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’un 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 editer 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 authentique 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.