Identification via tokens JWT dans une API REST en ASP.NET Core

Table des matières

Prérequis

Assurez-vous d'avoir la dernière version du SDK .NET Core (2.1.500 à l'heure de la redaction de ce guide). Si besoin est, installez la dernière version.
Créez un nouveau projet de type webapi à l'aide de la Command Line Interface (CLI) .NET Core.
Ouvrez ce projet à l'aide de Visual Studio Code.
Veillez à travailler dans un repository Git et à utiliser un fichier .gitignore bien configuré (gitignore.io, 2018).

Mise en place du projet

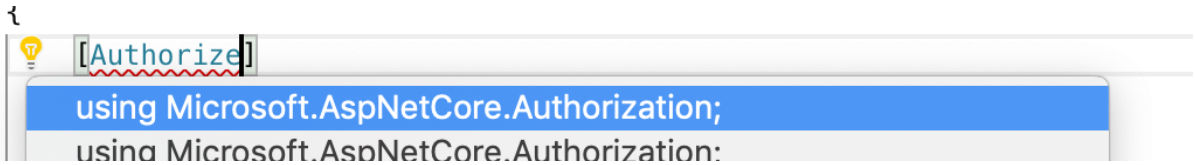
Lancez la restauration des packages

```
dotnet restore
```

Lancez ensuite le serveur de développement et essayez d'interroger le controller Values (via un simple Get) à l'aide de votre debugger http favori. A l'exception d'une éventuelle erreur liée au certificat utilisé (voir labos précédents), vous devriez obtenir une réponse avec un status code 200.

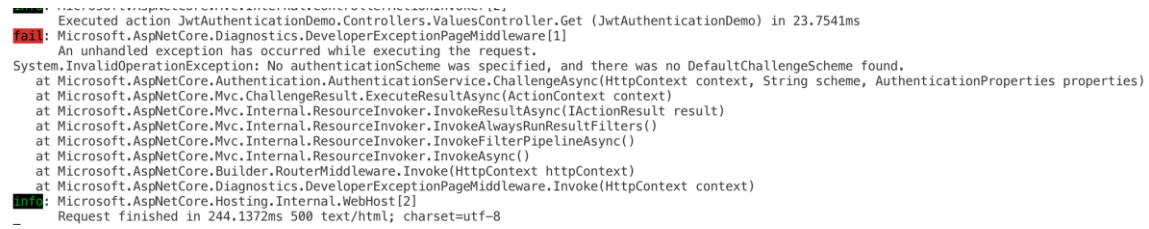
Nous allons dans un premier temps protéger ce controller afin que toute requête le ciblant doive être identifiée. Toute requête de laquelle l'identité de l'appelant ne peut être dérivée sera rejetée avec un status code http adéquat. Posez-vous la question : quel status code http représenterait un accès non-autorisé ? Si vous ne le savez pas, vous aurez la réponse un peu plus loin.

Afin de protéger le controller, décorez ce dernier avec un attribut *Authorize*.




```
[Authorize]
[Route("api/[controller]")]
[ApiController]
0 references
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    0 references
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "value1", "value2" };
    }
```

Relancez ensuite votre API et tentez d'interroger à nouveau votre controller. Vous obtiendrez une réponse 500 : Internal Server Error. Examinez la sortie produite par les logs de votre API :



Le runtime d'AspNetCore vous informe que vous avez protégé un controller, mais vous n'avez pas spécifié d'où extraire les informations d'identification. En effet, il ne suffit pas de dire que les informations d'identification doivent être extraites de la requête, il faut également spécifier de quelle manière : comment le client http passera-t-il ses informations d'identification à l'API, sous quel format ces informations seront-elles représentées. Nous allons nous y attarder dans la section suivante.

JWT

JWT signifie Json Web Tokens. Ce sont des structures de données représentées en Json, contenant de l'information (relative à l'utilisateur à l'origine de la requête http). Elles sont encodées dans un format spécifique et sont signées. Voici un exemple de token JWT.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJSb2xlIjpbIk1vZGVyYXRvciIsIkFkbWluIl0sInN1YiI6InNhbXVlbC5zY2hvbHRlcyIsImp0aSI6IjY5YWFlYjgyLTBlOTctNDA0OS1iZGE4LThjNDVmMzg0MmIzZSIsImlhdCI6MTU0MjgwNTIwMiwibmJmIjoxNTQyODA1MjAxLCJleHAiOjE1NDI4MDU1MDEsImlzcyI6IlN1cGVyQXdlc29tZVRva2VuU2VydmVyIiwiYXVkIjoiaHR0cDovL2xvY2FsaG9zdDo1MDAwLyJ9.mHH76WHl6JGBxGwQNJGhfNKw_kcqeuvyq9HOS1qhhXA

Remarquez que le token est constitué de trois parties, séparées par un point.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJSb2xlIjpbIk1vZGVyYXRvciIsIkFkbWluIl0sInN1YiI6InNhbXVlbC5zY2hvbHRlcyIsImp0aSI6IjY5YWFlYjgyLTBlOTctNDA0OS1iZGE4LThjNDVmMzg0MmIzZSIsImlhdCI6MTU0MjgwNTIwMiwibmJmIjoxNTQyODA1MjAxLCJleHAiOjE1NDI4MDU1MDEsImlzcyI6IlN1cGVyQXdlc29tZVRva2VuU2VydmVyIiwiYXVkIjoiaHR0cDovL2xvY2FsaG9zdDo1MDAwLyJ9.mHH76WHl6JGBxGwQNJGhfNKw_kcqeuvyq9HOS1qhhXA

Ces trois parties sont encodées en base64 et constituent :

1. Le header du token
2. Le corps du token (son contenu)
3. La signature du token

Allez sur (JWT.io, 2018a) et collez le token en question. Vous pourrez voir son contenu.

Le header du token contient des informations utiles au processus qui devra décoder le token :

- Type de token
- Algorithme utilisé pour la signature du token

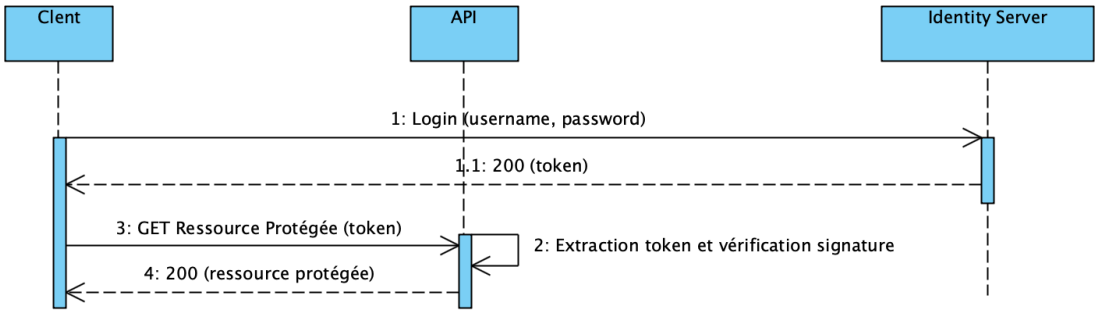
Le corps du token contient les informations utiles à l'identification de l'utilisateur (ex : son adresse e-mail, son ID dans la table des utilisateurs de la BD servant à l'identification…) Les

informations qu'on y retrouve sont des « **claims** ». Il en existe un ensemble défini mais cet ensemble est extensible (vous pouvez ajouter vos propres claims à un token JWT). Vous ajouterez des claims personnalisées plus tard. Pour la liste des claims déjà définies voir la section 4.1 de (Internet Engineering Task Force (IETF), 2015)

Pourquoi parle-t-on de claims ? L'utilisateur qui envoie la requête http *prétend* (de l'anglais *to claim*) être l'utilisateur X et avoir tel et tel droit. Rappelez-vous (voir support théorique + OWASP TOP 10): votre API ne doit pas croire aveuglément à ce que prétend un utilisateur : il doit le vérifier. Et c'est là que se trouve l'intérêt de la troisième partie du token : **la signature**.

La signature est utilisée pour vérifier que le token a bien été généré par un émetteur de confiance et qu'il n'a pas été altéré. Lorsque le token est généré par un fournisseur d'identité de confiance, la signature lui est ajoutée. Le tiers qui cherche à identifier l'utilisateur va vérifier si la signature est correcte avant de faire confiance aux informations contenues dans le token. La manière dont cette vérification de signature va se dérouler dépend de la manière dont le fournisseur de token a été configuré. C'est la raison pour l'inclusion du nom de l'algorithme de chiffrement au header du token JWT : le tiers cherchant à vérifier la signature sait sur quelle logique se baser.

Le diagramme suivant représente la dynamique d'identification dans un système « à jetons ».



L'application cliente qui souhaite accéder à une ressource protégée (notre ValuesController) doit se présenter avec un jeton d'identification, sans quoi elle sera rejetée avec un status code 401 (Unauthorized) par le serveur web[1]. Pour obtenir le jeton, elle doit s'adresser à un *Identity Server*. Les Identity Servers sont des services dont la responsabilité est de gérer l'identification. Les autres services (dont notre API) font confiance (déclaré dans la configuration de chacun de ces services) à 0-N Identity Servers pour l'identification : si une requête http arrive à un service, qu'elle contient un jeton d'identification fourni par un Identity Server de confiance (l'origine est vérifiée par le service à l'aide de la signature du jeton et d'une clé de chiffrement) alors le service accepte le jeton et en extrait les informations sur l'identité de l'utilisateur.

[1] Cette première tentative infructueuse n'est pas représentée ici.

La méthode utilisée pour vérifier la signature dépend de l'identity server utilisé. Elle sera tantôt basée sur des mécanismes de chiffrement symétriques ou asymétriques. Pour simplifier, si vous avez la main sur le serveur d'identité et sur les services qui l'utiliseront, alors le mécanisme symétrique (partage de la même clé de chiffrement côté chiffreur et déchiffreur) est suffisant. Dans le cas où le fournisseur d'identité n'est pas de votre ressort (ex : Facebook, Google ou autre Identity Server) alors un mécanisme asymétrique (clé privée + clé publique) est nécessaire.

L'identification dans les applications N-tiers modernes repose sur des principes de cryptographie vus en amont dans votre cursus. N'hésitez pas à relire vos cours.

Bien que des serveurs d'identité « clé en main » existent déjà, dans votre projet, vous intégrerez votre propre petit identity server et le configurerez, afin d'avoir une meilleure vue des implications d'une telle configuration. Vous fonctionnerez (par simplicité) avec un mécanisme de chiffrement symétrique (clé secrète partagée), puisque votre API génèrera les tokens et les validera également.

Prenez le temps de lire la brève introduction sur (JWT.io, 2018b) et poursuivez ensuite cet énoncé.

Remarque importante : JWT est portable. Il ne s'agit pas d'une spécificité .NET. Mais c'est dans votre implémentation d'une API REST en .NET Core que nous l'utiliserons.

Création d'un identity server dans votre API

Création du controller permettant de générer les tokens JWT

Créez un nouveau controller nommé JwtController.

Ce controller exposera une action nommée Login via la méthode HttpPost. Cette méthode prendra en entrée une instance d'une classe que vous définirez (ci-après LoginModel). Cette classe contiendra deux propriétés :

- Password
- UserName

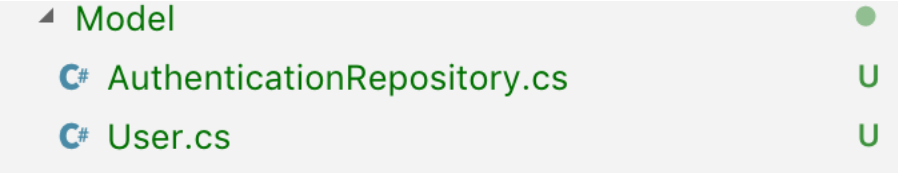
Question : à quoi ressemblera l'action Login (pensez aux attributs utiles au routage, pensez à la manière dont l'application cliente passera les informations au serveur, inspirez-vous du ValuesController et des exercices précédents…) ? Si vous ne trouvez pas, allez voir la Figure 1 - JWTController Login plus bas (mais faites l'effort de réfléchir et de trouver la solution par vous-même auparavant).

Cette action commencera par vérifier si l'objet passé en paramètre contient bien un UserName et un Password, sans quoi elle renverra une erreur 400 avec la raison du rejet.

Ensuite, elle recherchera dans la liste des utilisateurs connus une correspondance. Si elle ne retrouve aucun résultat, elle retournera une erreur 401. Dans le cas contraire, elle construira le jeton JWT et le retournera.

Pour simplifier les choses, nous allons créer une liste d'utilisateurs hardcodée. Dans la réalité, votre JwtController devrait dépendre d'une base de données stockant les utilisateurs, mais afin de limiter la complexité de l'exercice, nous nous passerons de l'accès aux données.

Créez dans votre projet un nouveau dossier nommé « Model ». Ce dossier contiendra deux classes : une classe *User* et une classe *AuthenticationRepository.*



```
1 reference
public class AuthenticationRepository
{

    1 reference
    private User[] _users=new User[]{
        new User(){ UserName="janedoe", Email="jane@doe.com", Id=1, Password="123"},
        new User(){ UserName="johndoe", Email="john@doe.com", Id=2, Password="456"},
    };
    1 reference
    public IEnumerable<User> GetUsers()
    {
        return _users;
    }
}
```

Le corps de votre action Login ressemblera à ceci (loginModel est le paramètre passé à l'action Login, ce que l'action a reçu de l'application cliente dans le corps de la requête).

```
// POST api/Jwt
[HttpPost]
0 references
public IActionResult Login([FromBody] DTO.LoginModel loginModel)
{
    if(!ModelState.IsValid)
        return BadRequest(ModelState);

    var repository=new JwtAuthenticationDemo.Model.AuthenticationRepository();
    Model.User userFound= repository.GetUsers().FirstOrDefault(user=>user.UserName==loginModel.UserName && user.Password==loginModel.Password );
    if(userFound==null)
        return Unauthorized();


    //TODO: build token and return it
    return Ok();
}
```

Figure 1 - JWTController Login

La classe DTO.LoginModel est à créer par vos soins. Elle contiendra deux propriétés publiques auto-implémentées : UserName (string) et Password (string).

Rien de compliqué jusqu'ici. Si vous vous demandez d'où vient « ModelState.IsValid » : il s'agit de la manière « standard » de procéder à de la validation simple en ASP.NET Core. Vous garnissez la classe modèle (ici DTO.LoginModel) d'attributs (des Data Annotations dans le jargon) qui représentent des contraintes. Si une instance de la classe ne respecte pas ces contraintes, la propriété IsValid du ModelState vaudra false. Voir (Microsoft, 2018a) et les annexes pour un exemple d'ajout de Data Annotations sur la classe DTO.LoginModel. Attention : rien de magique. Si votre classe LoginModel n'est pas décorée avec des Attributs de validation, aucune contrainte ne sera vérifiée et le ModelState sera toujours valide.

Il reste à construire le token et à le retourner. C'est l'objet de la section suivante.

Génération des tokens JWT

Bien que le token puisse être construit à la main (après tout, ce n'est qu'une question d'encodage et d'utilisation des algorithmes de chiffrement…), des implémentations « utilitaires » existent et facilitent la génération des tokens.

En .NET (pas seulement .NET Core), vous pouvez utiliser la classe JwtSecurityToken (assembly System.IdentityModel.Tokens.Jwt, namespace System.IdentityModel.Tokens.Jwt).

```
JwtSecurityToken token=new JwtSecurityToken(
    issuer: _jwtOptions.Issuer,
    audience: _jwtOptions.Audience,
    claims: allClaimsWithRoles,
    notBefore: _jwtOptions.NotBefore,
    expires: _jwtOptions.Expiration,
    signingCredentials: _jwtOptions.SigningCredentials
);
//TODO: build token and return it
return Ok();
```

Il faut instancier cette classe en lui passant plusieurs informations. Examinons-les :
Issuer : L'autorité qui a émis le jeton. L'identifiant de l'Identity Server. Cette valeur est configurée au démarrage de l'application web, voir plus bas.

- Audience : le service à qui le token est destiné.
- Claims : les Claims (voir définition ci-dessus). Il faut constuire la liste de claims que l'on souhaite inclure au jeton, voir plus bas).
- notBefore : début de validité du token.
- expires : date de fin de validité du token (expiration). La durée de validité d'un token est définie au démarrage de l'application web, voir plus bas.
- signingCredentials : informations utilisées pour signer le jeton émis.

La construction des claims est de votre ressort. Dans l'exemple ci-dessous, des claims sont extraites du profil utilisateur.

```
41          var claims = new[]
42          {
43              new Claim(JwtRegisteredClaimNames.Sub, userFound.UserName),
44              new Claim(JwtRegisteredClaimNames.Jti, await _jwtOptions.JtiGenerator()),
45              new Claim(JwtRegisteredClaimNames.Iat,
46                      ToUnixEpochDate(_jwtOptions.IssuedAt).ToString(),
47                      ClaimValueTypes.Integer64),
48          };
49          //IEnumerable<string> roles = await _userManager.GetRolesAsync(user);
50
51          JwtSecurityToken token = new JwtSecurityToken(
52              issuer: _jwtOptions.Issuer,
53              audience: _jwtOptions.Audience,
54              claims: claims,
55              notBefore: _jwtOptions.NotBefore,
56              expires: _jwtOptions.Expiration,
57              signingCredentials: _jwtOptions.SigningCredentials
58          );
59          var encodedJwt = new JwtSecurityTokenHandler().WriteToken(token);
60
61          // Sérialisation et retour
62          var response = new
63          {
64              access_token = encodedJwt,
65              expires_in = (int)_jwtOptions.ValidFor.TotalSeconds
66          };
67          return Ok(response);
68      }
```

Remarques :

- la classe Claim provient du namespace System.Security.Claims;
- pour la signification des différentes valeurs de JwtRegisteredClaimNames (structure définie dans le framework, allez voir sa definition), voir la documentation déjà donnée plus haut: (Internet Engineering Task Force (IETF), 2015)(section 4.1) et également (Richer & Sanso, 2017).
- la variable _jwtOptions est une variable d'instance du JwtController sur laquelle nous allons revenir.
- la méthode ToUnixEpochDate est une méthode utilitaire définie plus bas dans la classe JwtController.
- Notez l'utilisation du "await" pour le _jwtOptions.JtiGenerator() => votre action Login doit devenir asynchrone!

```
private static long ToUnixEpochDate(DateTime date)
  => (long)Math.Round((date.ToUniversalTime() -
                       new DateTimeOffset(1970, 1, 1,
                        0, 0, 0, TimeSpan.Zero))
                      .TotalSeconds);
}
```

Configuration de la generation des tokens JWT

La classe JwtController possède une variable d'instance nommée _jwtOptions. Il s'agit des options de generation des tokens JWT. Les valeurs de ces options sont définies par configuration, au démarrage de l'application web. Le controller reçoit ces options par injection (dans son constructeur).

```
[AllowAnonymous]
[Route("api/[controller]")]
[ApiController]
0 references
public class JwtController : ControllerBase
{

    9 references
    private readonly JwtIssuerOptions _jwtOptions;
    0 references
    public JwtController(IOptions<JwtIssuerOptions>
    jwtOptions)
    {
        _jwtOptions=jwtOptions.Value;
    }
```

Remarques:

- l'interface générique IOptions provient du namespace Microsoft.Extensions.Options.
- La classe JwtIssuerOptions est une classe que vous allez créer, dans votre projet et qui servira de conteneur pour les options de génération et validation JWT. https://gist.githubusercontent.com/williamhallatt/8e5e97dd27879b0500e73667e0336146/raw/72f048d28812d493843e4c98cea86162424b911b/JwtIssuerOptions.cs

La classe JwtIssuerOptions définit plusieurs valeurs par défaut. Examinez-les, vous vous en souviendrez peut-être plus tard lorsque vous vous poserez des questions telles que « combien de temps dure un token ?».

Pour que le controller puisse recevoir par injection les options de génération Jwt, il faut configurer l'application web. Rendez-vous dans la classe *Startup* et plus précisément dans la méthode ConfigureServices. C'est là que vous allez définir les valeurs des paramètres de génération Jwt qui seront injectés aux controllers de votre application.

```
0 references
public void ConfigureServices(IServiceCollection services)
{
    string SecretKey = "MaSuperCleSecreteANePasPublier";
    SymmetricSecurityKey _signingKey = new SymmetricSecurityKey
    (Encoding.ASCII.GetBytes(SecretKey));
    services.Configure<JwtIssuerOptions>(options =>
{
    options.Issuer = "MonSuperServeurDeJetons";
    options.Audience = "http://localhost:5000";
    options.SigningCredentials = new SigningCredentials(_signingKey,
    SecurityAlgorithms.HmacSha256);
});



    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

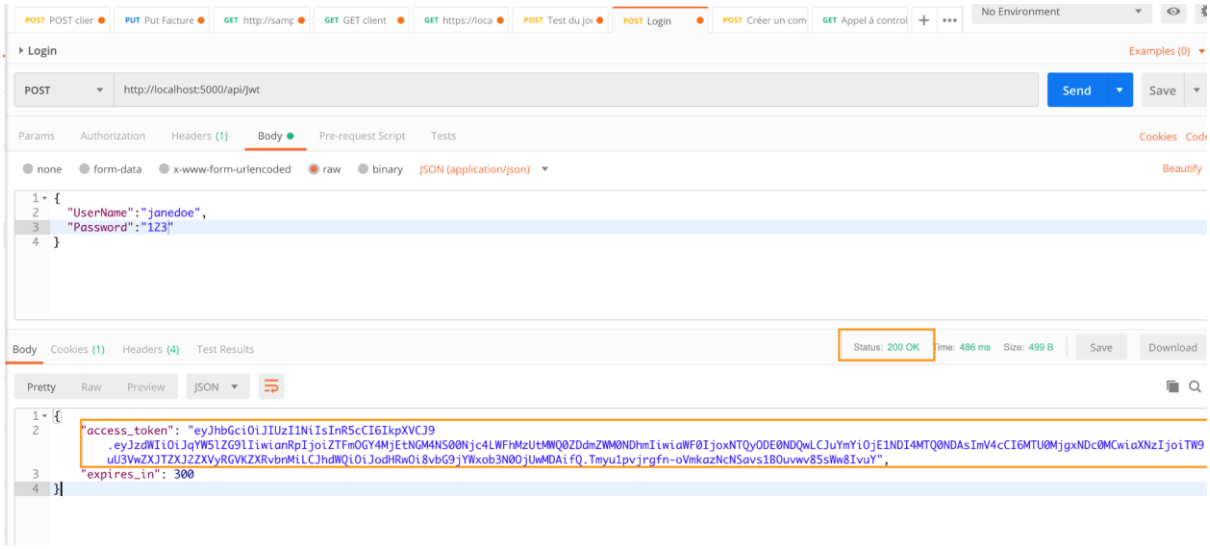
Comme précisé plus haut dans ce document, une clé secrète est utilisée pour signer le token et l'algorithme utilisé pour la signature doit être précisé.

Remarques :

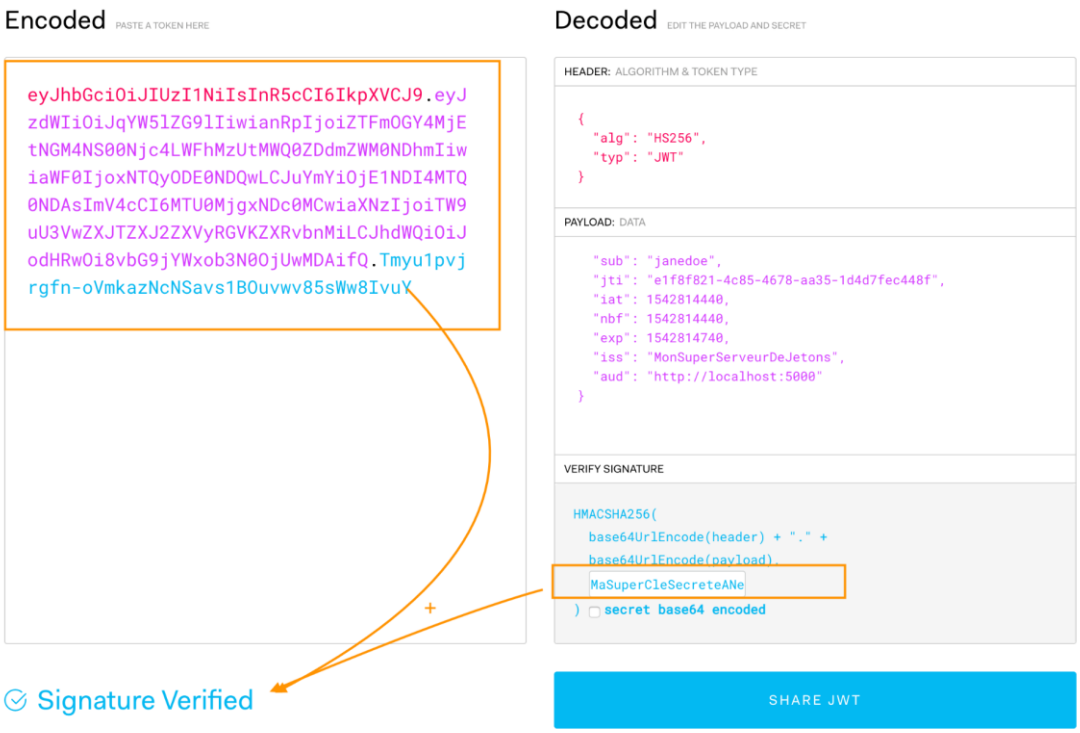
- La classe SymmetricSecurityKey est définie dans le namespace Microsoft.IdentityModel.Tokens.
- La classe Encoding est définie dans le namespace System.Text.
- Il conviendrait de ne pas hardcoder ces valeurs et de les récupérer par configuration, afin de rendre votre API déployable sur plusieurs environnements. Par souci de simplicité, cette étape de configuration a été laissée de côté.

Test de la génération de token

Tout est en place. Essayez de compiler puis de lancer votre application web. Invoquez à l'aide de votre debugger http favori l'action Login sur votre controller Jwt. Essayez les différents cas de figure (UserName/Password pas fourni, UserName + Password sans correspondance dans la base des utilisateurs, UserName+Password avec correspondance dans base utilisateurs). Dans ce dernier cas vous devriez obtenir en retour un token valide et sa date d'expiration.



Collez ce token dans le debugger disponible sur jwt.io (voir plus haut dans ce document) et examinez-en le contenu. Vérifiez également la signature en configurant le debugger avec la clé secrète (voir classe *Startup*) !



Vous avez terminé de travailler sur la partie Identity Server de cette introduction. Ce serveur est évidemment trop limité. Il conviendrait d'ajouter des fonctionnalités plus riches pour la gestion des utilisateurs (création d'utilisateurs, suppression…) et de rendre configurable les paramètres de génération du token. Mais pour l'heure, vous avez mis en place votre propre serveur d'identification en utilisant les classes mises à votre disposition par le .NET Framework. La section suivante vous montrera comment l'API qui fait confiance au

fournisseur d'identité que vous venez de créer pourra utiliser ce token pour identifier l'utilisateur.

Utilisation par une API du token émis par un fournisseur d'identité

Configuration de la logique de validation du token

Cette étape revient à :

1. extraire le token JWT de la requête http arrivant au serveur
2. valider la signature du token afin de s'assurer que c'est bien un Identity Server de confiance qui a émis le jeton et que personne ne l'a altéré.
3. initialiser un objet de type « User » que les controllers pourront utiliser pour connaître l'utilisateur à l'origine de la requête.

Avant toute chose, procédez à un commit de vos changements dans votre repository Git. Ceci vous permettra de revenir en arrière si vous pensez avoir loupé une étape.

Les étapes 1 à 3 sont implémentées en ASP.NET Core en ayant recours à du *Middleware*. Il s'agit de composants logiciels qui vont s'intégrer au pipeline de traitement d'une requête pour ajouter du comportement aux fonctionnalités standard d'ASP.NET Core.

Rendez-vous dans la classe Startup et ajoutez le code suivant à la méthode ConfigureServices (après le code déjà ajouté préalablement et avant l'appel à *services.AddMvc()*).

```
var tokenValidationParameters = new TokenValidationParameters
{
    ValidateIssuer = true,
    ValidIssuer = "MonSuperServeurDeJetons",

    ValidateAudience = true,
    ValidAudience = "http://localhost:5000",

    ValidateIssuerSigningKey = true,
    IssuerSigningKey = _signingKey,

    RequireExpirationTime = true,
    ValidateLifetime = true,

    ClockSkew = TimeSpan.Zero
};

services
    .AddAuthentication(
        options =>
        {
            options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
        })
    .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>
        {
            options.Audience = "http://localhost:5000";
            options.ClaimsIssuer = "MonSuperServeurDeJetons";
            options.TokenValidationParameters = tokenValidationParameters;
            options.SaveToken = true;
        });
```

Remarques :

- La classe *JwtBearerDefaults* vient du namespace Microsoft.AspNetCore.Authentication.JwtBearer
- La valeur des paramètres *Issuer* et *Audience* est la même que pour la configuration de la génération des tokens.

Spécification de la mécanique d'identification à utiliser pour protéger un controller

Retournez dans le ValuesController et modifiez l'attribut *Authorize* de ce dernier. Il doit ressembler à ceci.

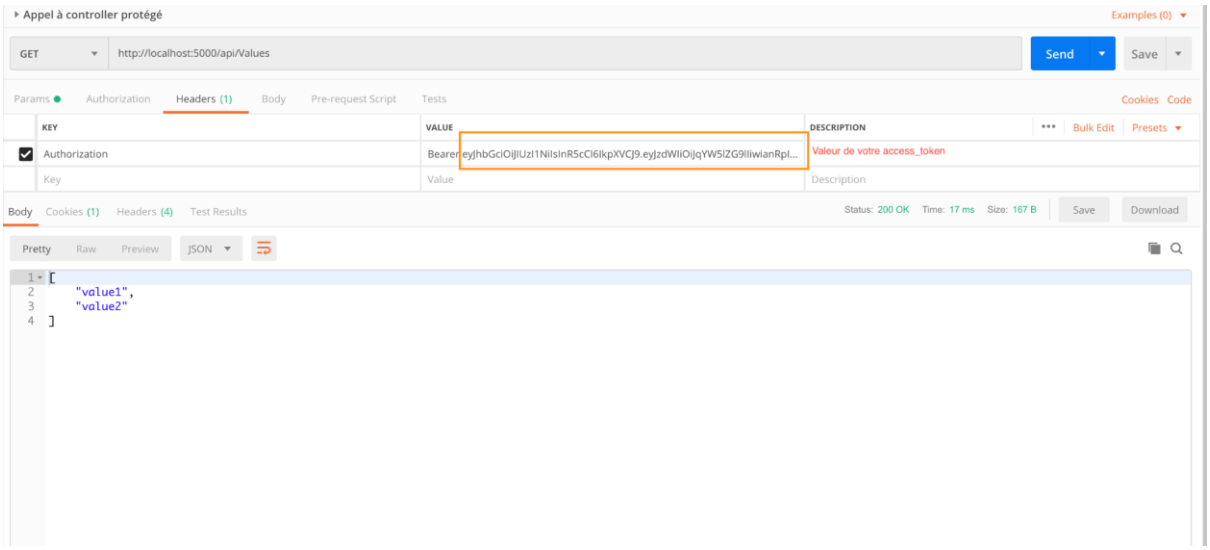
```
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
```

Cet attribut spécifie que pour l'identification, il faut passer par un jeton Jwt devant accompagner la requête. Pour d'autres manières de spécifier la mécanique d'identification à utiliser, voir https://docs.microsoft.com/en-us/aspnet/core/security/authorization/limitingidentitybyscheme?view=aspnetcore-2.1&tabs=aspnetcore2x

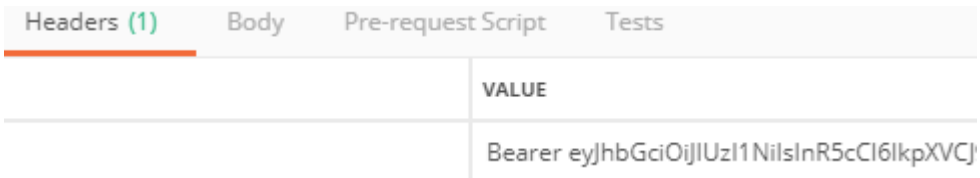
Vérification de l'implémentation de la protection

Lancez votre API et tentez à nouveau d'accéder au controller Values. Vous devriez cette fois obtenir une réponse 401. En effet, vous avez correctement configuré votre API en lui signalant que le mécanisme d'identification à utiliser était celui des tokens JWT présentés dans les requêtes http à traiter (voir *AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme* plus haut). Il vous reste à essayer d'obtenir un status code 200. Ce qui implique de générer un token, puis de le passer au serveur dans votre requête GET au controller Values.

Commencez par générer un token, comme déjà réalisé plus tôt. Copiez la valeur de la propriété « *access_token* » de l'objet Json retourné.
Composez ensuite une nouvelle requête au controller Values et à sa méthode Get. Dans les headers, ajoutez le suivant (exemple ci-dessous créé à l'aide de Postman, mais dans Fiddler c'est possible aussi).



<u>Attention à l'espace entrer Bearer et la valeur du token.</u>



Mise à disposition d'une propriété User garnie avec les informations du jeton

Ajoutez dans votre action Get du ValuesController le code suivant

```
// GET api/values
[HttpGet]
0 references
public ActionResult<IEnumerable<string>> Get()
{

    User.Claims.ToList().ForEach(claim=>Console.WriteLine($"Claim: {claim.Type}: {claim.Value}"));

    return new string[] { "value1", "value2" };
}
```

Tous vos controllers héritent de la classe *ControllerBase.* Nous en avons déjà parlé (voir méthodes utilitaires pour faciliter la construction des réponses http). Allez voir sa définition. Cette classe définit une propriété User qui expose les informations récupérées lors de l'identification de la requête. Tous vos controllers ont accès aux informations contenues dans le jeton : si vous y placez les rôles de l'utilisateur (voir construction du jeton, plus tôt dans ce document), vos controllers y auront également accès.

Conclusion

Ce document introductif a expliqué la structure des tokens JWT, vous a illustré la dynamique d'échange de jetons et de validation de ces derniers et vous en a proposé une implémentation en ASP.NET Core. Cette dynamique est un flux de OAuth. Vous en apprendrez davantage en lisant l'ouvrage disponible à la bibliothèque de l'IESN : OAuth in Action (https://bib.henallux.be/index.php?lvl=notice_display&id=549251).

Les serveurs d'identité sont utilisés pour extraire des différents applicatifs la logique d'identification et d'autorisation. Celle-ci peut alors être mutualisée (un identity server qui fournit la même identification à plusieurs applications). Pensez à une entreprise et à ses différents départements. Pour la gestion de ses activités chaque département utilise plusieurs applications développées au sein de l'entreprise. Chaque application va-t-elle proposer son propre mécanisme d'identification ? Chaque utilisateur va-t-il devoir connaître n logins/mot de passe ? Pensez au nombre d'applications qui à l'heure actuelle proposent l'identification via un fournisseur d'identité externe (Facebook, Google, Microsoft…) : elles le font car les utilisateurs n'ont pas un enième formulaire d'inscription à remplir => ils ne font pas directement demi-tour en voyant qu'ils vont à nouveau devoir créer un compte utilisateur, être imaginatif au niveau du mot de passe… Dans vos futures applications, pensez-y ! Offrez de l'identification mutualisable : http est votre protocole pour offrir une interface portable.

Ce document ayant une vocation introductive, certaines simplifications ont dû être opérées et des questions ont dû être mises en suspens. Ces points sont récapitulés ci-après.

Durée de vie et révocation du token

Que faire à l'expiration du token ? Dans la pratique, il conviendrait d'implémenter le mécanisme des « refresh tokens ». Ce sont des jetons « secondaires » qui sont utilisés pour récupérer un token « frais » une fois que le token d'origine a expiré. Par souci de simplicité, ce point n'est pas demandé dans votre projet Smart City. Si le token expire, redirigez l'utilisateur vers la page d'identification de votre application afin de récupérer un nouveau token. Si l'expiration du token vous ennuie, augmentez sa durée de vie. A noter : durant le

développement, il peut être intéressant de conserver une durée de vie courte pour le token, afin de tester correctement le comportement de l'application lors de l'expiration du token.

Pour être complet, il conviendrait également d'aborder la révocation des tokens. Imaginez en effet qu'un utilisateur se fasse pirater son compte. L'attaquant génère un token sur base de son username/password. Peu de temps après, l'utilisateur s'en rend compte et génère un nouveau mot de passe : il faut que le token généré par l'attaquant soit invalidé, afin qu'il ne puisse plus se faire passer pour l'utilisateur abusé.

Comment conditionner l'accès à certaines actions de vos controllers ?

Comment conditionner l'accès à certaines actions en fonction des rôles dont l'utilisateur est membre ? En utilisant **l'autorisation basée sur les rôles**. Voir (Microsoft, 2016c).

Comment aller plus loin et être plus granulaire dans les conditions d'accès aux actions de vos controllers ? En utilisant **l'autorisation claims-based**. Voir (Microsoft, 2016a) .

Les deux approches ci-dessus sont dites « déclaratives ». Mais vous pouvez coder ces conditions de manière impérative.

Stockage des utilisateurs

Enfin, pour être utilisable en production, votre serveur d'identification devrait s'intégrer à un véritable service de gestion des utilisateurs, persistant ces derniers dans une base de données, gérant le hashing des mots de passe…

Plusieurs librairies proposants ces fonctionnalités existent. L'une d'entre elles, particulièrement bien intégrée à ASP.NET Core et à Entity Framework Core est ASP.NET Core Identity : (Microsoft, 2016b).

Bibliographie

gitignore.io. (2018). Récupéré sur Gitignore.io: https://gitignore.io

Internet Engineering Task Force (IETF). (2015, 5). *JSON Web Tokens (JWT)*. Récupéré sur https://tools.ietf.org/html/rfc7519

JWT.io. (2018a). *JWT.io Debugger*. Récupéré sur https://jwt.io

JWT.io. (2018b). *JWT Introduction*. Récupéré sur https://jwt.io/introduction/

Microsoft. (2016a). *Claims-based authorization in ASP.NET Core*. Récupéré sur https://docs.microsoft.com/en-us/aspnet/core/security/authorization/claims?view=aspnetcore-2.1

Microsoft. (2016b). *Introduction to Identity on ASP.NET Core*. Récupéré sur https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-2.1&tabs=visual-studio

Microsoft. (2016c). *Role-based authorization in ASP.NET Core*. Récupéré sur https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles?view=aspnetcore-2.1

Microsoft. (2018a). *Model validation in ASP.NET Core MVC*. Récupéré sur https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-2.1

OWASP. (2018). *OWASP Top 10 Project*. Récupéré sur https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Richer, J., & Sanso, A. (2017). *OAuth 2 In Action.* Manning.

Annexes

```
using System.ComponentModel.DataAnnotations;

namespace JwtAuthenticationDemo.DTO
{
    1 reference
    public class LoginModel
    {
        [Required]
        1 reference
        public string UserName { get; set; }
        [Required]
        1 reference
        public string Password { get; set; }
    }
}
```

Figure 2 - Ajout de Data Annotations utilisées pour la validation du modèle