Le projet :

1- Créer une application Web en **en Razor** et une api en ASP.NET Web API 6.

2- Installer et configurer le serveur web **Apache2** sur le serveur linux **Debian 192.168.153.131** afin de pouvoir héberger l'app web.

3- Installer et configurer le serveur web **Nginx** pour héberger l'app web sur le serveur **CentOS 192.168.153.133.**

4- Installer et configurer le serveur web **IIS** sur le contrôleur de domaine **Windows 192.168.153.130.**

5- Installer et configurer un serveur **Jenkins** sur **192.168.153.132** qui permettra de déployer et intégrer de façon continue les sources de l'application web sur les différents nœuds (Linux **Debian, CentOS** et **Windows)**.

6- Faire la même chose que (5) mais déployer sur des workers d'un node master **Kubernetes**.

7- Configurer **Jenkins** pour déployer automatiquement sur le serveur linux **Debian et CentOS** les sources de l'application au moment du merge sur la branche master de **GitHub**.

8- Ne pas installer docker sur le serveur linux cible **Debian** mais d'ajouter un plugins **Docker** dans **Jenkins** afin de Builder le **Dockerfile** ou même de lancer le conteneur.

9- Créer un serveur de base de données **MSSQL** définit dans le contrôleur de domaine **192.168.153.130**.

10- Créer une image pour une **API** web en NET6.0 définit dans un service **Docker**.

11- Installer et configurer l'utilitaire **Traefik** permettant de concilier dans le même réseau les deux services **Docker** (**api** et **App**).

- Caching
- Répartition de charges entre services docker (notamment pour l'api .net6)
- SSL/TLS (on va le faire via haproxy)

12- On va utiliser **Traefik** de deux façons :

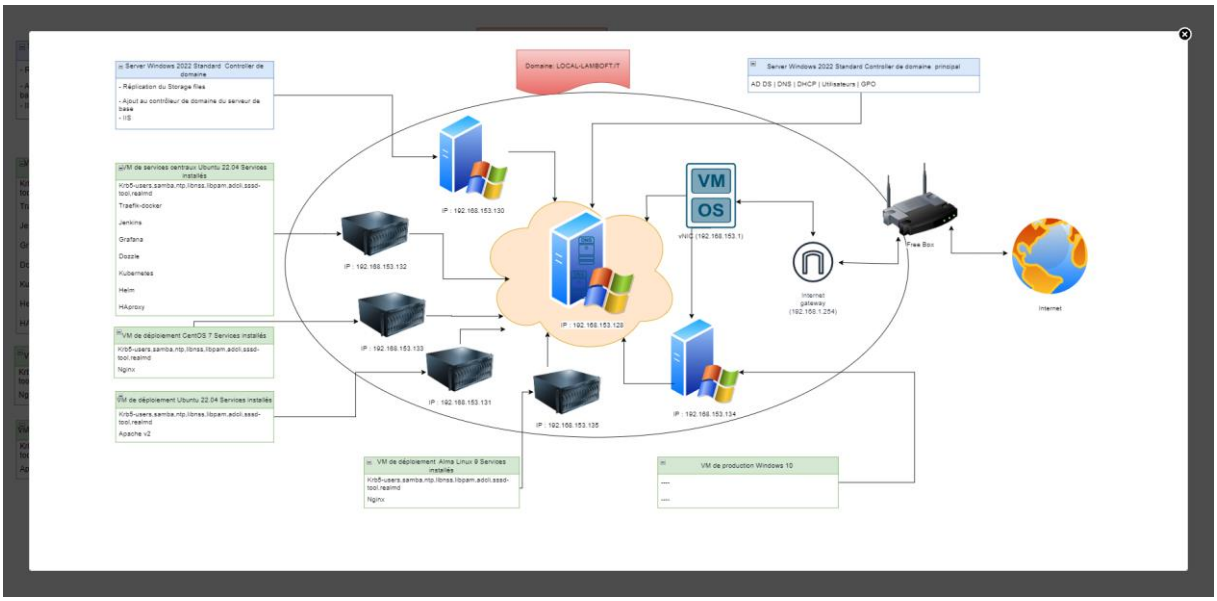
- Avec un fournisseur docker
- Avec un fournisseur kube

13- Installer et configurer un **DOZZLE** permettant de lire les logs des applications isolées dans **Docker**.

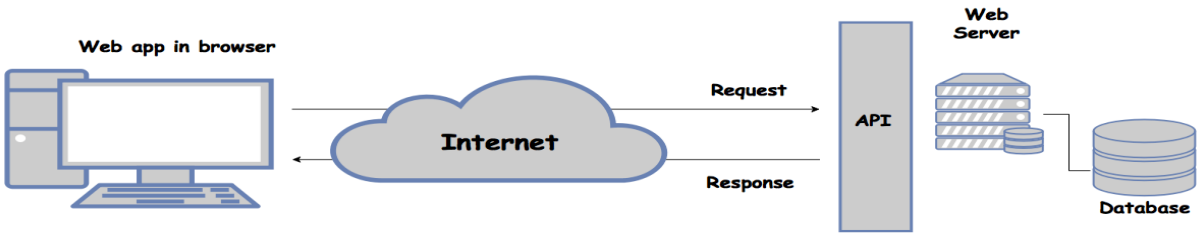
14- Utiliser **Ansible** notamment les rôles ansible pour déployer les configurations automatiquement sur le serveur Linux **Debian et CentOS et Windows** (la création de site web, pool d'application, chemin virtuel et physique**)**.

15- Installation de **Grafana** pour contrôler et monitorer les **VM.**

16- Plugger **prométhéums** , **Traefik** et **Grafana** ???



SWAGGER D'API .NET6



<u>Préparation et compréhension de l'api swagger NET6.0</u> :

On va développer une api web api en dotnet 6 et décrire son swagger (langage de programmation de description des actions à mener pour l'interrogation et la collecte des données via des méthodes GET, POST, PUT ,PATCH et DELETE pour des requêtes http et https).

1. **Kestrel** : Il s'agit du serveur web utilisé par défaut avec ASP.NET Core. Kestrel est responsable de la gestion des connexions entrantes, y compris les connexions HTTPS. Dans votre configuration Kestrel, vous spécifiez les paramètres de votre serveur HTTPS, notamment le certificat à utiliser pour sécuriser les connexions.
2. **AuthenticationCertification** : C'est une classe que vous avez créée pour gérer l'authentification par certificat dans votre application ASP.NET Core. Cette classe intervient après que Kestrel a établi une connexion HTTPS avec un client. Elle extrait le certificat client de cette connexion et effectue des vérifications personnalisées sur ce certificat pour authentifier le client.
3. **Chiffrement des données** : C'est le mécanisme par lequel on va sécuriser la transmission des informations entre client et serveur.

Il garantit :

- **Confidentialité : E**mpêcher la lecture de l'identité ou des données utilisateurs
- **Intégrité des données :** Données non modifiées
- **Authentification :** S'assurer que l'expéditeur de la donnée soit celui attendu
- **La non-répudiation** : L'expéditeur de la donnée ne peut pas nier son implication

En résumé, Kestrel est responsable de la gestion des connexions sécurisées HTTPS et de l'utilisation des certificats pour sécuriser ces connexions. Ensuite, une fois qu'une connexion HTTPS est établie, votre classe `AuthenticationCertification` intervient pour gérer l'authentification par certificat spécifiquement pour les demandes reçues via cette connexion HTTPS.

Vous avez besoin des deux éléments pour garantir un système d'authentification sécurisé par certificat dans votre application ASP.NET Core.

https://youtu.be/T7PvHcGGoNk?list=PLSuzYIVSEUT6SE-5dSZq-zbtItfHqWdqv

Brainstorming :

Créer son propre package nuget pour déployer en environnement de production ?

Comprendre comment fonctionne la cryptographie ?

Les clés symétriques et le handshake pour le certificat ?

Les types de certificats ?

Comment séparer ces différents fichiers de configuration et permettre à notre application d'utiliser leur configuration en fonction des environnements ?

Environnement de Développement : bd en mémoire, logs Informations

Environnement de Préproduction ou staging : bd en SQLite, logs Debug

Environnement de Production : bd en SQL dans Docker, logs Informations

Le fichier launchSettings.json :

- Est utilisé uniquement sur l'ordinateur de développement local.
- N'est pas déployé en production.
- Contient les paramètres de profil.

https://moodle.utc.fr/pluginfile.php/16777/mod_resource/content/0/SupportIntroSecu/co/CoursSecurite_2.html.

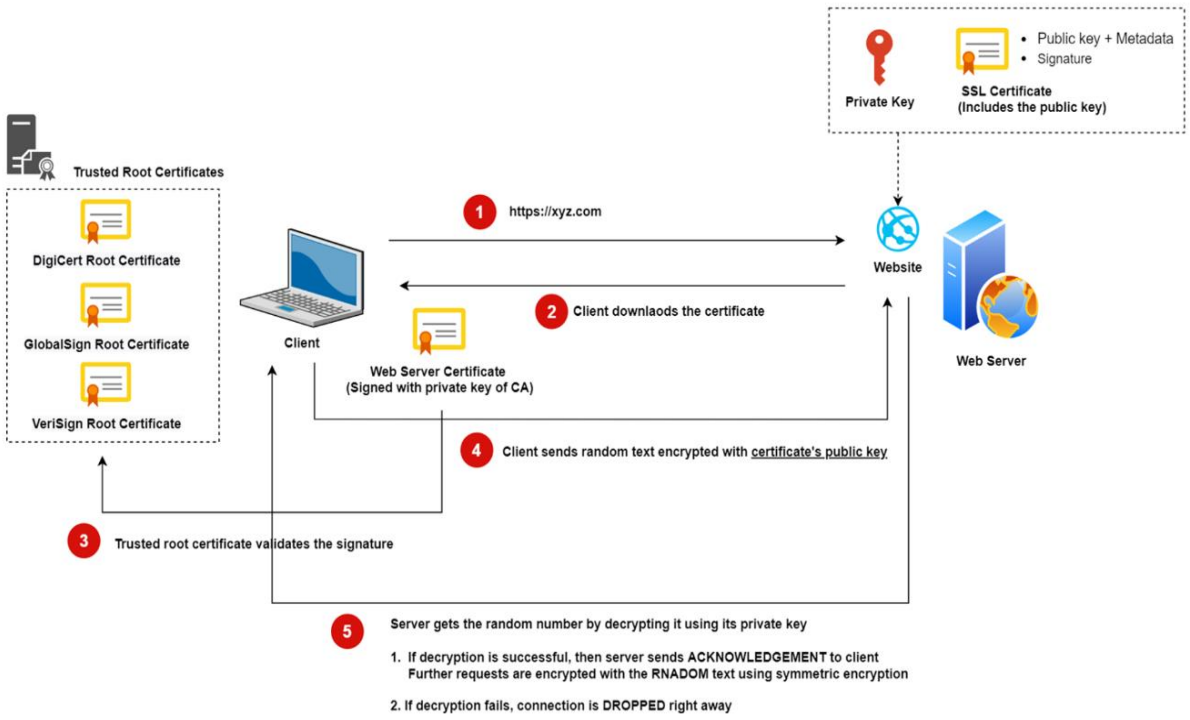
I. Mise en place d'un certificat auto-signé (TLS/SSL)

I Création du certificat

Certificat : C'est l'identité numérique d'une entité qui assure le(a) chiffrement/sécurité des communications entre un objet A et un objet B sur des pages web via le protocole https sur un navigateur web.

Objectifs :

- On souhaite encrypter les communications entrantes dans notre API.
- Protéger l'accès à nos points de terminaisons par un certificat valide.
- Accéder à l'api en Https



Ce schéma ci-dessus est une illustration de comment on peut faire pour créer son propre certificat et sa demande de signature auprès d'une CA que l'on a soi-même créé.

La création de certificat se fera en trois temps :

- Création d'une CA et son installation dans le magasin de certificats racine de Windows.

- sudo openssl genrsa -aes256 -out $CANAME.key 4096
- sudo openssl req -x509 -new -nodes -key $CANAME.key -sha256 -days 1826 -out $CANAME.crt
- sudo cp $CANAME.crt /d/Certificates/ (Dans un répertoire de Windows)

CANAME=LamboFT-RootCA

On va générer via ***openssl*** :

- Une clé privée pour notre CA
- A partir de la clé privée créer un certificat dont la date d'expiration est de 5 ans
- Copier ce certificat dans la partition de D de Windows

Sur ***Windows*** faudra :

- Cliquer sur le certificat
- L'installer sur l'os
- Choisir ordinateur local
- Choisir le magasin autorité de certificats racine

Sur Linux :

- Copier le certificat crt dans /usr/local/share/ca-certificates/
- **Sudo update-ca-certificate**

NB : lors de la création du certificat du CA des informations peuvent être demandées telles que :

- Le pays, région, la ville dans lequel on est
- L'email
- Le nom convivial du certificat
- Le mot de passe de la clé privée
- L'organisation pour laquelle on crée le certificat
- La section ou le département pour lequel le certificat est crée

On peut bien ne pas les renseigner excepté le mdp de la clé privée que notre CA sera toujours valide.

Exemple :

Comme on peut le voir dans la capture ci-dessus on a une hiérarchisation des certificats dans la mesure où notre certificat racine de confiance contient en son sein une demande de certificat pour un serveur en particulier.

- Création d'un certificat SSL pour le serveur sur lequel est hébergé notre application (API).

- **Sudo openssl req -new -nodes -out $MYCERT.csr -newkey rsa:4096 -keyout $MYCERT.key**

MYCERT=ApiNet6Certificate : le certificat pour notre serveur local.

Avec cette commande on va créer une demande de signature de certificat pour notre serveur en local.

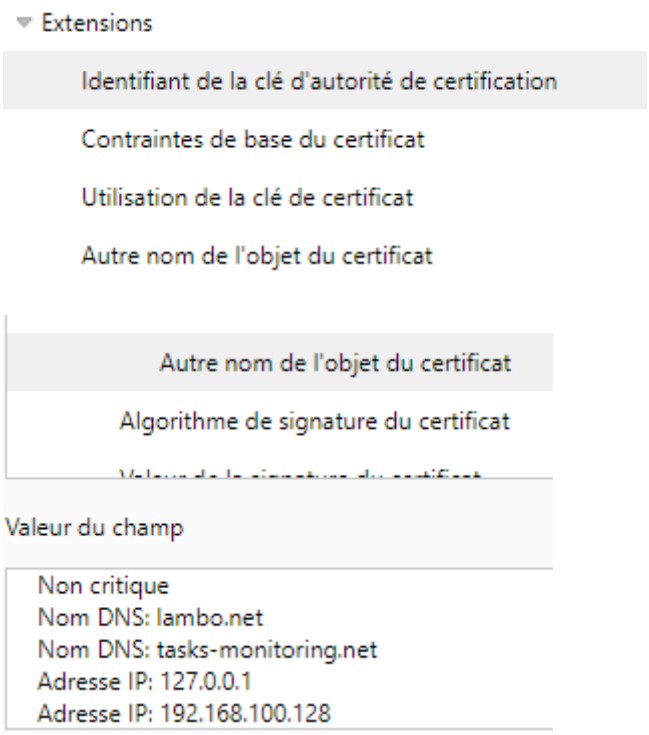
Rappel : on a pris soin de résoudre localement le nom de domaine de notre serveur 127.0.0.1 en **lambo.net** donc il s'agit bien de la demande de certificat pour **lambo.net**

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
subjectAltName = @alt_names
[alt_names]
DNS.1 = lambo.net
DNS.2 = tasks-monitoring.net
IP.1 = 127.0.0.1
IP.2 = 192.168.100.128
```

Dans un fichier qu'on va appeler **$**MYCERT.v3.ext on va définir les informations suivantes

Ce SAN représente les extensions pour le certificat

Exemple :



On précise les noms de domaines et adresses IP pour lesquels la demande de certificat est faite.

- Signature de la demande de certificat auprès de notre CA.

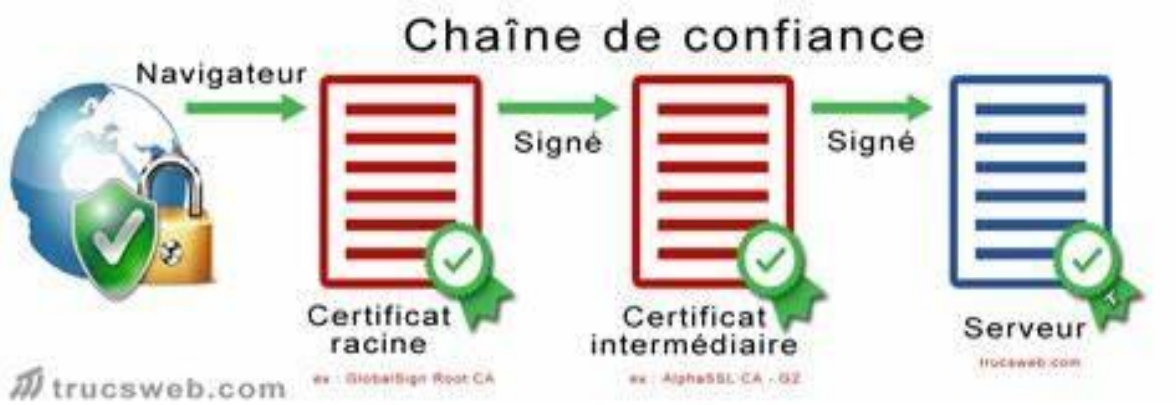
- sudo openssl x509 -req -in MYCERT.csr -CA CANAME.crt -CAkey CANAME.key -CAcreateserial -out MYCERT.crt -days 730 -sha256 -extfile $MYCERT.v3.ext
- sudo openssl pkcs12 -export -out MYCERT.pfx -inkey MYCERT.key -in MYCERT.crt

Avec ces deux commandes on a signé la demande de certificat de notre serveur local auprès du CA.

Ce qui a généré un fichier MYCERT.crt

On a extrait le fichier MYCERT.pfx du certificat à l'aide de sa clé privée et du certificat en lui-même

NB : c'est le fichier MYCERT.pfx qui est utilisé sur l'application (API) + le mdp de la clé privée.



II Configuration du serveur Kestrel

Kestrel ou crécerelle est le serveur web multiplateforme par défaut utilisé par ASP.NET il gère les connexions entrantes dans l'applications notamment les connexions en http et https.

Remarque : On a besoin de Kestrel comme serveur web pour lancer l'application (qu'elle soit dockerisée ou non).

afin de faire savoir qu'on souhaite utiliser un certificat https que l'on connait.

```
builder.Services.Configure<KestrelServerOptions>(options =>
{
   if (string.IsNullOrEmpty(certificateFile) || string.IsNullOrEmpty(certificatePassword))
   {
      throw new InvalidOperationException("Certificate path or password not configured");
   }
   options.ListenAnyIP(5195);
   options.ListenAnyIP(7251, listenOptions =>
   {
      listenOptions.UseHttps(certificateFile, certificatePassword);
   });
   options.Limits.MaxConcurrentConnections = 5;
   options.ConfigureHttpsDefaults(opt =>
   {
      opt.ClientCertificateMode = ClientCertificateMode.RequireCertificate;

   });
});
```

ClientCertificateMode : Permet de spécifier quel mode d'authentification le client certificate utilisera pour se connecter à kestrel

Il existe d'autres mode d'authentification pour le client certificate :

- RequireCertificate : le client certificat doit fournir un certificat valide avant de se connecter
- NoCertificate : Le client certificat n'a pas besoin de fournir un certificat
- AllowCertificate : Le certificat client sera demandé, cependant l'authentification n'échouera pas si jamais la demande de certificat n'est pas fournie.
- DelayCertificate : Le certificat client sera demandé plustard par l'application

Ps : Par défaut, pour le certificat https dans asp.net on n'a pas besoin qu'on configure kestrel

II. Multiple environnements (Dev, staging et prod)

Nous allons mettre en place les secrets utilisateurs pour l'environnement de développement.

Via User Secret Manager

- On initialise le projet :

```
<dotnet user-secrets init>
```

- On définit le secret :

```
< dotnet user-secrets set "TaskaManagement_API:ServicePass" <password> >
```

- Avec cette commande on aura dans le fichier de projet

```
<Nullable>enable</Nullable>
<UserSecretsId>e53b68bc-5d52-4258-9852-b800ed43886f</UserSecretsId>
```

- Le magasin de secrets se trouve dans

```
.microsoft/userSecrets/*
Et dans
<TasksManagement_API>.AssemblyInfo
```

Aussi nous allons nous intéresser à 03 fichiers principaux qui sont les fichiers dans lesquels on stocke les configurations globales de l'application en aspnet Core. Configurations telles que : les clés secrètes , la chaine de connexion à une base de données , les fichiers de logs et les niveaux de logs

Nous allons voir comment stocker de façon sécurisée ces configurations par environnement

A noter que : l'objectif est de devoir déployer l'application dans un conteneur docker donc notre fichier appsettings.Production.json sera pris en compte dans docker.

appsettings.json :

C’est le fichier principal des configurations d’une application en asp.net sa configuration est t’applicable sur l’ensemble de l’api et des environnements de Dev , de staging et de Prod .

Dans ce fichier on définit les niveaux de logs, le chemin pour les logs, les chaines de connexions à la bd, les paramètres pour les sécrets(clés d’api, clé secrète pour des token jwt, secret d’un certificat) .

A toute fin utile nous allons supprimer ce fichier afin d’utiliser nos propres fichiers de configuration par environnement.

Dans le fichier program.cs on va spécifier quel fichier utiliser par environnement

```
buil-
der.Configuration.AddJsonFile($"appsettings.{Environment.GetEnvironmentVariable("ASPNETCORE
_ENVIRONMENT") ?? "Development"}.json", optional: true, reloadOnChange: true);
```

```
buil-
der.Configuration.AddJsonFile($"appsettings.{Environment.GetEnvironmentVariable("ASPNETCORE
_ENVIRONMENT") ?? "Production"}.json",optional:true,reloadOnChange:true);
```

Par défaut c’est le fichier appsettings.Development.json

- **appsettings.Development.json : User Secret via Manage Secret Tools**

Uniquement pour l’environnement de Développement

- **appsettings.Preproduction.json : User Secret via les variables d’environnement**

  .

  .Environment-Dev-TasksManagement_API

  .Environment-Prod-TasksManagement_API

  .dockerignore

- **appsettings.Production.json : User Secret via Azure Key Vault**

NB : Ne jamais stocker les sécrets de production en claire dans le code source

On doit pouvoir utiliser les variables d’environnements ou Azure Key Vault pour le contrôle et la sécurisation des secrets.

- Pour la génération du Token JWT on va utiliser les variables d’environnements
- Pour la base de données on va utiliser Azure Key Vault
- Pour le serveur kestrel aussi on va utiliser les variables d’environnements

Dans l'environnement de production

Launchsettings.json

```
{
 "$schema": "https://json.schemastore.org/launchsettings.json",
 "iisSettings": {
  "windowsAuthentication": false,
  "anonymousAuthentication": true
 },
 "profiles": {
  "Run (Prod)": {
   "commandName": "Project",
   "applicationUrl": "https://192.168.153.131:7251;http://192.168.153.131:5195",
   "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Production"
   }
  },
  "IIS Express": {
   "commandName": "IISExpress",
   "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Production"
   }
  }
 }
}
```

Appsettings.json

```
{
 "AllowedHosts": "*",
 "Logging": {
  "LogLevel": {
   "Default": "Warning",
   "Microsoft": "Error",
   "Microsoft.Hosting.Lifetime": "Error"
  }
 },
 "ConnectionStrings": {
  "DefaultConnection": "Server=SRV-PROD;Database=TasksManagementDb;User
Id=sa;Password=password$1;"
 },
 "Caching": {
  "Enabled": true,
  "DurationMinutes": 60
 },
 "ErrorPages": {
  "ShowFriendlyErrors": true,
  "DiagnosticPagesEnabled": false
 },
```

```
 "JwtSettings": {
   "Issuer": "https://192.168.153.131:7250",
   "Audience": "https://192.168.153.131:7250",
   "JwtSecretKey": 64,
   "SecretPass" : "lambo"
 }
}
```

DataBaseContext -> DesignTimeDbContextFactory

```
   public class DesignTimeDbContextFactory : IDesignTimeDbContextFacto-
ry<DailyTasksMigrationsContext>
   {
      public DailyTasksMigrationsContext CreateDbContext(string[] args)
      {
         var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.Production.json")
            .Build();

         var optionsBuilder = new DbContextOptionsBuilder<DailyTasksMigrationsContext>();
         optionsBuilder.UseSqlServer(configuration.GetConnectionString("DefaultConnection"),
         sqlOptions => sqlOptions.EnableRetryOnFailure(
         maxRetryCount: 10,
         maxRetryDelay: TimeSpan.FromSeconds(40),
         errorNumbersToAdd: null));


         return new DailyTasksMigrationsContext(optionsBuilder.Options);
      }
   }
```

DataBaseContext ->DailyTasksMigrationsContext

```

public class DailyTasksMigrationsContext : DbContext
{
   public DailyTasksMigrationsContext(DbContextOptions<DailyTasksMigrationsContext> options)
      : base(options)
   {
   }

   public DbSet<Utilisateur> Utilisateurs { get; set; } = null!;
   public DbSet<Tache> Taches { get; set; } = null!;

   protected override void OnModelCreating(ModelBuilder modelBuilder)
   {
      modelBuilder.Entity<Utilisateur>(entity =>
       {
```

```
            entity.HasKey(u => u.ID);
            entity.Property(u => u.Nom).IsRequired();
            entity.Property(u => u.Email).IsRequired();
            entity.Property(u => u.Pass).IsRequired();
            entity.Property(u => u.Role).IsRequired();
        });

        modelBuilder.Entity<Tache>(entity =>
        {
            entity.HasKey(t => t.Matricule);
            entity.Property(t => t.Matricule).ValueGeneratedOnAdd();
            entity.Property(t => t.Titre).IsRequired();
            entity.Property(t => t.Summary);
            entity.Property(t => t.StartDateH).IsRequired()
            .HasColumnName("StartDateH")
            .HasColumnType("Datetime");
            entity.Property(t => t.EndDateH).IsRequired()
            .HasColumnName("EndDateH")
            .HasColumnType("Datetime");
        });


        base.OnModelCreating(modelBuilder);
    }
}
```

Et le dossier de Migrations

- InitialCreate
- **InitialCreate.Designer**
- DailyTasksMigrationsContextModelSnapshot

Dans l'environnement de production on a plus besoin d'indiquer le mot de passe et le Path du serveur Kestrel .

Tout est géré dans le Dockerfile sous forme de variable d'environnements

```
# Copier le certificat
COPY ApiNet6Certificate.pfx /https/certificate.pfx
ENV ASPNETCORE_Kestrel__Certificates__Default__Path=/https/certificate.pfx
ENV ASPNETCORE_Kestrel__Certificates__Default__Password=lambo
# Définissez la variable d'environnement
ENV ASPNETCORE_ENVIRONMENT=Production
```

Et aussi dans le fichier program.cs il faut :

Indiquer au serveur Kestrel de prendre en charge les variables d'environnements

```
var certificateFile=Environment.GetEnvironmentVariable("ASPNETCORE_Kestrel__Certificates__Default__Path");
var  certificatePassword=Environment.GetEnvironmentVariable("ASPNETCORE_Kestrel__Certificates__Default__Password");
builder.Services.Configure<KestrelServerOptions>(options =>
{
    if (string.IsNullOrEmpty(certificateFile) || string.IsNullOrEmpty(certificatePassword))
    {
        throw new InvalidOperationException("Certificate path or password not configured");
    }
    options.ListenAnyIP(5195);
    options.ListenAnyIP(7251, listenOptions =>
    {
        listenOptions.UseHttps(certificateFile, certificatePassword);
    });
    options.Limits.MaxConcurrentConnections = 5;
    options.ConfigureHttpsDefaults(opt =>
    {
        opt.ClientCertificateMode = ClientCertificateMode.RequireCertificate;

    });
});
```

Dans l'environnement de Développement

```
{
 "Logging": {
  "LogLevel": {
   "Default": "Debug",
   "Microsoft": "Information",
   "Microsoft.Hosting.Lifetime": "Information"
  }
 },
 "AllowedHosts": "*",
 "ConnectionStrings": {
  "DefaultConnection": "TasksManagement",
  "SecretApiKey": "lambo"
 },
 "JwtSettings": {
  "Issuer": "https://localhost:7082",
  "Audience": "https://localhost:7082",
  "JwtSecretKey": 64
 },
 "Kestrel": {
  "EndPoints": {
   "Https": {
    "Url": "https://localhost:7082",
    "Certificate": {
     "ClientCertificateMode":"AllowCertificate",
     "File": "/Certs/ApiNet6Certificate.pfx",
     "Password":"lambo",
     "CN":""
    }
   }
  }
 }
}
```

Et dans le program.cs de Dev

```
var kestrelSection=builder.Configuration.GetSection("Kestrel:EndPoints:Https");
var certificateFile = kestrelSection["Certificate:File"];
var certificatePassword = kestrelSection["Certificate:Password"];
builder.Services.Configure<KestrelServerOptions>(options =>
{
   if (string.IsNullOrEmpty(certificateFile) || string.IsNullOrEmpty(certificatePassword))
   {
      throw new InvalidOperationException("Certificate path or password not configured");
   }
   options.ListenAnyIP(7081, listenOptions =>
   {
      listenOptions.UseHttps(certificateFile, certificatePassword);
   });
   options.Limits.MaxConcurrentConnections = 5;
   options.ConfigureHttpsDefaults(opt =>
   {
      opt.ClientCertificateMode = ClientCertificateMode.RequireCertificate;

   });
});
```

launchSettings

```
{
 "$schema": "https://json.schemastore.org/launchsettings.json",
 "iisSettings": {
  "windowsAuthentication": false,
  "anonymousAuthentication": true,
  "iisExpress": {
   "applicationUrl": "http://localhost:17427",
   "sslPort": 44350
  }
 },
 "profiles": {
  "Run (Dev)": {
   "commandName": "Project",
   "dotnetRunMessages": true,
   "launchBrowser": true,
   "launchUrl": "",
   "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
   }
  },
  "IIS Express": {
   "commandName": "IISExpress",
   "launchBrowser": true,
   "launchUrl": "",
   "environmentVariables": {
```

```
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  }
 }
}
```

DataBaseContext

```
using Microsoft.EntityFrameworkCore;
namespace TasksManagement_API.Models;

public class DailyTasksMigrationsContext : DbContext
{
   public DailyTasksMigrationsContext(DbContextOptions<DailyTasksMigrationsContext> options)
      : base(options)
   {
   }

   public DbSet<Utilisateur> Utilisateurs { get; set; } = null!;
   public DbSet<Tache> Taches { get; set; } = null!;

   protected override void OnModelCreating(ModelBuilder modelBuilder)
   {
      modelBuilder.Entity<Tache>()
      .Property(t => t.TaksDate)
      .HasColumnName("DateH");
      base.OnModelCreating(modelBuilder);
   }
}
```

Comment j'ai fait pour arriver ici :

AccessToken

POST	/api/v1.0/AccessToken/Login	Permet de générer un token JWt pour l'utilisateur Admin en fonction de son adresse mail

TasksManagement

GET	/api/v1.0/TasksManagement/GetAllTasks	Affiche la liste de toutes les taches.
GET	/api/v1.0/TasksManagement/GetTaskByID/{Matricule}	Affiche les informations sur une tache précise.
POST	/api/v1.0/TasksManagement/CreateTask	Crée une tache.
DELETE	/api/v1.0/TasksManagement/DeleteTask/{Matricule}	Supprime une tache en fonction de son matricule.
PUT	/api/v1.0/TasksManagement/UpdateTask	Met à jour les informations d'une tache.

UsersManagement

GET	/api/v1.0/UsersManagement/GetAllUsers	Affiche la liste de tous les utilisateurs.
GET	/api/v1.0/UsersManagement/GetUserByID/{ID}	Affiche les informations sur un utilisateur en fonction de son ID.
POST	/api/v1.0/UsersManagement/CreateUser	Créée un utilisateur.
DELETE	/api/v1.0/UsersManagement/DeleteUser/{ID}	Supprime un utilisateur en fonction de son ID.
PATCH	/api/v1.0/UsersManagement/SetUserPassword	Met à jour le mot de passe d'un utilisateur en fonction de son nom

A) Installation des prérequis

Ces packages ci-dessous sont communs à tous les environnements.

```
<ItemGroup>
  <PackageReference Include="BCrypt.Net-Next" Version="4.0.3" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.0">
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    <PrivateAssets>all</PrivateAssets>
  </PackageReference>

  <PackageReference Include="Microsoft.Extensions.Logging" Version="6.0.0" />
  <PackageReference Include="Newtonsoft.Json" Version="13.0.3" />
  <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="6.0.0" />
</ItemGroup>
```

Ces packages ci-dessous sont utilisables uniquement en environnement de Developpement.

```
<ItemGroup Condition="'$(Configuration)' == 'Debug'">
  <PackageReference Include="Microsoft.EntityFrameworkCore.InMemory" Version="6.0.0" />
  <PackageReference Include="Moq" Version="4.16.1" />
  <PackageReference Include="xunit" Version="2.4.1" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.4.3" />
</ItemGroup>
```

Ces packages ci-dessous sont utilisables uniquement en environnement de Préproduction.

```
<ItemGroup Condition="'$(Configuration)' == 'Staging'">
  <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="6.0.0" />
</ItemGroup>
```

Ces packages ci-dessous sont utilisables à la fois en environnement de Préproduction et de Développement.

```
<ItemGroup Condition="'$(Configuration)' == 'Debug' Or '$(Configuration)' == 'Staging'">
  <PackageReference Include="Swashbuckle.AspNetCore" Version="6.0.0" />
  <PackageReference Include="Swashbuckle.AspNetCore.Annotations" Version="6.0.0" />
  <PackageReference Include="Microsoft.TestPlatform.TestHost" Version="17.7.0" />
</ItemGroup>
```

Ces packages ci-dessous sont utilisables uniquement en environnement de Production.

```
<ItemGroup Condition="'$(Configuration)' == 'Production'">
  <PackageReference Include="MySql.EntityFrameworkCore" Version="6.0.0" />
</ItemGroup>
```

Bonne pratique : Toujours commenter les packages installés dans le projet.

```
Include="Microsoft.EntityFrameworkCore.InMemory" Version="6.0.0" />
```

- Package EF pour la gestion des bd en mémoire

```
Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.0" />
```

- Package EF pour la gestion des bd SQL Server.

```
Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.20">
```

- Package EF pour les outils permettant de :
  - Générer un **DbContext** et les types d'entités à migrer dans une base de données.
  - Générer un script SQL pour une migration vers une base de données (équivalent au MPD dans MERISE?)
  - Créer une nouvelle migration d'entités dans un serveur de base de données

```
Include="Microsoft.AspNetCore.Rewrite" Version="2.0.3
```

- Package pour le service rewrite permettant la réécriture d'url

```
Include="Moq" Version="4.16.1" />
```

- Package Moq pour simuler les interfaces et les classes durant les tests unitaires

```
Include="xunit" Version="2.4.1" />
```

- Framework .NET pour les tests unitaires

```
Include="xunit.runner.visualstudio" Version="2.4.3">
```

- IHM permettant de visualiser les tests unitaires sur visual studio ou vscode

```
Include="Microsoft.TestPlatform.TestHost " Version=" 17.7.0" />
```

- Package nécessaire pour les tests d'intégration.

```
<GenerateDocumentationFile>true</GenerateDocumentationFile>
    <NoWarn>$(NoWarn);1591</NoWarn>
```

- Permet de générer automatiquement les commentaires sur les méthodes du contrôleur et les classes d'objets dans le swagger

```
Include="Microsoft.Extensions.Logging" Version="6.0.0" />
```

- Package pour gérer les logs et les niveaux de logs dans les fichiers de configurations et l'application.

```
Include="BCrypt.Net-Next" Version="4.0.3" />
```

- Package permettant de gérer l'encryptage de la propriété mot de passe.

```
Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="6.0.0" />
```

- Ce package permet de gérer l'authentification et l'autorisation du porteur **JWT.**

```
Include="coverlet.collector" Version="6.0.2">
<IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive
</IncludeAssets>
<PrivateAssets>all</PrivateAssets>
```

- Ce package permet de gérer la couverture de code pour le projet.

B) Gérer la description du swagger et génération du fichier xml

De base, le fichier <projetName>.*xml* n'existe pas.

On doit le générer. Il sera stocké dans /${HOME}/<projetName>/bin/Debug/net6.0/

Et rajouter dans le programme principal

```
var xmlFilename = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";

opt.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory, xmlFilename));
```

Dans program.**cs**

```
var builder = WebApplication.CreateBuilder(args);
var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(opt =>
{
    opt.SwaggerDoc("1.0", new OpenApiInfo
    {
        Title = "DailyTasks | Api",
        Description = "An ASP.NET Core Web API for managing Tasks App",
        Version = "1.0",
        Contact = new OpenApiContact
        {
            Name = "Artur Lambo",
            Email = "lamboartur94@gmail.com"
        }
    });
    opt.OperationFilter<RemoveParameterFilter>();
    var xmlFilename = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
    opt.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory, xmlFilename));
```

L'objectif ici c'est de générer un fichier **TasksManagement_API.xml** qui prend en charge la génération des commentaires apportés à notre api. Dans le fichier **TasksManagement_API *.csproj*** de la solution, rajouté ceci :

```
<GenerateDocumentationFile>true</GenerateDocumentationFile>
<NoWarn>$(NoWarn);1591</NoWarn>
```

<u>Conclusion</u> : Pour permettre la génération automatique des commentaires du swagger dans le fichier **TasksManagement_API.xml**

C) **Gérer la stratégie CORS** CORS (Cross Origin Ressources Sharing)

C'est une stratégie permettant de définir quel type de ressources autorisées sur un serveur via des requêtes http/s

Exemple :

Ces deux URL ont la même origine :

- https://example.com/foo.html
- https://example.com/bar.html

Ces URL ont des origines différentes des deux URL précédentes :

- https://example.net : Domaine différent
- https://www.example.com/foo.html : Sous-domaine différent
- http://example.com/foo.html : Schéma différent
- https://example.com:9000/foo.html : Port différent

Deux urls sont identiques si et seulement si elles ont :

- Le même type de protocole http ou https
- Le même hôte
- Le même port d'écoute

Dans le fichier **Program.cs** ce bout de code permet d'accepter la prise en charge des requêtes provenant des http et https. Si on ne le définit pas et bien seules les origines commençant par http + localhost + le port 5163 seront prises en compte.

Dans ce cas de figure, l'api acceptera toutes les IP qui l'appelleront ainsi que toutes les méthodes utilisées même les headers

```
builder.Services.AddCors(options =>
{
  options.AddPolicy(name: MyAllowSpecificOrigins,
            policy =>
            {
              policy.AllowAnyOrigin()
               .AllowAnyMethod()
               .AllowAnyHeader();
            });
});
```

```
//app.UseHttpsRedirection();
app.UseCors(MyAllowSpecificOrigins);
```

D) Gérer des interfaces

```
builder.Services.AddScoped<RemoveParametersInUrl>();
builder.Services.AddScoped<IRemoveParametersIn, RemoveParametersInUrl>();
builder.Services.AddScoped<IReadUsersMethods, UtilisateurService>();
builder.Services.AddScoped<IWriteUsersMethods, UtilisateurService>();
builder.Services.AddScoped<IReadTasksMethods, TacheService>();
builder.Services.AddScoped<IWriteTasksMethods, TacheService>();
builder.Services.AddTransient<IJwtTokenService, JwtBearerAuthentificationService>();
```

Il s'agit là d'une injection de dépendance à notre application. On associe les interfaces créées aux responsabilités (classes dépendantes des interfaces)

Durée de vie des conteneurs de DI

E) Gérer des Endpoints d'api

```
### 7.1- UsersManagementController

> - `/api/v1.0/UsersManagement/GetAllUsers`
> - `/api/v1.0/UsersManagement/GetSingleUser/{Nom}/{Role}`
> - `/api/v1.0/UsersManagement/CreateUser`
> - `/api/v1.0/UsersManagement/SetUserPassword/?{nom}&{mdp}`
> - `/api/v1.0/UsersManagement/Delete/{Nom}/{Role}`

### 7.2- TasksManagementController

> - `/api/v1.0/TasksManagement/GetAllTasks`
> - `/api/v1.0/TasksManagement/GetTaskByID/{Matricule}`
> - `/api/v1.0/TasksManagement/CreateTask`
> - `/api/v1.0/TasksManagement/DeleteTask/{Matricule}`
> - `/api/v1.0/TasksManagement/UpdateTask`
```

F) Gérer des politiques d'authentification

```
builder.Services.AddAuthorization(options =>
{
    // Politique d'autorisation pour les administrateurs
    options.AddPolicy("AdminPolicy", policy =>
        policy.RequireRole(nameof(Utilisateur.Privilege.Admin))
              .RequireAuthenticatedUser()
              .AddAuthenticationSchemes("JwtAuthorization"));


    // Politique d'autorisation pour les utilisateurs non-administrateurs
    options.AddPolicy("UserPolicy", policy =>
       policy.RequireRole(nameof(Utilisateur.Privilege.UserX))
              .RequireAuthenticatedUser()  // L'utilisateur doit être authentifié
              .AddAuthenticationSchemes("BasicAuthentication"));

    options.AddPolicy("CertPolicy", policy =>
       policy.RequireRole(nameof(Utilisateur.Privilege.Admin))
             .AuthenticationSchemes.Add("CertificateAuthentication"));

});
```

G) Organisation du code suivant le principe SOLID

On va appliquer le principe SOLID notamment les principes S et I

- ➔ **Single responsability** : chaque classe doit pouvoir implémenter une et une seule responsabilité.
- ➔ **Open Closed** : Une classe est ouverte à l'extensibilité mais fermée à la modification
- ➔ **Liskov substitution** :
- ➔ **Interfaces ségrégation** : Réorganisation des interfaces afin de séparer les méthodes qui n'ont aucun lien les unes des autres et de regrouper que celles implémentant la même responsabilité.
- ➔ **Dépendance inversion** : Eviter que le contrôleur dépende de l'implémentation d'un module de haut niveau définit dans un service mais plutôt de l'abstraction (une interface)

Exemple : On voit ici que dans le contrôleur UsersManagementController, pour la méthode GETALLUSERS

On implémente une méthode appelée GetUsers() qui provient de la classe UtilisateurService (qui dépend lui-même d'une abstraction appelée readMethods.

```
[HttpGet("GetAllUsers")]
  public async Task<ActionResult> GetUsers()
  {
    var users = await readMethods.GetUsers();
    if (users.Any()) { return Ok(users); }
    return NotFound();
  }
```

➔ Bonus : **Injection de dépendance :** C'est une technique qui permet de mettre en pratique le DIP

Exemple :

```
builder.Services.AddScoped<IReadUsersMethods, UtilisateurService>();
builder.Services.AddScoped<IWriteUsersMethods, UtilisateurService>();
builder.Services.AddScoped<IReadTasksMethods, TacheService>();
builder.Services.AddScoped<IWriteTasksMethods, TacheService>();
builder.Services.AddTransient<IJwtTokenService, JwtBearerAuthentificationService>();
```

On associe dans le programme principal l'abstraction au module de haut niveau

Créer des dossiers afin d'organiser le code suivant les principes S et I de SOLID

- ***Models*** : Dans lequel on va définir les classes et leurs propriétés
- ***DataBaseContext*** : On définit le contexte de base de données
- ***Interfaces*** : On va définir les contrats publics qui contient des méthodes de chaque interface permettant d'accéder à nos méthodes.
- ***ServicesRepositories*** : C'est un dossier dans lequel on définit les responsabilités de chaque classe créée dans le dossier Models.
- ***Controllers*** : C'est dans ce dossier qu'on va implémenter les méthodes nécessaires pour les requetes http/S (GET CREATE DELETE UPDATE).

PS : C'est dans le contrôleur qu'on va implémenter la logique faisant appel aux modules de hauts niveaux.

- ***Authentifications*** : Dans ce dossier on va créer les fichiers permettant de gérer les deux types d'authentifications à savoir basic authentification et bearer **JWT**. Aussi le fichier de génération de la clé sécrète.

A noter que : On pourra utiliser les GUID pour certaines propriétés dans nos classes

a) DataBaseContext

- DataBaseContext DailyTasksMigrationsContext

```
public class DailyTasksMigrationsContext : DbContext
{
    0 references
    public DailyTasksMigrationsContext(DbContextOptions<DailyTasksMigrationsContext> options)
        : base(options)
    {
    }

    8 references
    public DbSet<Utilisateur> Utilisateurs { get; set; } = null!;
    8 references
    public DbSet<Tache> Taches { get; set; } = null!;
```

On définit un contexte de base de données en spécifiant un **DataSet** pour la liste des utilisateurs et des taches.

Et dans program.cs

```
builder.Configuration.AddJsonFile($"appsettings.{Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT") ?? "Development"}.json",
optional: true, reloadOnChange: true
);
builder.Services.AddDbContext<DailyTasksMigrationsContext>(opt =>
{
    var item = builder.Configuration.GetSection("TasksManagement_API");
    var conStrings = item["DefaultConnection"];

    opt.UseInMemoryDatabase(conStrings);
});
```

b) ServicesRepositories

- ServicesRepositories UtilisateurService

```
public class UtilisateurRepository
{
    13 references
    private readonly DailyTasksMigrationsContext dataBaseMemoryContext;
    0 references
    public UtilisateurRepository(DailyTasksMigrationsContext dataBaseMemoryContext)
    {
        this.dataBaseMemoryContext = dataBaseMemoryContext;
    }
```

Comme dit chaque classe créée dans le Model, a une responsabilité dans le dossier Responsabilités

Dans cette classe on fait appel au contexte de BD

```
private readonly DailyTasksMigrationsContext dataBaseMemoryContext;
```

Cette instance va nous permettra de manipuler les dataSet contenu dans le contexte de BD et pouvoir implémenter les méthodes CRUD.

c) Interfaces

- Interfaces IUtilisateurRepository

On peut bien évidement créer pour chaque repository une I<class>Repository où on va définir les méthodes à implémenter dans le repository.

Dans notre cas on va plutôt créer des interfaces de lecture et d'écriture pour nos méthodes. Sachant que ces méthodes doivent être implémentées dans les repositories appropriés.

Tous deux sont des interfaces contenant des méthodes de lecture et d'écriture pour un utilisateur

Donc plus besoin de créer une interface IUtilisateurRepository

```
2 references
Task<string> GetToken(string email);
2 references
bool CheckUserSecret(string secretPass);
3 references
Task<List<Utilisateur>> GetUsers();
3 references
Task<Utilisateur> GetUserById(int id);
```

Plutôt avoir des interfaces de lecture et d'écriture (pour faire de la ségrégation d'interfaces)

```
8 references
public interface IWriteUsersMethods
{
    3 references
    string EncryptUserSecret(string plainText);
    1 reference
    string DecryptUserSecret(string cipherText);
    2 references
    Task<Utilisateur> CreateUser(Utilisateur utilisateur);
    2 references
    Task<Utilisateur> SetUserPassword(string nom, string mdp);
    2 references
    Task DeleteUserById(int id);
}
```

Dans le programme principal

```
builder.Services.AddScoped<IReadUsersMethods, UtilisateurService>();
builder.Services.AddScoped<IWriteUsersMethods, UtilisateurService>();
builder.Services.AddScoped<IReadTasksMethods, TacheService>();
builder.Services.AddScoped<IWriteTasksMethods, TacheService>();
builder.Services.AddTransient<IJwtTokenService, JwtBearerAuthentificationService>();
```

On applique là le principe I de SOLID la ségrégation d'interfaces on crée des bugs si jamais mal utilisé.

Le repository utilisateur va donc hériter de ces deux interfaces là et implémenter toutes les méthodes qui y sont définies

```
3 references
public class UtilisateurRepository:IReadUsersMethods,IWriteUsersMethods
{
    13 references
    private readonly DailyTasksMigrationsContext dataBaseMemoryContext;
    0 references
    public UtilisateurRepository(DailyTasksMigrationsContext dataBaseMemoryContext)
    3 references
    public async Task<List<Utilisateur>> GetUsers()
    4 references
    public async Task<Utilisateur> GetUserById(int id)

    2 references
    public async Task<Utilisateur> CreateUser(Utilisateur utilisateur)

    2 references
    public async Task DeleteUserById(int id)

    public async Task<Utilisateur> UpdateUser(Utilisateur utilisateur)
}
```

d) Controllers

- Controllers UsersManagementController

C'est dans ce dossier qu'on va créer les différents contrôleurs permettant d'exposer nos verbes http sur le swagger d'api.

On fait appel aux instances IReadUsersRepository et IWriteUsersRepository

```
public class UsersManagementController : ControllerBase
{
    1 reference
    private readonly IAuthentificationRepository authentification;
    6 references
    private readonly IReadUsersMethods readMethods;
    4 references
    private readonly IWriteUsersMethods writeMethods;
```

Transit DTO :

public class Utilisateur

public class DailyTasksMigrationsContext : DbContext

public class UtilisateurRepository:IReadUsersMethods,IWriteUsersMethods

public interface IWriteUsersMethods et public interface IReadUsersMethods

public class UsersManagementController : ControllerBase

On rajoute le middleware swagger en définissant les informations que l'on souhaite voir apparaitre sur la page swagger Titre, description, la version de swagger et le contact. En

Dans le fichier **program.cs** on devra activer swagger dans décris ci bas

Générer la persistance des données lors de la création d'une ressource dans la méthode POST.

Dans la classe de contexte de base de données, commencez par donner un nom au contexte de BD.

NB : dans la méthode POST il est interdit de passer les données par l'url.

Pb : j'ai écrit différentes méthodes GET POST etc … cependant les transactions entre ces différentes méthodes posent pb :

- Pour les bases de données en mémoire les transactions ne fonctionnent pas il faut SQLlite.car ces transactions sont gérer automatiquement.
- On n'a pas besoin du package *Add-Migrations InitialCreate* dans ce cas vue que la bd est en mémoire.

H) Les injections SQL et XSS

On n'a RIEN POUR LE MOMENT

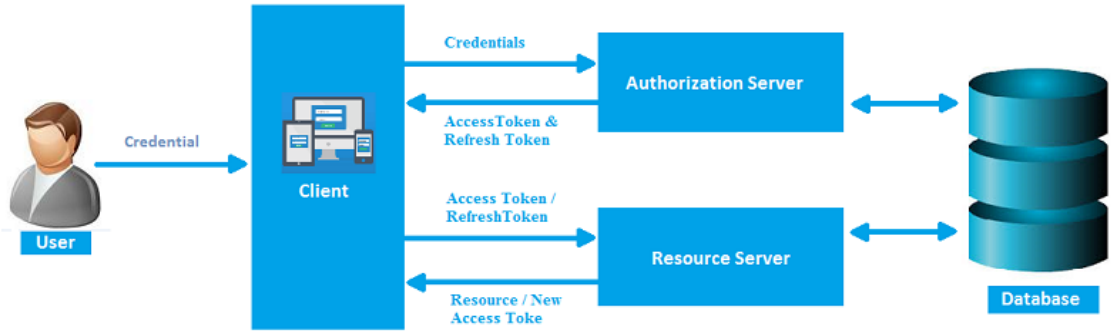
I Gestion de l'authentification sur l'api

Il en existe plusieurs types d'authentification à une Api tout dépend de ce qu'on veut faire.

Dans notre cas on veut :

- ➔ Une authentification de base pour un utilisateur lambda
- ➔ Une authentification Via token JWT pour un utilisateur administrateur
- ➔ Une authentification via un certificat pour gérer les transmissions https

Chaine de confiance est la suivante



➢ **Authorization Server**

C'est le serveur d'autorisation qui reçoit dans le header une paire id : password

- Réalise des traitements d'identification relatifs à ces credentials et renvoie à l'utilisateur un token et un refresh token de connexion
- Avec les credentials et le token ou le refresh token l'utilisateur peut accéder à la ressource

➢ **Ressource Server**

- Reçoit le token ou le refresh token
- Calcul le token reçu et établit la correspondance
- Libère la ressource dans la requête http ou libère un nouveau token

➢ **Token**

C'est l'identité numérique qui représente l'authentification et l'autorisation dans un système numérique.

- Sécuriser la communication entre un client et un serveur d'authentification
- Authentification (identité du demandeur de ressources)
- Autorisation (accès à aux ressources

Types de token : clé d'api, string, jeton JWT, jeton OAuth, cookie, etc.

- **Hashage de mot de passe**

Nb : Ne jamais stocker le mot de passe en claire dans la base de données toujours utiliser un algorithme de hashage.

a) Basic authentification

```
builder.Services.AddAuthentication("BasicAuthentication")
.AddScheme<AuthenticationSchemeOptions, AuthentificationBasic>("BasicAuthentication", options
=> {});
```

Dans le fichier program.cs on va rajouter un service pour l'authentification.

AuthenticationSchemeOptions

- Contiens des options qui vont nous permettre de créer des claims, des events et le principal

AuthentificationBasic

- C'est la classe dans laquelle on va implémenter la logique d'authentification basic

BasicAuthentication : C'est le nom du schéma d'authentification

Après avoir défini le schéma d'authentification, on doit pouvoir définir une stratégie d'autorisation en fonction du privilège utilisateur correspondant.

```
builder.Services.AddAuthorization(options =>
{
   options.AddPolicy("UserPolicy", policy =>
      policy.RequireRole(nameof(Utilisateur.Privilege.UserX))
          .RequireAuthenticatedUser()
          .AddAuthenticationSchemes("BasicAuthentication")) ;
```

- **UserPolicy** : C'est le nom de la stratégie d'authentification pour les utilisateurs non-administrateur.
- **policy.RequireRole** : Fonction permettant d'attribuée un rôle spécifique à un utilisateur.
- AddAuthenticationSchemes("BasicAuthentication"));});
  Cette fonction permet de faire la liaison entre notre politique d'authentification et le schéma d'authentification.

Implémentation de la classe AuthentificationBasic

```
protected override async Task<AuthenticateResult> HandleAuthenticateAsync()
```

Dans cette méthode protégée, on va définir la logique de code permettant de :

- Vérifier les informations d'un utilisateur au niveau du header de la requête http
- De créer le header du token de connexion
- Ajouter les revendications (claims)
- De créer la signature du ticket
- De vérifier les crédentials d'un utilisateur
- D'utiliser la classe AuthenticateResult pour retourner les différents types d'erreurs

```
var authHeader = AuthenticationHeaderValue.Parse(Request.Headers["Authorization"]);
if (authHeader.Scheme.Equals("Basic", StringComparison.OrdinalIgnoreCase))
{
    var credentialBytes = Convert.FromBase64String(authHeader.Parameter);//header du toke
    var credentials = Encoding.UTF8.GetString(credentialBytes).Split(':', 2);
    var username = credentials[0];
    var password = credentials[1];


    // Votre logique d'authentification ici
    if (await IsValidCredentials(username, password))
    {
        // Créer un ClaimsPrincipal avec le nom d'utilisateur playload du token
        var claims = new[]
        {
        new Claim(ClaimTypes.Name, username),
        new Claim(ClaimTypes.Role, Utilisateur.Privilege.UserX.ToString())
        };

        var identity = new ClaimsIdentity(claims, Scheme.Name);
        var principal = new ClaimsPrincipal(identity);

        // Assigner le principal à la propriété Principal de l'objet context
        var ticket = new AuthenticationTicket(principal, Scheme.Name);//signature header

        return AuthenticateResult.Success(ticket);
```

Cette méthode permet de renvoyer un **bool** avec login et mot de passe de l'utilisateur

```
private async Task<bool> IsValidCredentials(string username, string password)
{
    var utilisateur = dataBaseMemoryContext.Utilisateurs.FirstOrDefault(u => u.Nom == username);
    if (utilisateur != null)
    {
        return utilisateur.ChechHashPassword(password);
    }
    await Task.Delay(1000);
    return false;
```

NB : Gérer les requêtes https aussi

b) <u>Authentification via un porteur jwt</u>

C'est un moyen plus sécurisé qui permet d'avoir un accès sur les ressources de notre API tout en fournissant un token.

Pour atteindre cet idéal, nous allons définir le procédé dans 03 fichiers + ajout du package jwt

```
JwtBearerAuthentification : AuthenticationHandler<JwtBearerOptions>
```

Qui sera le serveur d'autorisation pour le token jwt lui-même qui hérite de la classe JwtBearerOptions.

```
JwtTokenService : IJwtTokenService
```

Qui sera le serveur d'authentification pour le client demandant une autorisation jwt elle-même hérite d'une interface.

```
public interface IJwtTokenService
{
    2 references
    string GetSigningKey();
    2 references
    string GenerateJwtToken(string email);
}
```

Interface qui contient 02 méthodes :

- GenerateJwtToken : Qui permet de générer le token jwt
- GetSigningKey : Pour récupérer la clé secrete pour le token jwt

Program :

Etapes à suivre :

- Définir une clé secrete, l'émetteur et l'audience dans le fichier de configuration *appsettings<env>.json*
- Créer un schéma d'authentification jwt bearer afin de valider les paramètres du token jwt.

```
builder.Services.AddAuthentication("JwtAuthentification")
    // On va ajouter le schéma d'authentification personnalisé JwtBearer avec les options par défaut
    .AddScheme<JwtBearerOptions, JwtBearerAuthentification>("JwtAuthentification", options =>
    {
        var JwtSettings = builder.Configuration.GetSection("JwtSettings");
        var secretKeyLength = int.Parse(JwtSettings["JwtSecretKey"]);
        var randomSecretKey = new RandomUserSecret();
        var signingKey = randomSecretKey.GenerateRandomKey(secretKeyLength);

        options.SaveToken = true;
        options.RequireHttpsMetadata = false;
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,              // Valider l'émetteur (issuer) du jeton
            ValidateAudience = true,            // Valider l'audience du jeton
            ValidateLifetime = true,            // Valider la durée de vie du jeton
            ValidateIssuerSigningKey = true,    // Valider la signature du jeton

            ValidIssuer = JwtSettings["Issuer"],        // Émetteur (issuer) valide
            ValidAudience = JwtSettings["Audience"],    // Audience valide
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(signingKey)) // Clé de signature
        };
    });
```

- Implémenter le serveur d'autorisation du porteur jwt

```
protected override Task<AuthenticateResult> HandleAuthenticateAsync()
{
    if (!Request.Headers.ContainsKey("Authorization"))
        return Task.FromResult(AuthenticateResult.Fail("Authorization header missing"));
```

Le serveur d'autorisation va vérifier si l'entête de la requête http contient le mot clé Autorisation.

```
var authHeader = AuthenticationHeaderValue.Parse(Request.Headers["Authorization"]);
if (!authHeader.Scheme.Equals("Bearer", StringComparison.OrdinalIgnoreCase))
{
    return Task.FromResult(AuthenticateResult.Fail("Schéma d'authentification invalide"));
}
```

S'il contient le mot clé recherché il vérifie que sa valeur c'est bien **Bearer.**

```
var authHeaderKey = authHeader.Parameter;
var tokenValidationParameters = Options.TokenValidationParameters;

if (tokenValidationParameters == null)
    return Task.FromResult(AuthenticateResult.Fail("Les paramètres du token de validation ne sont pas configurés"));
```

Il va récupérer les paramètres de validation de token dans le fichier program s'ils n'ont pas été défini il retourne une erreur.

```
var tokenHandler = new JwtSecurityTokenHandler();
SecurityToken securityToken;
var principal = tokenHandler.ValidateToken(authHeaderKey, tokenValidationParameters, out securityToken);
```

On crée un principal afin de valider les paramètres du token et dégager le token sécurisé.

```
// Créer un ticket d'authentification réussi avec le principal
var ticket = new AuthenticationTicket(principal, Scheme.Name);
return Task.FromResult(AuthenticateResult.Success(ticket));
```

Il crée un ticket contenant le principal et le nom du schéma et il renvoie le ticket.

En somme, le serveur d'autorisation va :

- ➔ Vérifier l'entête de la requête http et s'assurer qu'elle contient bien la paire **Authorization : Bearer**
- ➔ On récupère les paramètres de validation de token et on crée un principal
- ➔ On dégage le token sécurisé et on crée un ticket pour le principal

- Implémenter le serveur d'authentification du client demandant un token jwt

Il permet de vérifier l'existence d'un client http auprès de notre application. Si ce client existe on va pouvoir lui fournir un token de connexion lui permettant d'accéder aux ressources de notre application.

Il va récupérer la clé secrete dans le fichier configuration

```
public string GetSigningKey()
{
    var JwtSettings = configuration.GetSection("JwtSettings");
    int secretKeyLength = int.Parse(JwtSettings["JwtSecretKey"]);
    var randomSecretKey = new RandomUserSecret();
    string signingKey = randomSecretKey.GenerateRandomKey(secretKeyLength);
    return signingKey;
}
```

Il recherche l'utilisateur dont l'email correspond à l'email passé en paramètre et dont le rôle est admin.

```
SymmetricSecurityKey
```

Il va sécuriser cette clé secrete

```
var utilisateur = dataBaseMemoryContext.Utilisateurs
.Single(u => u.Email.ToUpper().Equals(email.ToUpper()) && u.Role.Equals(Utilisateur.Privilege.Admin));
if (utilisateur is null)
{
    throw new ArgumentException("Cet utilisateur n'existe pas");
}
var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(GetSigningKey()));
var tokenHandler = new JwtSecurityTokenHandler();
```

Il définit une liste de revendications pour le client qui souhaite avoir le token en précisant la date d'expiration du token , l'émetteur et l'audience pour ce token.

```
var tokenDescriptor = new SecurityTokenDescriptor
{
    Subject = new ClaimsIdentity(new[] {
        new Claim(ClaimTypes.Name, utilisateur.Nom),
        new Claim(ClaimTypes.Email, utilisateur.Email),
        new Claim(ClaimTypes.Role, utilisateur.Role.ToString())
        }
    ),
    Expires = DateTime.UtcNow.AddMinutes(50),
    SigningCredentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha512Signature),
    Audience = configuration.GetSection("JwtSettings")["Audience"],
    Issuer = configuration.GetSection("JwtSettings")["Issuer"],
};
```

Il va pouvoir générer le token si la clé secrète est bien celle attendue par le serveur d'autorisation.

```
var tokenCreation = tokenHandler.CreateToken(tokenDescriptor);
var token = tokenHandler.WriteToken(tokenCreation);
return token;
```

c) <u>Authentification via un certificat auto-signé</u>

II API Web ASP.NET de test et de débogage

C'est dans le **csproj** du projet principal qu'on rajoute les packages xunit, runner. visualstudio

a) Tests unitaires :

NB : On teste les actions results du Controller, et l'authentification.

b) Tests d'intégration : (On teste le fait que tous les composants fonctionnent bien ensemble dans l'application [base de données, API] on le fera uniquement pour le web app en html/JavaScripts/css
c) Tests de charges : threads

III Tests fonctionnelles de l'API

- Que se passe t'il si on essaye de générer un token jwt pour un utilisateur non-admin
- Pour un utilisateur qui n'existe pas

Fonctionnalités endogènes de l'api	Etat actuelle	Résultat après fixation
Génération d'un token pour user non-admin	Retourne un incident sans gestion de logs	Retourne résultat attendu avec gestion des logs
Génération d'un token pour un user inexistant	Retourne résultat attendu avec gestion des logs	RAS
Génération d'un token jwt pour un user admin mais dont le mot de passe secret user est erroné	Retourne le token sans gérer le mot de passe	Retourne bien le token en vérifiant que non seulement l'user est admin mais aussi que le mot de passe secret user est le bon
Peut-on modifier le mot de passe d'un utilisateur non-admin ?	Incohérence dans le code résultat attendu non satisfaisant	Retourne ce que l'on souhaite on peut bien changer le mot de passe pour un utilisateur non-admin
Deux utilisateurs peuvent-ils avoir la même adresse électronique ?	Oui c'est possible pour le moment	
Peut-on notifier à un utilisateur que son mot de passe a été changé	Non on voit le message dans les logs informations mais y'a pas de serveur SMTP	

IV Migration vers la base de données SQL Server

Pour ce faire, on a besoin de deux packages essentiels

```
Include="Microsoft.EntityFrameworkCore.SqlServer" Version="6.0.0" />
Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.20">
```

Nb : Toujours se rassurer que les versions de packages sont compatibles

Remarque 1 : C'est quand sur le serveur de base de données on utilise Windows authentification

```
"TasksManagement_API": {
  "DefaultConnection": "Ser-
ver=192.168.153.130;Database=TasksManagementDb;Trusted_Connection=True;",
```

Remarque 2 : C'est quand on utilise SQL authentification

```
"TasksManagement_API": {
  "DefaultConnection": "Server=192.168.153.130;Database=TasksManagementDb; User
Id=<username>;Password=<password>;",
```

TasksManagementDb : **Représente le nom de la base de données sur le serveur SQL.**

Remarque **3 :** Quand on essaye de créer un utilisateur pour notre base de données on reçoit le message suivant : "Nom d'objet 'Utilisateurs' non valide."

Ce pourquoi : Entity Framework Core n'arrive pas à créer le diagramme de base de données pour notre projet dans le DBMS.

On peut le voir dans les logs du conteneur docker que le contexte de base de données DailyTasksMigrationsContext n'est pas pris en compte.

```
   fail: Microsoft.EntityFrameworkCore.Database.Command[20102]
   Failed executing DbCommand (566ms) [Parameters=[], CommandType='Text', Command-
Timeout='30']
   SELECT [u].[ID], [u].[Email], [u].[Nom], [u].[Pass], [u].[Role]
   FROM [Utilisateurs] AS [u]
fail: Microsoft.EntityFrameworkCore.Query[10100 ]
   An exception occurred while iterating over the results of a query for context type 'TasksManage-
ment_API.Models.DailyTasksMigrationsContext'.
```

Solution : Utiliser la commande <dotnet ef migrations initialCreate > pour générer un diagramme de bd basé sur le contexte DailyTasksMigrationsContext

Contexte qui elle-même dépend du code.

C'est grâce à cette commande qu'on va établir et maintenir la synchronisation entre le code de l'application et la structure de base de données.

```
DesignTimeDbContextFactory
```

Nécessaire pour ajouter le contexte de bd à EF même lorsque l'application n'est pas en cours d'exécution.

Cette commande va :

- Créer dans le repertoire du projet à migrer un dossier **Migrations**

Dans le dossier **Migrations** on aura 03 fichiers crées **:**

- 20240806123745_InitialCreate.cs

  Description :

- 20240806123745_InitialCreate.Designer.cs

  Description :

- DailyTasksMigrationsContextModelSnapshot.cs

  Description :

Remarque : 20240806123745 c'est l'ID de la migration du contexte de base de données on pourrais donc avoir pour un même projet plusieurs migrations

V Grandes Finitions

a) Empêcher les scripts intersites (XSS)
b) Cacher le mot de passe sur le swagger mais pas dans l'url
c) Retirer un champ dans le swagger

Pour retirer un champ dans le swagger nous avons créé une classe RemoveParameterFilter qui hérite elle-même de l'interface IOperationFilter

```
public class RemoveParameterFilter : IOperationFilter
public void Apply(OpenApiOperation operation, OperationFilterContext context)
var parameters = operation.Parameters;

 var parameterToRemove = parameters.FirstOrDefault(p => p.Name == "identifiant");
        if (parameterToRemove != null)
        {
           parameters.Remove(parameterToRemove);
        }
```

Ceci dans l'optique de cacher l'identifiant d'un utilisateur lors de la création d'un utilisateur dans la méthode POST

d) Vérifier le format d'adresse électronique
e) Séparation du code Env-DEV et Env-Prod
f) Supprimer le code legacy
g)

Brainstorming les tests unitaires

On a reproduit le test illustré dans https://learn.microsoft.com/fr-fr/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2022

Brainstorming

On a décidé de créer un struct DateH à l'intérieur de la classe principale Tache :

EF n'arrive pas à faire la relation entre la classe Tache et DateH

Solution 1 : On pourrait créer un fichier appelé DateH.cs et implémenter notre struct en rajoutant le mot clé Partial pour dire qu'il s'agit d'une partie de la classe Tache

Solution 2 : Rajouter dans le contexte de bd la méthode **OnModelCreating** puis rajouter le mot clé HasColumnName en précisant le nom de la **struct**.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Tache>()
    .Property(t => t.TasksDate)
    .HasColumnName("DateH");
    base.OnModelCreating(modelBuilder);
}
```

```
"iisSettings": {
  "windowsAuthentication": false,
  "anonymousAuthentication": true,
```

Un utilisateur a une liste de taches on va changer le modèle de données et rajouter une collection pour la liste des taches d'un utilisateur

En gros il faut une relation entre les deux classes pour pouvoir générer un modèle de données qui prend en compte la liste des taches par utilisateur sans toutefois impacter sur le fait qu'on puisse créer un utilisateur sans ses taches.

Les fichiers concernés sont :

➔ Les classes utilisateur et tache

```
[Key]
    public int ID { get; set; }
    //public Guid UserId { get; set; } A revoir
    [Required]
    public string? Nom { get; set; }

    [Required]
#pragma warning disable CS8618 // Non-nullable field must contain a non-null
value when exiting constructor. Consider declaring as nullable.
    public string Email { get; set; }
#pragma warning restore CS8618 // Non-nullable field must contain a non-null
value when exiting constructor. Consider declaring as nullable.

    public enum Privilege { Administrateur, Utilisateur }
    [EnumDataType(typeof(Privilege))]
    [Required]
    public Privilege Role { get; set; }
```

```
    [Required]
    [Category("Security")]
    //[System.Text.Json.Serialization.JsonIgnore] // set à disable le mot de
passe dans la serialisation json
    public string? Pass { get; set; }
    public ICollection<Tache>? LesTaches { get; set; }
```

```
    [Key]
    public int? Matricule { get; set; }
    [Required]
    public string? Titre { get; set; }
    public string? Summary { get; set; }
    [Required(ErrorMessage = "Le format de date doit être comme l'exemple sui-
vant : 01/01/2024")]
    [DataType(DataType.Date)]
    public DateTime StartDateH { get; set; }

    [Required(ErrorMessage = "Le format de date doit être comme l'exemple sui-
vant : 01/01/2024")]
    [DataType(DataType.Date)]
    public DateTime EndDateH { get; set; }
    public int UserId { get; set; }
    public Utilisateur? utilisateur{ get; set; }
```

➔ Le contexte de base de données

```
        modelBuilder.Entity<Utilisateur>()
            .HasMany(u => u.LesTaches)
            .WithOne(t => t.utilisateur)
            .HasForeignKey(t => t.UserId);
```

On veut pourvoir :

- Ajouter une tache à un utilisateur déjà existant
- Supprimer une tache particulière à cet utilisateur
- Récupérer la liste des taches d'un utilisateur

```
 public class TacheUtilisateur
    {
        public int UserId { get; set; }
        public String? NomUtilisateur { get; set; }
        public String? EmailUtilisateur { get; set; }
        public ICollection<Tache>? LesTaches { get; set; }
    }
```

Cette classe a pour but de nous aider à gérer tout ce qui est DTO 5Data Transfert Object)

Pour une meilleure gestion du code mais en soi n'interfère pas avec le contexte de base de données.

NB : Avec EF le schéma de migration changera

Changer les listes en Collection

Critère	**ICollection<T>**	**List<T>**
Flexibilité	Permet d'utiliser différentes implémentations de collections	Restreint l'implémentation à une liste
Abstraction	Fournit une abstraction de base pour une collection	Implémentation concrète d'une liste
Compatibilité avec EF Core	Fortement recommandé pour les collections de navigation	Supporté mais moins flexible
Lazy Loading	Compatible avec Lazy Loading en EF	Supporté, mais moins flexible

Exemple de collections et leur implication

Collection	Duplicats autorisés ?	Ordre des éléments	Performance des opérations	Définition simple
List<T>	Oui	Conserve l'ordre	O(1) pour accès par index, O(n) pour recherche	Une liste ordonnée d'éléments où chaque élément peut être accédé par son index.
HashSet<T>	Non	Pas d'ordre	O(1) pour ajout, suppression et recherche	Un ensemble d'éléments uniques sans ordre particulier
Dictionary<TKey, TValue>	Non (sur les clés)	Pas d'ordre	O(1) pour accès via clé	Une collection de paires clé/valeur où chaque clé est unique, utilisée pour un accès rapide aux valeurs.
Queue<T> .	Oui	FIFO (Premier arrivé, premier sorti)	O(1) pour ajout/suppression	Une file d'attente où le premier élément ajouté est le premier à sortir
Stack<T>	Oui	LIFO (Dernier arrivé, premier sorti)	O(1) pour ajout/suppression	Une pile d'éléments où le dernier élément ajouté est le premier à sortir.
LinkedList<T>	Oui	Conserve l'ordre	O(1) pour insertion/suppression en début ou fin, O(n) pour accès	Une liste chaînée où chaque élément pointe vers le suivant, permettant des ajouts rapides en début ou fin.
ObservableCollection<T>	Oui	Conserve l'ordre	O(n) mais avec notifications de changement	Une collection qui notifie automatiquement les modifications (ajout, suppression) aux observateurs, souvent utilisée en UI.

Alpha zero : le code qui fonctionne et dont on peut pousser en production

Alpha one : du code révisé dont on peut analyser la performance et s'assurer de la qualité du code

Alpha two : du code dont peut évaluer la performance en fonction de l'environnement de production

Alpha three : Est-ce que le code est extensible

Fonction déléguée avec paramètre : On lui passe des paramètres optionnels

```
public async Task<List<Utilisateur>> GetUsers(Func<Utilisateur, bool>? filter
= null)
        {
var listUtilisateurs = await dataBaseSqlServerCon-
text.Utilisateurs.ToListAsync();

            // Si un filtre est fourni, appliquer ce filtre
            if (filter != null)
            {
                return listUtilisateurs.Where(filter).ToList();
            }
            // Retourner tous les utilisateurs si aucun filtre n'est donné
            return listUtilisateurs;
        }
```

Au lieu d'avoir ce bout de code

```
var user = await GetUsers(query => query.Where(u => u.Nom!.Equals(nom)));
var utilisateur = user.FirstOrDefault();
```

on pourrait le simplifier ainsi : Afin de moins utiliser les variables

```
var utilisateur = (await GetUsers(query => query.Where(u =>
u.Nom!.Equals(nom)))).FirstOrDefault();
```

```
public string Email { get; set; }=string.Empty;
```

Quand on ne souhaite pas initialiser la propriété dans le constructeur et on sait que cette valeur pourra être modifiée plustard

Nécessaire pour la gestion ou la centralisation des erreurs

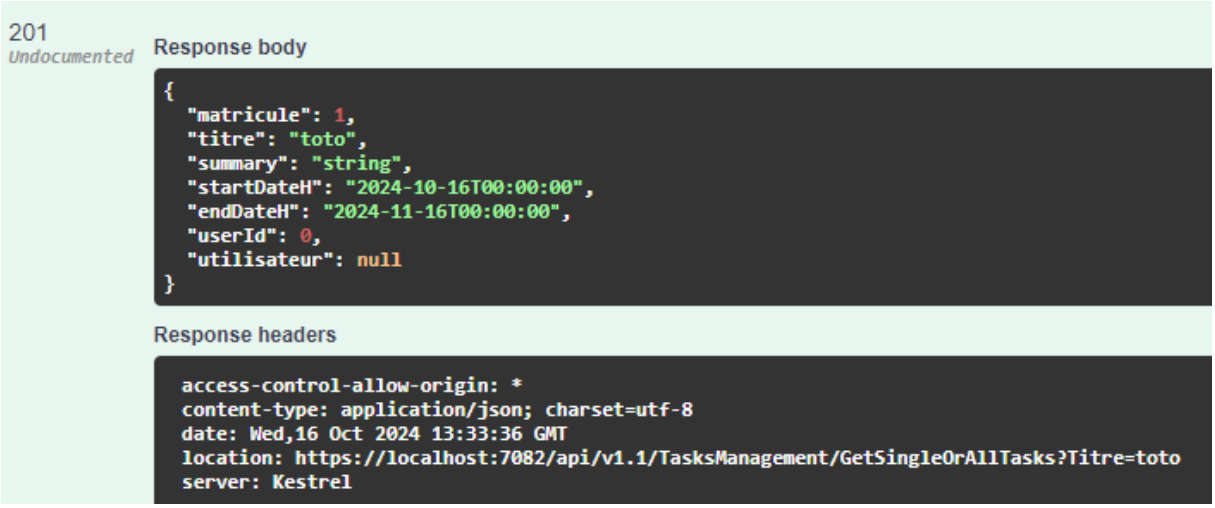
```
throw new (Exception)
```

Inconvénient on ne contrôle pas forcément le code d'erreur généralement utilisé dans un **Try Catch** afin de renvoyer les erreurs sur le server code status (500)

Gestion des codes de retours sur les controllers

Donc si je me limite aux 4 méthodes http je dois impérativement rajouter dans le code de retour pour la création d'une ressource

Une URI qui permet d'accéder facilement à cette dernière



201 :

```
await writeMethods.CreateTask(newTache);
return CreatedAtAction(nameof(GetSingleOrAllTasks), new { Titre = new-
Tache.Titre }, newTache);
```

200 :

204 : NoContent

Astuces : En REST, les URI doivent représenter des ressources, et non des actions. L'idée est que chaque URI identifie une ressource (par exemple, des tâches) plutôt que d'indiquer ce que l'on fait avec cette ressource (comme "obtenir" des tâches).

Donc en faisant un filtre sur la méthode GetSingleOrAllTask, on viole le principe SOLID (S) qui voudrait que chaque classe possède une et une seule responsabilité

On va garder cela pour les Taches mais pour les utilisateurs on va séparer les fonctions

- On va supprimer le controller dans les Endpoint

```
-   // NoContent(204) Vs OK(200) Vs NotFound(404)
-       /*
-           - 204: operation réussie mais pas de contenu à renvoyer
-           - 200 : opération réussie avec contenu à renvoyer
-           - 404 : opération échouée on arrive pas à retrouver la ressource
    demandée
-
-       */
```