[Robin van der Knaap](https://dev.to/robinvanderknaap)

[Robin van der Knaap](https://dev.to/robinvanderknaap)

Posted on Dec 11, 2020 • Updated on Dec 24, 2020

**Setting up an Authorization Server with OpenIddict - Part III - Client Credentials Flow**

[#dotnet](https://dev.to/t/dotnet)[#oauth2](https://dev.to/t/oauth2)[#openiddict](https://dev.to/t/openiddict)[#aspnet](https://dev.to/t/aspnet)

This article is part of a series called **Setting up an Authorization Server with OpenIddict**. The articles in this series will guide you through the process of setting up an OAuth2 + OpenID Connect authorization server on the the [ASPNET Core](https://dotnet.microsoft.com/apps/aspnet) platform using [OpenIddict](https://github.com/openiddict/openiddict-core).

* [Part I: Introduction](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-i-introduction-4jid)
* [Part II: Create ASPNET project](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-ii-create-aspnet-project-4949)
* **Part III: Client Credentials Flow**
* [Part IV: Authorization Code Flow](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iv-authorization-code-flow-3eh8)
* [Part V: OpenID Connect](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-v-openid-connect-a8j)
* [Part VI: Refresh tokens](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-vi-refresh-tokens-5669)

GitHub logo [robinvanderknaap](https://github.com/robinvanderknaap)/ [**authorization-server-openiddict**](https://github.com/robinvanderknaap/authorization-server-openiddict)

Authorization Server implemented with OpenIddict.

In this part we will add OpenIddict to the project and implement out first authorization flow: The [Client Credentials Flow](https://oauth.net/2/grant-types/client-credentials/).

**Add OpenIddict packages**

First, we need to install the OpenIddict NuGet packages:

dotnet add package OpenIddict

dotnet add package OpenIddict.AspNetCore

dotnet add package OpenIddict.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.InMemory

Besides the main library we also installed the OpenIddict.AspNetCore package, this package enables the integration of OpenIddict in an ASPNET Core host.

OpenIddict.EntityFrameworkCore package enables Entity Framework Core support. For now we will work with an in-memory implementation, for which we use the package Microsoft.EntityFrameworkCore.InMemory.

**Setup OpenIddict**

We will start with what is minimal required to get OpenIddict up and running. At least one OAuth 2.0/OpenID Connect flow must be enabled. We choose to enable the the [Client Credentials Flow](https://oauth.net/2/grant-types/client-credentials/), which is suitable for machine-to-machine applications. In the next part of this series we will implement the [Authorization Code Flow with PKCE](https://oauth.net/2/grant-types/authorization-code/) which is the recommended flow for Single Page Applications (SPA) and native/mobile applications.

Start to make the following changes to Startup.cs:

public void ConfigureServices(IServiceCollection services)

{

...

services.AddDbContext<DbContext>(options =>

{

// Configure the context to use an in-memory store.

options.UseInMemoryDatabase(nameof(DbContext));

// Register the entity sets needed by OpenIddict.

options.UseOpenIddict();

});

services.AddOpenIddict()

// Register the OpenIddict core components.

.AddCore(options =>

{

// Configure OpenIddict to use the EF Core stores/models.

options.UseEntityFrameworkCore()

.UseDbContext<DbContext>();

})

// Register the OpenIddict server components.

.AddServer(options =>

{

options

.AllowClientCredentialsFlow();

options

.SetTokenEndpointUris("/connect/token");

// Encryption and signing of tokens

options

.AddEphemeralEncryptionKey()

.AddEphemeralSigningKey();

// Register scopes (permissions)

options.RegisterScopes("api");

// Register the ASP.NET Core host and configure the ASP.NET Core-specific options.

options

.UseAspNetCore()

.EnableTokenEndpointPassthrough();

});

}

First, the DbContext is registered in the ConfigureServices method. OpenIddict natively supports Entity Framework Core, Entity Framework 6 and MongoDB out-of-the-box, and you can also provide your own stores.  
In this example, we will use Entity Framework Core, and we will use an in-memory database. The options.UseOpenIdDict call registers the entity sets needed by OpenIddict.

Next, OpenIddict itself is registered. The AddOpenIddict() call registers the OpenIddict services and returns a OpenIddictBuilder class with which we can configure OpenIddict.

The core components are registered first. OpenIddict is instructed to use Entity Framework Core, and use the aforementioned DbContext.

Next, the server components are registered and the Client Credentials Flow is enabled. For this flow to work, we need to register a token endpoint. We need to implement this endpoint ourselves. We'll do this later.

For OpenIddict to be able to encrypt and sign tokens we need to register two keys, one for encrypting and one for signing. In this example we'll use ephemeral keys. Ephemeral keys are automatically discarded when the application shuts down and payloads signed or encrypted using these key are therefore automatically invalidated. This method should only be used during development. On production, using a X.509 certificate is recommended.

RegisterScopes defines which scopes (permissions) are supported. In this case we have one scope called api, but the authorization server can support multiple scopes.

The UseAspNetCore() call is used to setup AspNetCore as a host for OpenIddict. We also call EnableTokenEndpointPassthrough otherwise requests to our future token endpoint are blocked.

To check if OpenIddict is properly configured we can start the application and navigate to: <https://localhost:5001/.well-known/openid-configuration>, the response should be something like this:

{

"issuer": "https://localhost:5001/",

"token\_endpoint": "https://localhost:5001/connect/token",

"jwks\_uri": "https://localhost:5001/.well-known/jwks",

"grant\_types\_supported": [

"client\_credentials"

],

"scopes\_supported": [

"openid",

"api"

],

"claims\_supported": [

"aud",

"exp",

"iat",

"iss",

"sub"

],

"id\_token\_signing\_alg\_values\_supported": [

"RS256"

],

"subject\_types\_supported": [

"public"

],

"token\_endpoint\_auth\_methods\_supported": [

"client\_secret\_basic",

"client\_secret\_post"

],

"claims\_parameter\_supported": false,

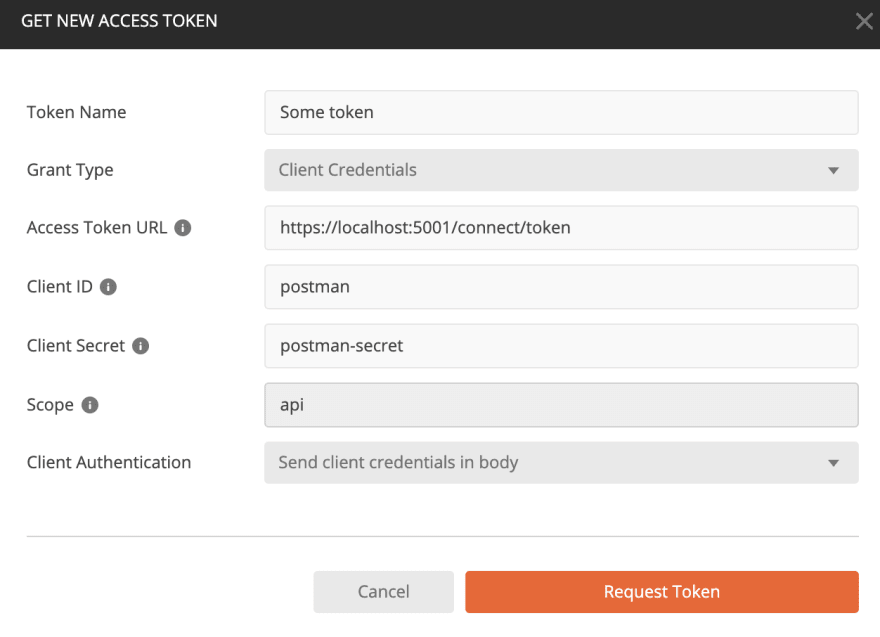
"request\_parameter\_supported": false,

"request\_uri\_parameter\_supported": false

}

In this guide, we will be using [Postman](https://www.postman.com/) to test the authorization server, but you could just as easily use another tool.

Below you find an example authorization request using Postman. Grant Type is the Client Credentials Flow. We specify the access token url, a client id and secret to authenticate our client. We also request access to the api scope.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--kWiRtqzh--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/bh84rtf29mcmn71mtul7.png)

If we request the token, the operation will fail: The client\_id is invalid. This makes sense, because we haven't registered any clients yet with our Authorization Server.  
We can create a client by adding it to the database. For that purpose we create a class called TestData. The test data implements the IHostedService interface, which enables us to execute the generation of test data in Startup.cs when the application starts.

using System;

using System.Threading;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using OpenIddict.Abstractions;

namespace AuthorizationServer

{

public class TestData : IHostedService

{

private readonly IServiceProvider \_serviceProvider;

public TestData(IServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

public async Task StartAsync(CancellationToken cancellationToken)

{

using var scope = \_serviceProvider.CreateScope();

var context = scope.ServiceProvider.GetRequiredService<DbContext>();

await context.Database.EnsureCreatedAsync(cancellationToken);

var manager = scope.ServiceProvider.GetRequiredService<IOpenIddictApplicationManager>();

if (await manager.FindByClientIdAsync("postman", cancellationToken) is null)

{

await manager.CreateAsync(new OpenIddictApplicationDescriptor

{

ClientId = "postman",

ClientSecret = "postman-secret",

DisplayName = "Postman",

Permissions =

{

OpenIddictConstants.Permissions.Endpoints.Token,

OpenIddictConstants.Permissions.GrantTypes.ClientCredentials,

OpenIddictConstants.Permissions.Prefixes.Scope + "api"

}

}, cancellationToken);

}

}

public Task StopAsync(CancellationToken cancellationToken) => Task.CompletedTask;

}

}

A client is registered in the test data. The client id and secret are used to authenticate the client with the authorization server. The permissions determine what this client's options are.  
In this case we allow for the client to use the Client Credentials Flow, access the token endpoint and allow the client to request the api scope.

Register the test data service in Startup.cs, so it is executed when the application starts:

public void ConfigureServices(IServiceCollection services)

{

...

services.AddHostedService<TestData>();

}

If we try to obtain an access token with Postman again, the request will still fail. This is because we haven't created the token endpoint yet. We'll do that now.

Create a new controller called AuthorizationController, we will host the endpoint here:

using System;

using System.Security.Claims;

using Microsoft.AspNetCore;

using Microsoft.AspNetCore.Mvc;

using OpenIddict.Abstractions;

using OpenIddict.Server.AspNetCore;

namespace AuthorizationServer.Controllers

{

public class AuthorizationController : Controller

{

[HttpPost("~/connect/token")]

public IActionResult Exchange()

{

var request = HttpContext.GetOpenIddictServerRequest() ??

throw new InvalidOperationException("The OpenID Connect request cannot be retrieved.");

ClaimsPrincipal claimsPrincipal;

if (request.IsClientCredentialsGrantType())

{

// Note: the client credentials are automatically validated by OpenIddict:

// if client\_id or client\_secret are invalid, this action won't be invoked.

var identity = new ClaimsIdentity(OpenIddictServerAspNetCoreDefaults.AuthenticationScheme);

// Subject (sub) is a required field, we use the client id as the subject identifier here.

identity.AddClaim(OpenIddictConstants.Claims.Subject, request.ClientId ?? throw new InvalidOperationException());

// Add some claim, don't forget to add destination otherwise it won't be added to the access token.

identity.AddClaim("some-claim", "some-value", OpenIddictConstants.Destinations.AccessToken);

claimsPrincipal = new ClaimsPrincipal(identity);

claimsPrincipal.SetScopes(request.GetScopes());

}

else

{

throw new InvalidOperationException("The specified grant type is not supported.");

}

// Returning a SignInResult will ask OpenIddict to issue the appropriate access/identity tokens.

return SignIn(claimsPrincipal, OpenIddictServerAspNetCoreDefaults.AuthenticationScheme);

}

}

}

One action is implemented, Exchange. This action is used by all flows, not only the Client Credentials Flow, to obtain an access token.

In the case of the Client Credentials Flow, the token is issued based on the client credentials. In the case of Authorization Code Flow, the same endpoint is used but then to exchange an authorization code for a token. We'll see that in [part IV](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iv-authorization-code-flow-3eh8).

For now, we need to focus on the Client Credentials Flow. When the request enters the Exchange action, the client credentials (ClientId and ClientSecret) are already validated by OpenIddict. So we don't need to authenticate the request, we only have to create a claims principal and use that to sign in.

The claims principal is not the same as we used in the Account controller, that one was based on the Cookie authentication handler and is only used within the context of the Authorization server itself to determine if the user has been authenticated or not.

The claims principal we have to create is based on the OpenIddictServerAspNetCoreDefaults.AuthenticationScheme. This way, when we call SignIn at the end of this method, the OpenIddict middleware will handle the sign in and return an access token as response to the client.

The claims defined in the claims principal are included in the access token only when we specify the destination. The some-value claim in the example will be added to the access token.  
The subject claim is required and you don't need to specify a destination, it will be included in the access token anyway.

We also grant all the requested scopes by calling claimsPrincipal.SetScopes(request.GetScopes());. OpenIddict has already checked if the requested scopes are allowed (in general and for the current client). The reason why we have to add the scopes manually here is that we are able to filter the scopes granted here if we want to.

**One token to rule them all**

Let's try to obtain an access token with Postman again, this time it should work.

As of v3 of OpenIddict, the access token is in Jason Web Token (JWT) format by default. This enables us to examine the token with [jwt.io](https://jwt.io/). (Thanks [Auth0](https://auth0.com/), for hosting that service!)

One problem, the token is not only signed, but also encrypted. OpenIddict encrypts the access token by default. We can disable this encryption when configuring OpenIddict in Startup.cs:

// Encryption and signing of tokens

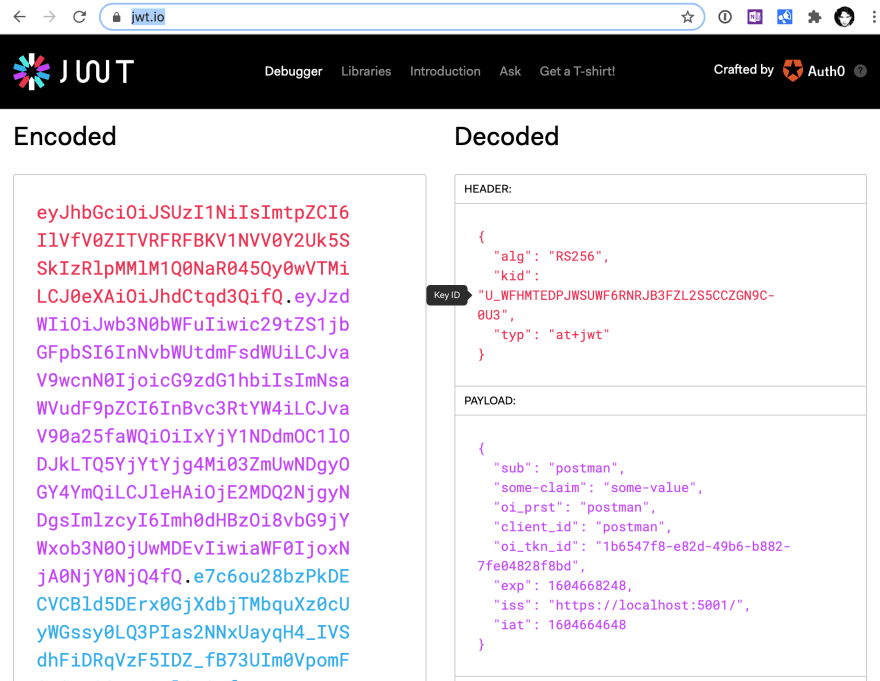
options

.AddEphemeralEncryptionKey()

.AddEphemeralSigningKey()

.DisableAccessTokenEncryption();

Now, when we restart our authorization server and request a new token. Paste the token to [jwt.io](https://jwt.io/) and view the content of the token:

[](https://res.cloudinary.com/practicaldev/image/fetch/s--4q9tnUjj--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/dev-to-uploads.s3.amazonaws.com/i/u8bfyj3blsccb5egd764.png)

As you can see the client id postman is set as subject sub. Also the some-claim claim is added to the access token.

**Next**

Congratulations, you have implemented the Client Credentials Flow with OpenIddict!

As you may have noticed, the login page is unused by the Client Credentials Flow. This flow exchanges the client credentials for a token immediately, which is suitable for machine-to-machine applications.

[Next up](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iv-authorization-code-flow-3eh8), we will implement the Authorization Code Flow with PKCE, which is the recommended flow for single page applications (SPA) and mobile apps. This flow will involve the user, so our login page will come in to play.

**Discussion (5)**

Subscribe

Top of Form





Bottom of Form

[hdsoftware profile image](https://dev.to/hdsoftware)

HD Software

•[Jul 12 '21](https://dev.to/hdsoftware/comment/1g7mn)

Hi.  
When I finnished up this page I get an error frm JWT.IO saying

Error: Looks like tour JWT payload is not a valid JSON object. JWT payloads must be top level JSON objects as per [tools.ietf.org/html/rfc7519#sectio...](https://tools.ietf.org/html/rfc7519#section-7.2)

Did I do something wrong in my coding or is this a normal behaviour ?

1 like[Reply](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iii-client-credentials-flow-55lp#/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iii-client-credentials-flow-55lp/comments/new/1g7mn)

[hdsoftware profile image](https://dev.to/hdsoftware)

HD Software

•[Jul 12 '21](https://dev.to/hdsoftware/comment/1g7mp)

hahaha. And this is what happens when you dont read the complete instructions :D  
Simply forgot to add the DisableEncryption method :D

2 likes[Reply](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iii-client-credentials-flow-55lp#/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iii-client-credentials-flow-55lp/comments/new/1g7mp)

[hero1992 profile image](https://dev.to/hero1992)

Hero

•[Feb 18](https://dev.to/hero1992/comment/1m6e3)

Hands are faster than brains

Like[Reply](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iii-client-credentials-flow-55lp#/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iii-client-credentials-flow-55lp/comments/new/1m6e3)

[two_kids_in_a_trenchcoat profile image](https://dev.to/two_kids_in_a_trenchcoat)

Peter Wone

•[Jun 11 '21 • Edited on Jun 11](https://dev.to/two_kids_in_a_trenchcoat/comment/1fa3a)

I wonder could you expand a little on how you got to that "GET NEW ACCESS TOKEN" UI in Postman. I've installed it but never used it and I can't figure out how to follow your instructions from that point. UPDATE I figured it out but others would probably appreciate knowing how to do the test.

2 likes[Reply](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iii-client-credentials-flow-55lp#/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iii-client-credentials-flow-55lp/comments/new/1fa3a)

[robinvanderknaap profile image](https://dev.to/robinvanderknaap)

Robin van der Knaap

•[Jun 21 '21](https://dev.to/robinvanderknaap/comment/1fhp4)

Hi Peter,

Here's an article which covers all you need to know about setting authorization headers for Postman requests: [learning.postman.com/docs/sending-...](https://learning.postman.com/docs/sending-requests/authorization/)

Focus on this part specifically: [learning.postman.com/docs/sending-...](https://learning.postman.com/docs/sending-requests/authorization/#oauth-20)

2 likes[Reply](https://dev.to/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iii-client-credentials-flow-55lp#/robinvanderknaap/setting-up-an-authorization-server-with-openiddict-part-iii-client-credentials-flow-55lp/comments/new/1fhp4)

[Code of Conduct](https://dev.to/code-of-conduct) • [Report abuse](https://dev.to/report-abuse)