Les architectures microservices gagnent en popularité dans le développement d’applications modernes.

1 – Création du projet et configuration de IdentityServer

1-1 Création du projet

Nous allons commencer à partir de zéro en créant une nouvelle application ASP.NET Core qui sera notre IdentityServer. Elle doit être basée sur le modèle « Vide » et n’avoir aucune authentification :



Une fois le nouveau projet créé, vous devez ajouter une référence au package IdentityServer4, en utilisant le gestionnaire de packages NuGet :



Si vous utilisez Visual Studio Code, vous pouvez utiliser la commande :

Dotnet add package IdentityServer4

1-2 Configuration d’IdentityServer

Vous aurez besoin d’enregistrer IdentityServer dans le conteneur d’injection de dépendances de ASP.NET Core et ajouter le middleware de ce dernier dans le pipeline HHTP du Framework.

Pour enregistrer IdentityServer, vous devez éditer le fichier Startup.cs et modifier la méthode ConfiguresServices comme suit :

public void ConfigureServices(IServiceCollection services)

{

services.AddIdentityServer()

.AddDeveloperSigningCredential();

}

AddIdentityServer est une méthode d’extension qui permet d’enregistrer IdentityServer dans le conteneur d’IoC.

La dépendance minimale que nous avons besoin pour l’instant est AddDeveloperSigningCredential(). Cette extension permet de créer une clé temporaire et le necessaire piour signer les jetons (Tokens). C’est pratique pour démarrer en environnement de développement. Mais, vous ne devez pas le laisser trainer la en production et vous devez fournir le nécessaire pour gérer cela.

Pour ajouter Ie middleware IdentityServer dans le pipeline HTTP de ASP.NET Core, vous devez modifier la méthode Configure() du fichier Startup.cs et ajouter la ligne de code :

app.UseIdentityServer();

Le code complet de cette méthode est le suivant :

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseIdentityServer();

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World!");

});

}

C’est tout. Nous venons de faire le minimun pour intergrer IdentityServer à notre projet. Nous pouvons desormais l’utiliser comme service de gestion sécuriser des accès à nos applications.

Mais avant, nous allons faire quelques modifications pour nous assurer que les clients pointeront toujours la bonne application.

1-3 Configuration des clients et les ressources

Tout client qui fait appel à notre serveur de gestion d’identité doit être un client de confiance. C’est pourquoi, ce dernier doit être référencé dans l’application IdentityServer.

Par ailleurs, toute ressources (API par exemple), donc l’accès est sécurisé, doit être repertorie dans l’application IdentityServer.

Pour cela, nous allons créer un classe Config.cs, qui aura une methode GetClients, qui permettra de retourner la liste des clients supportés par l’application et une méthode GetApiResources, qui retournra la liste des API qui nous voulons securiser l’accès. Pour l’instant, puisque nous n’avons pas encore developper nos clients et nos ressources, ces listes seront vides.

Vous devez donc ajouter un nouveau fichier Config.cs à votre application avec le code suivant :

using IdentityServer4.Models;

using System.Collections.Generic;

namespace AspNetCoreIdentityServer

{

public class Config

{

public static IEnumerable<Client> GetClients()

{

return new List<Client>

{

};

}

public static IEnumerable<ApiResource> GetApiResources()

{

return new List<ApiResource>

{

};

}

}

}

Une fois cela fait, vous devez editer le fichier Startup.cs et modifier la méthode ConfigureServices, pour configurer IdentityServer pour qu’il utilise la liste des clients et les ressources que nous avons definis :

public void ConfigureServices(IServiceCollection services)

{

//configure identity server with in-memory stores, keys, clients and resources

services.AddIdentityServer()

.AddDeveloperSigningCredential()

.AddInMemoryApiResources(Config.GetApiResources())

.AddInMemoryClients(Config.GetClients());

}

1-4 Modification de l’hote

Nous devons nous assurer que notre application IdentityServer sera toujours accessible via la même adresse lorsqu’elle est en execution. Par ailleurs, il environement de developpement et lorsqu’on est en mode apprentissage, il est interressant de voir en temps réel les logs de notre application dans la console.

Pour cela, nous allons accéder à l’onglet « Déboguer » dans les propriétés de notre projet. Nous allons dérouler la zone « Profil » et selectionner le nom de application (AspNetCoreIdentityServer).

Les URL suivantes doivent être définies dans le champ URL de l’application, si ce n’est pas le cas :

<https://localhost:5001;http://localhost:5000>



Générez et exécutez votre application.

NB : Les projets ASP.NET Core 2.1 sont configurés pour utiliser par defaut SSL. Pour éviter les avertissements SSL dans le navigateur, vous devez accepter le certificat auto-signé généré par ASP.NET Core. Une notification s’affichera à cet effet à la première exécution de votre application.

Ouvrez votre navigateur et saisissez l’adresse URL :

<http://localhost:5000/.well-known/openid-configuration>

ou

<https://localhost:5001/.well-known/openid-configuration>

Vous obtiendrez le résultat suivant :



Vous venez de mettre en place votre service de gestion sécurisé de jeton. Il est prêt pour la securisation de vos ressources et l’authentification de vos cliients.

2 – Protection d’une API en utilisant IdentityServer

Dans cette partie, nous verrons comment securiser une API en utilisant IdentityServer. Pour qu’un client puisse consomer l’API, il devra au prealable s’authentifier auprés de IdentityServer, pour obtenir un jeton d’accès qu’il utilisera pour acceder à l’API.

2-1 Création de l’API

La premiere chose à faire sera la création de l’API. Nous allons ajouter à notre solution une nouvelle application ASP.NET Core Web API :



Une fois l’application créée, nous allons ajouter un nouveau Controlleur d’API ayant pour nom SecureController, avec le code suivant :

[Route("api/[controller]")]

[Authorize]

public class SecureController : Controller

{

// GET: api/<controller>

[HttpGet]

public IActionResult Get()

{

return new JsonResult(from c in User.Claims select new { c.Type, c.Value });

}

}

Vous remarquerez que notre controlleur est décoré avec l’attribut [Authorize]. Ce qui veut dire que tout accès aux méthodes de cette classe est conditionné par l’obtention au prealable des droits. Pour un client ayant des accès, il pourra visualiser les informations de revendication (Claims) associée à son profile utilisateur.

2-2 Configuration d’IdentityServer

Nous allons configurer IdentityServer afin que ce dernier puisse proceder à la vilidation du jeton de securité d’un client, afin de s’assurer que ce dernier provient d’un client de confiance ayant les autorisations neccessaires pour acceder à l’API.

Pour cela, nous allons dans un premier temps ajouter le package IdentityServer4.AccessTokenValidation.



Si vous utilisez Visual Studio Code, vous devez executer dans le terminal intégré la commande : dotnet add package IdentityServer4.AccessTokenValidation

Ensuite, nous allons éditer le fichier Startup.cs et modifier la méthode ConfigureServices :

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

services.AddAuthentication("Bearer")

.AddIdentityServerAuthentication(options =>

{

options.Authority = "https://localhost:5001";

options.RequireHttpsMetadata = false;

options.ApiName = "testapi";

});

}

La méthode AddAuthentication() va permettre d’enregistrer aupres du conteneur d’IoC le service d’authentification en utilisant « Bearer » comme shema defaut. Cette configuration permet à notre application d’être basée sur une authentificationque par jeton (token authentication) basique. Avant d’acceder à notre API, le client sera donc obligé de passer un jeton d’autorisation dans l’entete HTTP.

La méthode AddIdentityServerAuthentication va permettre d’enregistrer le service qui permettra de valider le jeton fourni par le client par IdentityServer.

Vous devez aussi modifier la méthode Configure de la classe Startup pour ajouter au pipeline http d’ASP.NET Core le middleware d’authentification :

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseAuthentication();

app.UseMvc();

}

Pour finir, je vous recommande également de modifier les propriétés du projet pour que l’API soit toujours accessible via les adresses :

<https://localhost:5003>

et

<http://localhost:5002>



2-3 : Reférencement de l’API dans IdentityServer

Pour que notre API soit protégée, nous devons la référencer auprès de IdentityServer comme ressources.

Pour cela, nous allons simplement editer le fichier Config.cs dans l’application IdentityServer et ajouter notre API dans la liste des ApiResources :

public static IEnumerable<ApiResource> GetApiResources()

{

return new List<ApiResource>

{

new ApiResource("testapi", "My Test API")

};

}

Le premier parametre est le nom de l’API. Il doit être identique au nom d’API defini dans les options lors l’appel à AddIdentityServerAuthentication dans le projet d’API.

Si vous essayez d’acceder à l’adresse <https://localhost:5003/api/secure> dans votre navigateur, vous aurez l’erreur 401. Cela signifie qu’une authenfication est neccessaire pour acceder à cette ressource.

NB : pour deboguer et executer vos deux applications simultanement. Vous pouvez ouvrir deux invites de commandes, chacune positionnée sur le repertoire de chaque projet et executer la commande dotnet run.

C’est tout pour cette partie. Dans la prochaine partie, nous verrons comment mettre en place le client et lui donner les autorisations necceaires pour acceder à l’API.

3 - Creation et configuration du client

Dans cette partie, nous allons créer le client qui sera une application Console .NET Core. Nous allons ecrire le code neccessaire pour permettre à ce denier de demander un jeton de sécurité à l’application Identitty Server, ensuite utiliser ce dernier pour s’authentifier auprès de l’API et acceder aux fonctionnalités de cette derniere.

3-1 Création du client

Pour commencer, nous allons créer une nouvelle application console .NET Core 2.1 :



Ensuite, nous devons ajouter le package Newtonsoft.Json à cette derniere en utilisant le gestionnaire de packages NuGet.



Une fois cela fait, nous allons editer le fichier Program.cs et ajouter le code neccessaire pour appeler notre API. Le code complet de la classe Program est le suivant :

class Program

{

static void Main(string[] args) => CallWebApi().GetAwaiter().GetResult();

static async Task CallWebApi()

{

var client = new HttpClient();

var response = await client.GetAsync("https://localhost:5003/api/secure");

if (!response.IsSuccessStatusCode)

{

Console.WriteLine(response.StatusCode);

}

else

{

var content = await response.Content.ReadAsStringAsync();

Console.WriteLine(JArray.Parse(content));

}

}

}

Enregistrez et exécutez votre application.

Vous aurez le message suivant à l’écran :



L’accès à la ressource pour notre application a été refusée. Nous allons maintenant configurer cette dernièere pour qu’elle demande un jeton d’authentification à IdentityServer et qu’elle utilise ce dernier pour acceder à la ressource voulue.

3-2. Configuration du client

Pour acceder facilement à IdentityServer dans notre client, nous allons utiliser la librairie IdentityModel. La premiere chose à faire sera donc l’ajout du package correspondant à notre application en utilisant NuGet :



Nous allons utiliser la méthode GetAsync() de la classe DiscoveryClient pour recuperer les métadonnées exposées par le EndPoint de l’application IdentityServer. Cette méthode prend en parametre l’URL de notre de l’application IdentityServer. Nous devons nous assurer que le EndPoint est accessible avant de continuer :

var disco = await DiscoveryClient.GetAsync("https://localhost:5001");

if (disco.IsError)

{

Console.WriteLine(disco.Error);

return;

}

Ensuite, nous devons initialiser un nouvel objet TokenClient, en lui passant en paramètre le TokenEndpoint, l’ID du client et le secret.

Nous allons utiliser la méthode RequestClientCredentialsAsync pour demander un jeton d’authentification pour acceder à l’API. Cette méthode prend en parametre le nom de l’API auquel on veut acceder, tel qu’il est repertorier auprès de IdentityServer. Nous devons nous assurer que le jeton a été obtenu avant de continuer :

// request token

var tokenClient = new TokenClient(disco.TokenEndpoint, "consoleappclient", "secret");

var tokenResponse = await tokenClient.RequestClientCredentialsAsync("testapi");

if (tokenResponse.IsError)

{

Console.WriteLine(tokenResponse.Error);

return;

}

Console.WriteLine(tokenResponse.Json);

Nous allons pour finir utiliser la méthode SetBearerToken de HttpClient() pour inscrire le jeton dans l’entete HTTP de notre requette :

var client = new HttpClient();

client.SetBearerToken(tokenResponse.AccessToken);

Le code complet de la méthode CallWebApi devient cecci :

static async Task CallWebApi()

{

// discover endpoints from metadata

var disco = await DiscoveryClient.GetAsync("https://localhost:5001");

if (disco.IsError)

{

Console.WriteLine(disco.Error);

return;

}

// request token

var tokenClient = new TokenClient(disco.TokenEndpoint, "consoleappclient", "secret");

var tokenResponse = await tokenClient.RequestClientCredentialsAsync("testapi");

if (tokenResponse.IsError)

{

Console.WriteLine(tokenResponse.Error);

return;

}

Console.WriteLine(tokenResponse.Json);

// call api

var client = new HttpClient();

client.SetBearerToken(tokenResponse.AccessToken);

var response = await client.GetAsync("https://localhost:5003/api/secure");

if (!response.IsSuccessStatusCode)

{

Console.WriteLine(response.StatusCode);

}

else

{

var content = await response.Content.ReadAsStringAsync();

Console.WriteLine(JArray.Parse(content));

}

}

3-2 Mise à jour de IdentityServer pour reconnaitre le client

Toute la configuration necessaire pour acceder à l’API a été éffectuée coté client. Toutefois, si ce dernier essaye d’acceder à la ressource, il n’aura toujours pas le droit. Cela est du est fait qu’il n’est pas encore connu par IdentityServer. Nous devons donc enregistrer ce dernier et definir à quoi il a accès.

Pour cela, nous devons editer le fichier Config.cs et ajouter un nouveau client à la liste des clients. Nous devons lui donner le même nom et le même secret que nous avons passé en parametrre en initialisant le tokenclient dans l’application console :

var tokenClient = new TokenClient(disco.TokenEndpoint, "consoleappclient", "secret");

Le code de la méthode GetClients de la classe Config devrait donc ressembler à ceci :

public static IEnumerable<Client> GetClients()

{

return new List<Client>

{

new Client

{

ClientId = "ConsoleAppClient",

AllowedGrantTypes = GrantTypes.ClientCredentials,

ClientSecrets =

{

new Secret("secret".Sha256())

},

AllowedScopes = { "testapi" }

}

};

}

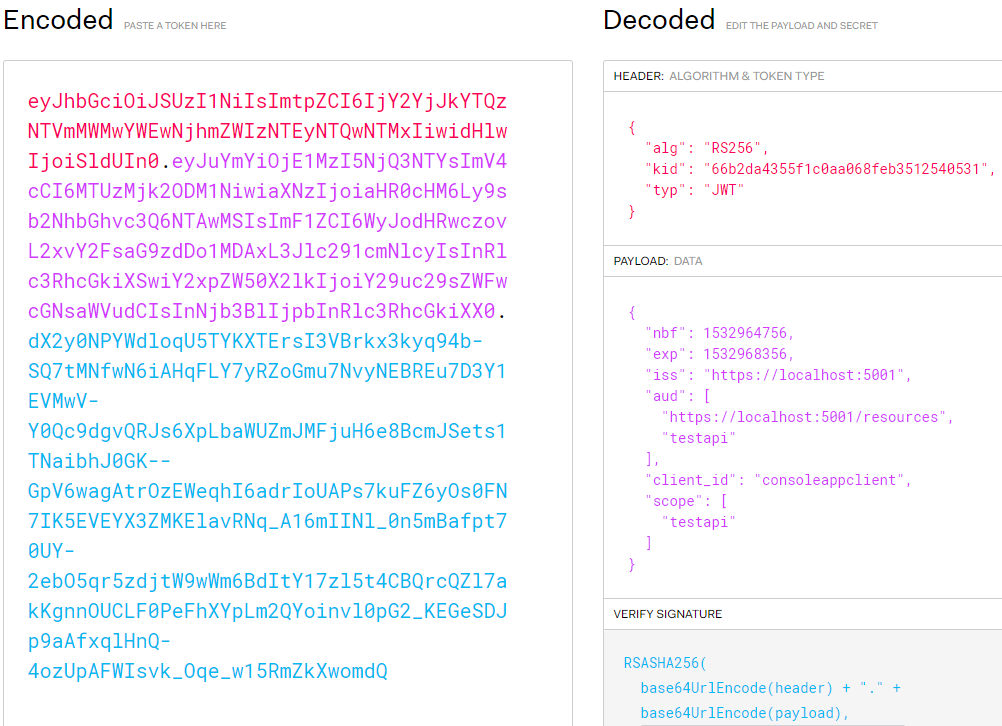
Vous remarquerez que nous avons également definir la ressource à laquelle le client doit acceder.

Enregistrez les modifications. Exécutez l’application IdentityServer, l’API et enfin l’application console.

Vous aurez le resultat suivant :



Le jeton qui est généré par IdentityServer et utilisé par le client pour acceder à la ressource est au format JWT (). Il s’agit d’un jeton securisé qui contient toutes les informations neccessaire pour confirmer l’identité du client et lui donner acceès à la ressource demandée. Si vous decodez le jeton avec <https://jwt.io>, vous obtiendrez ce qui suit :



Nous venons d’acceder à notre ressource securisé en utilisant un jeton de securité provenant d’IdentityServer.

Dans la prochaine partie, nous verrons comment mettre en place l’authentification pour une application ASP.NET Core MVC en utilisant OpenID Connect.

4 – Authentification d’un utilisateur avec OpenID Connect

Nous disposons d’une application Web que voulons securiser certaines pages. Pour acceder à ces pages, l’utilisateur doit s’authentifier au prealable en utilisant son compte.

Nous devons donc integrer cette nouvelle application à notre application IdentityServer. Lorsque l’utilisateur vaudra acceder à une page protegée, il sera redirigé vers IdentityServer qui affichera une fenetre d’authentification. Une fois ce dernier authentifié, il sera redirigé vers la page à laquelle il voulait acceder.

4-1 – Ajout de l’interface d’authentification

Actuellement notre solution IdentityServer ne dispose d’aucune interface. Tout ce qu’elle est en mesure d’afficher dans un navigateur c’est un « Hello World! ».

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage()

;

}

app.UseIdentityServer();

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World!");

});

}

Elle ne dispose donc d’aucune vue, controlleur, etc. Poutant, nous avons besoin d’une solution permettant d’authentifier un utilisateur via un formulaire, mettre fin à sa session, etc. Pour mettre cela en place, nous allons nous appuyer sur un modèle Quickstart existant fourni par les développeurs de IdentityServer4.

Ce modele est disponible dans le repository GitHub suivant <https://github.com/IdentityServer/IdentityServer4.Quickstart.UI/tree/release/Views>. Vous pouvez le telecharger et copier/coller les dossiers Quickstart, Views et wwwroot dans votre projet.

Vous pouvez aussi simplement ouvrir le terminal Powershell dans le dossier racine du projet et executer la commande :

iex ((New-Object System.Net.WebClient).DownloadString('https://raw.githubusercontent.com/IdentityServer/IdentityServer4.Quickstart.UI/release/get.ps1'))

Je n’entrerais pas dans les détails d’implémentation de ce modèle.

4-2. Transformation du projet en solution MVC

Le Quickstart que nous avons intégré repose sur ASP.NET MVC. Nous allons apporter quelques modifications à notre projet pour prendre en charge ASP.NET Core MVC.

La premiere chose à faire sera de modifier la méthode ConfigureServices() et ajouter la ligne de code suivante au début :

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

Cette ligne de code permet d’enregistrer les services pour la prise en charge de ASP.NET Core MVC.

Par la suite, vous devez modifier la méthode Configure pour integrer les middlewares neccessaires à la gestion des fichiers statistiques et le routage. Par ailleurs, vous devez supprimer ligne de code permettant d’afficher le « Hello Word! » :

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage()

;

}

app.UseStaticFiles();

app.UseIdentityServer();

app.UseMvcWithDefaultRoute();

}

4-3. Ajout du support pour OpenID Connect

OpenID Connect est une couche d'identification basée sur le protocole OAuth 2.0, qui autorise les clients à vérifier l'identité d'un utilisateur final en se basant sur l'authentification fournie par un serveur d'autorisation, dont IdentityServer. L’authentification d’un utilisateur via un formulaire avec IdentityServer repose sur OpenID Connect.

L’implementation repose sur le concept de scopes (portés). Nous definir comme nous l’avons fait l’API les ressources auxquelles le client doit acceder. Sauf qu’ici, il s’agit des informations du profile utilisateur (id, nom, email, etc.) que nous souhaitons partager avec le client.

Pour le faire, nous allons editer le fichier Config.cs et ajouter les lignes de code suivantes :

**public** **static** IEnumerable<IdentityResource> GetIdentityResources()

{

**return** **new** List<IdentityResource>

{

**new** IdentityResources.OpenId(),

**new** IdentityResources.Profile(),

};

}

Une fois cela fait, nous devons modifier la méthode ConfigureServices du fichier Startup.cs pour ajouter cette nouvelle ressource à la configuration d’IdentityServer. Cela se fait en utilisant la méthode d’extension AddInMemoryIdentityResources lors de l’appel de AddIdentityServer() :

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

//configure identity server with in-memory stores, keys, clients and resources

services.AddIdentityServer()

.AddDeveloperSigningCredential()

.AddInMemoryIdentityResources(Config.GetIdentityResources())

.AddInMemoryApiResources(Config.GetApiResources())

.AddInMemoryClients(Config.GetClients());

}

4-4. Ajout des utilisateurs pour les tests

Les informations saisies par l’utilisateur dans le formulaire d’authentification doivent être validées avant de lui donner les accés. Cependant nous ne disposons pas d’une base données utilisateurs que nous pouvons utiliser. IdentityServer pour des besoins de tests permet de definir et utiliser une liste de type TestUser.

Nous allons une nouvelle fois modifier le fichier Config.cs et ajouter le code suivant pour definir notre liste d’utilisateurs :

public static List<TestUser> GetUsers()

{

return new List<TestUser>

{

new TestUser{SubjectId = "818727", Username = "alice", Password = "alice",

Claims =

{

new Claim(JwtClaimTypes.Name, "Alice Smith"),

new Claim(JwtClaimTypes.GivenName, "Alice"),

new Claim(JwtClaimTypes.FamilyName, "Smith"),

new Claim(JwtClaimTypes.Email, "AliceSmith@email.com"),

new Claim(JwtClaimTypes.EmailVerified, "true", ClaimValueTypes.Boolean),

new Claim(JwtClaimTypes.WebSite, "http://alice.com"),

new Claim(JwtClaimTypes.Address, @"{ 'street\_address': 'One Hacker Way', 'locality': 'Heidelberg', 'postal\_code': 69118, 'country': 'Germany' }", IdentityServer4.IdentityServerConstants.ClaimValueTypes.Json)

}

},

new TestUser{SubjectId = "88421113", Username = "bob", Password = "bob",

Claims =

{

new Claim(JwtClaimTypes.Name, "Bob Smith"),

new Claim(JwtClaimTypes.GivenName, "Bob"),

new Claim(JwtClaimTypes.FamilyName, "Smith"),

new Claim(JwtClaimTypes.Email, "BobSmith@email.com"),

new Claim(JwtClaimTypes.EmailVerified, "true", ClaimValueTypes.Boolean),

new Claim(JwtClaimTypes.WebSite, "http://bob.com"),

new Claim(JwtClaimTypes.Address, @"{ 'street\_address': 'One Hacker Way', 'locality': 'Heidelberg', 'postal\_code': 69118, 'country': 'Germany' }", IdentityServer4.IdentityServerConstants.ClaimValueTypes.Json),

new Claim("location", "somewhere")

}

}

};

}

Nous devons également modifier la méthode ConfigureServices du fichier Startup.cs pour ajouter cette liste à la configuration d’IdentityServer. Cela se fait en utilisant la méthode d’extension AddTestUsers lors de l’appel de AddIdentityServer() :

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

//configure identity server with in-memory stores, keys, clients and resources

services.AddIdentityServer()

.AddDeveloperSigningCredential()

.AddInMemoryIdentityResources(Config.GetIdentityResources())

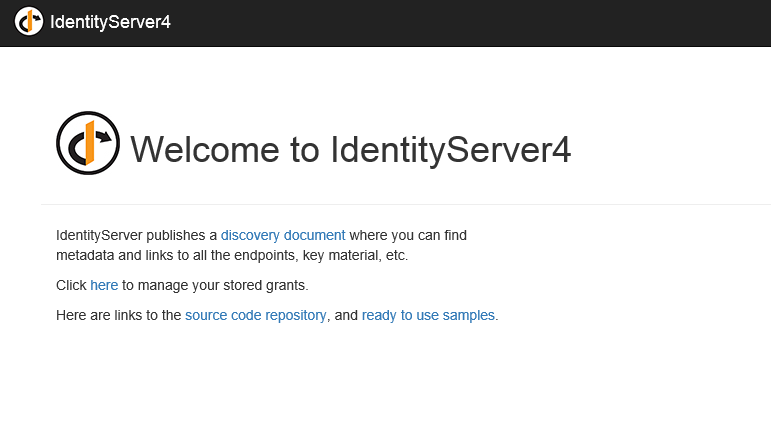
.AddInMemoryApiResources(Config.GetApiResources())

.AddInMemoryClients(Config.GetClients())

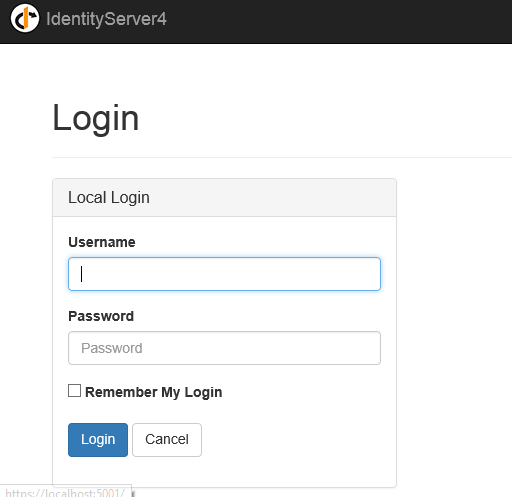
.AddTestUsers(Config.GetUsers()); ;

}

Enregistrer les modifications et executez l’application. Vous aurez le resultat suivant affiché dans le navigateur.



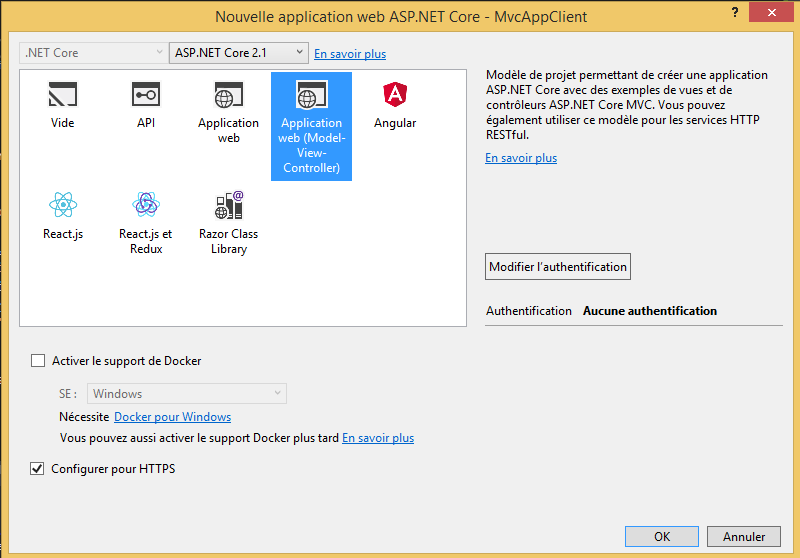
Si vous cliquez sur « Click here to manage your stored grants. », vous essayerez d’acceder à une page protégée. De ce fait, vous serez redirigé vers la page d’authentification :



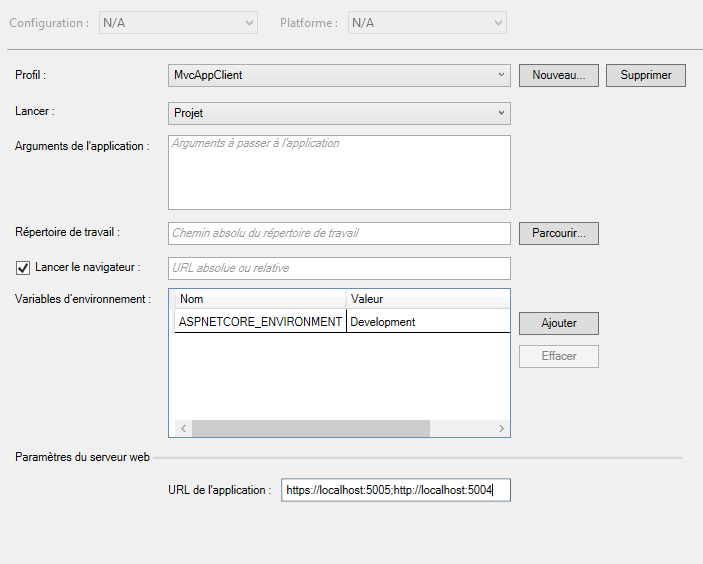
5 – Creation et configuration d’un client utilisant OpenID Connect

Nous allons maintenant créer une nouvelle application ASP.NET Core MVC. L’utilisateur pour acceder à une page securisée de cette application, devra au prealable s’authentifier. Et l’authentification sera gérée par notre solution d’authentification centraliseée.

Vous allez donc ajouter une nouvelle application ASP.NET Core MVC à votre solution :



Vous devez modfier l’application pour utiliser les ports 5004 et 5005 respectivement en HTTP et en HTTPS :



5-1  Configuration de l’authentification OpenID Connect

Nous allons configurer notre application pour utiliser l’authentification OpenID Connect, car c’est ce qui est supporté par notre serveur d’auhentification IdentityServer.

Pour cela, nous allons editer le fichier Startup.cs et ajouter les lignes de code suivantes dans la méthode ConfigureServices :

JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

services.AddAuthentication(options =>

{

options.DefaultScheme = "Cookies";

options.DefaultChallengeScheme = "oidc";

})

.AddCookie("Cookies")

.AddOpenIdConnect("oidc", options =>

{

options.SignInScheme = "Cookies";

options.Authority = "https://localhost:5001";

options.RequireHttpsMetadata = false;

options.ClientId = "mvcappclient";

options.SaveTokens = true;

});

La méthode AddAuthentication() va permettre d’enregistrer auprès du conteneur d’IoC le service d’authentification en utilisant « Cookies » par défaut. Nous avons besoin de definir DefaultChallengeScheme à « oidc » parce que nous utilisons OpenID Connect.

L’extension AddCookie va permettra d’ajouter le handler pour la gestion des cookies.

L’extension AddOpenIdConnect, quant à elle, va permettre de configurer le handler pour utiliser le protocole OpenID Connect. Nous devons spécifier l’URL de l’application IdentityServer (Authority) et l’ID du client (ClientID). Nous devons renseigner le meme ID Client dans IdentityServer. Par ailleurs, SaveTokens est utilisé pour assurer la persitant du token retourné par IdentityServer dans le cookies et SignInScheme pour utiliser le handler de gestion de cookies pour générer un cookie lorsque le processus OpenID Connect est completé.

Le code complet de cette méthode ConfigureServices est le suivant :

public void ConfigureServices(IServiceCollection services)

{

services.Configure<CookiePolicyOptions>(options =>

{

// This lambda determines whether user consent for non-essential cookies is needed for a given request.

options.CheckConsentNeeded = context => true;

options.MinimumSameSitePolicy = SameSiteMode.None;

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

services.AddAuthentication(options =>

{

options.DefaultScheme = "Cookies";

options.DefaultChallengeScheme = "oidc";

})

.AddCookie("Cookies")

.AddOpenIdConnect("oidc", options =>

{

options.SignInScheme = "Cookies";

options.Authority = "https://localhost:5001";

options.RequireHttpsMetadata = false;

options.ClientId = "mvcappclient";

options.SaveTokens = true;

});

}

Pour s’assurer que les services de gestion d’authentification sont executés à chaque requette, nous allons ajouter le middleware pour l’authentification dans le pipeline HTTP de ASP.NET Core.

Cela se fait en ajoutant UseAuthentication dans la méthode Configure du fichier Startup :

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Home/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseAuthentication();

app.UseStaticFiles();

app.UseCookiePolicy();

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

}

Le middleware d’authentification doit être ajouté avant le middleware MVC (UseMVC) et de gestion des fichiers statisques (UseStaticFiles).

5-2 : Ajout d’une page securisé

Maintenant, nous allons ajouter une page dont l’accés est securisé. Dans le HomeController, ajoutez une nouvelle méthode d’action Secure, avec le code suivant :

[Authorize]

public IActionResult Secure()

{

ViewData["Message"] = "Secure page.";

return View();

}

Elle doit être décorée avec l’attribut [Autorize]. C’est cet attribut qui permet au système d’authentification de ASP.NET de savoir que l’accès à cette resource est securisée et que l’utilisateur doit obtenir au prealable les droits neccessaires.

Faites un clic droit sur cette méthode, puis sélectionnez ajouter une vue. Remplacez le code de cette vue par ce qui suit :

@{

ViewData["Title"] = "Secure";

}

<h2>@ViewData["Title"]</h2>

<h3>User claims</h3>

<dl>

@foreach (var claim in User.Claims)

{

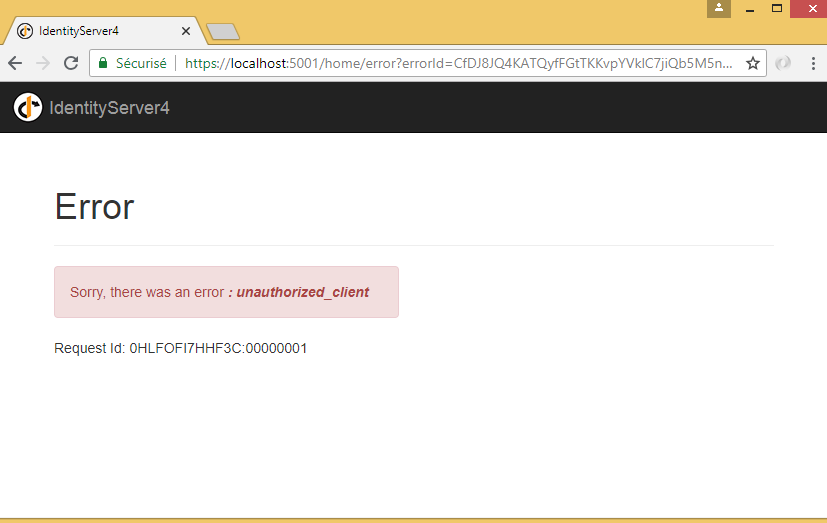
<dt>@claim.Type</dt>

<dd>@claim.Value</dd>

}

</dl>

Enregistrez et exécutez votre application, ainsi que l’application IdentityServer en utilisant l’invite de commande et la commande Dotnet Run. Si vous entrez l’URL suivante <https://localhost:5005/Home/Secure> dans votre navigateur, vous serez redirigé vers la page suivante :



IdentityServer ne reconnait pas notre client. Celui-ci doit faire partie de sa liste des clients.

5-3 : Enregistrement du client MVC dans IdentityServer

Vous allez éditer le fichier Config.cs et ajouter notre application dans la liste des clients :

public static IEnumerable<Client> GetClients()

{

return new List<Client>

{

new Client

{

ClientId = "consoleappclient",

AllowedGrantTypes = GrantTypes.ClientCredentials,

ClientSecrets =

{

new Secret("secret".Sha256())

},

AllowedScopes = { "testapi" }

},

// OpenID Connect implicit flow client (MVC)

new Client

{

ClientId = "mvcappclient",

ClientName = "MVC Client",

AllowedGrantTypes = GrantTypes.Implicit,

RedirectUris = { "https://localhost:5005/signin-oidc" },

PostLogoutRedirectUris = { "https://localhost:5005/signout-callback-oidc" },

AllowedScopes =

{

IdentityServerConstants.StandardScopes.OpenId,

IdentityServerConstants.StandardScopes.Profile

}

}

}

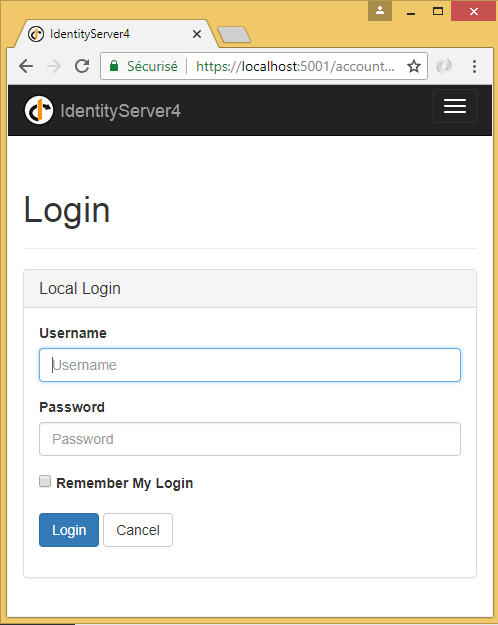
;

}

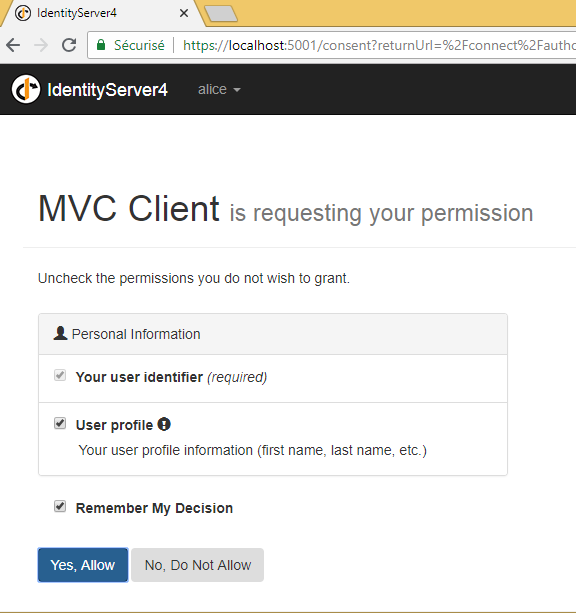
L’ID doit être le même que celui specifié lors de la configuration de l’authentification coté client. Le « Grant Type » doit être à Implicit. Il s’agit du mode le plus optimisé pour les applications Web. Ce mode permet le transfert de tous les jetons via le navigateur.

Dans AllowedScopes permet de definir les informations qui doivent être partagées avec le client. Nous voulons que les données de profil de l’utilisateur puissent être partagées et que la transaction se fasse via le protocole OpenID.

Une fois cela fait, executez de nouveau vos application et accedez à la page securisée (<https://localhost:5005/Home/Secure>). Vous serez maintenant redirigé vers la page de connexion :



Une fois connecté une page de consentement s’affichera afin d’obtenir l’approbation de l’utilisateur avant de partager ses données de profile avec l’application MVC :



Par defaut le consentement est à « True ». Si vous ne voulez pas que le consentement de l’utilisateur soit demandé au prélalble, vous devez mettre la propriété RequireConsent à False lors de l’enregistrement du client :

new Client

{

ClientId = "mvcappclient",

ClientName = "MVC Client",

AllowedGrantTypes = GrantTypes.Implicit,

RequireConsent = false,

RedirectUris = { "https://localhost:5005/signin-oidc" },

PostLogoutRedirectUris = { "https://localhost:5005/signout-callback-oidc" },

AllowedScopes =

{

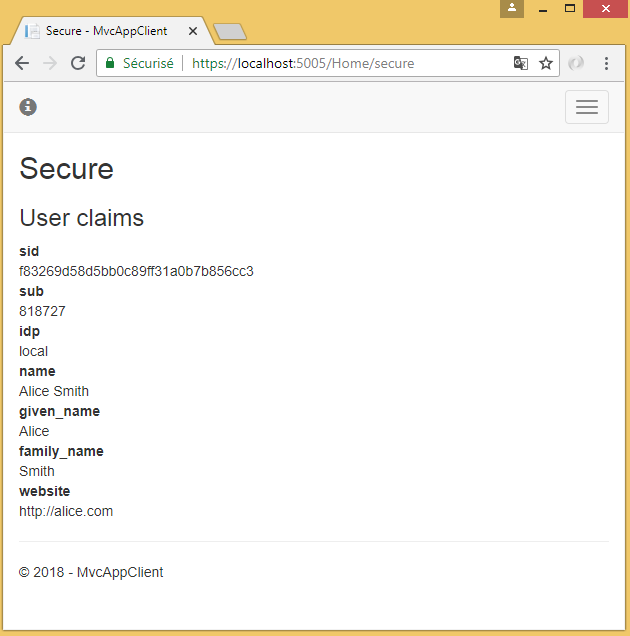
IdentityServerConstants.StandardScopes.OpenId,

IdentityServerConstants.StandardScopes.Profile

}

}

Lorsque vous cliquez sur « Yes, Allow », vous êtes redirigé vers la page à laquelle vous vouliez acceder :



5-3 Déconnexion

La deconnexion est IdentityServer est aussi simple que supprimer les cookies d’authentification. Pour mettre en place la déconnexion, vous devez simplement la méthode d’action et les lignes de code suivantes coté client :

public async Task Logout()

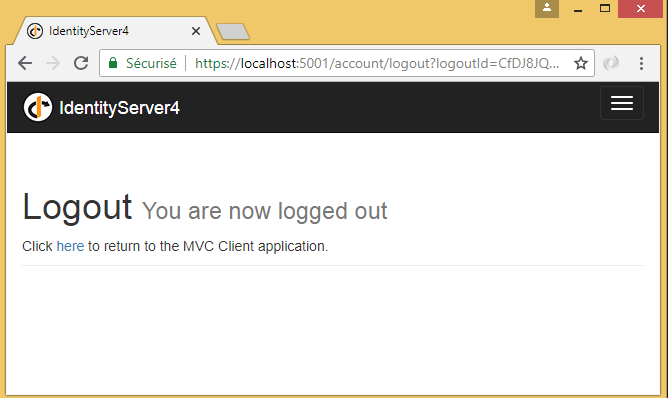
{

await HttpContext.SignOutAsync("Cookies");

await HttpContext.SignOutAsync("oidc");

}

Si vous exécutez votre application et essayez de vous déconnecter, vous allez vous rendre compte que vous allez être automatiquement redirigé vers la page de confirmation de déconnexion de l’application IdentityServer (cette page est déjà mise en place dans le modèle Quickstart que nous avons intégré dans notre application).



En cliquant sur le lien affiché, vous êtes redirectionné vers la page d’accueil du client MVC. Vous devez avoir au préalable defini l’URL de retour. Nous l’avons fait lors de l’enrtegistement du client.

// OpenID Connect implicit flow client (MVC)

new Client

{

ClientId = "mvcappclient",

ClientName = "MVC Client",

AllowedGrantTypes = GrantTypes.Implicit,

RequireConsent = false,

RedirectUris = { "https://localhost:5005/signin-oidc" },

PostLogoutRedirectUris = { "https://localhost:5005/signout-callback-oidc" },

AllowedScopes =

{

IdentityServerConstants.StandardScopes.OpenId,

IdentityServerConstants.StandardScopes.Profile

}

}

Pour que la redirection soit automtique, vous devez simplement editer le ficher AccountOptions.cs qui se trouve dans le dossier Quickstart/Account et mettre à true le champ

public class AccountOptions

{

public static bool AllowLocalLogin = true;

public static bool AllowRememberLogin = true;

public static TimeSpan RememberMeLoginDuration = TimeSpan.FromDays(30);

public static bool ShowLogoutPrompt = true;

public static bool AutomaticRedirectAfterSignOut = true;

// specify the Windows authentication scheme being used

public static readonly string WindowsAuthenticationSchemeName = Microsoft.AspNetCore.Server.IISIntegration.IISDefaults.AuthenticationScheme;

// if user uses windows auth, should we load the groups from windows

public static bool IncludeWindowsGroups = false;

public static string InvalidCredentialsErrorMessage = "Invalid username or password";

}

6 : Autoriser l’application MVC à acceder à l’API

Nous avons développé un client MVC capable d’utiliser IdentityServer et OpenID Connect pour s’authentifier via un formulaire et obtenir l’accès aux ressources sécurisées. Nous voulons maintenant qu’une fois l’utilisateur authentifié, ce dernier puisse obtenir un jeton d’accès pour appeler l’API que nous avons développée précédemment.

IdentityServer implémente le protocole OAuth 2.0 pour la gestion des jetons. Nous aurons donc recours à cette étape à un mode hybride qui utilise à la fois OpenID Connect et OAUth 2.0. Voyons comment procéder.

6-1 : Modification de la configuration du client

Dans l’application IdentityServer, nous allons modifier la configuration de l’application MVC dans le fichier Config.cs. La nouvelle configuration est la suivante :

// OpenID Connect implicit flow client (MVC)

new Client

{

ClientId = "mvcappclient",

ClientName = "MVC Client",

AllowedGrantTypes = GrantTypes.HybridAndClientCredentials,

ClientSecrets =

{

new Secret("secret".Sha256())

},

RequireConsent = false,

RedirectUris = { "https://localhost:5005/signin-oidc" },

PostLogoutRedirectUris = { "https://localhost:5005/signout-callback-oidc" },

AllowedScopes =

{

IdentityServerConstants.StandardScopes.OpenId,

IdentityServerConstants.StandardScopes.Profile,

"testapi"

},

AllowOfflineAccess = true

}

Vous allez remarquer que nous avons changé le GrantTypes de « Implicit » à « HybridAndClientCredentials ». Le mode implicite est pratique pour la transmission des jetons d’identification via le navigateur. Par contre, en ce qui concerne les jetons d’accès (dont nous aurons besoin pour accéder à l’API), qui sont plus sensibles que les jetons d’identification ce mode est moins pratique. Pour parlier à cela, un mode hybride est offert.

Le mode hybride permet au client d’obtenir dans un premier temps un jeton d’identification qui est transmis par le navigateur. En fois l’authenticité du jeton validé, le client ouvre un canal de retour avec IdentityServer pour obtenir un jeton d’accès.

Nous avons ensuite fourni un mot de passe (ClientSecrets) qui sera utilisé pour fournir le jeton d’accés via la cannal de retour.

Enfin, nous avons ajouté l’API dans les scopes afin que le client puisse accedder à ce dernier et mettre « AllowOfflineAccess » à « True ». Cela permet de demander l’actualisation des jetons pour des accès de longue durée aux APIs.

C’est tout ce qui est à modifier coté IdentityServer.

6-2 Installation du package IdentityModel

La première chose à faire côté client est l’installation du package IdentityModel dans l’application MVC. Il s’agit de la librairie utilisée coté client pour acceder facilement au « discovery endpoint » d’IdentityServer.



6-2 : Modification de la configuration de l’authentification

Nous devons modifier la méthode ConfigureServices du fichier Startup.cs et remplacer le code de la configuration de l’authentification par ce qui suit :

services.AddAuthentication(options =>

{

options.DefaultScheme = "Cookies";

options.DefaultChallengeScheme = "oidc";

})

.AddCookie("Cookies")

.AddOpenIdConnect("oidc", options =>

{

options.SignInScheme = "Cookies";

options.Authority = "https://localhost:5001";

options.RequireHttpsMetadata = false;

options.ClientId = "mvcappclient";

options.ClientSecret = "secret";

options.ResponseType = "code id\_token";

options.SaveTokens = true;

options.GetClaimsFromUserInfoEndpoint = true;

options.Scope.Add("testapi");

options.Scope.Add("offline\_access");

});

Nous avons ajouté le ClientSecret afin qu’il puisse correspondre à la valeur definie dans IdentityServer. Nous avons également ajouté l’ID de l’API (testapi) et « offline\_access » dans les scopes pour avoir les mêmes niveaux d’accès des deux côtés. La valeur « code id\_token » pour le champ ResponseType permet de specifier que nous utilisons un mode hydride.

6-3 Appel de l’API

Nous pouvons utiliser deux approches pour appeler l’API, selon nos besoins.

6-4 : Utilisation du justificatif d’Identité (credential) du client

Maintenant que le client à des justificatifs d’identité (ClientSecret) pour s’identifier, il peut utiliser ces informations pour s’identifier auprès d’IdentityServer et obtenir un jeton d’accès qu’il utiliser pour appeler l’API. Le code pour effectuer cela est le suivant :

public async Task<IActionResult> CallApiUsingClientCredentials()

{

var tokenClient = new TokenClient("https://localhost:5001/connect/token", "mvcappclient", "secret");

var tokenResponse = await tokenClient.RequestClientCredentialsAsync("testapi");

var client = new HttpClient();

client.SetBearerToken(tokenResponse.AccessToken);

var content = await client.GetStringAsync("https://localhost:5003/api/secure");

ViewBag.Json = JArray.Parse(content).ToString();

return View();

}

Ajoutez celui-ci dans comme méthode d’action du HomeController, puis ajouter la vue correspondante dans le dossier Views/Home avec le suivant :

@{

ViewData["Title"] = "CallApiUsingClientCredentials";

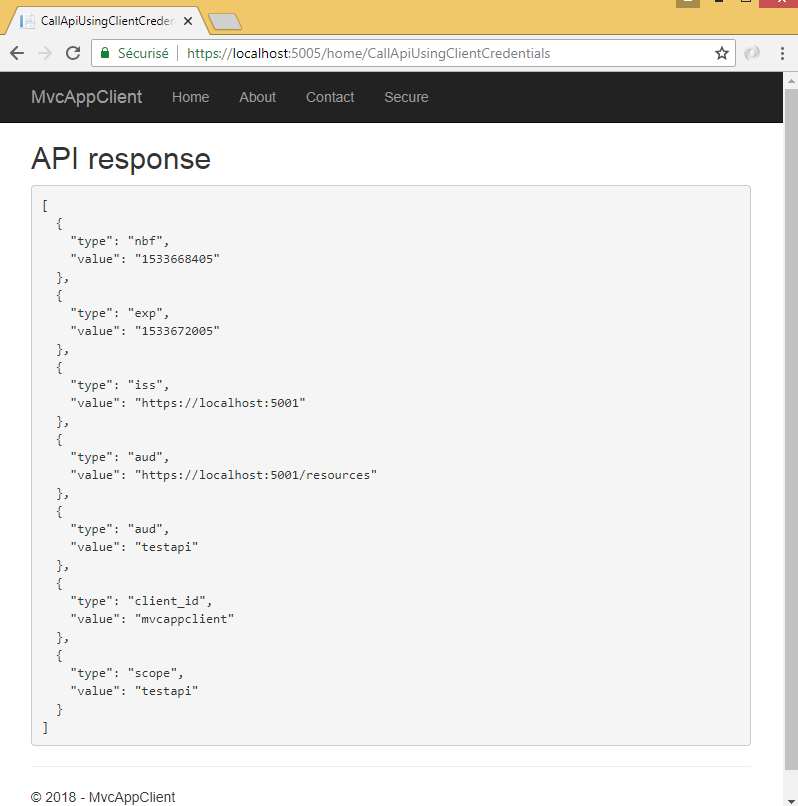
}

<h2>API response</h2>

<pre>@ViewBag.Json</pre>

Pour tester l’application, vous devez executer à la fois les trois applications (IdentityServer, TestAPI et le client MVC). Une fois les applications en cours d’execution, utilisez le lien suivant pour visualiser les résultats :

<https://localhost:5005/home/CallApiUsingClientCredentials>



6-4 : Utilisation d’un jeton d’accès

Dans la seconde approche, le client devra d’abord s’authentifier par formulaire en utilisant OpenID Connect. Il recevra ensuite jeton d’accès qu’il pourra utiliser pour accéder à l’API. Le code correspondant est le suivant :

public async Task<IActionResult> CallApiUsingUserAccessToken()

{

var accessToken = await HttpContext.GetTokenAsync("access\_token");

var client = new HttpClient();

client.SetBearerToken(accessToken);

var content = await client.GetStringAsync("https://localhost:5003/api/secure");

ViewBag.Json = JArray.Parse(content).ToString();

return View();

}

Ce mode est pratique si on a besoin que l’utilisateur soit identifié avant de pouvoir acceder à l’API. Vous pouvez ajouter la vue correspondante et affectuer vos tests.

Connexion avec un compte Microsoft

Dans la section précédente, nous avons vue comment mettre en place l’authentification par formulaire en utilisant OpenID Connect. Maintenant, nous voulons offrir l’opportunité à l’utilisateur de se connecter en utilisant son compte Microsoft.

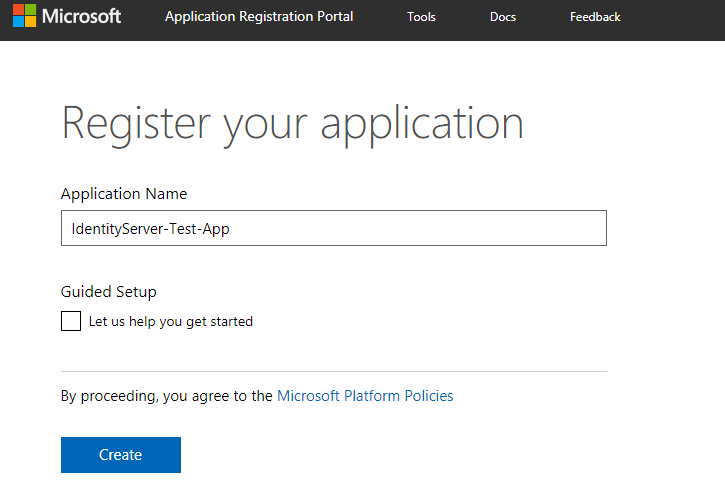
IdentityServer offre la prise en charge en charge de l’authentification en utilisant un provider externe. La mise en place de cela est assez simple dans la mesure ou ASP.NET Core offre en natif la prise en charge de l’authentification avec un compte Google, Facebook, Twitter et Microsoft.

Enregistrement de l’application

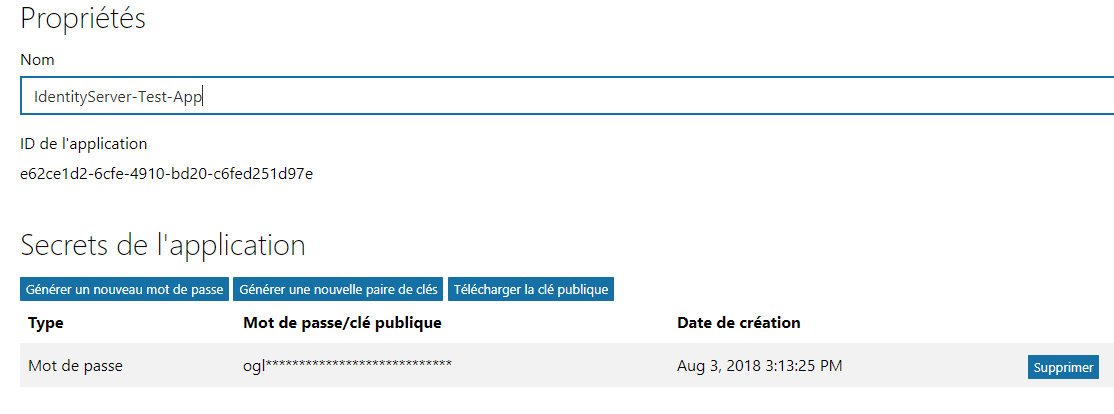
Avant de configurer l’authentification Microsoft pour notre application dans IdentityServer, nous devons d’abord enregistrer notre application dans le portail developper de Microsoft.

Vous devez vous rendre sur la page suivante en utilisant votre compte Microsoft : <https://apps.dev.microsoft.com/>

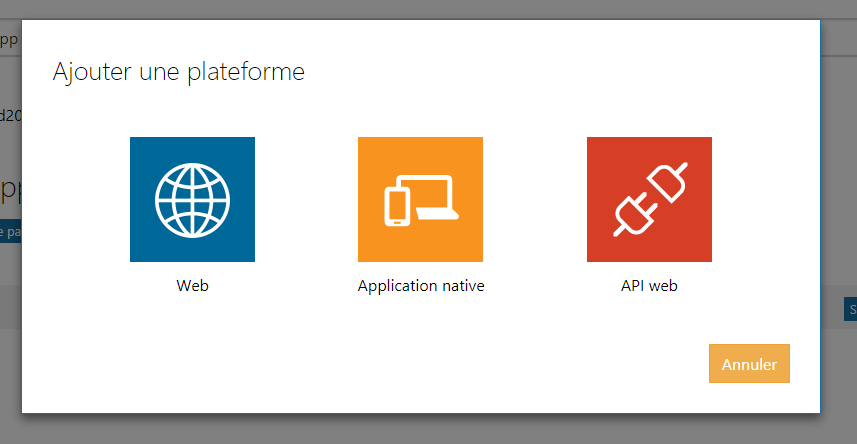
Ensuiter créer une nouvelle application :



Une fois l’application créée, vous serez redirigé avec une nouvelle page vous permettant de générer un mot de passe. Cliquez sur le bouton « Générer un nouveau mot de passe ». Copiez le mot de passe qui sera généré et sauvegardez-le. C’est l’unique fois que vous le verrez en clair dans ce portail. Vous en aurez besoin plus tard pour la configuration de votre application.



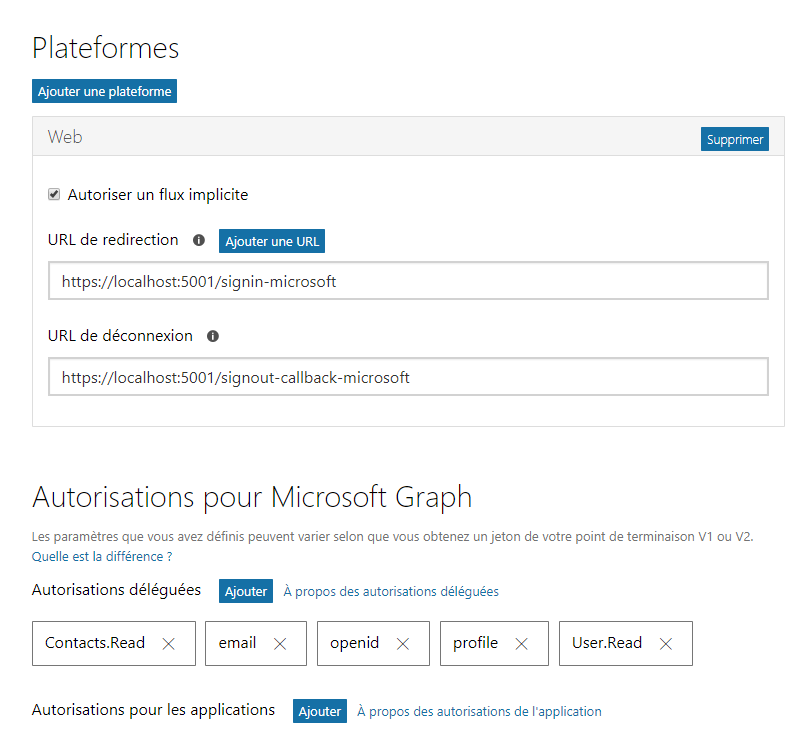
Maintenant, vous devez ajouter une nouvelle plateforme à votre application. Cliquez sur Ajouter une plateforme, puis selectionnez Web :



Ensuite vous devez saisir l’URL de redirection. Il s’agit de l’URL de l’application IdentityServer suivi de signin-microsoft : <https://localhost:5001/signin-microsoft>. Il s’agit du « callback » par défaut pour l’authentification en utilisant le provider Microsoft.

Vous devez également saisir l’url de déconnexion qui sera au format suivant : <https://localhost:5001/signout-callback-microsoft>.

Definissez les autorisations et les infomations du profil utilisateur auxquelles l’application va demander les accés, puis enregistrez.



Revenez dans votre application IdentityServer et ouvrez le fichier Startup.cs. Vous devez modifier la méthode ConfigureServices pour ajouter les services et configurer le provider pour l’authentification Microsoft :

services.AddAuthentication().AddMicrosoftAccount("Microsoft", microsoftOptions =>

{

microsoftOptions.SignInScheme = IdentityServerConstants.ExternalCookieAuthenticationScheme;

microsoftOptions.ClientId = "e62ce1d2-6cfe-4910-bd20-c6fed251d97e";

microsoftOptions.ClientSecret = "ogldbT256%:|sqJGNQOK13\*";

});

Vous devez renseigner l’ID de votre application créée sur le portail Microsoft dans le champ ClientId, puis le mot de passe dans le champ ClientSecret.

Le SignInScheme permet de specifier le nom du cookie qui sera généré et permettra de sauvegarder temporairement les informations provenant du fournisseur d’authentification externe. IdentityServer offre la constante IdentityServerConstants.ExternalCookieAuthenticationScheme pour permettre de définir ce nom.

Le code complet de cette méthode est le suivant.

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

//configure identity server with in-memory stores, keys, clients and resources

services.AddIdentityServer()

.AddDeveloperSigningCredential()

.AddInMemoryIdentityResources(Config.GetIdentityResources())

.AddInMemoryApiResources(Config.GetApiResources())

.AddInMemoryClients(Config.GetClients())

.AddTestUsers(Config.GetUsers());

services.AddAuthentication().AddMicrosoftAccount("Microsoft", microsoftOptions =>

{

microsoftOptions.SignInScheme = IdentityServerConstants.ExternalCookieAuthenticationScheme;

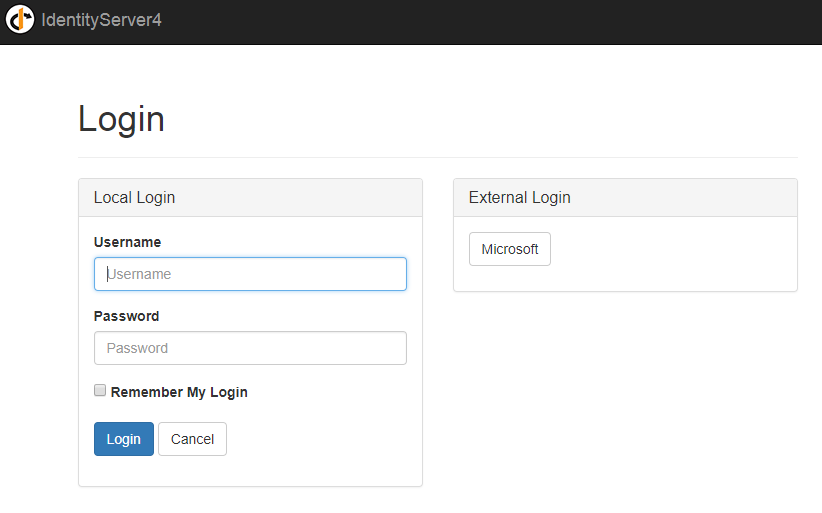
microsoftOptions.ClientId = "e62ce1d2-6cfe-4910-bd20-c6fed251d97e";

microsoftOptions.ClientSecret = "ogldbT256%:|sqJGNQOK13\*";

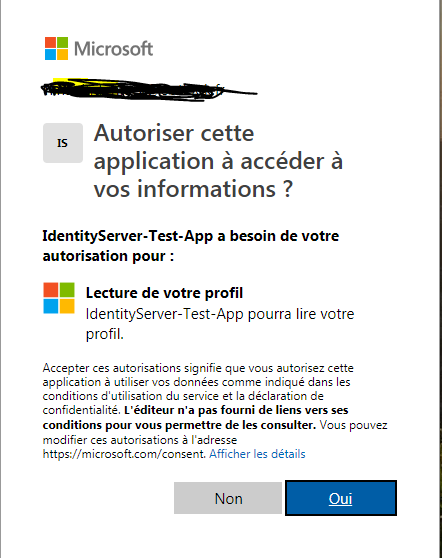
});

}

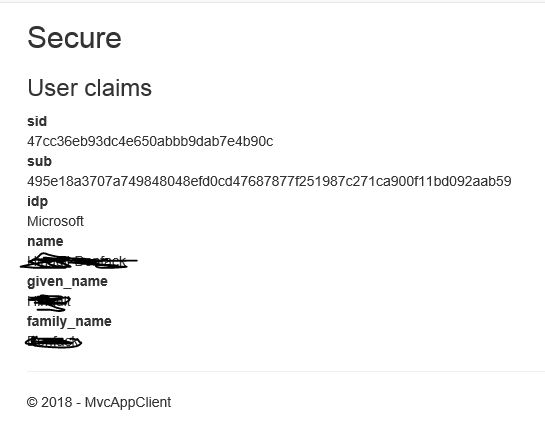
Executez votre application. Dans la page de connexion, vous aurez desormais un bouton pour Microsoft.



Lorsque vous cliquez dessus, vous êtes redirigé vers la page de connexion de Microsoft. Une fois connecté avec votre compte Microsoft, une page s’afficher pour demander votre autorisation à partager vos informations avec l’application.



Une fois votre consentement donné, vous êtes redirigé vers la page à laquelle vous vouliez acceder initialement.



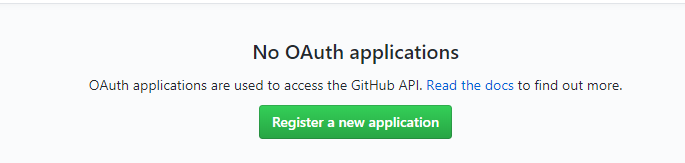
Connexion avec un compte GitHub

Dans le billet précédent, nous avons vue comment mettre en place l’authentification en utilisant Microsoft comme fournisseur d’authentification. Dans ce billet, nous verrons comment faire la même chose en utilisant cette fois GitHub comme fournisseur d’authentification.

IdentityServer offre la prise en charge en charge de l’authentification en utilisant un provider externe. La mise en place de cela est assez simple dans la mesure où ASP.NET Core offre en natif la prise en charge de l’authentification avec un compte Google, Facebook, Twitter et Microsoft.

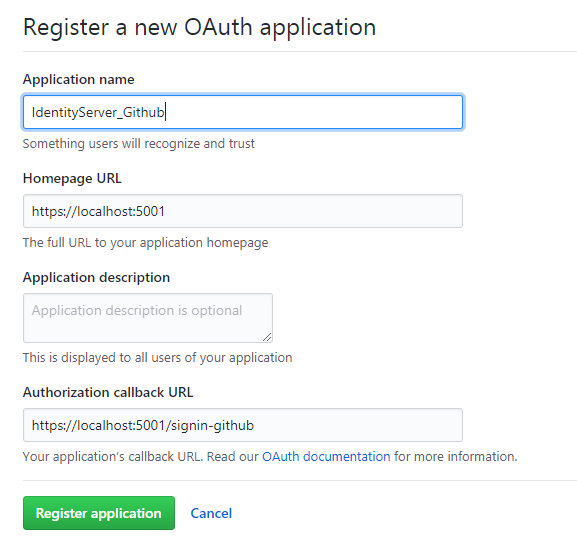
Enregistrement de l’application

La première étape dans le processus est l’enregistrement de votre application. Cela se passe sur cette page : <https://github.com/settings/developers>.



Dans la fenetre qui va s’afficher, vous devez renseigner le nom de votre application, l’URL de la page d’accueil de l’application IdentityServer (<https://localhost:5001>, pour notre cas) et l’URL de CallBack. Il s’agit de l’URL de l’application IdentityServer suivi de signin-github : https://localhost:5001/signin-github. Il s’agit du « callback » par défaut pour l’authentification en utilisant le provider GitHub. Vous pouvez renseigner toute autre valeur, à condition que lors de la configuration de l’authentification GitHub dans votre code, vous renseigniez la même valeur pour l’option CallBackPath.

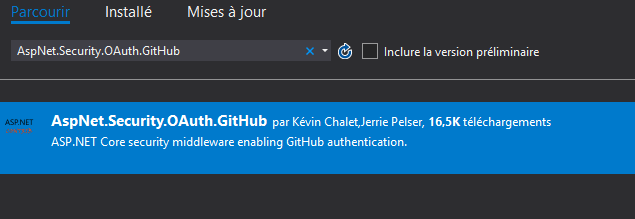
La definition de l’URL de CallBack permet au middleware d’authentification du provider externe d’intercepter le retour après s’être authentifié pour effectuer le traitement neccessaire afin de générer les revendications (Claims) qu’il faut.



Après avoir créé l’application, vous allez obtenir un Client ID et un Client Secret. Copiez ces informations, vous en aurez besoin plus tard.

Configuration de l’authentification GitHub

Revenez dans votre application IdentityServer. GitHub n’est pas supporté par defaut comme fournisseur d’authentification. Vous devez donc installer le package AspNET.Security.OAuth.GitHub :



Une fois le package installé, ouvrez le fichier Startup.cs. Vous devez modifier la méthode ConfigureServices pour ajouter les services et configurer le provider pour l’authentification GitHub :

services.AddAuthentication()

.AddGitHub("GitHub", githubtOptions =>

{

githubtOptions.SignInScheme = IdentityServerConstants.ExternalCookieAuthenticationScheme;

githubtOptions.ClientId = "35a2660287e1914a87da";

githubtOptions.ClientSecret = "de09982aae2bbae90f8ad106e48b135f00761bb3";

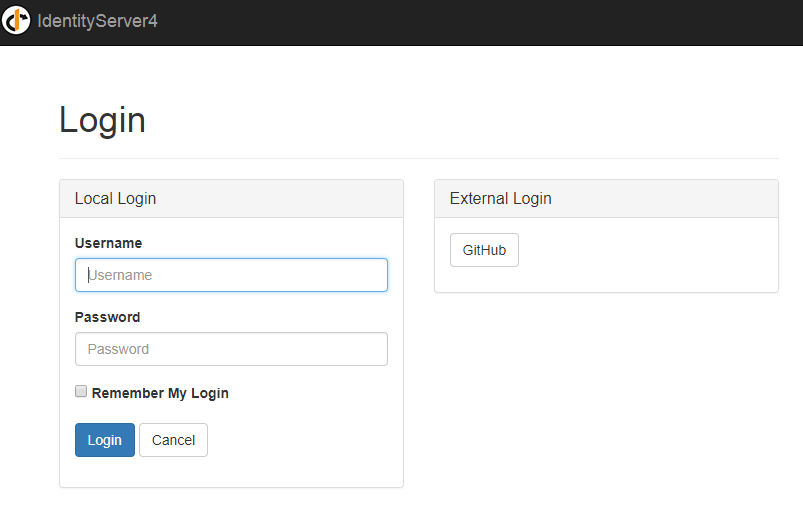
githubtOptions.CallbackPath = "/signin\_github";

});

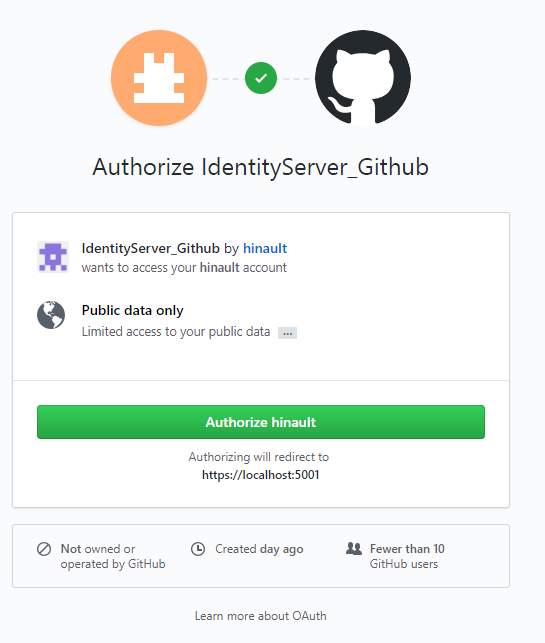
Vous devez renseigner le Client ID et le Client Secret de l’application GitHub respectivement dans les champs ClientId, et ClientSecret.

Le SignInScheme permet de spécifier le nom du cookie qui sera généré et permettra de sauvegarder temporairement les informations provenant du fournisseur d’authentification externe. IdentityServer offre la constante IdentityServerConstants.ExternalCookieAuthenticationScheme pour permettre de définir ce nom.

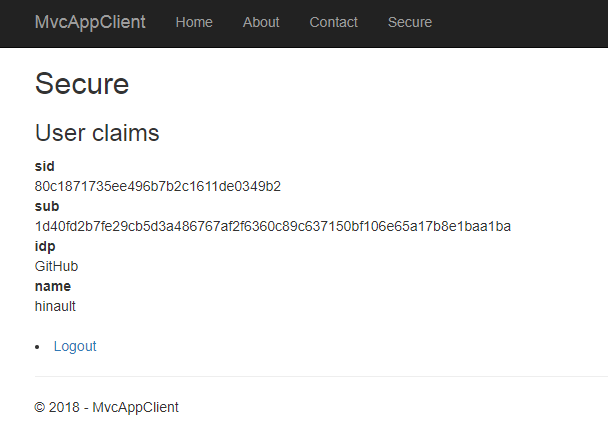
Enregistrez les modifications. Exécutez votre application. Dans la page de connexion, vous aurez désormais un bouton pour GitHub :



Lorsque vous cliquez dessus, vous êtes redirigé vers la page de connexion de GitHub. Une fois connecté avec votre compte GitHub, une page s’affiche pour demander votre autorisation à partager vos informations avec l’application



Une fois votre consentement donné, vous êtes redirigé vers la page à laquelle vous vouliez accéder initialement.



IdentityServer4 : persistance des données de configuration avec EntityFramework Core

Depuis le debut de notre ballade dans l’univers fascinant d’IdentityServer, nous avons utilisé une classe Config pour definir les informations de configuration de notre serveur de gestion d’identité :

public class Config

{

public static IEnumerable<Client> GetClients()

{

return new List<Client>

{

new Client

{

ClientId = "consoleappclient",

AllowedGrantTypes = GrantTypes.ClientCredentials,

ClientSecrets =

{

new Secret("secret".Sha256())

},

AllowedScopes = { "testapi" }

},

// OpenID Connect implicit flow client (MVC)

new Client

{

ClientId = "mvcappclient",

ClientName = "MVC Client",

AllowedGrantTypes = GrantTypes.HybridAndClientCredentials,

ClientSecrets =

{

new Secret("secret".Sha256())

},

RequireConsent = false,

RedirectUris = { "https://localhost:5005/signin-oidc" },

PostLogoutRedirectUris = { "https://localhost:5005/signout-callback-oidc" },

AllowedScopes =

{

IdentityServerConstants.StandardScopes.OpenId,

IdentityServerConstants.StandardScopes.Profile,

"testapi"

},

AllowOfflineAccess = true

}

}

;

}

public static IEnumerable<ApiResource> GetApiResources()

{

return new List<ApiResource>

{

new ApiResource("testapi", "My Test API")

};

}

public static IEnumerable<IdentityResource> GetIdentityResources()

{

return new List<IdentityResource>

{

new IdentityResources.OpenId(),

new IdentityResources.Profile(),

};

}

public static List<TestUser> GetUsers()

{

return new List<TestUser>

{

new TestUser{SubjectId = "818727", Username = "alice", Password = "alice",

Claims =

{

new Claim(JwtClaimTypes.Name, "Alice Smith"),

new Claim(JwtClaimTypes.GivenName, "Alice"),

new Claim(JwtClaimTypes.FamilyName, "Smith"),

new Claim(JwtClaimTypes.Email, "AliceSmith@email.com"),

new Claim(JwtClaimTypes.EmailVerified, "true", ClaimValueTypes.Boolean),

new Claim(JwtClaimTypes.WebSite, "http://alice.com"),

new Claim(JwtClaimTypes.Address, @"{ 'street\_address': 'One Hacker Way', 'locality': 'Heidelberg', 'postal\_code': 69118, 'country': 'Germany' }", IdentityServer4.IdentityServerConstants.ClaimValueTypes.Json)

}

},

new TestUser{SubjectId = "88421113", Username = "bob", Password = "bob",

Claims =

{

new Claim(JwtClaimTypes.Name, "Bob Smith"),

new Claim(JwtClaimTypes.GivenName, "Bob"),

new Claim(JwtClaimTypes.FamilyName, "Smith"),

new Claim(JwtClaimTypes.Email, "BobSmith@email.com"),

new Claim(JwtClaimTypes.EmailVerified, "true", ClaimValueTypes.Boolean),

new Claim(JwtClaimTypes.WebSite, "http://bob.com"),

new Claim(JwtClaimTypes.Address, @"{ 'street\_address': 'One Hacker Way', 'locality': 'Heidelberg', 'postal\_code': 69118, 'country': 'Germany' }", IdentityServer4.IdentityServerConstants.ClaimValueTypes.Json),

new Claim("location", "somewhere")

}

}

};

}

}

Avec cette approche, chaque fois que nous aurons une nouvelle application qui veut utiliser notre solution IdentityServer, nous devons modifier la classe Config pour ajouter ce nouveau client, ensuite regenerer et deployer notre application. Idem pour toute ressource (une API par exemple), que nous voulons sécuriser l’accès.

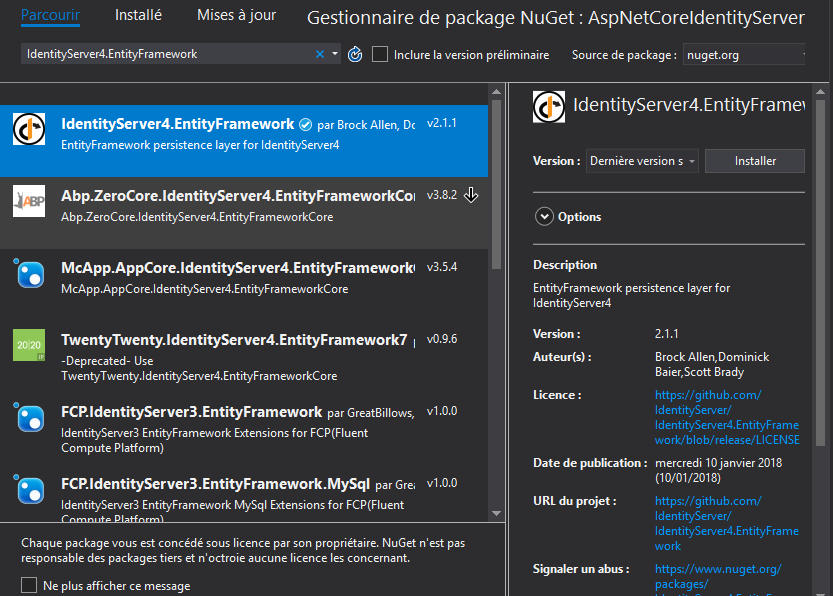
Cette approche n’est pas très pratique, car elle ne permet pas de livrer une solution clé en main.

Dans ce billet, nous verrons comment utiliser une base de données SQL Server pour le stockage des informations de configuration et Entity Framework pour l’accès.

Support d’Entity Framework dans IdentityServer 4

IdentityServer offre un package qui fournit des API permettant d’assurer facilement la persistante des données de configuration en utilisant EntityFramework. Ces API offrent des implémentations des interfaces IClientStore et IResourceStore pour la gestion de la persistant des clients, des ressources d’API et des ressources d’identité. Ces dernières utilisent la classe ConfigurationDbContext qui hérite du DbContext.

Pour commencer, nous allons installer le package IdnetityServer4.EntityFramework à notre application IdentityServer, via le gestionnaire de package :



Vous devez aussi installer le package Microsoft.EntityFrameworkCore.Sqlite.

Configuration du Store

Nous allons utiliser une base de données SQLite pour le stockage de nos données. Après l’ajout des packages, la seconde chose à faire est l’ajout d’un fichier de parametres d’application (appsettings.json) pour la définition de la chaine de connexion :

{

"ConnectionStrings": {

"DefaultConnection": "Data Source=IdentityServer.db"

}

}

Suite à cela, nous devons ajouter un constructeur à la classe Startup.cs afin que les données de configuration soient obtenues au démarrage de l’application :

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

Ensuite, nous devons modifier la méthode ConfigureServices pour remplacer AddInMemoryClients, AddInMemoryIdentityResources et AddInMemoryApiResources avec notre nouveau store de configuration, en utilisant la méthode d’extension AddIdentityServer :

.AddConfigurationStore(options =>

{

options.ConfigureDbContext = builder =>

builder.UseSqlite(Configuration.GetConnectionString("DefaultConnection"),

sql => sql.MigrationsAssembly(typeof(Startup).GetTypeInfo().Assembly.GetName().Name));

})

Vous devez ajouter les namespaces suivants :

using Microsoft.Extensions.Configuration;

using Microsoft.EntityFrameworkCore;

using System.Reflection;

Le code complet de la classe Startup devient ceci :

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

// This method gets called by the runtime. Use this method to add services to the container.

// For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

//configure identity server with in-memory stores, keys, clients and resources

services.AddIdentityServer()

.AddDeveloperSigningCredential()

.AddConfigurationStore(options =>

{

options.ConfigureDbContext = builder =>

builder.UseSqlite(Configuration.GetConnectionString("DefaultConnection"),

sql => sql.MigrationsAssembly(typeof(Startup).GetTypeInfo().Assembly.GetName().Name));

})

.AddTestUsers(Config.GetUsers());

}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage()

;

}

app.UseStaticFiles();

app.UseIdentityServer();

app.UseMvcWithDefaultRoute();

}

}

Configuration de la migration

Le package IdentityServer4.EntityFramework contient les classes d’entité correspodant au modele des données de configuration d’EntityFramework. Nous allons utiliser la migration et les outils Entity Framework en ligne de commande pour créer la base de données avec un schema de données correspondant à celui défini par ce package.

Depuis la version 2.1.3 du kit de développement de .NET Core, les commandes dotnet ef sont intégrées par défaut et supportent Entity Framework Core 2.0 et versions ultérieures. Si vous avez une version égale ou supérieure à cette version, aucune installation supplémentaire n’est nécessaire pour pouvoir utiliser les outils en ligne de commande. Sinon, vous devez installer le package Microsoft.EntityFrameworkCore.Tools.DotNet.

L’approche utilisée ici est CodeFirst. Nous devons exécuter la commande « dotnet ef migrations add » en invite de commande à partir du dossier contenant le fichier .csproj. Cette commande va permettre de générer le code correspondant au modèle de données.

La commande compléte à executée est la suivante :

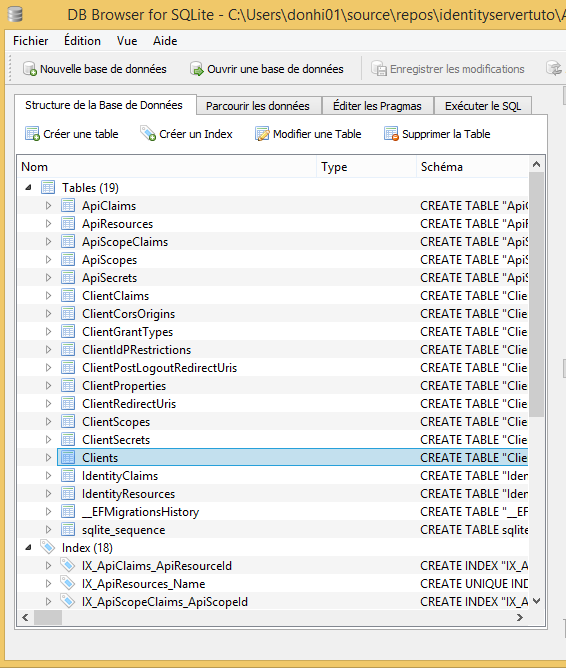
dotnet ef migrations add InitialIdentityServerConfigurationDbMigration -c ConfigurationDbContext -o Data/Migrations/IdentityServer/ConfigurationDb

Vous devez ensuite exécuter la commande :

Dotnet ef database update

Cette commande va permettre d’utiliser le code généré précédemment pour créér la base de données et toutes les tables correspondates au modèle de données.

Une fois cette commande executée, un fichier de base de données « IdentityServer.db » sera créé à la racine du dossier du projet. Si vous ouvrez ce fichier avec « DB Browser for SQLite », vous aurez les tables suivantes :



Initialisation de la base de données

Nous allons maintenant initailiser la base données avec les données qui ont été definies dans le fichier Config.cs. Vous devez editer le fichier Startup.cs et ajouter la méthode suivante :

private void InitializeDatabase(IApplicationBuilder app)

{

using (var serviceScope = app.ApplicationServices.GetService<IServiceScopeFactory>().CreateScope())

{

var context = serviceScope.ServiceProvider.GetRequiredService<ConfigurationDbContext>();

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();

}

}

}

Puis appeler cette dernière dans la méthode Configure :

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

InitializeDatabase(app);

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage()

;

}

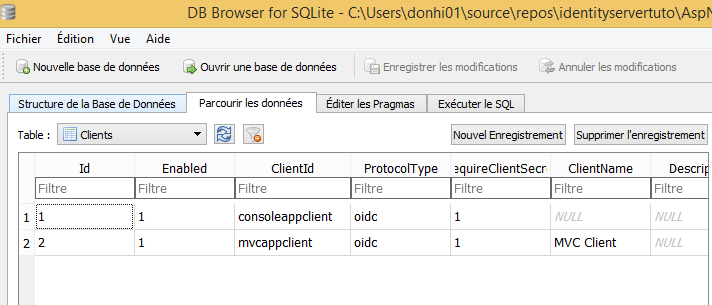
app.UseStaticFiles();

app.UseIdentityServer();

app.UseMvcWithDefaultRoute();

}

Cela fait, executez votre application. Toutes les données presentes dans le fichier Config.cs, en dehors de la liste des utilisateurs seront enregistrés dans la base de données.



Desormais, si vous avez une nouvelle ressource ou un nouveau client à configurer, vous devez simplement enregistrer ce dernier dans la base de données pour que celui-ci soit pris en charge par IdentityServer.

Dans le prochain billet, nous verrons comment utiliser une base de données pour le stockage des données utilisateurs.

IdentityServer4 : utiliser ASP.NET Core Identity pour l’authentification

Dans le billet précédent, nous avons vu comment utiliser OpenID et permettre à l’utilisateur de s’authentifier via un formulaire. Pour la mise en place de la fenêtre de connexion, de déconnexion, etc. nous avons utilisé un Quickstart offert par IdentityServer. Ce modèle repose sur TestUserStore, qui nous permet de définir et charger nos utilisateurs depuis fichier inclut dans le projet.

Maintenant, nous voulons que les utilisateurs soient stockés dans une base de données et que des interfaces soient disponibles pour l’enregistrement d’un nouvel utilisateur, la connexion, la gestion de son compte, etc.

Toutefois, nous ne voulons non plus developper ces fonctionnalités à partir de zero, pourtant ASP.NET Core nous offre le neccessaire pour la gestion d’identité.

A propos de ASP.NET Core Identity

ASP.NET Core Identity est un système de gestion d’Identité qui ajoute à une application ASP.NET Core des fonctionnalités de connexion, gestion de compte, modification de mot de passe, reinitialisation de mot de passe, etc. ASP.NET Core Identity est configuré par defaut pour utiliser EntityFramework Core pour assurer la persistance des données. Les informations d’Identité (nom d’utilisateur, mot de passe, données de profil) peuvent être stockées dans toute base de données supportées par EntityFramework Core.

Avec la version 2.1 de ASP.NET Core, ASP.NET Core Identity est desormais disponible comme une bibliotheques de classes Razor. Ce qui signifie qu’il est desormais possible d’utiliser le scaffolding pour le générer le code de gestion d’identité dans une application existante.

ASP.NET Core Identity et IdentityServer4

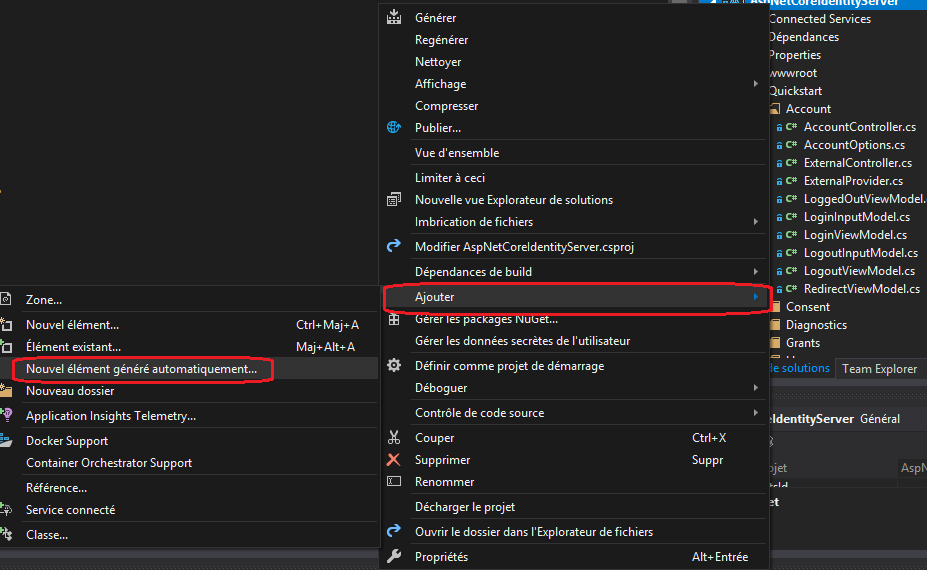
IdentityServer a été concu pour être extensible et flexible. De fait, il est possible d’utiliser IdentityServer tout en deleguant le stockage de données et la gestion des comptes à Identity Server. Voyons comment proceder.

Intégration de ASP.NET Core Identity au projet

Reprenons notre projet precedent. Vous pouvez le telecharger sur mon compte GitHub à l’adresse suivante.

Pour la persistante, nous allons utiliser une base de données SQLite. La premiere chose à faire sera d’ajouter le package correspondant : Microsoft.EntityFrameworkCore.Sqlite.

L’étape suivante sera la génération du code pour Identity. Pour cela, vous devez faire un clic droit sur votre projet, aller sur Ajouter, ensuite cliquer sur « Nouvel élément généré automatiquement… ».

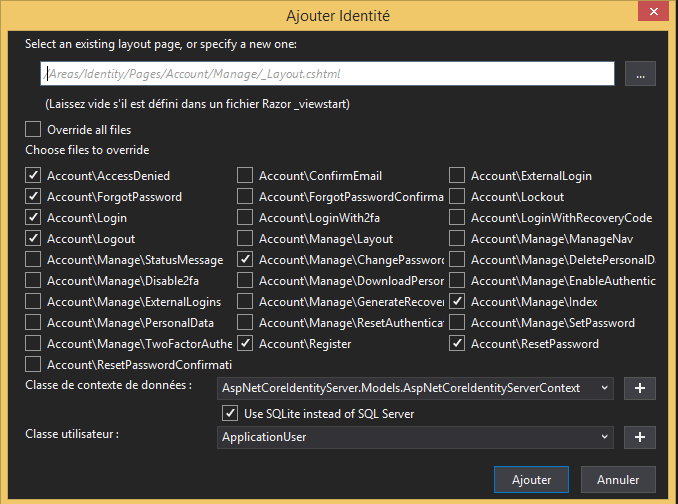


Dans la liste des modèles installés, vous devez sélectionner Identité et cliquer sur Ajouter. Une fenetre va s’afficher permettant de selectionner les fichiers à générer (AccessDenied, ForgotPassword, Login, Logout, ChangePassoword, Register, Index, ResetPassword).

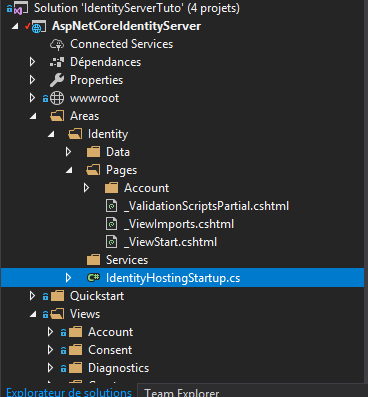
Ensute, nous devons definir DataContext que nous souhaitons utiliser pour la persitance des données. Nous devons en créer un nouveau. Vous devez cliquer sur le « + » dans la ligne correspondante puis sur Ajouter.

Nous allons utiliser SQLite comme base de données. Vous devez donc cocher la case « Use SQLite instead of SQL Server ».

Nous allons également ajouter une nouvelle classe utilisateur ayant pour nom « ApplicationUser ». Une fois cela effectué, cliquez sur Ajouter.



Le code permettant d’intégrer la gestion d’identité dans votre projet est généré dans le dossier Areas/Identity. Ce code a été dévéloppé en utilisant Razor Pages. Il faut noter qu’ASP.NET Core MVC, Web API et Razor Page peuvent cohabiter sans aucun probleme dans le même projet.



La configuration des services d’Identité se trouve dans le fichier Areas/Identity/IdentityHostingStartup.cs. Je ne vais pas m’attarder sur les details d’implementation de ASP.NET Core Identity. Pour en savoir plus sur le sujet, vous pouvez consulter le tutoriel suivant.

Génération de la base données

Nous allons utiliser la migration pour générer la base de données SQLite. Dans la console de gestion de package de Visual Studio, vous devez executer les deux commandes suivantes.

1. Add-Migration CreateIdentitySchema

2. Update-Database

Si vous voulez le faire en utilisant l’invite de commande, vous devez executer les deux commandes suivantes :

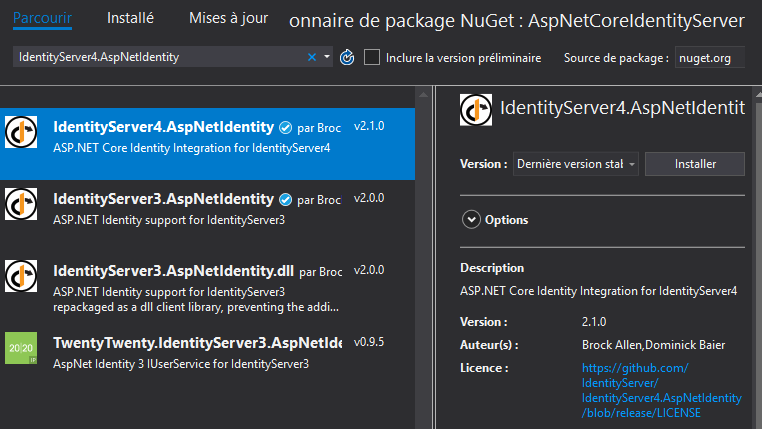
1. dotnet ef migrations add CreateIdentitySchema

2. dotnet ef database update

Un fichier AspNetCoreIdentityServer.db sera créé à la racine de votre projet.

Configuration d’IdentityServer pour utiliser ASP.NET Core Identity

A cette étape, nous allons dans un premier temps ajouter à notre projet le package IdentityServer4.AspNetIdentity :



Une fois le package installer, nous allons editer le fichier Startup.cs et remplacer AddTestUsers(Config.GetUsers()) par AddAspNetIdentity<ApplicationUser>(). Il faudra également ajouter le service suivant AddInMemoryPersistedGrants()

Parceque généré par ASP.NET Core Identity est placé dans le dossier Areas/Identity, nous devons faire un override la configuration par defaut des URL de login et logout de IdentityServer :

services.AddIdentityServer(options =>

{

options.UserInteraction.LoginUrl = "/Identity/Account/Login";

options.UserInteraction.LogoutUrl = "/Identity/Account/Logout";

})

Le code complet de la méthode ConfigureServices devient ceci :

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

//configure identity server with in-memory stores, keys, clients and resources

services.AddIdentityServer(options =>

{

options.UserInteraction.LoginUrl = "/Identity/Account/Login";

options.UserInteraction.LogoutUrl = "/Identity/Account/Logout";

})

.AddDeveloperSigningCredential()

.AddInMemoryPersistedGrants()

.AddInMemoryIdentityResources(Config.GetIdentityResources())

.AddInMemoryApiResources(Config.GetApiResources())

.AddInMemoryClients(Config.GetClients())

.AddAspNetIdentity<ApplicationUser>();

}

Vous devez modifier par defaut la configuration de ASP.NET Core Identity pour definir l’entitée utilisé pour la gestion de role. Pour cela, devez editer le fichier IdentityHostingStartup.cs et changer services.AddDefaultIdentity<ApplicationUser>() par services.AddIdentity<ApplicationUser,IdentityRole>().

Le code de cette classe devient ce qui suit :

public void Configure(IWebHostBuilder builder)

{

builder.ConfigureServices((context, services) => {

services.AddDbContext<AspNetCoreIdentityServerContext>(options =>

options.UseSqlite(

context.Configuration.GetConnectionString("AspNetCoreIdentityServerContextConnection")));

services.AddDefaultIdentity<ApplicationUser>()

.AddEntityFrameworkStores<AspNetCoreIdentityServerContext>()

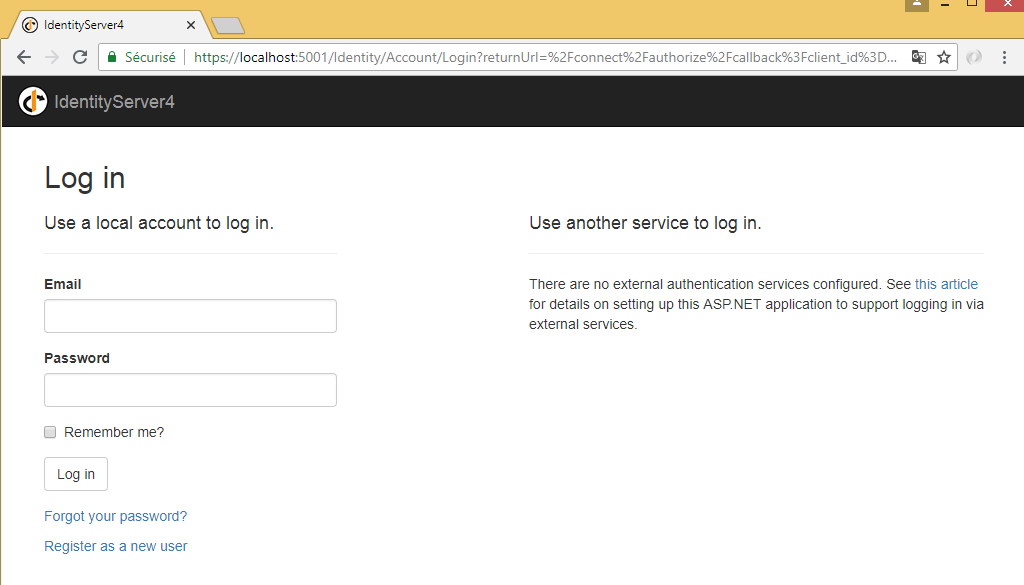
.AddDefaultTokenProviders()

.AddDefaultUI();

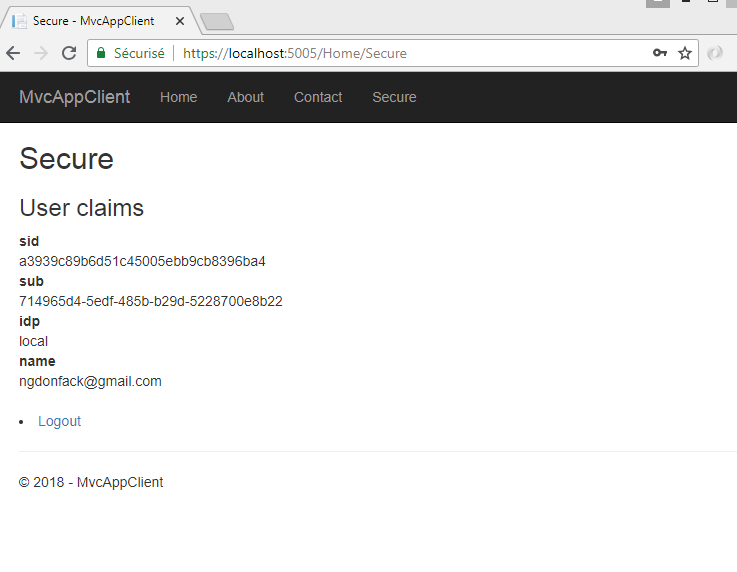
});

}

Sauvegardez et executez votre application. Lorsque vous aller essayer d’acceder à une ressource securisée, vous verrez le formualire de Login d’ASP.NET Core Identity.



Si nous nous connectons avec l’application MVCAppClient une fois un utilisateur créée, nous verrons ce qui suit :



IdentityServer4 : Propre base de données

Dans le billet précédent, nous avons vu comment utiliser OpenID et permettre à l’utilisateur de s’authentifier via un formulaire. Pour la mise en place de la fenêtre de connexion, de déconnexion, etc. nous avons utilisé un Quickstart offert par IdentityServer. Ce modèle repose sur TestUserStore, qui nous permet de définir et charger nos utilisateurs depuis fichier inclut dans le projet.

Le TestUserStore est offert à des fins de tests pour permetre aux developpeurs de demarrer facilement avec la prise en main de l’outil. Dans un projet concret d’entreprise, voous aurez votre propre base de données utilisateur et utiliserez ce dernier pour l’authentification.

Dans ce billet, nous verrons comment authentifier l’utilisateur en utilisateur notre propre service d’accès aux données et comment definir les revendications de l’utilisateur.

Nous utiliserons comme projet de base la solution suivante qui est disponible sur mon GitHub.

1 – Création du service

La premiere chose à faire sera de definir notre classe entité Utilisateur. Pour cela, nous allons créer dans le dossier Model la classe CustomUser suivante :

public class CustomUser

{

public string SubjectId { get; set; }

public string UserName { get; set; }

public string Password { get; set; }

public string FirstName { get; set; }

public string Email { get; set; }

}

La deuxieme étape sera la créatiion du UserRepository. Nous allons tout d’abord créé l’interface IUserRepository dans le dossier Repository du projet :

public interface IUserRepository

{

CustomUser FindByUserName(string userName);

CustomUser FindBySubjectId(string subjectId);

bool ValidateCredentials(string userName, string password);

}

Ensuite, nous allons ajouter l’implementation de cette interface :

public class UserRepository : IUserRepository

{

private List<CustomUser> \_customUsers = new List<CustomUser> {

new CustomUser

{

SubjectId = "1111",

UserName = "alice",

FirstName = "Alice Smith",

Email = "AliceSmith@email.com",

Password = "alice"

},

new CustomUser

{

SubjectId = "2222",

UserName = "bob",

FirstName = "Bob Smith",

Email = "BobSmith@email.com",

Password = "bob"

}

};

public CustomUser FindBySubjectId(string subjectId)

{

return \_customUsers.Find(x => x.SubjectId.Equals(subjectId));

}

public CustomUser FindByUserName(string userName)

{

return \_customUsers.Find(x=>x.UserName.Equals(userName));

}

public bool ValidateCredentials(string userName, string password)

{

var customUser = FindByUserName(userName);

return customUser != null && customUser.Password.Equals(password);

}

}

Afin que l’initialisation de ce service puisse se faire correctmenet, nous devons enregistrer celui-ci dans notre conteneur d’IoC ASP.NET Core. Vous devez editer la méthode ConfigureServices du fichier Startup.cs et ajouter la ligne de code suivante :

services.AddTransient<IUserRepository, UserRepository>();

Pour en savoir plus sur l’injection de dépendances avec ASP.NET Core, veuillez consulter mon billet de blog suivant :

Le controlleur AccourntController doit être modifié pour utiliser le UserRepository et sa méthode ValidateCredentials() pour valider l’identité de l’utilisateur.

private readonly IUserRepository \_userRepository;

public AccountController(

IIdentityServerInteractionService interaction,

IClientStore clientStore,

IAuthenticationSchemeProvider schemeProvider,

IEventService events,

IUserRepository userRepository)

{

\_interaction = interaction;

\_clientStore = clientStore;

\_schemeProvider = schemeProvider;

\_events = events;

\_userRepository = userRepository;

}

/// <summary>

/// Entry point into the login workflow

/// </summary>

[HttpGet]

public async Task<IActionResult> Login(string returnUrl)

{

// build a model so we know what to show on the login page

var vm = await BuildLoginViewModelAsync(returnUrl);

if (vm.IsExternalLoginOnly)

{

// we only have one option for logging in and it's an external provider

return RedirectToAction("Challenge", "External", new { provider = vm.ExternalLoginScheme, returnUrl });

}

return View(vm);

}

/// <summary>

/// Handle postback from username/password login

/// </summary>

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Login(LoginInputModel model, string button)

{

// check if we are in the context of an authorization request

var context = await \_interaction.GetAuthorizationContextAsync(model.ReturnUrl);

// the user clicked the "cancel" button

if (button != "login")

{

if (context != null)

{

// if the user cancels, send a result back into IdentityServer as if they

// denied the consent (even if this client does not require consent).

// this will send back an access denied OIDC error response to the client.

await \_interaction.GrantConsentAsync(context, ConsentResponse.Denied);

// we can trust model.ReturnUrl since GetAuthorizationContextAsync returned non-null

if (await \_clientStore.IsPkceClientAsync(context.ClientId))

{

// if the client is PKCE then we assume it's native, so this change in how to

// return the response is for better UX for the end user.

return View("Redirect", new RedirectViewModel { RedirectUrl = model.ReturnUrl });

}

return Redirect(model.ReturnUrl);

}

else

{

// since we don't have a valid context, then we just go back to the home page

return Redirect("~/");

}

}

if (ModelState.IsValid)

{

// validate username/password against in-memory store

if (\_userRepository.ValidateCredentials(model.Username, model.Password))

{

var user = \_userRepository.FindByUserName(model.Username);

await \_events.RaiseAsync(new UserLoginSuccessEvent(user.UserName, user.SubjectId, user.UserName));

// only set explicit expiration here if user chooses "remember me".

// otherwise we rely upon expiration configured in cookie middleware.

AuthenticationProperties props = null;

if (AccountOptions.AllowRememberLogin && model.RememberLogin)

{

props = new AuthenticationProperties

{

IsPersistent = true,

ExpiresUtc = DateTimeOffset.UtcNow.Add(AccountOptions.RememberMeLoginDuration)

};

};

// issue authentication cookie with subject ID and username

await HttpContext.SignInAsync(user.SubjectId, user.UserName, props);

if (context != null)

{

if (await \_clientStore.IsPkceClientAsync(context.ClientId))

{

// if the client is PKCE then we assume it's native, so this change in how to

// return the response is for better UX for the end user.

return View("Redirect", new RedirectViewModel { RedirectUrl = model.ReturnUrl });

}

// we can trust model.ReturnUrl since GetAuthorizationContextAsync returned non-null

return Redirect(model.ReturnUrl);

}

// request for a local page

if (Url.IsLocalUrl(model.ReturnUrl))

{

return Redirect(model.ReturnUrl);

}

else if (string.IsNullOrEmpty(model.ReturnUrl))

{

return Redirect("~/");

}

else

{

// user might have clicked on a malicious link - should be logged

throw new Exception("invalid return URL");

}

}

await \_events.RaiseAsync(new UserLoginFailureEvent(model.Username, "invalid credentials"));

ModelState.AddModelError("", AccountOptions.InvalidCredentialsErrorMessage);

}

Implémentation de l’interface IProfileService

Nous voulons que certaines informations de l’utilisateur (Email, nom, etc.) soient partagées avec les applications qui viennent s’authentifier via IdentityServer. Ces informations doivent être inclusent dans les Claims (Revendications). Pour mettre cela en place, nous devons fournir notre propre implementation de l’interface IProfileService :

public class ProfileService : IProfileService

{

private IUserRepository \_userRepository { get; set; }

public ProfileService(IUserRepository userRepository)

{

\_userRepository = userRepository;

}

public Task GetProfileDataAsync(ProfileDataRequestContext context)

{

var custormUser = \_userRepository.FindBySubjectId(context.Subject.FindFirst(x => x.Type == "sub").Value);

if (custormUser != null)

{

context.IssuedClaims = GetClaims(custormUser);

}

return Task.FromResult(0);

}

public Task IsActiveAsync(IsActiveContext context)

{

return Task.FromResult(0);

}

private List<Claim> GetClaims(CustomUser customUser) => new List<Claim> {

new Claim(JwtClaimTypes.Name, customUser.UserName),

new Claim(JwtClaimTypes.FamilyName, customUser.FirstName),

new Claim(JwtClaimTypes.Email, customUser.Email)

};

}

Modification de la configuration d’IdentityServer

Nous devons maintenant modifier la configuration d’IdentityServer pour enregistrer notre implementation de IProfileService et supprimer l’enregistrement du TestUserStore :

services.AddIdentityServer()

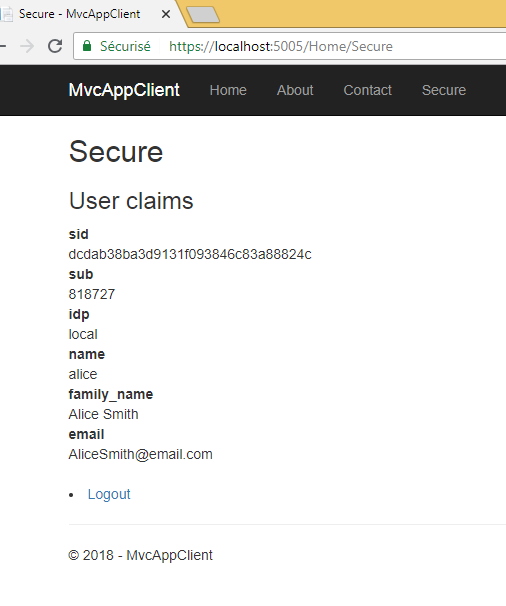
.AddDeveloperSigningCredential()

.AddInMemoryIdentityResources(Config.GetIdentityResources())

.AddInMemoryApiResources(Config.GetApiResources())

.AddProfileService<ProfileService>();

C’est tout. Vous pouvez tester votre application et vous authentifier en utilisant l’application MvcAppClient :



Implémentation de l’interface IResourceOwnerPasswordValidator

Su vous souhaitez que d’autres clients puissent obtenir des jetons d’authentification en fournissant directement leur nom d’utilisateur et leur mot de passe au token endpoint, vous devez fournir votre propre implémentation de l’interface IResourceOwnerPasswordValidator.

Supposons que nous voulons que l’application ConsoleAppClient utilise ce mode. Nous allons dans un premier temps changer son GrantTypes dans le Config.cs pour utiliser ResourceOwnerPassword :

new Client

{

ClientId = "consoleappclient",

AllowedGrantTypes = GrantTypes.ResourceOwnerPassword,

ClientSecrets =

{

new Secret("secret".Sha256())

},

AllowedScopes = { "testapi" }

},

Ensuite implémenter l’interface IResourceOwnerPasswordValidator pour valider l’identité de l’utilisateur en utilisant notre UserRepository :

public class ResourceOwnerPasswordValidator : IResourceOwnerPasswordValidator

{

private IUserRepository \_userRepository { get; set; }

public ResourceOwnerPasswordValidator(IUserRepository userRepository)

{

\_userRepository = userRepository;

}

public Task ValidateAsync(ResourceOwnerPasswordValidationContext context)

{

var customUser = \_userRepository.FindByUserName(context.UserName);

if (customUser != null && customUser.Password.Equals(context.Password))

{

context.Result = new GrantValidationResult(

subject: customUser.SubjectId,

authenticationMethod: OidcConstants.AuthenticationMethods.Password);

}

else

{

context.Result = new GrantValidationResult(

TokenRequestErrors.InvalidGrant,

"invalid credential");

}

return Task.FromResult(0);

}

}

Modifier la configuration de IdentityServer pour utiliser notre implementation de cette interface :

services.AddIdentityServer()

.AddDeveloperSigningCredential()

.AddInMemoryIdentityResources(Config.GetIdentityResources())

.AddInMemoryApiResources(Config.GetApiResources())

.AddResourceOwnerValidator<ResourceOwnerPasswordValidator>()

.AddProfileService<ProfileService>();

Maintenant, il ne nous reste plus qu’à mettre à jour le client pour passer ses iinformations d’identification lors de l’appel d’une ressource securisée :

static async Task CallWebApiR()

{

// discover endpoints from metadata

var disco = await DiscoveryClient.GetAsync("https://localhost:5001");

if (disco.IsError)

{

Console.WriteLine(disco.Error);

return;

}

// request token

var tokenClient = new TokenClient(disco.TokenEndpoint, "consoleappclient", "secret");

var tokenResponse = await tokenClient.RequestResourceOwnerPasswordAsync("alice", "alice", "testapi");

if (tokenResponse.IsError)

{

Console.WriteLine(tokenResponse.Error);

return;

}

Console.WriteLine(tokenResponse.Json);

// call api

var client = new HttpClient();

client.SetBearerToken(tokenResponse.AccessToken);

var response = await client.GetAsync("https://localhost:5003/api/secure");

if (!response.IsSuccessStatusCode)

{

Console.WriteLine(response.StatusCode);

}

else

{

var content = await response.Content.ReadAsStringAsync();

Console.WriteLine(JArray.Parse(content));

}

}

A l’exécution, on obtient ce qui suit :



Conclusion

<http://docs.identityserver.io/en/release/index.html>

<https://github.com/IdentityServer/IdentityServer4.Quickstart.UI>

<https://github.com/IdentityServer/IdentityServer4>