ASP.NET Core : Sécurité

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

1. Introduction
2. JWT
3. Installation
4. Connexion statique
5. Intégration de l'autorisation dans Swagger
6. Gestion de rôles
7. Bcrypt pour le hachage de mots de passe
8. Connexion dynamique
9. `AutoMapper`

ASP.NET Core

But de la sécurité

Interdire, à un utilisateur, l'accès à une ressource à laquelle il n'a pas droit

ASP.NET Core

But de la sécurité

Interdire, à un utilisateur, l'accès à une ressource à laquelle il n'a pas droit

Deux étapes

- Qui veut accéder à la ressource ? (**Authentification**)
- A t-il le droit d'y accéder ? (**Autorisation**)

ASP.NET Core

Deux systèmes d'authentification

- **Stateful** (avec état) : stocke les données de l'utilisateur dans les sessions et les cookies
- **Stateless** (sans état) : stocke les données dans un jeton (token) qui sera délivré au client

ASP.NET Core

Authentification **Stateful**

- Le client envoie une requête **HTTP** de type `POST` contenant le nom d'utilisateur et le mot de passe.
- Le serveur reçoit la requête et récupère les rôles de l'utilisateur.
- Dans un **Map** (`SESSION_ID` $\Rightarrow$ clé, données utilisateur $\Rightarrow$ valeur) nommé `sessions`, il ajoute une `entry` associée à l'utilisateur connecté.
- Le serveur retourne une réponse avec le code 200 et stocke dans les cookies du navigateur le `SESSION_ID` attribué à l'utilisateur.
- Chaque fois que le client envoie une requête au serveur, le `SESSION_ID` sera automatiquement attaché à l'entête de la requête.
- Le serveur vérifiera chaque fois que le `SESSION_ID` est bien présent dans le dictionnaire `sessions`.

Authentification **Stateless**

- Le client envoie une requête **HTTP** de type `POST` contenant le nom d'utilisateur et le mot de passe.
- Le serveur reçoit la requête et récupère les rôles de l'utilisateur.
- Le serveur génère un jeton (token) contenant les données utilisateur + signature
- Le serveur retourne une réponse avec le code 200 et un jeton. Le client doit stocker le jeton pour le réutiliser (dans le **Local Storage** par exemple).
- Chaque fois que le client envoie une requête au serveur, il doit ajouter le jeton à l'entête de la requête.
- Le serveur vérifiera chaque fois que la signature du jeton.

ASP.NET Core

Remarque

En utilisant un système de connexion de type **Stateful** et les cookies, on est exposé à une faille de sécurité appelé **CSRF**

ASP.NET Core

Remarque

En utilisant un système de connexion de type **Stateful** et les cookies, on est exposé à une faille de sécurité appelé **CSRF**

CSRF ou **C**ross-**S**ite **R**equest **F**orgery (via un exemple)

- Un utilisateur non authentifié `X` veut exécuter une action sur une application alors qu'il n'est pas authentifié
- Cette application utilise un système d'authentification **Stateful** (avec les cookies)
- `X` transmet alors à un utilisateur `U` authentifié à cette même application une image (par exemple) derrière laquelle se cache un script
- `U` clique sur l'image et envoie donc une requête **HTTP** avec son `SESSION_ID` sans qu'il le sache

ASP.NET Core

Quelles solutions pour les attaques **CSRF** ?

- Éviter d'utiliser la méthode **HTTP** `GET` pour l'exécution d'une action.
- Demander une confirmation pour chaque action critique
- Utiliser un jeton de validité (un jeton intégré dans tous les formulaires comme champ caché)
- ...

ASP.NET Core

Configuration de la sécurité

- Avec des données statiques (en mémoire, dans un fichier) [**InMemory**]
- Avec des données dynamiques (provenant d'une base de données) [**InDatabase**]

ASP.NET Core

Configuration de la sécurité

- Avec des données statiques (en mémoire, dans un fichier) [**InMemory**]
- Avec des données dynamiques (provenant d'une base de données) [**InDatabase**]

Dans **ASP.NET Core**

L'authentification est gérée par le `IAuthenticationService` : utilisé par le middleware d'authentification.

ASP.NET Core

JWT : **J**SON **W**eb **T**oken

- Librairie d'échange sécurisé d'informations
- Utilisant des algorithmes de cryptage comme **HMAC SHA256** ou **RSA**
- Utilisant les jetons (tokens)
- Un jeton est composé de trois parties séparées par un point :
  - entête (header) : objet **JSON** décrivant le jeton encodé en base 64
  - charge utile (payload) : objet **JSON** contenant les informations du jeton encodé en base 64
  - Une signature numérique = concaténation de deux éléments précédents séparés par un point + une clé secrète (le tout crypté par l'algorithme spécifié dans l'entête)
- Documentation officielle : `https://jwt.io/introduction/`

ASP.NET Core

Entête : exemple

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

ASP.NET Core

Entête : exemple

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Charge utile : exemple

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

ASP.NET Core

Exemple de construction de signature en utilisant l'algorithme précisé dans l'entête

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

ASP.NET Core

Exemple de construction de signature en utilisant l'algorithme précisé dans l'entête

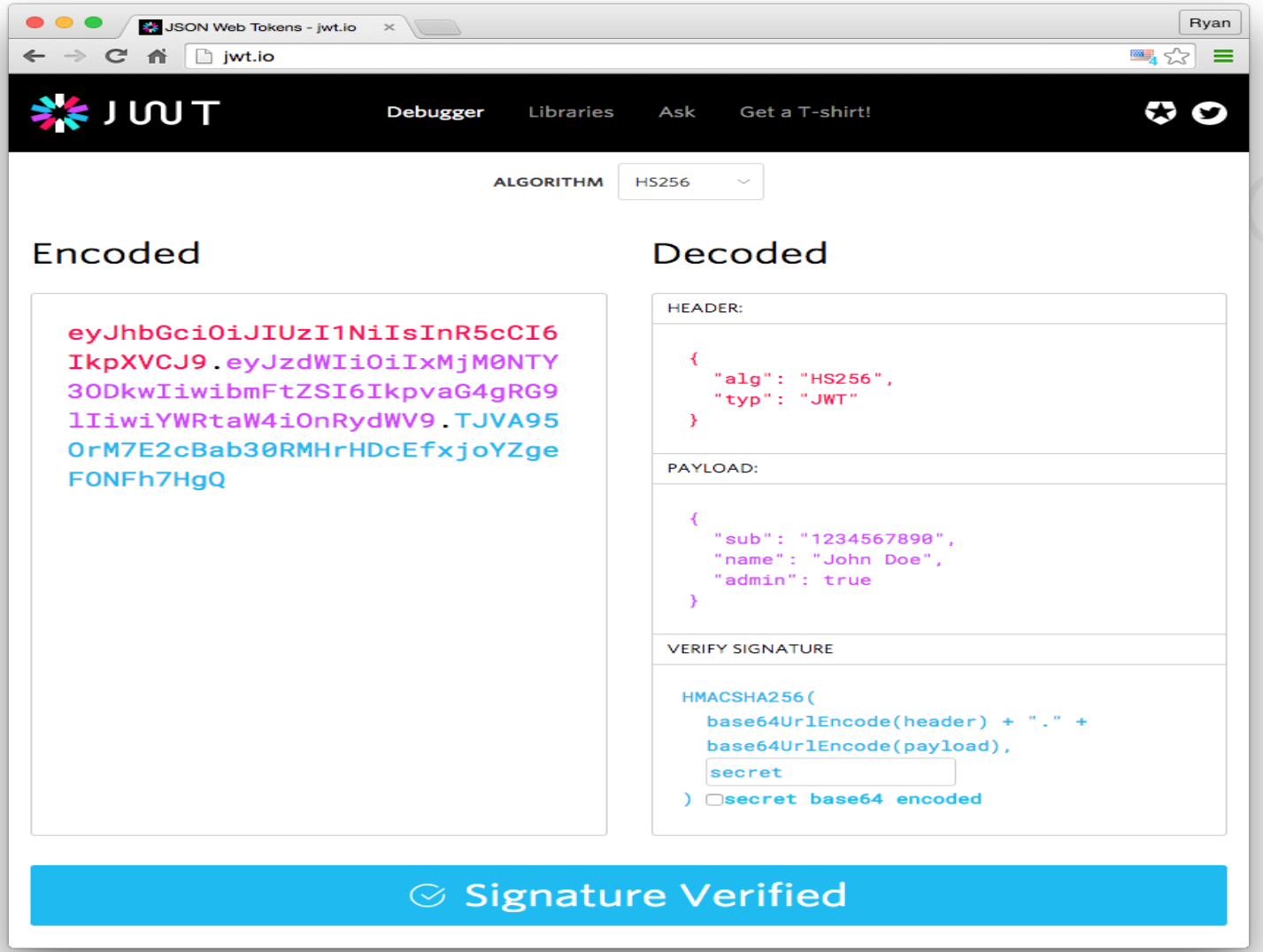
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Résultat

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4

ASP.NET Core

Exemple complet (pour tester `https://jwt.io/#debugger-io`)



ASP.NET Core

Pour décoder les deux premières parties d'un jeton

`http://calebb.net/`

ASP.NET Core

Remarque

Pour tester, utilisons le projet `CoursWebApiEF` créé dans le chapitre précédent.

ASP.NET Core

Remarque

Pour tester, utilisons le projet `CoursWebApiEF` créé dans le chapitre précédent.

Assemblies à installer

- **Microsoft.AspNetCore.Authentication.JwtBearer** (version 8.0.1)
- **Microsoft.IdentityModel.Tokens** (version 7.2.0)

ASP.NET Core

Rappel

Pour intégrer des nouveaux assemblies dans le projet, on utilise **NuGet**.

ASP.NET Core

Rappel

Pour intégrer des nouveaux assemblies dans le projet, on utilise **NuGet**.

NuGet

- Gestionnaire de paquets, par défaut, pour **.NET**
- Open-source et gratuit
- Inclus dans *Visual Studio* depuis 2012
- Utilisable aussi en ligne de commande

ASP.NET Core

Utiliser **NuGet** pour installer les assemblies

- Faire un clic droit sur `Dépendances` dans l'`Explorateur de solution`
- Choisir `Gérer les packages NuGet`
- Aller dans l'onglet `Parcourir` et chercher **Microsoft.AspNetCore.Authentication.JwtBearer**
- Choisir la version (version 8.0.1) et installer
- Accepter, attendre la fin de l'installation
- Vérifier l'ajout du package installé dans `Dépendances/packages`

ASP.NET Core

Utiliser **NuGet** pour installer les assemblies

- Faire un clic droit sur `Dépendances` dans l'`Explorateur de solution`
- Choisir `Gérer les packages NuGet`
- Aller dans l'onglet `Parcourir` et chercher **Microsoft.AspNetCore.Authentication.JwtBearer**
- Choisir la version (version 8.0.1) et installer
- Accepter, attendre la fin de l'installation
- Vérifier l'ajout du package installé dans `Dépendances/packages`

Refaire la même chose pour **Microsoft.IdentityModel.Tokens** (version 7.2.0).

ASP.NET Core

Démarche

- Créer une classe `UserDto` : **D**ata **T**ransfer **O**bject
- Ajouter les propriétés `Username`, `Password` qui nous permettrons de récupérer les identifiants de l'utilisateur non-connecté (enregistré ou non-enregistré dans notre système).
- Créer une classe `User` (entité) : qui contient les données d'un l'utilisateur connecté (enregistré dans notre système).

ASP.NET Core

Commençons par créer une classe `UserDtoRequest` qui va nous permettre de récupérer les identifiants de l'utilisateur qui souhaite se connecter

```
using System.ComponentModel.DataAnnotations;

namespace CoursWebApiEF.Models
{
    public class UserDtoRequest
    {
        [Required]
        public string Username { get; set; }
        [Required]
        public string Password { get; set; }
    }
}
```

ASP.NET Core

Commençons par créer une classe `UserDtoRequest` qui va nous permettre de récupérer les identifiants de l'utilisateur qui souhaite se connecter

```
using System.ComponentModel.DataAnnotations;

namespace CoursWebApiEF.Models
{
    public class UserDtoRequest
    {
        [Required]
        public string Username { get; set; }
        [Required]
        public string Password { get; set; }
    }
}
```

Les deux propriétés sont obligatoires pour la connexion.

ASP.NET Core

Créons aussi une classe `User` qui contiendra les données d'un utilisateur connecté

```
using System.Text.Json.Serialization;

namespace CoursWebApiEF.Models
{
    public class User
    {
        public int? Id { get; set; }
        public string Username { get; set; }
        public string Password { get; set; }
        public string? Nom { get; set; }
    }
}
```

ASP.NET Core

Dans `appsettings.json`, ajoutons la clé qui va nous permettre de décoder et encoder le jeton

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "CoursWebApiEFContext": "Server=(localdb)\\mssqllocaldb;
       Database=FormationWS;Trusted_Connection=True;"
  },
  "Jwt": {
    "Key": "UtiliseUneClePlusSecurisee"
  }
}
```

ASP.NET Core

Dans `Interfaces`**, créons l'interface** `IAuthenticationService`

```
namespace CoursWebApiEF.Interfaces
{
    public interface IAuthenticationService
    {
        User? CheckUser(UserDtoRequest userDto);
        string GenerateJWT(User user);
    }
}
```

ASP.NET Core

Dans `Services`, créons une classe `AuthenticationService` qui implémente l'interface `IAuthenticationService`

```
using CoursWebApiEF.Interfaces;

namespace CoursWebApiEF.Services
{
    public AuthenticationService :  IAuthenticationService
    {
    }
}
```

ASP.NET Core

Commençons par injecter `IConfiguration` pour récupérer les propriétés définies dans `appsettings.json`

```
using CoursWebApiEF.Interfaces;

namespace CoursWebApiEF.Services
{
    public class AuthenticationService :  IAuthenticationService
    {
        private readonly IConfiguration? _configuration = null;

        public List<User> Users { get; set; }

        public AuthenticationService(IConfiguration configuration)
        {
            _configuration = configuration;
        }
    }
}
```

ASP.NET Core

Définissons notre tableau d'utilisateurs dans le constructeur

```
public AuthenticationService(IConfiguration configuration)
{
    _configuration = configuration;
    Users = [
        new() { Id = 1, Nom = "Admin",  Username = "admin", Password = "admin" },
        new() { Id = 2, Nom = "User", Username = "user", Password = "user" }
    ];
}
```

ASP.NET Core

Définissons notre tableau d'utilisateurs dans le constructeur

```
public AuthenticationService(IConfiguration configuration)
{
    _configuration = configuration;
    Users = [
        new() { Id = 1, Nom = "Admin",  Username = "admin", Password = "admin" },
        new() { Id = 2, Nom = "User", Username = "user", Password = "user" }
    ];
}
```

Implémentons ensuite les méthodes concernant le `User`

```
public User? CheckUser(UserDtoRequest u)
{
    return Users.Find(elt => elt.Username == u.Username && elt.Password == u.Password);
}
```

Et enfin la méthode `GenerateJWT`

```
public string GenerateJWT(User user)
{

    var key = _configuration["Jwt:Key"];
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
    var token = new JwtSecurityToken(
         signingCredentials: credentials
    );
    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Et enfin la méthode `GenerateJWT`

```
public string GenerateJWT(User user)
{

    var key = _configuration["Jwt:Key"];
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
    var token = new JwtSecurityToken(
         signingCredentials: credentials
    );
    return new JwtSecurityTokenHandler().WriteToken(token);
}
```


Explication

- On commence par récupérer la clé et ensuite on construit un objet `SymmetricSecurityKey` pour la création et la validation du jeton.
- On spécifie l'algorithme de hachage
- On choisit ce qu'il faut ajouter dans le jeton
- On crée et on retourne le jeton

ASP.NET Core

Dans `Program.cs`**, définissons ce qu'il faut injecter pour** `IUserService`

```
builder.Services.AddScoped<IAuthenticationService, AuthenticationService>();
```

ASP.NET Core

Dans `Controllers`**, créons un contrôleur** `AuthenticationController`

```
namespace CoursWebApiEF.Controllers
{
    [Route("[controller]")]
    [ApiController]
    public class AuthenticationController : ControllerBase
    {
    }
}
```

ASP.NET Core

Dans `AuthenticationController`**, injectons l'interface** `UsersController`

```
namespace CoursWebApiEF.Controllers
{
    [Route("[controller]")]
    [ApiController]
    public class AuthenticationController : ControllerBase
    {
        private readonly IAuthenticationService _authenticationService;

        public AuthenticationController(IAuthenticationService authenticationService)
        {
            _authenticationService = authenticationService;
        }
    }
}
```

ASP.NET Core

Ajoutons maintenant la méthode d'authentification

```
[AllowAnonymous]
[HttpPost]
public IActionResult Login([FromBody] UserDtoRequest userDto)
{
    var user = _authenticationService.CheckUser(userDto);
    if (user != null)
    {
        var tokenString = _authenticationService.GenerateJWT(user);
        return Ok(tokenString);
    }
    return Unauthorized();
}
```

ASP.NET Core

On peut aussi retourner le jeton dans un objet JSON

```
[AllowAnonymous]
[HttpPost]
public IActionResult Login([FromBody] UserDto userDto)
{
    var user = _authenticationService.CheckUser(userDto);
    if (user != null)
    {
        var tokenString = _authenticationService.GenerateJWT(user);
        return Ok(new { token = tokenString });
    }
    return Unauthorized();
}
```

ASP.NET Core

Dans `Program.cs`, préparons les paramètres de validation de tokens (reçus)

```
var key = Encoding.ASCII.GetBytes(builder.Configuration["Jwt:Key"]);

var tokenParams = new TokenValidationParameters()
{
    ValidateIssuer = false,
    ValidateAudience = false,
    ValidateLifetime = false,
    ValidateIssuerSigningKey = true,
    IssuerSigningKey = new SymmetricSecurityKey(key),
};
```

ASP.NET Core

Dans `Program.cs`, précisons maintenant l'utilisation de `JWT` et chargeons les paramètres de validation

```
builder.Services.AddAuthentication(JwtBearerDefaults.
   AuthenticationScheme)
     .AddJwtBearer(options =>
     {
         options.RequireHttpsMetadata = true;
         options.TokenValidationParameters = tokenParams;
     });
```

ASP.NET Core

Dans `Program.cs`, veillons à bien respecter l'ordre de chargement de nos middlewares

```
app.UseHttpsRedirection();
app.UseCors("CorsPolicy");
app.UseAuthentication();
app.UseAuthorization();
app.MapControllers();
```

ASP.NET Core

Remarque 1

Avant de tester, décorons le contrôleur `PersonnesController` par l'attribut `[Authorize]` pour exigé que l'utilisateur soit authentifié et autorisé pour y accéder.

ASP.NET Core

Remarque 1

Avant de tester, décorons le contrôleur `PersonnesController` par l'attribut `[Authorize]` pour exigé que l'utilisateur soit authentifié et autorisé pour y accéder.

Remarque 2

Vérifiez que toutes les actions du contrôleur `PersonnesController` ne sont plus accessibles : **401 Unauthorized**.

ASP.NET Core

Pour obtenir le jeton avec **Postman**

- Dans la liste déroulante, choisir `POST` puis saisir l'url vers le web service `http://localhost:<port>/api/Users`
- Dans `Headers`, saisir `Content-Type` comme `Key` et `application/json` comme `Value`
- Ensuite cliquer sur `Body`, cocher `raw`, choisir `JSON (application/json)` et saisir des données sous format `JSON` correspondant à l'objet personne à ajouter

```
{
    "Username": "admin",
    "Password": "admin",
}
```

- Cliquer sur `Send` puis copier le jeton

ASP.NET Core

Pour obtenir la liste des personnes avec **Postman**

- Dans la liste déroulante, choisir `GET` puis saisir l'url vers notre web service `http://localhost:<port>/api/personnes`
- Dans `Headers`, saisir `Content-Type` comme `Key` et `application/json` comme `Value`
- Dans `Authorization`, cliquer sur `Type`, choisir `Bearer Token` et coller le token
- Cliquer sur `Send`

ASP.NET Core

Pour ajouter des données concernant l'utilisateur dans le jeton, on utilise la section `claims`

```
public string GenerateJWT(User user)
{
  var key = _configuration["Jwt:Key"];
  var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));
  var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
  Claim idClaim = new("id", user.Id.ToString());
  Claim nomClaim = new("nom", user.Nom);
  var token = new JwtSecurityToken(
    claims: [ idClaim, nomClaim ],
    signingCredentials: credentials
  );
  return new JwtSecurityTokenHandler().WriteToken(token);
}
```

ASP.NET Core

Ajoutons aussi une date d'expiration (dans notre exemple, le jeton expire dans 5 minutes)

```
public string GenerateJWT(User user)
{
  var key = _configuration["Jwt:Key"];
  var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));
  var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
  Claim idClaim = new("id", user.Id.ToString());
  Claim nomClaim = new("nom", user.Nom);
  var token = new JwtSecurityToken(
    claims: [ idClaim, nomClaim ],
    expires: DateTime.Now.AddMinutes(5),
    signingCredentials: credentials
  );
  return new JwtSecurityTokenHandler().WriteToken(token);
}
```

ASP.NET Core

Pour tester

- Utiliser **Postman** pour vous connecter avec les identifiants suivants

```
{
    "Username": "admin",
    "Password": "admin",
}
```

- Copier le jeton
- Aller à `http://calebb.net/`
- Coller le jeton et vérifier la présence de `"Id": 1` dans la deuxième partie.

ASP.NET Core

Pour renforcer la sécurité de notre jeton, utilisons `issuer` et `audience` pour les inclure dans la génération et la validation du token (contenu de `appsettings.json`)

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Jwt": {
    "Key": "UtiliseUneClePlusSecurisee",
    "Issuer": "https://localhost:7255",
    "Audience": "http://localhost:4200"
  }
}
```

ASP.NET Core

Incluons `issuer` et `audience` dans la génération du token

```
public string GenerateJWT(User user)
{
    var key = _configuration["Jwt:Key"];
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256
       );
    Claim idClaim = new("id", user.Id.ToString());
    Claim nomClaim = new("nom", user.Nom);
    var token = new JwtSecurityToken(
      issuer: _configuration["Jwt:Issuer"],
      audience: _configuration["Jwt:Audience"],
      claims: new Claim[] { idClaim, nomClaim },
      expires: DateTime.Now.AddMinutes(5),
      signingCredentials: credentials
    );
    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

ASP.NET Core

Dans `Program.cs`, modifions les paramètres de validation de tokens

```
var key = Encoding.ASCII.GetBytes(builder.Configuration["Jwt:Key"]);

var tokenParams = new TokenValidationParameters()
{
    ValidIssuer = builder.Configuration["Jwt:Issuer"],
    ValidAudience = builder.Configuration["Jwt:Audience"],
    ValidateIssuer = true,
    ValidateAudience = true,
    ValidateLifetime = true,
    ValidateIssuerSigningKey = true,
    IssuerSigningKey = new SymmetricSecurityKey(key),
};
```

ASP.NET Core

Il faut également remettre à zéro la durée par défaut (5 minutes) de validation de jeton

```
var key = Encoding.ASCII.GetBytes(builder.Configuration["Jwt:Key"]);

var tokenParams = new TokenValidationParameters()
{
    ValidIssuer = builder.Configuration["Jwt:Issuer"],
    ValidAudience = builder.Configuration["Jwt:Audience"],
    ValidateIssuer = true,
    ValidateAudience = true,
    ValidateLifetime = true,
    ValidateIssuerSigningKey = true,
    ClockSkew = TimeSpan.FromMinutes(0),
    IssuerSigningKey = new SymmetricSecurityKey(key),
};
```

ASP.NET Core

Remarque

Initialement, **Swagger** ne permet pas l'envoi de jetons.

ASP.NET Core

Remarque

Initialement, **Swagger** ne permet pas l'envoi de jetons.

Solution

Le reconfigurer

ASP.NET Core

Dans `Program.cs`, remplaçons la ligne suivante

```
builder.Services.AddSwaggerGen();
```

ASP.NET Core

Par le code suivant

```
builder.Services.AddSwaggerGen(swagger =>
{
    var jwtSecurityScheme = new OpenApiSecurityScheme
    {
        Scheme = "bearer",
        BearerFormat = "JWT",
        Name = "JWT Authentication",
        In = ParameterLocation.Header,
        Type = SecuritySchemeType.Http,
        Description = "Coller ici votre jeton",
        Reference = new OpenApiReference
        {
            Id = JwtBearerDefaults.AuthenticationScheme,
            Type = ReferenceType.SecurityScheme
        }
    };

    swagger.AddSecurityDefinition(jwtSecurityScheme.Reference.Id, jwtSecurityScheme);

    swagger.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        { jwtSecurityScheme, Array.Empty<string>() }
    });
});
```

ASP.NET Core

Relancez le projet et vérifiez la présence de la nouvelle section **Authorization**.

ASP.NET Core

Rôle

- Propriété qui permet d'autoriser ou d'interdire, un utilisateur connecté, à une ressource.
- Peut être spécifié dans le décorateur `[Authorize]`.

ASP.NET Core

Dans `User`, ajoutons une propriété `Role`

```
using System.Text.Json.Serialization;

namespace CoursWebApiEF.Models
{
    public class User
    {
        public int? Id { get; set; }
        public string Username { get; set; }
        public string Password { get; set; }
        public string? Nom { get; set; }
        public string Role { get; set; } = "ROLE_USER";
    }
}
```

ASP.NET Core

Dans le constructeur de `UserService`, ajoutons deux rôles différents à nos deux utilisateurs

```
public UserService(IConfiguration configuration)
{
    _configuration = configuration;
    Users = [
        new() { Id = 1, Nom = "Admin", Username = "admin", Password = "admin", Role = "
           ROLE_ADMIN" },
        new() { Id = 2, Nom = "User", Username = "user", Password = "user" }
    ];
}
```

ASP.NET Core

Il faut intégrer les rôles dans le token

```
public string GenerateJWT(User user)
{
    var key = _configuration["Jwt:Key"];
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256
       );
    Claim idClaim = new("id", user.Id.ToString());
    Claim nomClaim = new("nom", user.Nom);
    Claim roleClaim = new("role", user.Role);
    var token = new JwtSecurityToken(
      issuer: _configuration["Jwt:Issuer"],
      audience: _configuration["Jwt:Audience"],
      claims: [idClaim, nomClaim, roleClaim],
      expires: DateTime.Now.AddMinutes(5),
      signingCredentials: credentials
    );
    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

ASP.NET Core

Pour tester

- Utiliser **Postman** pour vous connecter avec les identifiants suivants

```
{
    "Username": "admin",
    "Password": "admin",
}
```

- Copier le jeton
- Aller à `http://calebb.net/`
- Coller le jeton et vérifier la présence de `"role": Admin` dans la deuxième partie.

ASP.NET Core

Dans `PersonnesController`, ajoutons les décorateurs `[Authorize]`) aux deux actions `Get`

```
[Authorize(Roles = "ROLE_ADMIN")]
[HttpGet]
public async Task<ActionResult<IEnumerable<Personne>>> GetPersonnes()
{
    return await _context.Personnes.ToListAsync();
}

[Authorize(Roles = "ROLE_USER")]
[HttpGet("{id}")]
public async Task<ActionResult<Personne>> GetPersonne(int id)
{
    var personne = await _context.Personnes.FindAsync(id);

    if (personne == null)
    {
        return NotFound();
    }

    return personne;
}
```

ASP.NET Core

Pour tester

- Connectez-vous avec `(admin, admin)` et vérifiez
  - que vous pouvez accéder à la route `/api/personnes`
  - qu'un message `403 Forbidden` s'affiche lorsque vous essayez d'accéder à la route `/api/personnes/1`
- Connectez-vous avec `(user, user)` et vérifiez
  - que vous pouvez accéder à la route `/api/personnes/1`
  - qu'un message `403 Forbidden` s'affiche lorsque vous essayez d'accéder à la route `/api/personnes`

ASP.NET Core

Remarque

Pour hacher le mot de passe, on peut utiliser `Bcrypt`.

ASP.NET Core

Remarque

Pour hacher le mot de passe, on peut utiliser `Bcrypt`.

Assembly à installer

`Bcrypt.Net-core` (version 1.6.0)

ASP.NET Core

Dans `AuthenticationService`, hachons les mots de passe dans le constructeur s

```
public AuthenticationService(IConfiguration configuration)
{
    _configuration = configuration;
    Users = [
        new()
        {
            Id = 1,
            Nom = "Admin",
            Username = "admin",
            Password = BCrypt.Net.BCrypt.HashPassword("admin"),
            Role = "ROLE_ADMIN"
        },
        new()
        {
            Id = 2,
            Nom = "User",
            Username = "user",
            Password = BCrypt.Net.BCrypt.HashPassword("user")
        }
    ];
}
```

ASP.NET Core

Utilisons la méthode `Verify` pour comparer les mots de passe saisis avec les mots de passe hachés

```
public User? CheckUser(UserDto data)
{
    return Users.Find(elt => elt.Username == data.Username
    && BCrypt.Net.BCrypt.Verify(data.Password, elt.Password));
}
```

ASP.NET Core

Objectif

Faire les modifications nécessaires pour que les utilisateurs soient enregistrés en base de données.

ASP.NET Core

Exercice : étape 1

- créer l'interface générique `IRepository` [voir page suivante]
- créer une classe générique `Repository` implémentant `IRepository`
- créer l'interface générique `IService` [voir page suivante]
- créer une classe générique `Service` implémentant `IService`

ASP.NET Core

Contenu de `IRepository`

```
namespace CoursWebApiEF.Repositories
{
    public interface IRepository<T> where T : class
    {

        IQueryable<T> GetAll();

        T? GetById(int id);

        T Add(T entity);

        T Update(T entity);

        void Delete(T entity);

    }
}
```

ASP.NET Core

Contenu de `IService`

```
namespace CoursWebApiEF.Interfaces
{
    public interface IService<T> where T : class
    {

        IQueryable<T> GetAll();

        T? GetById(int id);

        T Add(T entity);

        T Update(T entity);

        bool Delete(int id);

    }
}
```

ASP.NET Core

Considérons le contenu suivant de `IUserRepository`

```
using CoursWebApiEF.Models;

namespace CoursWebApiEF.Repositories
{
    public interface IUserRepository
    {
        public User? FindByUsernameAndPassword(string username, string password);
    }
}
```

ASP.NET Core

Et le contenu de `UserRepository` **implémentant** `IUserRepository`

```
using CoursWebApiEF.Configs;
using CoursWebApiEF.Models;

namespace CoursWebApiEF.Repositories
{
    public class UserRepository : Repository<User>, IUserRepository
    {
        public UserRepository(ApplicationContext context) : base(context)
        {
        }

        public User? FindByUsernameAndPassword(string username, string password)
        {
            var user = _dbSet.Where(elt => elt.Username == username).FirstOrDefault();
            if (user == null || !BCrypt.Net.BCrypt.Verify(password, user.Password))
            {
                return null;
            }
            return user;

        }
    }
}
```

ASP.NET Core

Exercice : étape 2

- Le contrôleur `PersonnesController` ne doit plus utiliser `ApplicationContext`, ni `Repository`, il doit uniquement utiliser le service.
- Créer `UsersController` qui doit également utiliser le service : vérifier que le mot de passe ajouté ou modifié est toujours crypté.

ASP.NET Core

N'oublions pas de déclarer les services dans `Program.cs`

```
builder.Services.AddDbContext<ApplicationContext>();
builder.Services.AddScoped<IUserRepository, UserRepository>();
builder.Services.AddScoped<IAuthenticationService, AuthenticationService>();
builder.Services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
builder.Services.AddScoped(typeof(IService<>), typeof(Service<>));
```

ASP.NET Core

Question

Comment faire pour ne pas inclure les `password` dans la réponse ?

ASP.NET Core

Question

Comment faire pour ne pas inclure les `password` dans la réponse ?

Réponse

Utiliser les Mappers.

ASP.NET Core

Assemblies à installer

- **AutoMapper**
- **AutoMapper.Extensions.Microsoft.DependencyInjection**

ASP.NET Core

Commençons par créer la classe `UserDtoResponse` qui contient toutes les propriétés de `User` sauf le `Password`

```
namespace CoursWebApiEF.Models
{
    public class UserDtoResponse
    {
        public int Id { get; set; }
        public string Username { get; set; }
        public string Nom { get; set; }
        public string Role { get; set; }
    }
}
```

ASP.NET Core

Dans un répertoire `Mappers`**, créons la classe** `UserProfile`

```
namespace CoursWebApiEF.Mappers
{
    public class UserProfile
    {
    }
}
```

ASP.NET Core

La classe `UserProfile` doit hériter de `Profile`

```
using AutoMapper;

namespace CoursWebApiEF.Mappers
{
    public class UserProfile : Profile
    {

    }
}
```

ASP.NET Core

Définissons dans le constructeur la source et la destination en utilisant `CreateMap`

```
using AutoMapper;
using CoursWebApiEF.Models;

namespace CoursWebApiEF.Mappers
{
    public class UserProfile : Profile
    {
        public UserProfile()
        {
            CreateMap<User, UserDtoResponse>();
        }
    }
}
```

ASP.NET Core

Dans `Program`, définissons `UserProfile` comme service

```
builder.Services.AddAutoMapper(typeof(UserProfile));
```

ASP.NET Core

Dans `UsersController`**, injectons** `IMapper`

```
using AutoMapper;
// + les autres using

namespace CoursWebApiEF.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class UsersController : ControllerBase
    {
        private readonly IService<User> _userService;
        private readonly IMapper _mapper;

        public UsersController(IService<User> userService, IMapper mapper)
        {
            _userService = userService;
            _mapper = mapper;
        }

        // + les autres actions

    }
}
```

ASP.NET Core

Modifions les actions de `UsersController` et commençons par la première

```
[HttpGet]
public IEnumerable<UserDtoResponse> GetUsers()
{
    var users = _userService.GetAll();
    var usersDto = _mapper.Map<List<UserDtoResponse>>(users);
}
```

ASP.NET Core

Modifions les actions de `UsersController` et commençons par la première

```
[HttpGet]
public IEnumerable<UserDtoResponse> GetUsers()
{
    var users = _userService.GetAll();
    var usersDto = _mapper.Map<List<UserDtoResponse>>(users);
}
```


Pour tester, aller à

`https://localhost:<port>/api/users`

ASP.NET Core

Exercice

Utiliser le mapper dans les autres actions de `UsersController`.

ASP.NET Core

Question

Serait-il possible de renommer certaines propriétés ?

ASP.NET Core

Question

Serait-il possible de renommer certaines propriétés ?

Réponse

Oui, on doit aussi l'indiquer dans le mapper.

ASP.NET Core

Commençons par renommer `Username` **en** `NomUtilisateur` **dans** `UserDtoResponse`

```
namespace CoursWebApiEF.Models
{
    public class UserDtoResponse
    {
        public int Id { get; set; }
        public string NomUtilisateur { get; set; }
        public string Nom { get; set; }
        public string Role { get; set; }
    }
}
```

ASP.NET Core

Définissons le mapping entre `NomUtilisateur` **et** `Username` **dans** `UserProfile`

```
using AutoMapper;
using CoursWebApiEF.Models;

namespace CoursWebApiEF.Mappers
{
    public class UserProfile : Profile
    {
        public UserProfile()
        {
            CreateMap<User, UserDtoResponse>()
                .ForMember(
                    dest => dest.NomUtilisateur,
                    opt => opt.MapFrom(src => src.Username)
                );
        }
    }
}
```

ASP.NET Core

Définissons le mapping entre `NomUtilisateur` **et** `Username` **dans** `UserProfile`

```
using AutoMapper;
using CoursWebApiEF.Models;

namespace CoursWebApiEF.Mappers
{
    public class UserProfile : Profile
    {
        public UserProfile()
        {
            CreateMap<User, UserDtoResponse>()
                .ForMember(
                    dest => dest.NomUtilisateur,
                    opt => opt.MapFrom(src => src.Username)
                );
        }
    }
}
```

Pour tester, aller à

`https://localhost:<port>/api/users`

ASP.NET Core

Remarque

Il est possible d'enchaîner plusieurs `ForMember` dans un mapper.