Les archictectures microservices gagenent en popularité dans le developpement d’applications modernes.

1 – Creation du projet et configuration de IdentityServer

1-1 Creation du projet

Nous allons commencer à partir de zéro en creant une nouvelle application ASP.NET Core qui sera notre IdentityServer. Elle doit être basée sur le modele « 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’enregisterer 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’enregister IdentityServer dans le conteneur d’IoC.

La dépendance minimale que nous avons besoin pour l’instant est AddDeveloperSigningCredential(). Cette extension permet de créer une clée temporaire et le necessaire piour signer les jetons (Tokens). C’est pratique pour demarrer en enviroenment de develloppement. Mais, vous ne devez pas le laisser trainer la en production et vous devez fournir le nécessaire pour gerer 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 obbligé 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" }

}

};

}

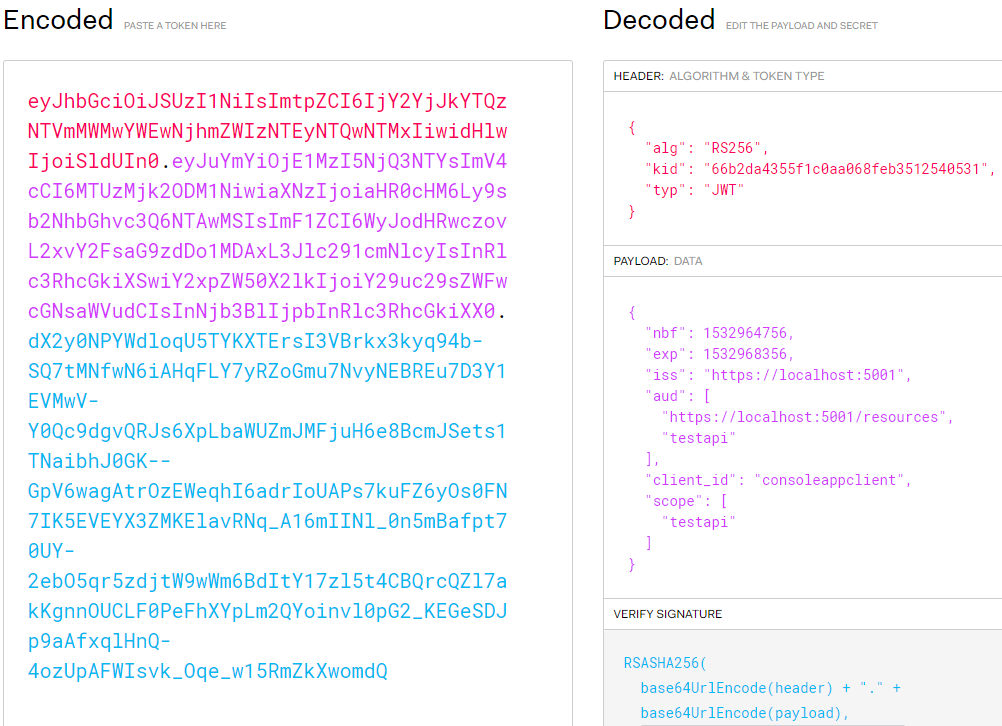
Vous remarquerez que nous avons également definir la ressource à laquelle le client doit acceder.

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'))

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 affichier la ligne de code permettant d’affiche 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())

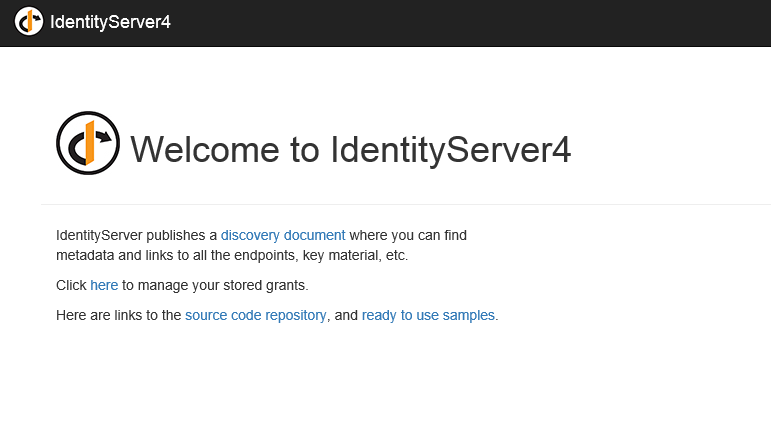
.AddInMemoryApiResources(Config.GetApiResources())

.AddInMemoryClients(Config.GetClients())

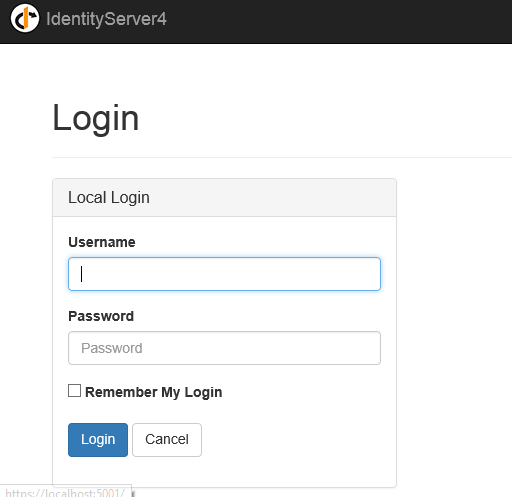
.AddTestUsers(Config.GetUsers()); ;

}

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



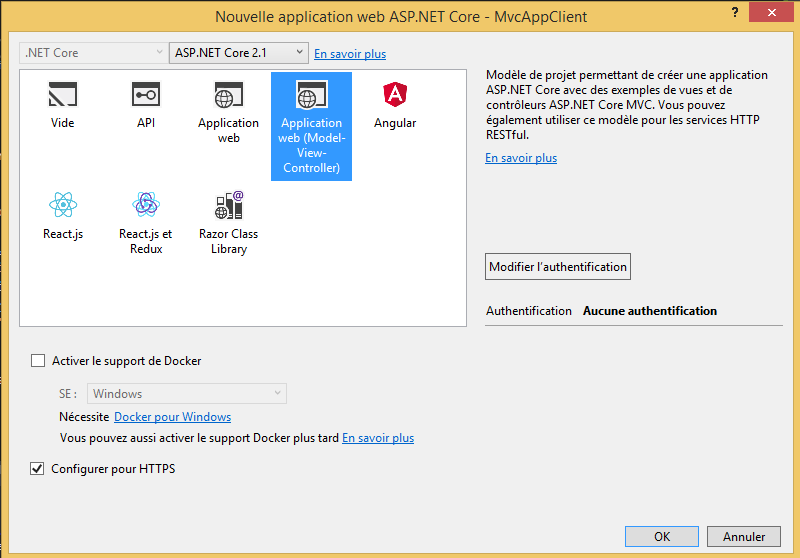
Si vous cliquez sur « Click here to manage your stored grants. », vous essayerez d’acceder à une page protégée. De ce fait, vous serez redirigé vers la page d’authentification :



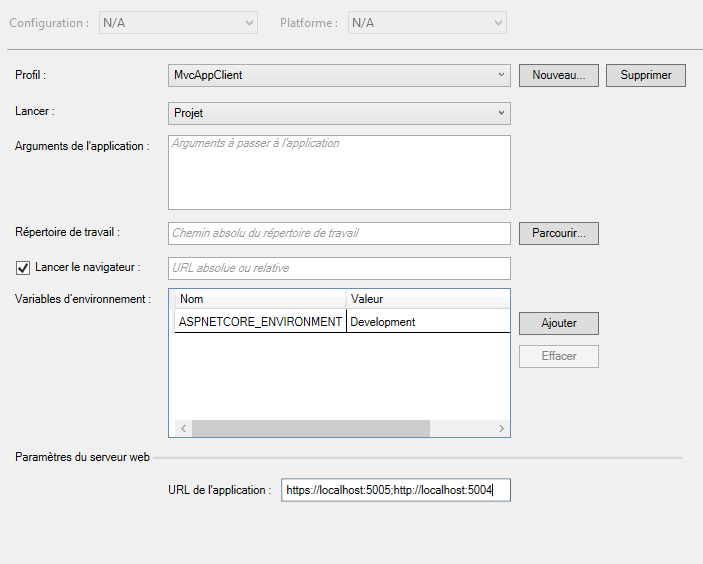
5 – Creation et configuration d’un client utilisant OpenID Connect

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.