo().Assembly.GetName().Name;

    // configure identity server with in-memory users, but EF stores for clients and resources
    services.AddIdentityServer()
        .AddTemporarySigningCredential()
        .AddTestUsers(Config.GetUsers())
        .AddConfigurationStore(builder =>
            builder.UseSqlServer(connectionString, options =>
                options.MigrationsAssembly(migrationsAssembly)))
        .AddOperationalStore(builder =>
            builder.UseSqlServer(connectionString, options =>
                options.MigrationsAssembly(migrationsAssembly)));
}
```

The above code is hard-coding a connection string, which you should feel free to change if you wish. Also, the calls to `AddConfigurationStore` and `AddOperationalStore` are registering the EF-backed store implementations.

The "builder" callback function passed to these APIs is the EF mechanism to allow you to configure the `DbContextOptionsBuilder` for the `DbContext` for each of these two stores. This is how our `DbContext` classes can be configured with the database provider you want to use. In this case by calling `UseSqlServer` we are using SqlServer. As you can also tell, this is where the connection string is provided.

The "options" callback function in `UseSqlServer` is what configures the assembly where the EF migrations are defined. EF requires the use of migrations to define the schema for the database.

---

**Note:** It is the responsibility of your hosting application to define these migrations, as they are specific to your database and provider.

---

We'll add the migrations next.

### 6.15.4 Adding migrations

To create the migrations, open a command prompt in the IdentityServer project directory. In the command prompt run these two commands:

```
dotnet ef migrations add InitialIdentityServerPersistedGrantDbMigration -c PersistedGrantDbContext -o
dotnet ef migrations add InitialIdentityServerConfigurationDbMigration -c ConfigurationDbContext -o
```

It should look something like this:


You should now see a *~/Data/Migrations/IdentityServer* folder in the project. This contains the code for the newly created migrations.

### 6.15.5 Initialize the database

Now that we have the migrations, we can write code to create the database from the migrations. We will also seed the database with the in-memory configuration data that we defined in the previous quickstarts.

In *Startup.cs* add this method to help initialize the database:

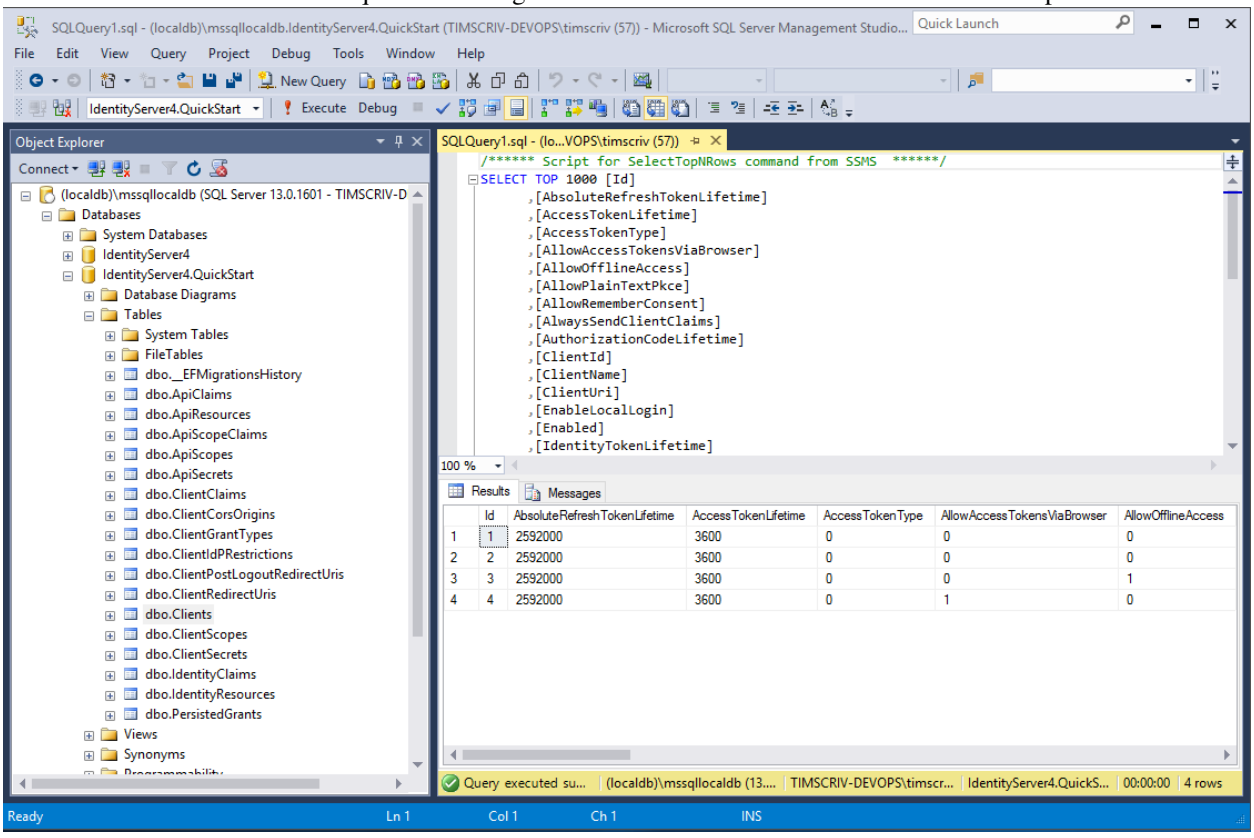
```
private void InitializeDatabase(IApplicationBuilder app)
{
    using (var serviceScope = app.ApplicationServices.GetService<IServiceScopeFactory>().CreateScope
    {
        serviceScope.ServiceProvider.GetRequiredService<PersistedGrantDbContext>().Database.Migrate()

        var context = serviceScope.ServiceProvider.GetRequiredService<ConfigurationDbContext>();
        context.Database.Migrate();
        if (!context.Clients.Any())
        {
            foreach (var client in Config.GetClients())
            {
                context.Clients.Add(client.ToEntity());
            }
            context.SaveChanges();
        }

        if (!context.IdentityResources.Any())
        {
            foreach (var resource in Config.GetIdentityResources())
```

```
            {
                context.IdentityResources.Add(resource.ToEntity());
            }
            context.SaveChanges();
        }

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

And then we can invoke this from the `Configure` method:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    // this will do the initial DB population
    InitializeDatabase(app);

    // the rest of the code that was already here
    // ...
}
```

Now if you run the IdentityServer project, the database should be created and seeded with the quickstart configuration data. You should be able to use SqlServer Management Studio or Visual Studio to connect and inspect the data.



### 6.15.6 Run the client applications

You should now be able to run any of the existing client applications and sign-in, get tokens, and call the API – all based upon the database configuration.

## 6.16 Startup

IdentityServer is a combination of middleware and services. All configuration is done in your startup class.

### 6.16.1 Configuring services

You add the IdentityServer services to the DI system by calling:

```
public void ConfigureServices(IServiceCollection services)
{
    var builder = services.AddIdentityServer();
}
```

Optionally you can pass in options into this call. See *here* for details on options.

This will return you a builder object that in turn has a number of convenience methods to wire up additional services.

**Key material**

- **`AddSigningCredential`** Adds a signing key service that provides the specified key material to the various token creation/validation services. You can pass in either an `X509Certificate2`, a `SigningCredential` or a reference to a certificate from the certificate store.
- **`AddTemporarySigningCredential`** Creates temporary key material at startup time. This is for dev only scenarios when you don't have a certificate to use.
- **`AddValidationKeys`** Adds keys for validating tokens. They will be used by the internal token validator and will show up in the discovery document. This is useful for key roll-over scenarios.

**In-Memory/Test stores**

- `AddInMemoryClients`
- `AddInMemoryIdentityResources`
- `AddInMemoryApiResources`
- `AddTestUsers`

**Additional services**

- `AddExtensionGrantValidator`
- `AddSecretParser`
- `AddSecretValidator`
- `AddResourceOwnerValidator`
- `AddProfileService`
- `AddAuthorizeInteractionResponseGenerator`
- `AddCustomAuthorizeRequestValidator`
- `AddCustomTokenRequestValidator`

**Caching**

- `AddClientStoreCache`
- `AddResourceStoreCache`

### 6.16.2 Configuring the pipeline

You need to add IdentityServer to the pipeline by calling:

```
public void Configure(IApplicationBuilder app)
{
    app.UseIdentityServer();
}
```

Be aware that order matters in the pipeline. You want to add IdentitySever e.g. before the UI framework that implementes the login screen etc.

## 6.17 Defining Resources

The first thing you will 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:** You can define resources using a C# object model - or load them from a data store. An implementation of `IResourceStore` deals with these low-level details. For this document we are using the in-memory implementation.

---

### 6.17.1 Defining identity resources

Identity resources are data like user ID, name, or email address of a user. An identity resource has a unique name, and you can assign arbitrary claim types to it. These claims will then be included in the identity token for the user. The client will use the `scope` parameter to request access to an identity resource.

The OpenID Connect specification specifies a couple of standard identity resources. The minimum requirement is, that you provide support for emitting a unique ID for your users - also called the subject id. This is done by exposing the standard identity resource called `openid`:

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

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

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

```
        new IdentityResources.Email(),
        new IdentityResources.Profile(),
        new IdentityResources.Telephone(),
        new IdentityResources.Address()
    };
}
```

### 6.17.2 Defining custom identity resources

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

```
public static IEnumerable<IdentityResource> GetIdentityResources()
{
    var customProfile = new IdentityResource(
        name: "custom.profile",
        displayName: "Custom profile",
        claimTypes: new[] { "name", "email", "status" });

    return new List<IdentityResource>
    {
        new IdentityResources.OpenId(),
        new IdentityResources.Profile(),
        customProfile
    };
}
```

See the *reference* section for more information on identity resource settings.

### 6.17.3 Defining API resources

To allow clients to request access tokens for APIs, you need to define API resources, e.g.:

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

```
public static IEnumerable<ApiResource> GetApis()
{
    return new[]
    {
        // simple API with a single scope (in this case the scope name is the same as the api name)
        new ApiResource("api1", "Some API 1"),

        // expanded version if more control is needed
        new ApiResource
        {
            Name = "api2",

            // secret for using introspection endpoint
            ApiSecrets =
            {
                new Secret("secret".Sha256())
            },

            // include the following using claims in access token (in addition to subject id)
            UserClaims = { JwtClaimTypes.Name, JwtClaimTypes.Email }
```

```
            },

            // this API defines two scopes
            Scopes =
            {
                new Scope()
                {
                    Name = "api2.full_access",
                    DisplayName = "Full access to API 2",
                },
                new Scope
                {
                    Name = "api2.read_only",
                    DisplayName = "Read only access to API 2"
                }
            }
        }
    };
}
```

See the *reference* section for more information on API resource settings.

## 6.18 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.

---

### 6.18.1 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 Secret("secret".Sha256()) },

                AllowedGrantTypes = GrantTypes.ClientCredentials,
                AllowedScopes = { "api1", "api2.read_only" }
            }
        };
    }
}
```

### 6.18.2 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 =           { "http://localhost:7017/index.html" },
    PostLogoutRedirectUris = { "http://localhost:7017/index.html" },
    AllowedCorsOrigins =     { "http://localhost:7017" },

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        IdentityServerConstants.StandardScopes.Email,

        "api1", "api2.read_only"
    }
};
```

### 6.18.3 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",
    ClientUri = "http://identityserver.io",

    AllowedGrantTypes = GrantTypes.Hybrid,
    AllowOfflineAccess = true,
    ClientSecrets = { new Secret("secret".Sha256()) },

    RedirectUris =           { "http://localhost:21402/signin-oidc" },
```

```
    PostLogoutRedirectUris = { "http://localhost:21402/" },
    LogoutUri =                "http://localhost:21402/signout-oidc",

    AllowedScopes =
    {
        IdentityServerConstants.StandardScopes.OpenId,
        IdentityServerConstants.StandardScopes.Profile,
        IdentityServerConstants.StandardScopes.Email,

        "api1", "api2.read_only"
    },
};
```

## 6.19 Connecting an MVC Application

You can integrate identityserver into your MVC application to authenticate users and request access token.

An MVC application typically uses the hybrid flow - use *this* sample to register the client.

In your MVC application startup, you can use the standard Microsoft ASP.NET OpenID Connect middleware to connect to identityserver:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

        app.UseStaticFiles();

        app.UseCookieAuthentication(new CookieAuthenticationOptions
        {
            AuthenticationScheme = "cookies",
            AutomaticAuthenticate = true,
        });

        var oidcOptions = new OpenIdConnectOptions
        {
            AuthenticationScheme = "oidc",
            SignInScheme = "cookies",

            Authority = "https://demo.identityserver.io",
            ClientId = "mvc",
            ClientSecret = "secret",
            ResponseType = "code id_token",
            SaveTokens = true,
            GetClaimsFromUserInfoEndpoint = true,

            TokenValidationParameters = new TokenValidationParameters
            {
                NameClaimType = "name",
                RoleClaimType = "role"
            }
        };

        oidcOptions.Scope.Clear();
        oidcOptions.Scope.Add("openid");
```

```
        oidcOptions.Scope.Add("profile");
        oidcOptions.Scope.Add("api1");

        app.UseOpenIdConnectAuthentication(oidcOptions);

        app.UseMvcWithDefaultRoutes();
    }
}
```

## 6.20 Protecting APIs

Protecting APIs with access tokens issued by your identityserver is easy - simply add our token validation middleware to the ASP.NET Core pipeline and configure the identityserver base address and the scope:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseIdentityServerAuthentication(new IdentityServerAuthenticationOptions
        {
            Authority = "https://demo.identityserver.io",
            AllowedScopes = { "api1" },
        });

        app.UseMvc();
    }
}
```

You can get the middleware from nuget or github.

## 6.21 Grant Types

Grant types are a way to specify how a client wants to interact with IdentityServer. The OpenID Connect and OAuth 2 specs define the following grant types:

- Implicit
- Authorization code
- Hybrid
- Client credentials
- Resource owner password
- Refresh tokens
- Extension grants

You can specify which grant type a client can use via the `AllowedGrantTypes` property on the `Client` configuration.

A client can be configured to use more than a single grant type (e.g. Hybrid for user centric operations and client credentials for server to server communication). The `GrantTypes` class can be used to pick from typical grant type combinations:

```
Client.AllowedGrantTypes = GrantTypes.HybridAndClientCredentials;
```

You can also specify the grant types list manually:

```
Client.AllowedGrantTypes = GrantTypes.List(
    GrantTypes.Hybrid,
    GrantType.ClientCredentials,
    "my_custom_grant_type");
```

If you want to transmit access tokens via the browser channel, you also need to allow that explicitly on the client configuration:

```
Client.AllowAccessTokensViaBrowser = true;
```

**Note:** For security reasons, not all grant type combinations are allowed. See below for more details.

For the remainder, the grant types are briefly described, and when you would use them. It is also recommended, that in addition you read the corresponding specs to get a better understanding of the differences.

### 6.21.1 Client credentials

This is the simplest grant type and is used for server to server communication - tokens are always requested on behalf of a client, not a user.

With this grant type you send a token request to the token endpoint, and get an access token back that represents the client. The client typically has to authenticate with the token endpoint using its client ID and secret.

See the *Client Credentials Quick Start* for a sample how to use it.

### 6.21.2 Resource owner password

The resource owner password grant type allows to request tokens on behalf of a user by sending the user's name and password to the token endpoint. This is so called "non-interactive" authentication and is generally not recommended.

There might be reasons for certain legacy or first-party integration scenarios, where this grant type is useful, but the general recommendation is to use an interactive flow like implicit or hybrid for user authentication instead.

See the Resource Owner Password Quick Start for a sample how to use it. You also need to provide code for the username/password validation which can be supplied by implementing the `IResourceOwnerPasswordValidator` interface. You can find more information about this interface *here*.

### 6.21.3 Implicit

The implicit grant type is optimized for browser-based applications. Either for user authentication-only (both server-side and JavaScript applications), or authentication and access token requests (JavaScript applications).

In the implicit flow, all tokens are transmitted via the browser, and advanced features like refresh tokens are thus not allowed.

*This* quickstart shows authentication for service-side web apps, and *this* shows JavaScript.

### 6.21.4 Authorization code

Authorization code flow was originally specified by OAuth 2, and provides a way to retrieve tokens on a back-channel as opposed to the browser front-channel. It also support client authentication.

While this grant type is supported on its own, it is generally recommended you combine that with identity tokens which turns it into the so called hybrid flow. Hybrid flow gives you important extra features like signed protocol responses.

### 6.21.5 Hybrid

Hybrid flow is a combination of the implicit and authorization code flow - it uses combinations of multiple grant types, most typically `code id_token`.

In hybrid flow the identity token is transmitted via the browser channel and contains the signed protocol response along with signatures for other artifacts like the authorization code. This mitigates a number of attacks that apply to the browser channel. After successful validation of the response, the back-channel is used to retrieve the access and refresh token.

This is the recommended flow for native applications that want to retrieve access tokens (and possibly refresh tokens as well) and is used for server-side web applications and native desktop/mobile applications.

See *this* quickstart for more information about using hybrid flow with MVC.

### 6.21.6 Refresh tokens

Refresh tokens allow gaining long lived access to APIs.

You typically want to keep the lifetime of access tokens as short as possible, but at the same time don't want to bother the user over and over again with doing a front-channel roundtrips to IdentityServer for requesting new ones.

Refresh tokens allow requesting new access tokens without user interaction. Every time the client refreshes a token it needs to make an (authenticated) back-channel call to IdentityServer. This allows checking if the refresh token is still valid, or has been revoked in the meantime.

Refresh tokens are supported in hybrid, authorization code and resource owner password flows. To request a refresh token, the client needs to include the `offline_access` scope in the token request (and must be authorized to for that scope).

### 6.21.7 Extension grants

Extension grants allow extending the token endpoint with new grant types. See *this* for more details.

### 6.21.8 Incompatible grant types

Some grant type combinations are forbidden:

- Mixing implicit and authorization code or hybrid would allow a downgrade attack from the more secure code based flow to implicit.
- Same concern exists for allowing both authorization code and hybrid

## 6.22 Secrets

In certain situations, clients need to authenticate with identityserver, e.g.

- confidential applications (aka clients) requesting tokens at the token endpoint
- APIs (aka resource scopes) validating reference tokens at the introspection endpoint

For that purpose you can assign a list of secrets to a `Client` or a `Scope`.

Secret parsing and validation is an extensibility point in identityserver, out of the box it supports shared secrets (stored hashed or plaintext - but defaults to hashed) as well as transmitting the shared secret via a basic authentication header or the POST body.

### 6.22.1 Creating a shared secret

The following code sets up a hashed shared secret:

```
var secret = new Secret("secret".Sha256());
```

This secret can now be assigned to either a `Client` or a `Scope`. Notice that both do not only support a single secret, but multiple. This is useful for secret rollover and rotation:

```
var client = new Client
{
    ClientId = "client",
    ClientSecrets = new List<Secret> { secret },

    AllowedGrantTypes = GrantTypes.ClientCredentials,
    AllowedScopes = new List<string>
    {
        "api1", "api2"
    }
};
```

In fact you can also assign a description and an expiration date to a secret. The description will be used for logging, and the expiration date for enforcing a secret lifetime:

```
var secret = new Secret(
    "secret".Sha256(),
    "2016 secret",
    new DateTime(2016, 12, 31));
```

### 6.22.2 Authentication using a shared secret

You can either send the client id/secret combination as part of the POST body:

```
POST /connect/token

client_id=client1&
client_secret=secret&
...
```

..or as a basic authentication header:

```
POST /connect/token

Authorization: Basic xxxxx

...
```

You can manually create a basic authentication header using the following C# code:

```
var credentials = string.Format("{0}:{1}", clientId, clientSecret);
var headerValue = Convert.ToBase64String(Encoding.UTF8.GetBytes(credentials));

var client = new HttpClient();
client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Basic", headerValue);
```

The IdentityModel library has helper classes called `TokenClient` and `IntrospectionClient` that encapsulate both authentication and protocol messages.

## 6.23 Extension Grants

OAuth 2.0 defines standard grant types for the token endpoint, such as `password`, `authorization_code` and `refresh_token`. Extension grants are a way to add support for non-standard token issuance scenarios like token translation, delegation, or custom credentials.

You can add support for additional grant types by implementing the `IExtensionGrantValidator` interface:

```
public interface IExtensionGrantValidator
{
    /// <summary>
    /// Handles the custom grant request.
    /// </summary>
    /// <param name="request">The validation context.</param>
    Task ValidateAsync(ExtensionGrantValidationContext context);

    /// <summary>
    /// Returns the grant type this validator can deal with
    /// </summary>
    /// <value>
    /// The type of the grant.
    /// </value>
    string GrantType { get; }
}
```

The `ExtensionGrantValidationContext` object gives you access to:

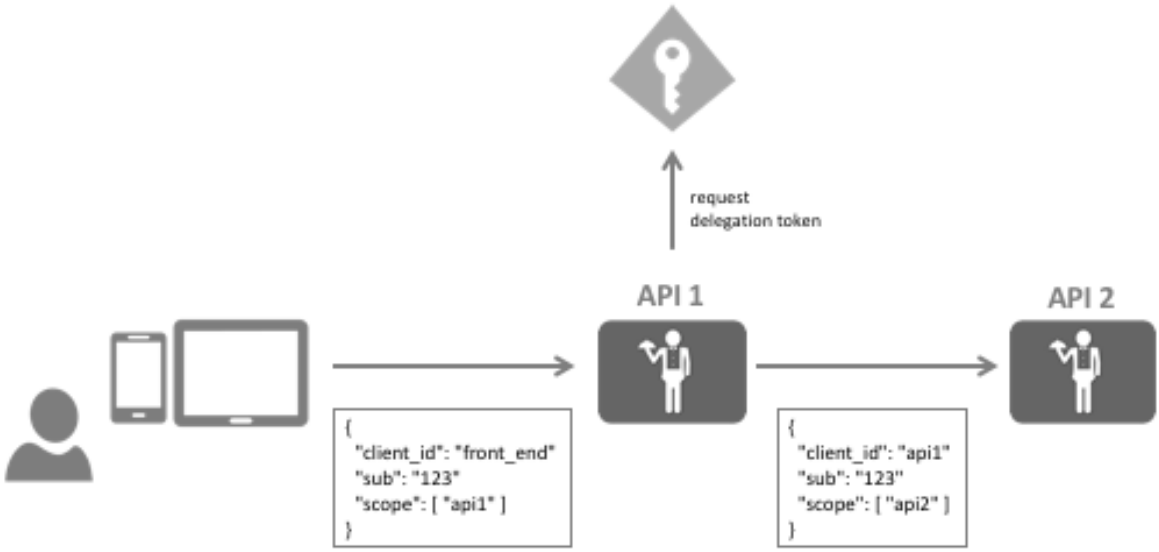
- the incoming token request - both the well-known validated values, as well as any custom values (via the `Raw` collection)
- the result - either error or success
- custom response parameters

To register the extension grant, add it to DI:

```
builder.AddExtensionGrantValidator<MyExtensionsGrantValidator>();
```

### 6.23.1 Example: Simple delegation using an extension grant

Imagine the following scenario - a front end client calls a middle tier API using a token acquired via an interactive flow (e.g. hybrid flow). This middle tier API (API 1) now wants to call a back end API (API 2) on behalf of the interactive user:




In other words, the middle tier API (API 1) needs an access token containing the user's identity, but with the scope of the back end API (API 2).

---

**Note:** You might have heard of the term *poor man's delegation* where the access token from the front end is simply forwarded to the back end. This has some short comings, e.g. *API 2* must now accept the *API 1* scope which would allow the user to call *API 2* directly. Also - you might want to add some delegation specific claims into the token, e.g. the fact that the call path is via *API 1*.

---

#### Implementing the extension grant

The front end would send the token to API 1, and now this token needs to be exchanged at IdentityServer with a new token for API 2.

On the wire the call to token service for the exchange could look like this:

```
POST /connect/token

grant_type=delegation&
scope=api2&
token=...&
client_id=api1.client
client_secret=secret
```

It's the job of the extension grant validator to handle that request by validating the incoming token, and returning a result that represents the new token:

```
public class DelegationGrantValidator : IExtensionGrantValidator
{
    private readonly ITokenValidator _validator;
```

```
    public DelegationGrantValidator(ITokenValidator validator)
    {
        _validator = validator;
    }

    public string GrantType => "delegation";

    public async Task ValidateAsync(ExtensionGrantValidationContext context)
    {
        var userToken = context.Request.Raw.Get("token");

        if (string.IsNullOrEmpty(userToken))
        {
            context.Result = new GrantValidationResult(TokenRequestErrors.InvalidGrant);
            return;
        }

        var result = await _validator.ValidateAccessTokenAsync(userToken);
        if (result.IsError)
        {
            context.Result = new GrantValidationResult(TokenRequestErrors.InvalidGrant);
            return;
        }

        // get user's identity
        var sub = result.Claims.FirstOrDefault(c => c.Type == "sub").Value;

        context.Result = new GrantValidationResult(sub, "delegation");
        return;
    }
}
```

Don't forget to register the validator with DI.

**Registering the delegation client**

You need a client registration in IdentityServer that allows a client to use this new extension grant, e.g.:

```
var client = new client
{
    ClientId = "api1.client",
    ClientSecrets = new List<Secret>
    {
        new Secret("secret".Sha256())
    },

    AllowedGrantTypes = GrantTypes.List("delegation"),

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

**Calling the token endpoint**

In API 1 you can now construct the HTTP payload yourself, or use the *IdentityModel* helper library:

```
public async Task<TokenResponse> DelegateAsync(string userToken)
{
```

```
    var payload = new
    {
        token = userToken
    };

    // create token client
    var client = new TokenClient(disco.TokenEndpoint, "api1.client", "secret");

    // send custom grant to token endpoint, return response
    return await client.RequestCustomGrantAsync("delegation", "api2", payload);
}
```

The `TokenResponse.AccessToken` will now contain the delegation access token.

## 6.24 Resource Owner Password Validation

If you want to use the OAuth 2.0 resource owner password credential grant (aka `password`), you need to implement and register the `IResourceOwnerPasswordValidator` interface:

```
public interface IResourceOwnerPasswordValidator
{
    /// <summary>
    /// Validates the resource owner password credential
    /// </summary>
    /// <param name="context">The context.</param>
    Task ValidateAsync(ResourceOwnerPasswordValidationContext context);
}
```

On the context you will find already parsed protocol parameters like `UserName` and `Password`, but also the raw request if you want to look at other input data.

Your job is then to implement the password validation and set the `Result` on the context accordingly. See the *GrantValidationResult* documentation.

## 6.25 Cryptography, Keys and HTTPS

IdentityServer relies on a couple of crypto mechanisms to do its job.

### 6.25.1 Token signing and validation

IdentityServer needs an asymmetric key pair to sign and validate JWTs. This keypair can can be a certificate/private key combination or raw RSA keys. In any case it must support RSA with SHA256.

Loading of signing key and the corresponding validation part is done by implementations of `ISigningCredentialStore` and `IValidationKeysStore`. If you want to customize the loading of the keys, you can implement those interfaces and register them with DI.

The DI builder extensions has a couple of convenience methods to set signing and validation keys.

`AddSigningCredential` allows setting either an RSA key or a certificate from the store or a file.

`AddTemporarySigningCredential` creates a fresh RSA key pair on every startup. This is useful for development situations where you don't have access to key material.

Example:

```
services.AddIdentityServer()
  .AddSigningCredential("CN=sts");
```

### 6.25.2 Signing key rollover

While you can only use one signing key at a time, you can publish more than one validation key to the discovery document. This is useful for key rollover.

A rollover typically works like this:

1. you request/create new key material
2. you publish the new validation key in addition to the current one. You can use the `AddValidationKeys` builder extension method for that.
3. all clients and APIs now have a chance to learn about the new key the next time they update their local copy of the discovery document
4. after a certain amount of time (e.g. 24h) all clients and APIs should now accept both the old and the new key material
5. keep the old key material around for as long as you like, maybe you have long-lived tokens that need validation
6. retire the old key material when it is not used anymore
7. all clients and APIs will "forget" the old key next time they update their local copy of the discovery document

This requires that clients and APIs use the discovery document, and also have a feature to periodically refresh their configuration.

### 6.25.3 Data protection

We use the ASP.NET Core data protection API. For the most parts this requires no manual configuration - some adjustments might be needed depending on your deployment scenario (e.g. self-hosted web farms). See here for more information.

### 6.25.4 HTTPS

We don't enforce the use of HTTPS, but for production it is mandatory for every interaction with IdentityServer.

HTTPS is typically provided by the reverse proxy that sits in front of ASP.NET Core's built-in webser, here are some instructions for using IIS, here for Apache.

## 6.26 Deployment

## 6.27 External Identity Provider

## 6.28 Sign-out

## 6.29 Logging

## 6.30 Refresh Tokens

## 6.31 Reference Tokens

## 6.32 Windows Authentication

On supported platforms, you can use IdentityServer to authenticate users using Windows authentication (e.g. against Active Directory). Currently Windows authentication is available when you host IdentityServer using:

- Kestrel on Windows using IIS and the IIS integration package
- WebListener on Windows

In both cases, Windows authentication is treated as external authentication that has to be invoked using an ASP.NET authentication manager challenge command. The account controller in our quickstart UI implements the necessary logic.

### 6.32.1 Using WebListener

When using WebListener you need to enable Windows authentication when setting up the host, e.g.:

```
var host = new WebHostBuilder()
    .UseWebListener(options =>
    {
        options.ListenerSettings.Authentication.Schemes = AuthenticationSchemes.Negotiate | Authentic
        options.ListenerSettings.Authentication.AllowAnonymous = true;
    })
    .UseUrls("https://myserver:443")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseStartup<Startup>()
    .Build();
```

The WebListener plumbing will insert Windows authentication middleware for each authentication scheme you selected. You can enumerate the schemes by using the ASP.NET Core authentication manager `GetAvailableSchemes` method, and invoke it using the `ChallengeAsync` method.

### 6.32.2 Using Kestrel

When using Kestrel, you must run "behind" IIS and use the IIS integration:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://localhost:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

Also the virtual directory in IIS (or IIS Express) must have Windows and anonymous authentication enabled.

Just as WebListener, the IIS integration will insert a Windows authentication middleware into the HTTP pipeline that can be invoked via the authentication manager.

## 6.33 Discovery Endpoint

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

The discovery endpoint is available via */.well-known/openid-configuration* relative to the base address, e.g.:

```
https://demo.identityserver.io/.well-known/openid-configuration
```

### 6.33.1 IdentityModel

You can programmatically access the discovery endpoint using the IdentityModel library:

```
var discoveryClient = new DiscoveryClient("https://demo.identityserver.io");
var doc = await discoveryClient.GetAsync();

var tokenEndpoint = doc.TokenEndpoint;
var keys = doc.KeySet.Keys;
```

For security reasons DiscoveryClient has a configurable validation policy that checks the following rules by default:

- HTTPS must be used for the discovery endpoint and all protocol endpoints
- The issuer name should match the authority specified when downloading the document (that's actually a MUST in the discovery spec)
- The protocol endpoints should be "beneath" the authority – and not on a different server or URL (this could be especially interesting for multi-tenant OPs)
- A key set must be specified

If for whatever reason (e.g. dev environments) you need to relax a setting, you can use the following code:

```
var client = new DiscoveryClient("http://dev.identityserver.internal");
client.Policy.RequireHttps = false;

var disco = await client.GetAsync();
```

Btw – you can always connect over HTTP to localhost and 127.0.0.1 (but this is also configurable).

## 6.34 Authorize Endpoint

The authorize endpoint can be used to request tokens or authorization codes via the browser. This process typically involves authentication of the end-user and optionally consent.

---

**Note:** IdentityServer supports a subset of the OpenID Connect and OAuth 2.0 authorize request parameters. For a full list, see here.

---

**client_id** identifier of the client (required).

**scope** one or more registered scopes (required)

**redirect_uri** must exactly match one of the allowed redirect URIs for that client (required)

**response_type** `id_token` requests an identity token (only identity scopes are allowed)

`token` requests an access token (only resource scopes are allowed)

`id_token token` requests an identity token and an access token

`code` requests an authorization code

`code id_token` requests an authorization code and identity token

`code id_token token` requests an authorization code, identity token and access token

**response_mode** `form_post` sends the token response as a form post instead of a fragment encoded redirect (optional)

**state** identityserver will echo back the state value on the token response, this is for round tripping state between client and provider, correlating request and response and CSRF/replay protection. (recommended)

**nonce** identityserver will echo back the nonce value in the identity token, this is for replay protection)

*Required for identity tokens via implicit grant.*

**prompt** `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** sends the code challenge for PKCE

**code_challenge_method** `plain` indicates that the challenge is using plain text (not recommended) `S256` indicates the the challenge is hashed with SHA256

**login_hint** can be used to pre-fill the username field on the login page

**ui_locales** gives a hint about the desired display language of the login UI

**max_age** if the user's logon session exceeds the max age (in seconds), the login UI will be shown

**acr_values** allows passing in additional authentication related information - identityserver special cases the following proprietary acr_values:

`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 login UI

**Example**

```
GET /connect/authorize?
    client_id=client1&
    scope=openid email api1&
    response_type=id_token token&
    redirect_uri=https://myapp/callback&
    state=abc&
    nonce=xyz
```

(URL encoding removed, and line breaks added for readability)

### 6.34.1 IdentityModel

You can programmatically create URLs for the authorize endpoint using the IdentityModel library:

```
var request = new AuthorizeRequest(doc.AuthorizeEndpoint);
var url = request.CreateAuthorizeUrl(
    clientId:     "client",
    responseType: OidcConstants.ResponseTypes.CodeIdToken,
    responseMode: OidcConstants.ResponseModes.FormPost,
    redirectUri: "https://myapp.com/callback",
    state:       CryptoRandom.CreateUniqueId(),
    nonce:       CryptoRandom.CreateUniqueId());
```

..and parse the response:

```
var response = new AuthorizeResponse(url);

var accessToken = response.AccessToken;
var idToken = response.IdentityToken;
var state = response.State;
```

## 6.35 Token Endpoint

The token endpoint can be used to programmatically request tokens. It supports the `password`, `authorization_code`, `client_credentials` and `refresh_token` grant types). Furthermore the token endpoint can be extended to support extension grant types.

---

**Note:** IdentityServer supports a subset of the OpenID Connect and OAuth 2.0 token request parameters. For a full list, see here.

---

**`client_id`** client identifier (required)

**`client_secret`** client secret either in the post body, or as a basic authentication header. Optional.

**`grant_type`** `authorization_code`, `client_credentials`, `password`, `refresh_token` or custom

**`scope`** one or more registered scopes. If not specified, a token for all explicitly allowed scopes will be issued.

**`redirect_uri`** required for the `authorization_code` grant type

**`code`** the authorization code (required for `authorization_code` grant type)

**`code_verifier`** PKCE proof key

**`username`** resource owner username (required for `password` grant type)

**password** resource owner password (required for `password` grant type)

**acr_values** allows passing in additional authentication related information for the `password` grant type - identityserver special cases the following proprietary acr_values:

`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 token endpoint

**refresh_token** the refresh token (required for `refresh_token` grant type)

### 6.35.1 Example

```
POST /connect/token

    client_id=client1&
    client_secret=secret&
    grant_type=authorization_code&
    code=hdh922&
    redirect_uri=https://myapp.com/callback
```

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

### 6.35.2 IdentityModel

You can programmatically access the token endpoint using the IdentityModel library:

```
var client = new TokenClient(
    doc.TokenEndpoint,
    "client_id",
    "secret");

var response = await client.RequestClientCredentialsAsync("scope");
var token = response.AccessToken;
```

## 6.36 UserInfo Endpoint

The UserInfo endpoint can be used to retrieve identity information about a user (see spec).

The caller needs to send a valid access token representing the user. Depending on the granted scopes, the UserInfo endpoint will return the mapped claims (at least the *openid* scope is required).

### 6.36.1 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"
    ]
}
```

### 6.36.2 IdentityModel

You can programmatically access the userinfo endpoint using the IdentityModel library:

```
var userInfoClient = new UserInfoClient(doc.UserInfoEndpoint, token);

var response = await userInfoClient.GetAsync();
var claims = response.Claims;
```

## 6.37 Introspection Endpoint

The introspection endpoint is an implementation of RFC 7662.

It can be used to validate reference tokens (or JWTs if the consumer does not have support for appropriate JWT or cryptographic libraries). The introspection endpoint requires authentication using a scope secret.

### 6.37.1 Example

```
POST /connect/introspect
Authorization: Basic xxxyyy

token=<token>
```

A successful response will return a status code of 200 and either an active or inactive token:

```
{
    "active": true,
    "sub": "123"
}
```

Unknown or expired tokens will be marked as inactive:

```
{
    "active": false,
}
```

An invalid request will return a 400, an unauthorized request 401.

### 6.37.2 IdentityModel

You can programmatically access the introspection endpoint using the IdentityModel library:

```
var introspectionClient = new IntrospectionClient(
    doc.IntrospectionEndpoint,
    "scope_name",
    "scope_secret");

var response = await introspectionClient.SendAsync(
    new IntrospectionRequest { Token = token });

var isActice = response.IsActive;
var claims = response.Claims;
```

## 6.38 Revocation Endpoint

This endpoint allows revoking access tokens (reference tokens only) and refresh token. It implements the token revocation specification (RFC 7009).

**token** the token to revoke (required)

**token_type_hint** either `access_token` or `refresh_token` (optional)

### 6.38.1 Example

```
POST /connect/revocation HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

token=45ghiukldjahdnhzdauz&token_type_hint=refresh_token
```

### 6.38.2 IdentityModel

You can programmatically revoke tokens using the IdentityModel library:

```
var revocationClient = new TokenRevocationClient(
    RevocationEndpoint,
    "client",
    "secret");

var response = await revocationClient.RevokeAccessTokenAsync(token);
```

## 6.39 Identity Resource

This class model an identity resource.

**Enabled** Indicates if this resource is enabled and can be requested. Defaults to true.

**Name** The unique name of the identity resource. This is the value a client will use for the scope parameter in the authorize request.

**DisplayName** This value will be used e.g. on the consent screen.

**Description** This value will be used e.g. on the consent screen.

**Required** Specifies whether the user can de-select the scope on the consent screen (if the consent screen wants to implement such a feature). Defaults to false.

**Emphasize** Specifies whether the consent screen will emphasize this scope (if the consent screen wants to implement such a feature). Use this setting for sensitive or important scopes. Defaults to false.

**ShowInDiscoveryDocument** Specifies whether this scope is shown in the discovery document. Defaults to true.

**UserClaims** List of associated user claim types that should be included in the identity token.

## 6.40 API Resource

This class model an API resource.

**Enabled** Indicates if this resource is enabled and can be requested. Defaults to true.

**Name** The unique name of the API. This value is used for authentication with introspection and will be added to the audience of the outgoing access token.

**DisplayName** This value can be used e.g. on the consent screen.

**Description** This value can be used e.g. on the consent screen.

**ApiSecrets** The API secret is used for the introspection endpoint. The API can authenticate with introspection using the API name and secret.

**UserClaims** List of associated user claim types that should be included in the access token.

**Scopes** An API must have at least one scope. Each scope can have different settings.

### 6.40.1 Scopes

In the simple case an API has exactly one scope. But there a cases where you might want to sub-divide the functionality of an API, and give different clients access to different parts.

**Name** The unique name of the scope. This is the value a client will use for the scope parameter in the authorize/token request.

**DisplayName** This value can be used e.g. on the consent screen.

**Description** This value can be used e.g. on the consent screen.

**Required** Specifies whether the user can de-select the scope on the consent screen (if the consent screen wants to implement such a feature). Defaults to false.

**Emphasize** Specifies whether the consent screen will emphasize this scope (if the consent screen wants to implement such a feature). Use this setting for sensitive or important scopes. Defaults to false.

**ShowInDiscoveryDocument** Specifies whether this scope is shown in the discovery document. Defaults to true.

**UserClaims** List of associated user claim types that should be included in the access token. The claims specified here will be added to the list of claims specified for the API.

## 6.41 Client

The `Client` class models an OpenID Connect or OAuth 2.0 client - e.g. a native application, a web application or a JS-based application.

### 6.41.1 Basics

**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.

**RequireClientSecret** Specifies whether this client needs a secret to request tokens from the token endpoint (defaults to `true`)

**AllowedGrantTypes** Specifies the grant types the client is allowed to use. Use the `GrantTypes` class for common combinations.

**RequirePkce** Specifies whether clients using an authorization code based grant type must send a proof key

**AllowPlainTextPkce** Specifies whether clients using PKCE can use a plain text code challenge (not recommended - and default to `false`)

**RedirectUris** Specifies the allowed URIs to return tokens or authorization codes to

**AllowedScopes** By default a client has no access to any resources - specify the allowed resources by adding the corresponding scopes names

**AllowOfflineAccess** Specifies whether this client can request refresh tokens (be requesting the `offline_access` scope)

**AllowAccessTokenViaBrowser** Specifies whether this client is allowed to receive 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.

### 6.41.2 Authentication/Logout

**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.

**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.

### 6.41.3 Token

**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*).

**note** production usage of reference tokens requires and implementation of `ITokenHandleStore`.

**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*.

### 6.41.4 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*.

**note** production usage of that features requires an implementation of `IConsentStore`.

**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)

## 6.42 GrantValidationResult

The `GrantValidationResult` class models the outcome of grant validation for extensions grants and resource owner password grants.

The most common usage is to either new it up using an identity (success case):

```
context.Result = new GrantValidationResult(
    subject: "818727",
    authenticationMethod: "custom",
    claims: optionalClaims);
```

...or using an error and description (failure case):

```
context.Result = new GrantValidationResult(
    TokenRequestErrors.InvalidGrant,
    "invalid custom credential");
```

In both case you can pass additional custom values that will be included in the token response.

## 6.43 IdentityServer Options

- **IssuerUri** Set the issuer name that will appear in the discovery document and the issued JWT tokens. It is recommended to not set this property, which infers the issuer name from the host name that is used by the clients.
- **ProtocolLogoutUrls`** todo

### 6.43.1 Endpoints

Allows enabling/disabling individual endpoints, e.g. token, authorize, userinfo etc.

By default all endpoints are enabled, but you can lock down your server by disbling endpoint that you don't need.

### 6.43.2 Discovery

Allows enabling/disabling various sections of the discovery document, e.g. endpoints, scopes, claims, grant types etc.

The `CustomEntries` dictionary allows adding custom elements to the discovery document.

### 6.43.3 Authentication

- **AuthenticationScheme** If set, specifies the cookie middleware you want to use. If not set, IdentityServer will use a built-in cookie middleware with default values.
- **RequireAuthenticatedUserForSignOutMessage`** Indicates if user must be authenticated to accept parameters to end session endpoint. Defaults to false.
- **FederatedSignOutPaths** todo

### 6.43.4 Events

todo.

### 6.43.5 InputLengthRestrictions

Allows setting length restrictions on various protocol parameters like client id, scope, redirect URI etc.

### 6.43.6 UserInteraction

- **LoginUrl**, **LogoutUrl**, **ConsentUrl**, **ErrorUrl** Sets the the URLs for the login, logout, consent and error pages.
- **LoginReturnUrlParameter** Sets the name of the login return URL parameter. Default to *returnUrl*
- `LogoutIdParameter`
- `ConsentReturnUrlParameter`
- `ErrorIdParameter`
- `CustomRedirectReturnUrlParameter`
- `CookieMessageThreshold`

### 6.43.7 Caching

- `ClientStoreExpiration`
- `ResourceStoreExpiration`

### 6.43.8 CORS

- `CorsPolicyName`
- `CorsPaths`

## 6.44 Training

### 6.44.1 Our workshop

Brock and Dominick are regularly doing workshops around identity & acess control for modern applications. Check the agenda and upcoming dates here.

### 6.44.2 PluralSight courses

- Introduction to OAuth2, OpenID Connect and JSON Web Tokens (JWT)
- Web API v2 Security
- Using OAuth to Secure Your ASP.NET API
- OAuth2 and OpenID Connect Strategies for Angular and ASP.NET

## 6.45 Blog posts

### 6.45.1 Team posts

- Announcing IdentityServer4 RC1
- Platforms where you can run IdentityServer4
- Optimizing Tokens for size
- Identity vs Permissions
- Bootstraping OpenID Connect: Discovery

### 6.45.2 What's new posts

- New in IdentityServer4: Clients without Secrets
- New in IdentityServer4: Default Scopes
- New in IdentityServer4: Support for Extension Grants
- New in IdentityServer4: Resource Owner Password Validation
- New in IdentityServer4: Resource-based Configuration

### 6.45.3 Community posts

- Getting Started with IdentityServer 4
- Angular2 OpenID Connect Implicit Flow with IdentityServer4
- Full Server Logout with IdentityServer4 and OpenID Connect Implicit Flow
- IdentityServer4, ASP.NET Identity, Web API and Angular in a single Project

## 6.46 Videos

### 6.46.1 2016

- Authentication & secure API access for native & mobile Applications - Dominick Baier
- ASP.NET Identity 3 - Brock Allen
- Introduction to IdentityServer3 - Brock Allen

### 6.46.2 2015

- Securing Web APIs – Patterns & Anti-Patterns - Dominick Baier
- Authentication and authorization in modern JavaScript web applications – how hard can it be? - Brock Allen

### 6.46.3 2014

- Unifying Authentication & Delegated API Access for Mobile, Web and the Desktop with OpenID Connect and OAuth 2 - Dominick Baier