

*@ Utopios Consulting*



# C#

**Création et Sécurisation d'API avec ASP.NET**

*@ Utopios Consulting*

# **Les APIs en ASP.NET**



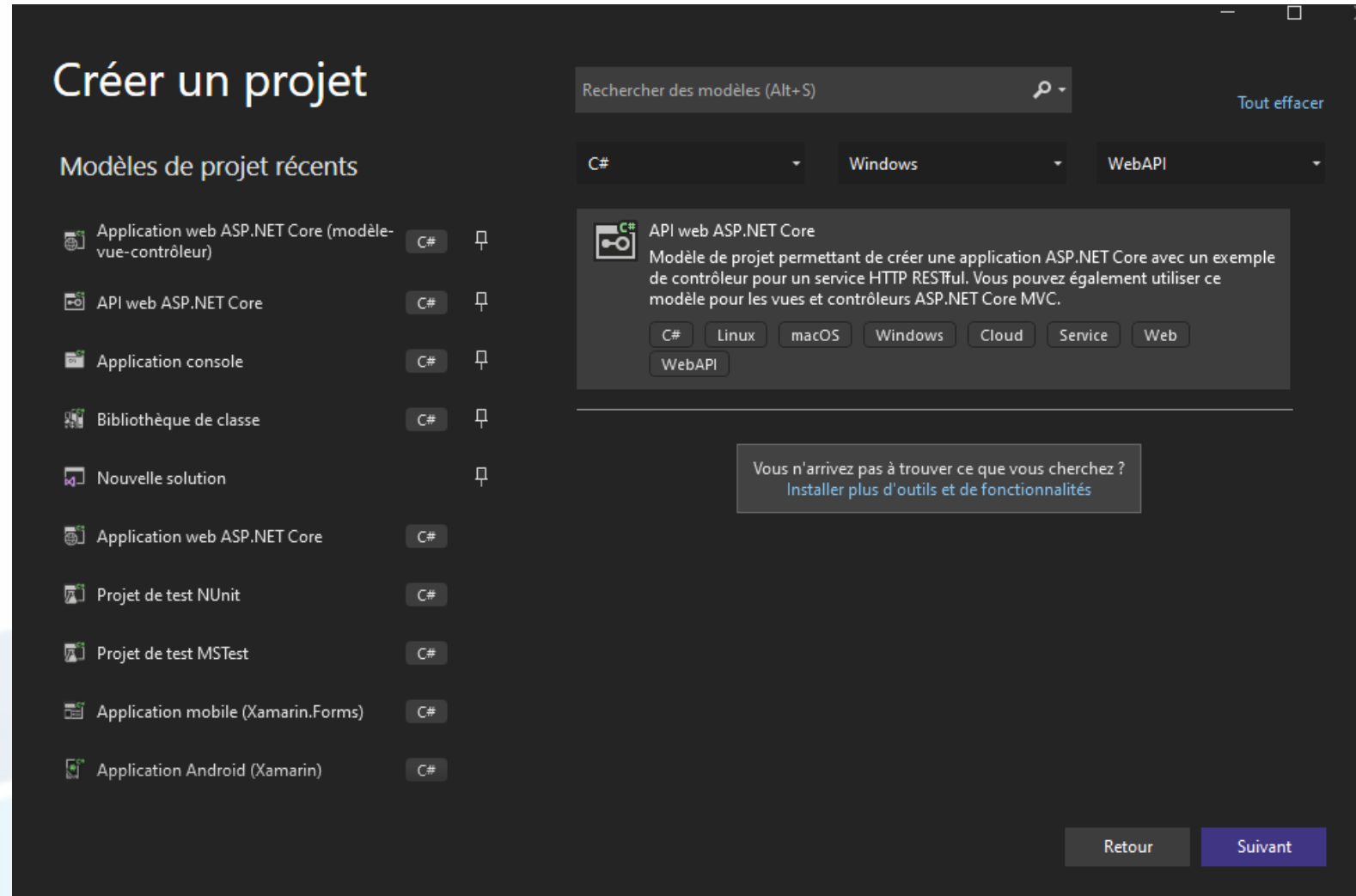
# Création de Projet d'API

Pour créer un projet d'API Web avec ASP.NET, il n'y a rien de plus simple. Il suffit de créer un nouveau projet dans Visual Studio et de sélectionner l'option API web ASP.NET Core.

Pour trouver cette option plus facilement, on peut utiliser les filtres:

- Langage : **C#**
- Plateforme : **Windows**
- Type : **WebAPI**

Une fois l'API choisie, on peut appuyer sur le bouton **Suivant**.

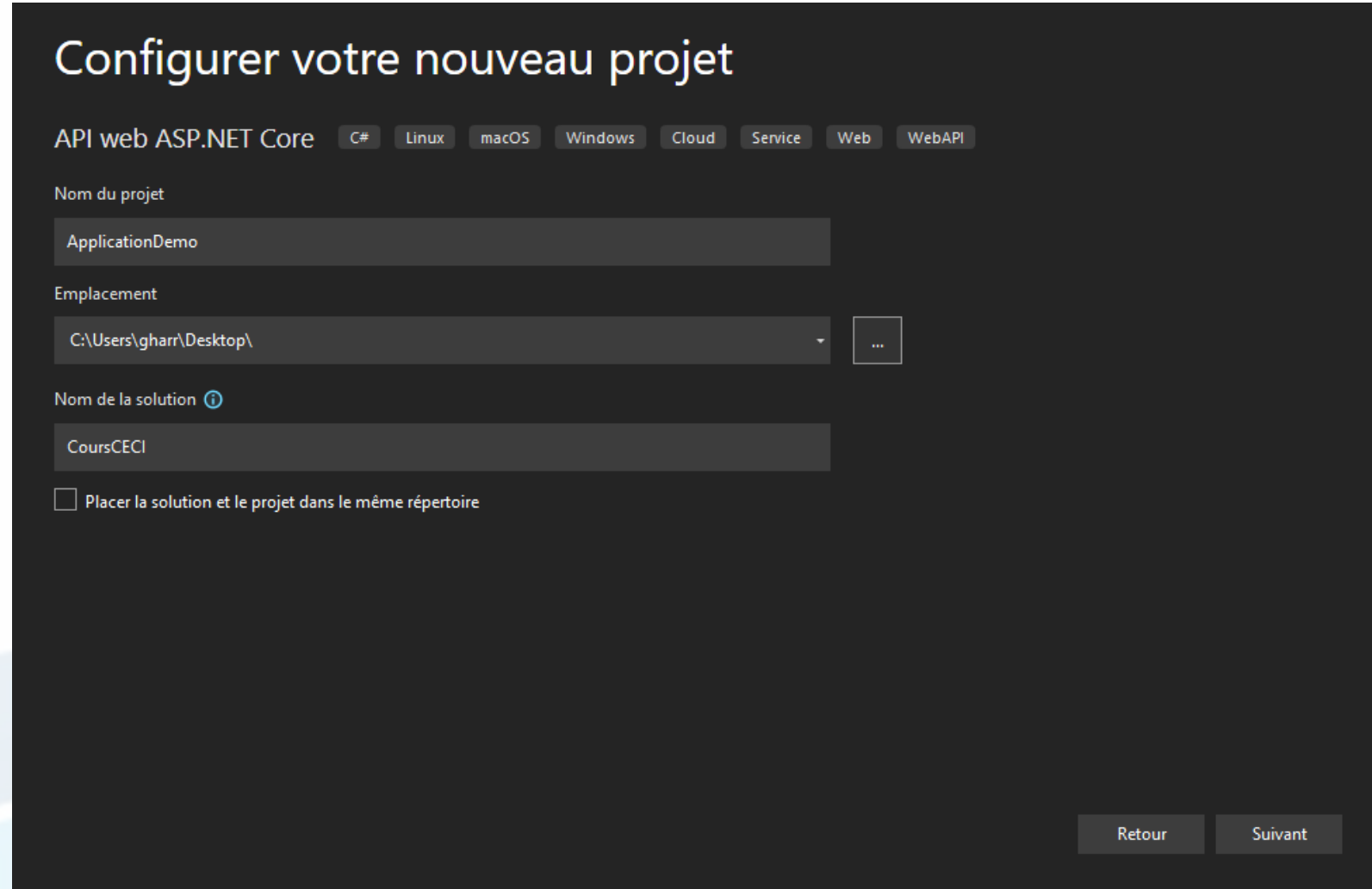


# Création de Projet d'API

Une fois fait, il nous est demandé de bien vouloir configurer notre projet d'API web.

Nous devons donc, comme pour chaque projet Visual Studio, choisir le **nom du projet**, l'**emplacement de la solution** et le **nom de la solution**.

Une fois fait, nous pouvons de nouveau appuyer sur le bouton **Suivant**.



The screenshot shows the 'Configurer votre nouveau projet' (Configure your new project) dialog box in Visual Studio. The title bar is dark gray. The main content area is white. At the top, there's a title 'Configurer votre nouveau projet'. Below it, there's a row of tabs: 'API web ASP.NET Core' (selected), 'C#', 'Linux', 'macOS', 'Windows', 'Cloud', 'Service', 'Web', and 'WebAPI'. The 'API web ASP.NET Core' tab is active. Below the tabs, there are three input fields: 'Nom du projet' (Project name) with the value 'ApplicationDemo', 'Emplacement' (Location) with the value 'C:\Users\gharr\Desktop\' and a folder selection button, and 'Nom de la solution' (Solution name) with the value 'CoursCECI'. There is a checkbox labeled 'Placer la solution et le projet dans le même répertoire' (Place the solution and the project in the same directory) which is currently unchecked. At the bottom right, there are two buttons: 'Retour' (Back) and 'Suivant' (Next).

Configurer votre nouveau projet

API web ASP.NET Core C# Linux macOS Windows Cloud Service Web WebAPI

Nom du projet

ApplicationDemo

Emplacement

C:\Users\gharr\Desktop\

Nom de la solution ⓘ

CoursCECI

☐ Placer la solution et le projet dans le même répertoire

Retour Suivant

# Création de Projet d'API

Enfin, avant de finaliser la création du projet, on nous demande de sélectionner:

- **La version de .NET** : 6.0
- Le type d'authentification : Aucune
- **Prise en charge HTTPS** : Oui
- Docker : Non
- **Modèle MVC** : Oui
- **OpenAPI** : Oui

Il nous est donc finalement possible de créer le projet en appuyant sur le bouton **Créer** qui lancera la création et la structuration du projet d'API web ASP.NET

## Informations supplémentaires

API web ASP.NET Core C# Linux macOS Windows Cloud Service Web WebAPI

Framework ⓘ  
.NET 6.0 (Prise en charge à long terme)

Type d'authentification ⓘ  
Aucun

☒ Configurer pour HTTPS ⓘ

☐ Activer Docker ⓘ

OS Docker ⓘ  
Windows

☒ Utiliser des contrôleurs (décocher pour utiliser un minimum d'API) ⓘ

☒ Activer la prise en charge d'OpenAPI ⓘ

Retour Créer

# Structure d'un projet d'API

Lorsque le projet est finalement créé, nous nous retrouvons avec une API de base fournie à titre d'exemple par Microsoft : **WeatherForecast**

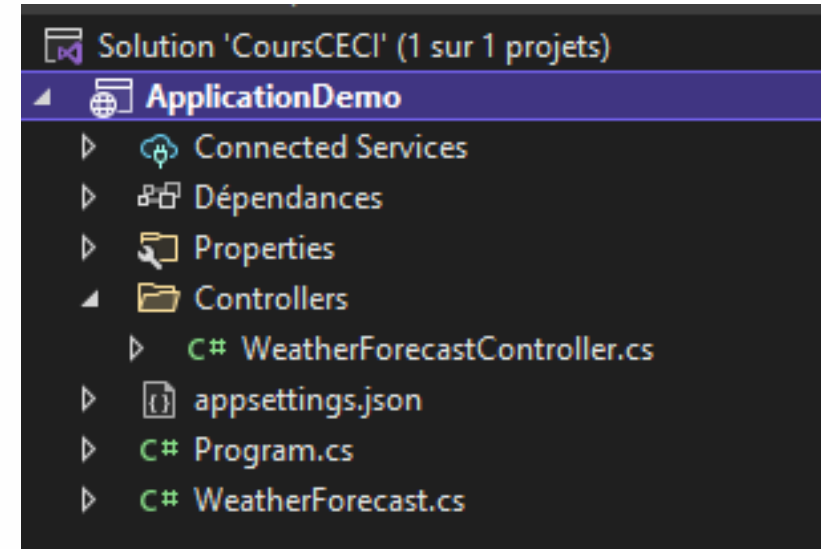
Cette API a pour fonction de renvoyer 5 prédictions météorologiques aléatoires générées à partir du modèle WeatherForecast (une classe C#) se trouvant dans la racine du projet.

La création d'une Web API n'utilisant pas le système de minimal API se base sur le modèle d'architecture **MVC**. Ce modèle d'archi utilise de son côté trois composants essentiels pour fonctionner et est utilisé fréquemment en corrélation avec ASP.NET dans le but de réaliser soit des applications web complexes, soit des APIs comme c'est le cas ici.

Le **Modèle** est l'élément servant de base représentative des objets. C'est une **classe** C# ou un **record** qui est généralement là pour servir d'intermédiaire en mémoire entre les éléments en base de données et l'application.

Le **Contrôleur** est en quelque sorte le centre logique des fonctionnalités de notre API. Il sert à l'API à **manipuler** les modèles à partir des **requêtes** utilisateurs et d'y répondre par des abstractions basées sur l'interface **IActionResult** (qui est la même que pour les applications web classiques MVC).

Les **Vues** ne seront pas utilisées ici car une API ne propose pas de vues à la manière d'un site web. Il est cependant intéressant de noter qu'on peut faire une application Web disposant à la fois d'une API et d'une version « à vues ».



# Le fichier Program.cs

Ce fichier sert de point de configuration de ce que l'on appelle le pipeline des middlewares d'ASP.NET.

En effet, ASP.NET se base de son côté sur une série de middlewares (interlogiciels) pour fonctionner. Ce sont ces middlewares qui vont récupérer la requête utilisateur, la traiter et l'envoyer à notre contrôleur avant de faire le chemin inverse pour envoyer une réponse depuis le contrôleur vers l'utilisateur.

Les middlewares sont une série de services qu'il nous est possible d'ajouter à la suite à notre application en fonction de nos besoins. De base, ils ne sont pas tous ajoutés dans le but de réduire la taille et d'améliorer les performances de notre application web

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```



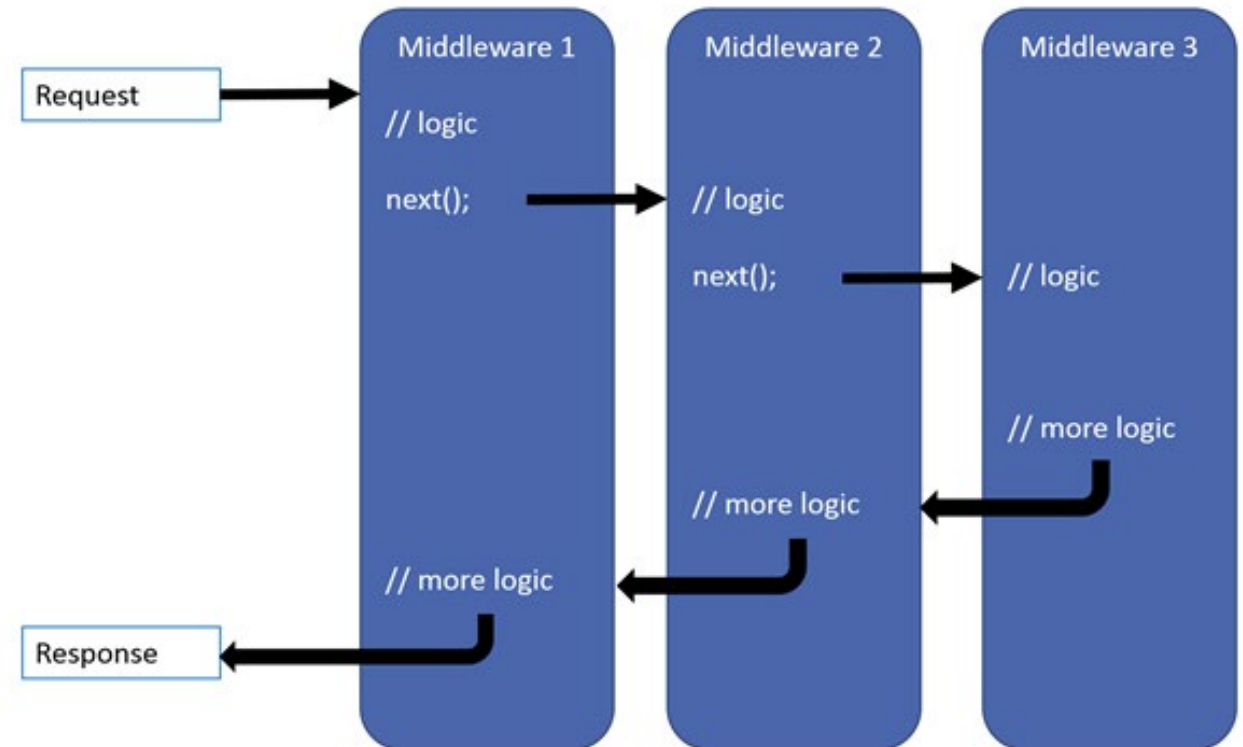
# Les Middlewares

Parmi les middlewares les plus commun, nous trouvons par exemple les deux middlewares **Authentification** et **Authorization**. Authentification permet de son côté de gérer l'utilisation de l'authentification par cookie ou JWT (JSON web token) dans notre application. Pour son homologue Authorization, qui se base sur l'Authentification pour fonctionner (il ne peut pas y avoir l'un sans l'autre, et il faut d'ailleurs toujours utiliser Authentification avant Authorization), il permet de spécifier les accès à tel ou tel utilisateur en se basant sur son rôle, son nom, son ancienneté ou tout autre paramètre que l'on jugerait essentiel au tri des accès.

L'autre middleware qui est fréquemment utilisé dans le cadre d'un projet d'API est le middleware gérant l'accès à notre API en se basant sur la politique des **CORS**.

Enfin, il nous faudra également par la suite utiliser et paramétrer le middleware **UseDbContext()** dans le but de paramétrer l'accès à notre base de donnée via **Entity Framework**.

Les middlewares ont de leur côté des **cycles de vie**, qui définissent le temps qu'ils restent actifs et la durée de leur instantiation lors des requêtes utilisateur à l'API.



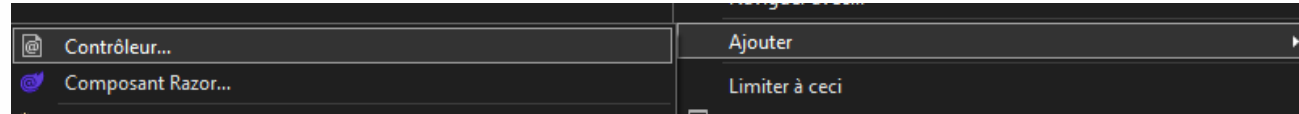


# Les Contrôleurs

Utopios Consulting

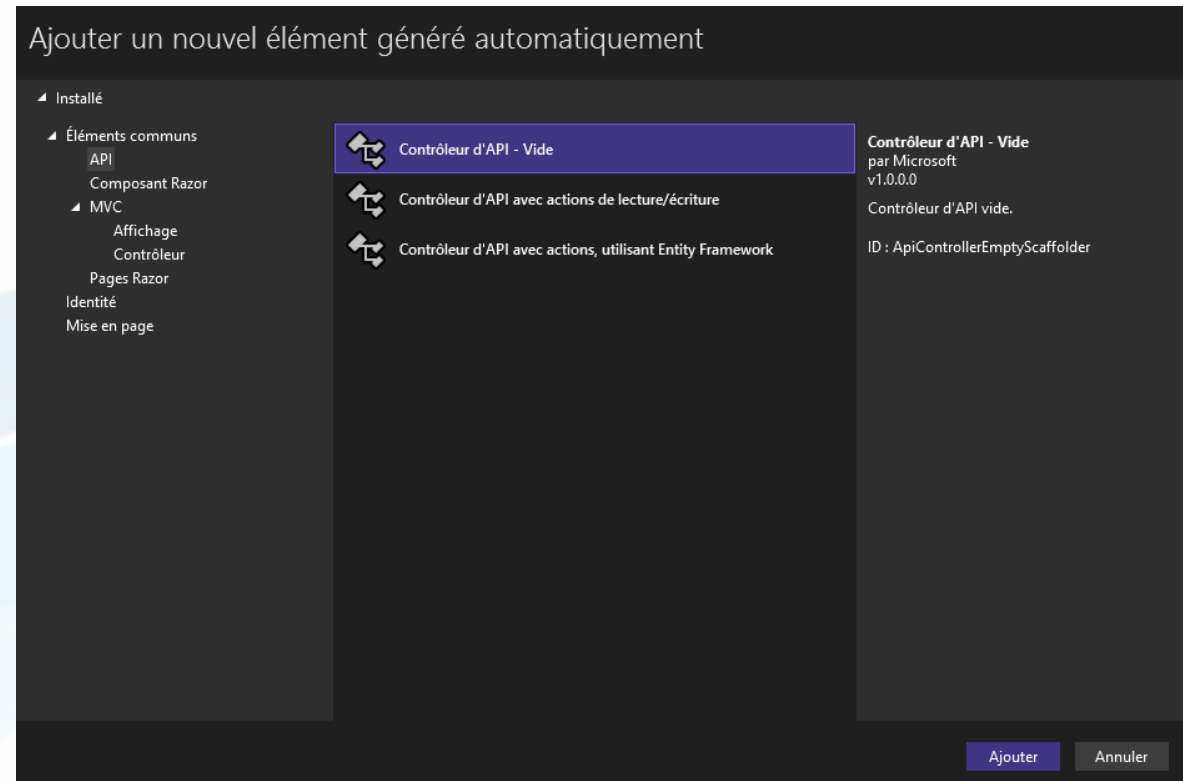
Lorsque l'on réalise un projet d'API avec ASP.NET Core, nous avons souvent besoin d'utiliser des contrôleurs. Ces contrôleurs sont le centre logique du fonctionnement de notre API, et ce sont eux qui réceptionnent les requêtes utilisateurs avant de renvoyer des résultats (la plupart du temps sous la forme d'un JSON).

Lorsque l'on veut créer un contrôleur, il suffit de faire un clic droit sur le dossiers des contrôleurs de notre application et de demander **Ajouter > Contrôleur...** :



Une fois ceci fait, nous nous retrouvons dans l'écran de création d'un contrôleur :

Il est alors possible de créer un contrôleur possédant déjà une série d'instructions en se basant sur la réalisation des verbes communs d'une API REST, ou alors en se basant sur EntityFramework. Cependant, dans un soucis d'apprentissage, nous allons réaliser des contrôleurs vides.



# Les verbes de Requêtes

Une fois notre contrôleur créé, on se retrouve avec une classe qui hérite alors de **ControllerBase**.

Cette classe servira bien souvent pour l'injection de dépendance, notamment pour y injecter notre DataContext utilisé dans le cadre d'une application utilisant **EntityFramework**, ou pour y injecter un mock de base de donnée injecté de façon Singleton pour conserver les datas.

On peut également voir que notre classe possède des séries de méthodes annotées des verbes d'une API Rest tels que :

- **HttpDelete** pour supprimer des données
- **HttpGet** pour permettre l'accès aux données et leur envoi à l'utilisateur
- **HttpPost** pour ajouter des données
- **HttpPut** pour éditer les données ou les ajouter si non existantes
- **HttpPatch** pour éditer simplement les données

Ces annotations peuvent également avoir entre parenthèse le routing de l'endpoint tel que :

```
[Route("api/[controller]")]
[ApiController]
1 référence
public class NewController : ControllerBase
{
    private readonly ApplicationDbContext context;

    0 références
    public NewController(ApplicationDbContext context)
    {
        this.context = context;
    }

    [HttpGet]
    0 références
    public IActionResult GetAll()
    {
        var dogList = context.Dogs.ToList();

        return Ok(dogList);
    }
}
```

```
[HttpPatch("dog/{id}")]
0 références
```

# Les paramètres de Requêtes

Dans le but de manipuler une certaine partie de notre application, d'accéder à telle ou telle valeur, il faut avoir recours à des paramètres. Pour les utiliser via ASP.NET, il n'y a rien de plus simple, il suffit de simplement les ajouter dans les paramètres de la méthode de notre contrôleur servant à retourner une réponse :

Lorsque l'on se sert d'un routing, il faut faire attention à ce que le nom du paramètre saisi entre crochets soit le même que celui que l'on donne à la méthode pour que l'association se fasse.

Il est possible de précéder nos paramètres de différentes annotations, telles que **[Bind]**, **[FromBody]** ou **[FromForm]** :

- **Bind("propA, propB")** va permettre de le lier qu'une partie des propriétés de l'objet récupéré, les autres se voyant attribuer leurs valeurs par défaut
- **FromBody** sert à récupérer les informations à partir du corps de la requête sous la forme d'un JSON. C'est la valeur par défaut de récupération des objets complexes
- **FromForm** va récupérer les informations à partir d'un corps de type form-data (formulaire). C'est la valeur par défaut de récupération des variables primitives tel qu'un Integer

```
[HttpPatch("dog/{id}")]
0 références
public IActionResult EditDog(int id, Dog newValues)
{
    var found = context.Dogs.FirstOrDefault(dog => dog.Id == id);
    if (found is not null)
    {
        found.Name = newValues.Name;
        found.Breed = newValues.Breed;

        if (context.SaveChanges() > 0)
        {
            return Ok(new
            {
                Message = "Chien édité avec succès",
                Dog = found
            });
        }
    }

    return NotFound(new
    {
        Message = "Il n'y a pas de chien avec cet ID !"
    });
}
```

# L'Injection de Dépendance

Au niveau de nos contrôleurs, nous pouvons avoir besoin de services pour faire fonctionner notre API. Un service peut simplement représenter une fonctionnalité comme le cryptage d'un mot de passe, l'upload d'un fichier ou encore une connexion en base de données. Pour réaliser cette association et utiliser ce service dans un contrôleur, nous allons avoir recours à l'injection de dépendance pour améliorer la scalabilité et la maintenabilité de notre application. En effet, l'injection de dépendance assure via l'utilisation des interfaces la possibilité de modifier aisément et à moindre coût les classes de services sans avoir à retoucher à nos contrôleurs, ce qui aide dans le cas de gros projets ou du travail en équipe.

Pour réaliser une injection de dépendance, il n'y a rien de plus simple. Il suffit de passer le service que l'on veut utiliser dans le constructeur de notre classe de contrôleur comme ci-dessous :

```
private readonly IUploadService uploadService;

0 références
public ValuesController(IUploadService uploadService)
{
    this.uploadService = uploadService;
}
```

Une fois l'injection de dépendance effectuée, on peut passer les déléguer les fonctionnalités souhaitées de notre contrôleur au service prévu à cet effet, permettant ainsi de se concentrer sur le contrôleur et non ces fonctionnalités.

# L'Injection de Dépendance

Les services sont généralement utilisés par les contrôleurs de notre application via ce que l'on appelle l'**injection de dépendance**. Malheureusement pour nous, nous n'avons pas de contrôle réel de l'instanciation des services car il s'agit de l'**orchestrateur d'ASP.NET** qui s'en charge (**inversion de contrôle**). Cependant, nous pouvons les injecter via les trois méthodes ci-dessous :

```
builder.Services.AddTransient<UploadService>();  
builder.Services.AddTransient<IUploadService, UploadService>();  
  
builder.Services.AddScoped<UploadService>();  
builder.Services.AddScoped<IUploadService, UploadService>();  
  
builder.Services.AddSingleton<UploadService>();  
builder.Services.AddSingleton<IUploadService, UploadService>();
```

Ces méthodes génériques peuvent soit avoir en paramètre de type une classe simple (qui sera instanciée lorsqu'un contrôleur y fera appel), soit une interface ainsi qu'une classe. Dans ce second cas, lorsqu'un contrôleur fera appel à cette interface, on injectera automatiquement la classe que l'on aura spécifiée en second paramètre. Les parenthèse servent à modifier les constructeurs lors de l'instanciation.

Nous pouvons ici voir que nous avons à notre disposition trois cycles de vie distincts que nous allons explorer : **Transient**, **Scoped** et **Singleton**.



# Cycles de vie des Services

Les trois cycles de vie disponibles lors de l'ajout des services au pipeline des middlewares sont utilisés en fonction de notre besoin par rapport aux dits services. Il est important de connaître les spécificités de l'injection de dépendance car en cas de mauvaise utilisation, nous pourrions avoir le mauvais service qui serait retourné par le conteneur de dépendance lors de la construction d'un contrôleur suite à une requête utilisateur (résultant par exemple en l'utilisation d'une nouvelle instance de base de données ou d'un type conteneur alors qu'il nous aurait fallu une persistance des données).

Le cycle de vie **Transient** va recréer le service à chaque fois que l'on pourrait en avoir besoin.

Le cycle de vie **Scope** va quant à lui refaire le service à chaque scope (une requête constitue un scope).

Le cycle de vie **Singleton** créera le service une fois durant tout le cycle de vie de l'application, et se servira de celui-ci à chaque fois.

Si l'on prend l'exemple d'un service servant à retourner simplement une Guid, on va rapidement se rendre compte de la différence lorsque l'on fera des appels API. En effet, là où la GUID changera à chaque rafraichissement voir à chaque nouvelle instantiation du service lors du contrôleur si l'on se sert respectivement des cycles de vie **Scoped** ou **Transient**, on va avoir une GUID identique à la précédente si l'on se sert d'un cycle de vie de type **Singleton**.

Pour savoir quel type de cycle de vie utiliser, il faut en général faire attention aux fonctionnalités qu'ils prennent en compte. Si l'on a pas besoin de conserver des états, on se sert généralement des services de type Transient. Si par contre on a besoin de ne pas utiliser trop de mémoire ou de conserver des informations, on peut utiliser des services **Scoped** voire **Singleton**.

# Exercice – API de Contacts

Vous devrez réaliser une API servant à un utilisateur d'accéder à un répertoire de contacts. La personne devra pouvoir opérer un CRUD de base (Ajout, Lecture, Edition et Suppression de contacts depuis l'API). Pour ce faire, vous devrez utiliser un contrôleur permettant de réaliser ce CRUD (qui héritera de **BaseController**).

L'API devra communiquer avec l'utilisateur via des JSON et être compatible avec Postman (La communication se fera alors au format **raw** de type **application/json**). La base de données sera un **Mock** injecté dans le contrôleur pour lui en permettre l'accès (pensez à vérifier votre cycle de vie de service lors de l'injection de dépendance), et les méthodes du CRUD se feront via ce service qui servira au contrôleur de faux contexte de données.

Vous devrez donc à terme avoir un **ContactsController** qui gèrera toutes les fonctionnalités énoncées à partir d'une **FakeDb** possédant la **collection** des contacts. Les contacts auront comme champs leur **nom**, leur **prénom**, leur **nom complet**, leur **date de naissance**, leur **âge**, leur **sexe** et leur **avatar** sous la forme d'une URL.



*@ Utopios Consulting*

# **Sécuriser une API**

# L'Authentification

geoparcos Consulting

Le middleware d'Authentification va être utilisé par l'API pour récupérer et/ou traiter des cookies ou des JWT. Dans le cadre de nos applications, nous allons nous servir de JWT pour les sécuriser. La différence majeure entre un JWT et un cookie est qu'un cookie ne peut transiter entre deux serveurs, et il n'est donc pas souvent utilisé dans le cadre d'une API qui pourrait être amenée à gérer une authentification de type OAuth2 ou à faire des appels sécurisés à d'autres APIs externes.

Pour faire appel à l'interlogiciel d'Authentification, il faut ajouter le service dans les services du fichier **Program.cs** via **UseAuthentication()**

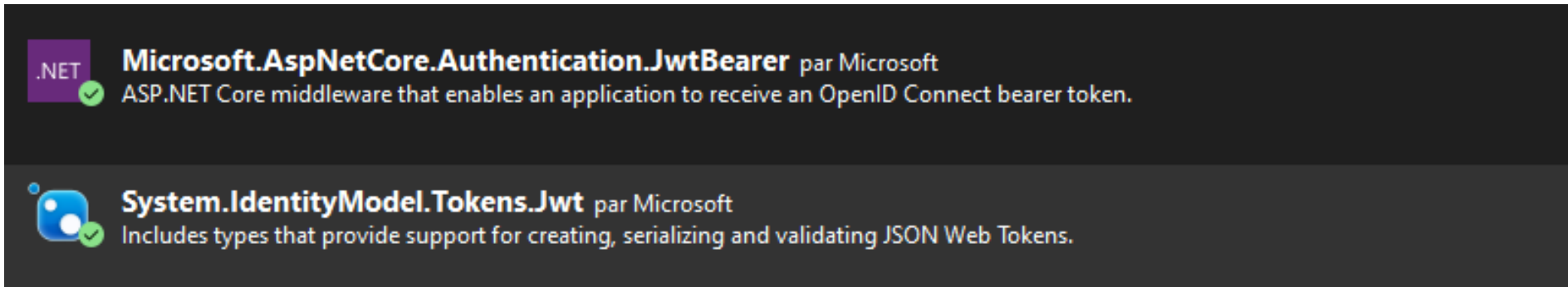
Une Authentification est gérée dans ASP .NET Core par ce que l'on appelle un gestionnaire d'authentification. Ce gestionnaire sert à confirmer ou non l'identité d'un utilisateur, de même que pour lui renvoyer (dans le cadre d'un site web) vers des pages de login ou d'erreur de type 5xx.

Pour fonctionner, le gestionnaire d'identification va se servir de schémas que l'on peut ou non créer nous-même de façon personnalisées. Cependant, on a plus fréquemment recours aux schéma de type Cookie ou JWT via l'utilisation des méthodes **AddJwtBearer()** ou **AddCookie()** à la suite de l'ajout du service via **AddAuthentication()**.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme,
        options => builder.Configuration.Bind("JwtSettings", options))
    .AddCookie(CookieAuthenticationDefaults.AuthenticationScheme,
        options => builder.Configuration.Bind("CookieSettings", options));
```

# Paramétrage de l'Authentification

Pour paramétrer l'Authentification, cela se fait comme d'habitude au niveau de l'inscription des services dans le fichier **Program.cs** Pour avoir accès à l'utilisation et le paramétrage du JWT, il faut avant tout ajouter ces deux packages NuGet :



Puis il faut modifier le fichier de configuration de la sorte :

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultSignInScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(builder.Configuration["JWT:SecretKey"])),
        ValidateLifetime = true,
        ValidateAudience = true,
        ValidAudience = builder.Configuration["JWT:ValidAudience"],
        ValidateIssuer = true,
        ValidIssuer = builder.Configuration["JWT:ValidIssuer"],
        ClockSkew = TimeSpan.Zero
    };
});
```

# Les options de l'Authentification

La première des deux méthodes chaînées est là pour permettre à ASP.NET d'apprécier le schéma d'Authentification du JWT. Ces options sont ici laissées par défaut car ce ne sont pas elles qui nous intéressent ici. La seconde méthode **AddJwtBearer** est celle que laquelle nous allons nous étendre. On remarque qu'elle a besoin d'un objet de type `TokenValidationParameters` que l'on peut instancier et paramétrer nous même.

- **ValidateIssuerSigningKey** sert à demander la vérification d'une clé de signature
- **IssuerSigningKey** permet de configurer la fameuse clé de signature
- **ValidateLifetime** sert à vérifier les dates d'expirations du token
- **ValidateAudience** permet de configurer la vérification d'une audience (les personnes ayant droit de se servir du JWT)
- **ValidAudience** configure les personnes ayant droit d'utilisation du token
- **ValidateIssuer** permet de configurer la vérification du service générant le token
- **ValidIssuer** paramètre le service devant avoir généré le token
- **ClockSkew** sert à paramétrer l'écart type des date et heures d'expiration du token

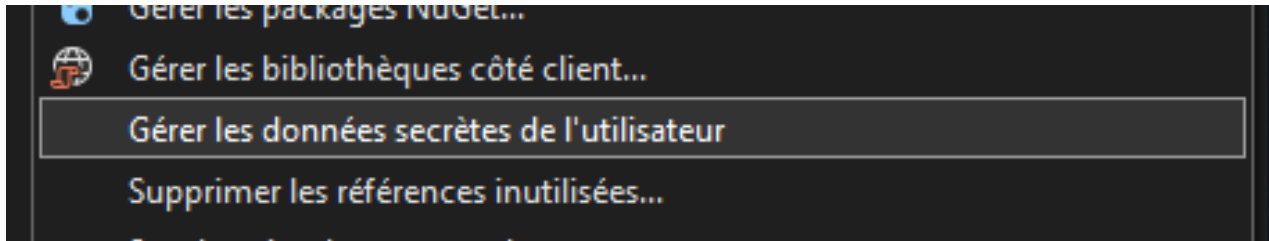
On remarque dans la méthode que l'on se sert également du dictionnaire de configuration. Ce dictionnaire est alimenté selon un ordre particulier en fonction de la présence des valeurs demandées :

- Les variables d'environnement système
- Les valeurs présentes dans le fichier `secrets.json` associé au projet
- Les valeurs présentes dans le fichier `appsettings.json` associé au projet

# Le fichier secrets.json

Ce fichier est un fichier caché utilisé de façon à sécuriser les valeurs que l'on ne souhaite pas voir apparaître dans un repository public de notre code. Par exemple, on y trouve fréquemment les chaînes de connexion à nos bases de données ou les valeurs de clés de cryptage ou de clés secrètes de nos services de cryptage.

Pour accéder à ce fichier, il suffit de faire un clic droit sur notre projet et de sélectionner l'option « **Gérer les données secrètes de l'utilisateur...** » :



Ceci fait, on se retrouve avec un fichier JSON permettant d'entrer des valeurs qui seront lues lorsque l'on se sert des méthodes **GetSection()**, **GetConnectionString()** ou de la notation entre crochets de notre configuration.

Il est également intéressant de noter que les éléments descendant hiérarchiquement dans le JSON peuvent être accédés dans la notation entre crochet de la sorte :

```
e = true,  
builder.Configuration["JWT:ValidAudience"],  
= true
```

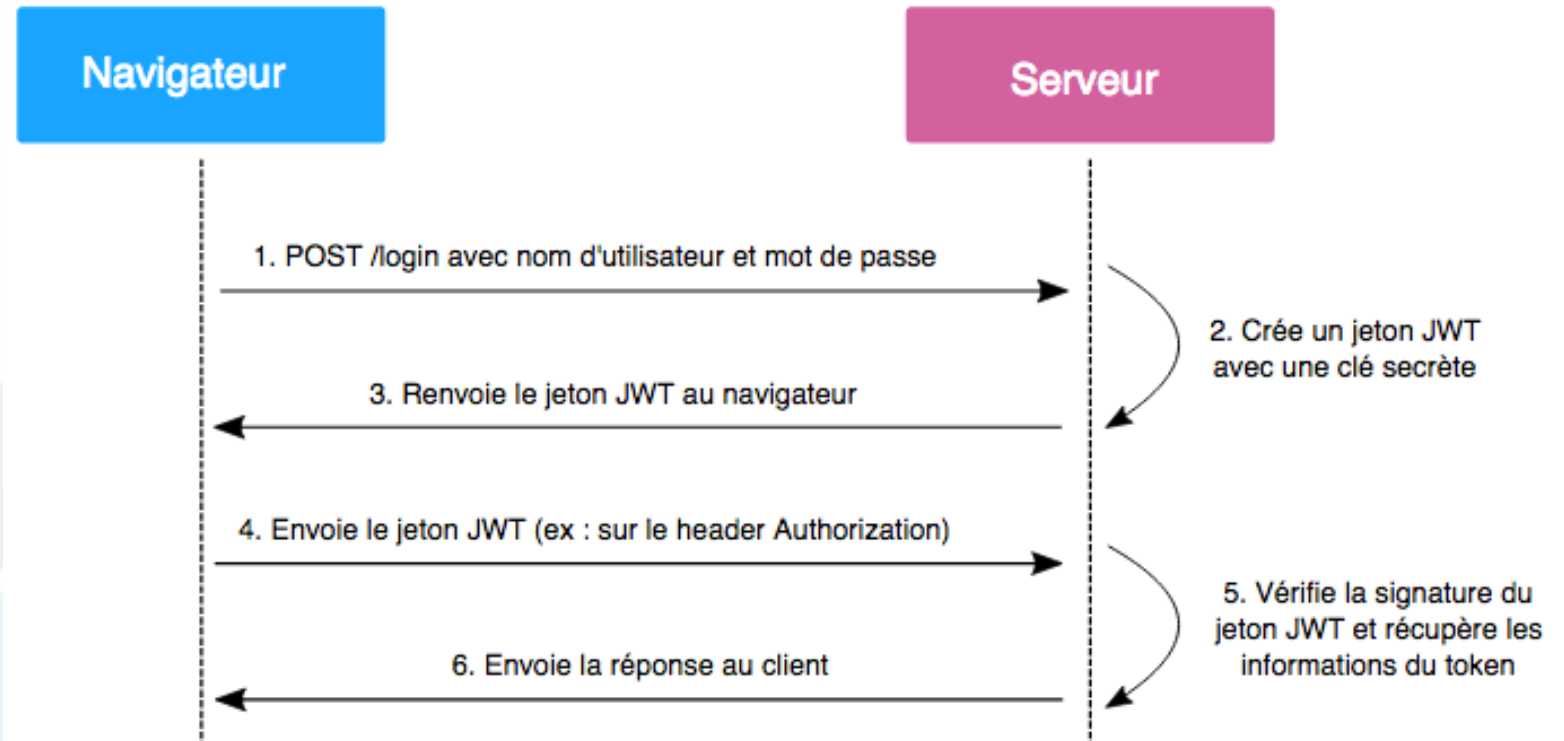
On peut ainsi enchaîner les hiérarchies complexes sans problème. Cependant, dans un souci de rapidité, le fichier **appsettings.json**, fonctionnant de la même façon, est plus aisé lorsque l'on souhaite travailler en groupe, car il n'y a ainsi pas besoin de réécrire à chaque fois les paramètres, que l'on supprimera simplement du fichier public lors de la production.

# Le JWT

@ Utopios Consulting

Le JWT répond à un standard (RFC 7519) dans le but de permettre un transit simple et sécurisé lors des échanges internet des services et applications. Il existe deux grands types de JWT :

- Les JWT **signés**, qui sont utilisés dans le but de vérifier que les informations des utilisateurs transitant avec eux (celles-ci n'étant pas masquées et pouvant facilement être lues via un simple décryptage de l'algorithme qui est souvent du **HMCHA256**) sont authentiques. Il ne faut donc pas mettre d'informations sensibles dans ces tokens et ne pas divulguer la clé de cryptage permettant de vérifier l'authenticité du token dans le but d'obtenir la 3<sup>ème</sup> partie de celui-ci.
- Les JWT cryptés se servant d'une clé privée et usant de RSA ou de ECDSA pour masquer à autrui les informations qui transiteraient avec eux.

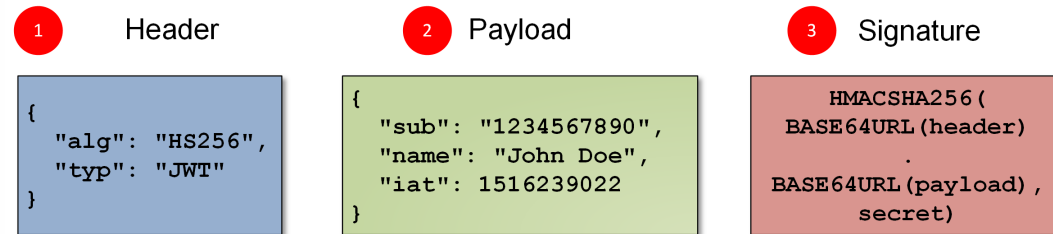




# Le JWT

@ Utopios Consulting

Dans le cadre de ce cours, nous nous intéresseront aux JWT signés et à l'utilisation d'une clé de sécurité pour vérifier qu'un utilisateur n'a pas falsifié les informations du token dans le but de se voir attribuer des privilèges supérieurs.



La structure d'un JWT est toujours la même. Elle consiste en une succession de trois strings correspondant au Header, au Payload et à la Signature.

- Le **Header** n'est constitué que de deux choses : l'algorithme de cryptage utilisé pour le l'objet JSON, et le type d'objet : JWT
- Le **Payload** contient quant à lui les claims de l'utilisateur, c'est-à-dire en quelque sorte ses droits. Il peut y avoir trois types de claims : des publiques , des enregistrées (comme la date d'expiration), et des privées.
  - Les claims **publiques** peuvent être définies en fonction des envies de l'utilisateur du JWT. Cependant, dans un soucis d'éviter les collisions, elles doivent être définies selon un schéma respectant le **Registre des Tokens JSON IANA**
  - Les claims **enregistrées** sont des claims prédéfinies non obligatoires mais recommandées lorsque l'on utilise les JWTs comme l'audience, les utilisateurs autorisés, la date d'expiration ou encore l'identité du service créateur du token
  - Les claims **privées** sont des claims utilisées par les personnes acceptant d'utiliser ce JWT mais qui ne sont ni enregistrées ni publiques.
- La **Signature** est quant à elle obtenue via le cryptage via la clé secrète des deux parties précédentes, et sert ainsi à attester de l'authenticité du token (si l'utilisateur falsifie les informations, le résultat de la signature sera différent de celui créé par utilisation de la bonne clé)



# Les Claims et les Rôles

Lorsque l'on se sert d'Authentification, on a besoin également de se servir de Claims. Pour faire simple, les claims ne sont ni plus ni moins que des lignes qui pourraient se situer sur un document d'identité comme un passeport ou un permis de conduire (par exemple le nom, le prénom, la date de naissance, la nationalité, le travail, le type de véhicule conduit, etc.). Lorsque le logiciel va lire les Claims de l'utilisateur, il va par exemple savoir s'il s'agit d'un manager, et pourra par exemple lui donner accès à tel ou tel type de contenu. A contrario, si l'utilisateur ne possède pas la claim requise, alors le logiciel ne lui permettra pas d'accéder à une fonctionnalité spécifique.

Il existe une série de Claims déjà présentés dans une Enum **ClaimTypes** comme par exemple l'Username, l'Email, etc... dont nous pouvons nous servir et qui sont l'un des piliers d'utilisation d'un module du .NET se nommant Identity. Il est cependant possible pour nous de créer nos propres claims, en faisant simplement appel au constructeur de Claim qui prend deux paramètres de type **string** en valeur comme par exemple **Claim("Profession", "Manager")**

Lorsque l'on définit la sécurité de notre API, on peut également se servir de Roles. Des rôles ne sont ni plus ni moins qu'une nouvelle claim qui a comme premier paramètre le schéma Microsoft des Roles et comme second une string représentant le rôle de la personne.

Une personne ne peut donc avoir qu'un seul rôle mais autant de Claims que l'on le veut.

# L'Authorization

© Utopios Consulting

Une fois l'Authentification paramétrée et mise en place, il est possible d'avoir recours à ce qui s'appelle l'Autorisation. Cette autorisation peut prendre autant de forme que l'on le souhaite mais se base la plupart du temps sur l'utilisation des claims ou des rôles (n'oublions pas que les rôles ne sont ni plus ni moins qu'un certain type de claims). A cela peut s'ajouter des paramètres personnalisés, comme par exemple des conditions booléennes de type « est présent dans la société depuis 1 ans au moins » ou encore « est majeur ».

Pour réaliser l'Autorisation, il faut tout d'abord l'ajouter dans le pipeline des services au niveau du fichier **Program.cs**. Surtout, il faut bien faire attention à l'ajouter à la chaîne des utilisations de services APRES le service gérant l'Authentification, sous peine d'avoir des problèmes lors de l'utilisation de notre API :

```
app.UseAuthentication();  
app.UseAuthorization();
```

Une fois cela fait, il est alors possible d'avoir recours à des autorisations que l'on peut paramétrer soit au niveau des contrôleurs, soit au niveau des méthodes. Pour ce faire, on dispose de plusieurs Data Annotations :

- **AllowAnonymous** permet lorsque l'on est au niveau d'un sous-ensemble d'outrepasser les règles édictées précédemment
- **Authorize** sert à demander le login d'un utilisateur avant d'accéder à cette partie de l'application
- **Authorize(Role="MonRole")** sert à n'autoriser à cet emplacement que les personnes disposant du rôle demandé
- **Authorize(Policy="NomPolicy1")** sert à autoriser les utilisateurs respectant la règle demandée.

Il est possible de paramétrer l'Autorisation et créer nous même autant de politiques que l'on le souhaite et de les rendre aussi complexes que l'on le voudrait.

# Les Politiques

@ Utopios Consulting

Une règle d'Autorisation peut se présenter de plusieurs façon, mais devra de toute manière être paramétrée lors de la configuration des services dans le fichier **Program.cs**

On peut alors avoir des règles se contentant de demander la présence d'une claim spécifique chez l'utilisateur, ou d'autres demandant que ces claims aient en plus une valeur définie.

Il est possible de demander ainsi plusieurs claims, voir la présence d'un rôle et d'une série de claims.

```
builder.Services.AddAuthorization(options => {  
  
    options.AddPolicy("AdminOnly",  
        policy => policy.RequireClaim("Admin"));  
  
    options.AddPolicy("MustBelongToHRDepartment",  
        policy => policy.RequireClaim("Department", "HR"));  
  
    options.AddPolicy("HRManagerOnly", policy => policy  
        .RequireClaim("Department", "HR")  
        .RequireClaim("Manager")  
        .Requirements.Add(new HRManagerProbationRequirement(3)));  
  
});
```

De plus, nous pouvons créer des conditions booléennes nous même, vu que les claims ne peuvent contenir que des association de deux strings. Pour ce faire, il faut créer une classe implémentant l'interface **IAuthorizationRequirement** ainsi qu'une autre classe servant à gérer les paramètres de la première. Cette seconde classe devra de son côté obligatoirement implémenter **AuthorizationHandler<ClasseDeBase>** et posséder une méthode asynchrone **HandleRequirementAsync()** de la sorte :

# Les Requirements personnalisés

Nous pouvons créer des conditions booléennes nous même, vu que les claims ne peuvent contenir que des association de deux strings. Pour ce faire, il faut créer une classe implémentant l'interface **IAuthorizationRequirement** ainsi qu'une autre classe servant à gérer les paramètres de la première. Cette seconde classe devra de son côté obligatoirement implémenter **AuthorizationHandler<ClasseDeBase>** et posséder une méthode asynchrone **HandleRequirementAsync()** de la sorte :

```
4 références
public class HRManagerProbationRequirement : IAuthorizationRequirement
{
    1 référence
    public HRManagerProbationRequirement(int probationMonths)
    {
        ProbationMonths = probationMonths;
    }

    2 références
    public int ProbationMonths { get; }
}

0 références
public class HRManagerProbationRequirementHandler : AuthorizationHandler<HRManagerProbationRequirement>
{
    0 références
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, HRManagerProbationRequirement requirement)
    {
        if (context.User.HasClaim(x => x.Type == "EmploymentDate"))
        {
            var empDate = DateTime.Parse(context.User.FindFirst(x => x.Type == "EmploymentDate").Value);
            var period = DateTime.Now - empDate;
            if (period.Days > 30 * requirement.ProbationMonths)
            {
                context.Succeed(requirement);
            }
        }

        return Task.CompletedTask;
    }
}
```

# Les Cors et l'accès distant

Lorsque l'on souhaite utiliser notre API via des frameworks front de type React ou Angular, il arrive fréquemment que l'on se heurte à un soucis de taille : Les CORS policies. Ces Cors policies sont par exemple là dans le but de sécuriser l'accès de notre navigateur à des APIs potentiellement malveillantes, et au niveau de notre API pour éviter que des adresses IPs non voulues puissent faire des requêtes d'un type paramétré.

Pour paramétrer les CORS, il faut encore une fois se rendre dans le fichier Program.cs et ajouter un service que l'on devra configurer nous même :

```
builder.Services.AddCors(options => {  
    options.AddPolicy("MyPolicy", builder =>  
    {  
        builder.WithOrigins("https://example.com", "http://example2.fr")  
        .WithMethods("POST", "GET")  
        .WithHeaders("custom-header");  
    });  
  
    options.AddPolicy("allConnections", builder =>  
    {  
        builder.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader();  
    });  
});
```

Une fois cela fait, il suffit d'ajouter une Data Annotation au niveau de notre contrôleur ou de ses méthodes tel que :

- **EnableCors("NomPolicy")** pour activer les vérification de requêtes croisées à ce niveau
- **DisableCors** pour annuler la vérification des Cors à ce niveau

# Exercice – API d'Authentification

Vous devrez réaliser une application permettant à un utilisateur logué d'accéder à une liste d'albums de musiques. Ces albums contiendront un **titre**, un **artiste**, un **prix** et l'URL de la **couverture de l'album**. Un utilisateur de type **administrateur** pourra en plus de cela **ajouter**, **éditer** ou **supprimer** des albums. Pour ce faire, vous utiliserez un **Mock** de base de données (via un service que vous pourrez nommer **FakeDb**) dont vous devrez penser au cycle de vie pour vous offrir une pseudo-persistance entre les différentes requêtes API.

L'API devra évidemment stocker aussi les **utilisateurs**, qui seront définis en tant qu'administrateur ou non par un utilisateur déjà administrateur (pensez donc à avoir de base dans votre base de donnée un utilisateur de ce type afin de pouvoir manipuler les autres). Les administrateurs pourront également **éditer** ou **supprimer** des utilisateurs, alors qu'un utilisateur lambda ne pourra que se **loguer** pour obtenir son JWT utilisé par la suite dans le reste de l'API.



*@ Utopios Consulting*

# **Entity Framework Core**

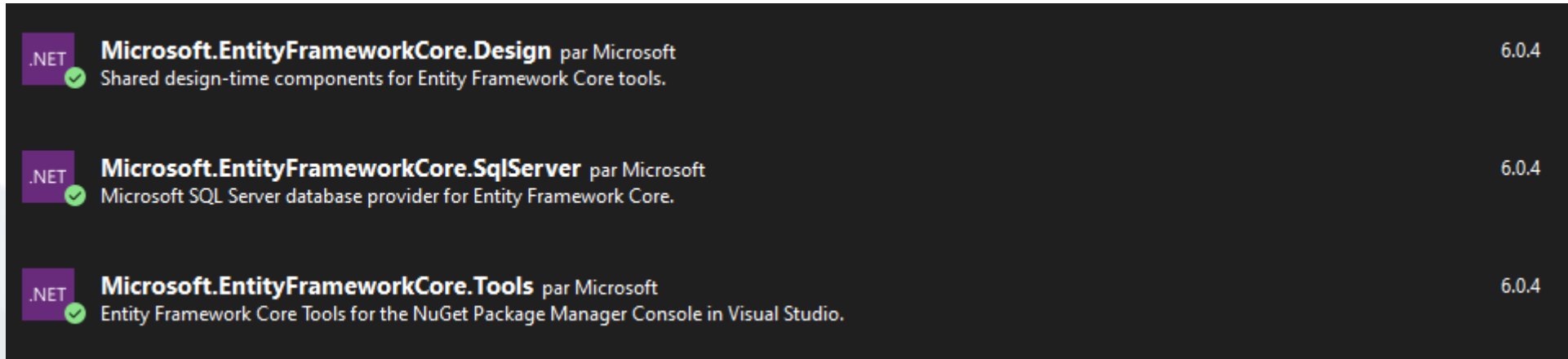
The bottom of the slide features several overlapping, wavy, horizontal bands in shades of light blue and white, creating a modern, fluid background element.



# Qu'est-ce qu'Entity Framework ?

Entity Framework est ce que l'on appelle communément un ORM (Object Relation Mapping) permettant de synchroniser efficacement les manipulation des classes et des objets en mémoire avec leur représentation en base de données dans un soucis de persistance sans avoir besoin d'écrire manuellement des centaines de requêtes SQL. Bien évidemment, l'utilisation d'un ORM a un impact sur les performance et la rapidité de l'utilisation de la base de données, mais également, et heureusement pour nous, il s'agit d'un gain de temps considérable lorsque l'on se met à coder un programme ayant besoin d'une persistance de données comme c'est le cas la plupart du temps d'une API.

Pour utiliser Entity Framework Core, il faut utiliser au minimum un package NuGet, mais il est préférable d'en installer trois tels que :



Le NuGet le plus important est ici le deuxième, qu'il convient de choisir en fonction du **fournisseur de base de données** que l'on souhaite se servir. Ici, il s'agira donc de **SqlServer**, mais il en existe pour d'autres tels que **PostgreSQL** ou **MySQL**. Le NuGet **Tools** sert à ne pas avoir à utiliser le terminal mais plutôt la console de gestionnaire de package dans le but de réaliser des migrations, et le module **Design** sert pour la visualisation des diagrammes de cardinalités pour notre application.

# Le DbContext

Utopios Consulting

Lorsque l'on veut se servir d'EntityFramework, il va encore une fois falloir modifier notre fichier Program.cs dans le but de lui ajouter un nouveau service qui est injecter par la méthode **AddDbContext()**. Cette méthode générique va prendre en typage la classe de notre contexte de base de donnée, qui par convention va se nommer soit **DataContext**, soit **ApplicationDbContext** :

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration["ConnectionString:Default"]);
});
```

L'ajout de ce service se fera de façon **Scoped**, car il ne faut pas utiliser toujours la même connexion en base de données. L'ajout du paramètre options vient de la façon dont la classe **ApplicationDbContext** a été définie et la façon dont son constructeur est implémenté :

```
3 références
public class ApplicationDbContext : DbContext
{
    0 références
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
    {
    }
}
```

Via l'utilisation d'un tel constructeur, il est ainsi possible de sortir l'utilisation de la méthode surchargée depuis la classe **DbContext** de notre classe **ApplicationDbContext**. Ceci est réalisé pour permettre un paramétrage de la **chaine de connexion** plus facile par accès au fichiers **secrets.json**, **appsettings.json** ou encore des **variables d'environnement**.

# Les DbSets

@ Utopios Consulting

Une fois notre classe **ApplicationDbContext** créée, nous pouvons y ajouter des variables de type **DbSet<T>** publique dans le but d'avoir un accès aux différentes tables de notre base de données. Un DbSet permettra ainsi un fonctionnement similaire à une liste qui sera synchronisé automatiquement par gestion des évènements de modification lorsque nous utiliserons les méthodes **SaveChanges()** ou **SaveChangesAsync()**. Par exemple, la simple réalisation d'un CRUD d'un chenil peut se faire aussi aisément que selon les lignes d'instructions ci-dessous, et ce sans aucune écriture de requête SQL de notre part :

```
[HttpGet]
public IActionResult Index()
{
    List<Dog> dogs = _context.Dogs.ToList();

    return View(dogs);
}
```

```
[HttpPost]
public IActionResult Create([Bind("Name, CollarColor, Age")] Dog newDog)
{
    if (!ModelState.IsValid)
    {
        return View(newDog);
    }

    _context.Dogs.Add(newDog);

    _context.SaveChanges();

    return RedirectToAction("Index");
}
```

```
[HttpPost]
public IActionResult Edit([Bind("Id, Name, CollarColor, Age")] Dog newDog)
{
    if (!ModelState.IsValid)
    {
        return View(newDog);
    }

    Dog found = _context.Dogs.FirstOrDefault(x => x.Id == newDog.Id);

    found.Name = newDog.Name;
    found.Age = newDog.Age;
    found.CollarColor = newDog.CollarColor;

    _context.Dogs.Update(found);
    _context.SaveChanges();

    return RedirectToAction("Index");
}
```

# Le Code First

@ Utopios Consulting

Après l'ajout de **DbSet** publiques dans notre classe ApplicationDbContext, il ne nous reste plus que deux grandes étapes pour permettre un paramétrage de notre application avec EF Core en mode Code-First.

La première des deux grandes étapes est le paramétrage de nos modèles, via une série de **Data Annotations** dans le but de permettre la configuration de nos entités en base de données. Les exemples les plus souvent rencontrés est alors soit le **type** de variable, soit la **précision** des décimales ou encore la **longueur** des strings. Pour ce faire, il suffit d'ajout des annotations entre crochets comme ci-après :

Il est intéressant de noter qu'Entity Framework est suffisamment intelligent pour comprendre qu'une propriété nommé **Id** ou **ClassId** pour une classe nommée « Classe » sera automatiquement affublée du paramétrage d'une clé primaire. De plus, l'ajout d'une cardinalité en joignant une propriété de type classe **ClasseB** dans **ClasseA** va créer une « propriété de navigation » servant à réaliser des jointures. L'ajout d'une propriété du type de la clé secondaire n'est pas obligatoire, mais permet la modification simple et rapide des associations avant la sauvegarde des changements en base de données.

```
public class ApplicationDbContext : DbContext
{
    public DbSet<Dog> Dogs { get; set; }

    public DbSet<Master> Masters { get; set; }

    public DbSet<Address> Addresses { get; set; }

    public DbSet<Account> Accounts { get; set; }
}
```

```
public class Dog
{
    [Key]
    public int Id { get; set; }

    [StringLength(50)]
    2 références
    public string Name { get; set; }

    [StringLength(100)]
    2 références
    public string Breed { get; set; }

    2 références
    public int MasterId { get; set; }

    [ForeignKey("MasterId")]
    2 références
    public virtual Master Master { get; set; }
}
```

# Les Migrations

@ Utopios Consulting

Une fois nos DbSets créés et notre série de modèles paramétrés, il est nécessaire de synchroniser notre base de données avec les classes que l'on va utiliser dans l'application. Pour ce faire, il va falloir utiliser ce qu'on appelle une **migration**. Cette migration, dans le cas où l'on a bien installé le package Tools d'Entity Framework Core, se passe dans la console de gestionnaire des packages, ouvrable via l'onglet de Visual Studio **Affichage > Autre fenêtres > Console de Gestionnaire de Packages**

Une fois ouverte, il ne nous reste plus qu'à utiliser la commande **Add-Migration <Nom de la migration>** (que l'on nommera Initial par convention lors de la première convention) pour qu'Entity Framework parcourt nos modèles associés à des DbSets dans la classe d'ApplicationDbContext injectée au niveau du Program.cs. Une fois ceci fait, nous aurons alors un script de migration possédant deux méthodes : **Up()** et **Down()**

Ces deux méthodes sont utilisées pour appliquer ou retirer les changements (lorsque l'on veut par exemple passer à une migration antérieure ou faire une migration ajoutée par autrui).

Une fois le script parcouru et les modifications en Base de Données approuvées par le développeur, il nous suffit alors d'entrer la commande **Update-Database** dans la console de gestionnaire de packages pour les appliquer. Dans le cas où notre script de migration ne nous convient pas, on peut réaliser la commande **Remove-Migration** pour l'annuler et le supprimer du dossier Migrations de notre projet. Il est également possible d'appliquer les changements jusqu'à une migration voulue en ajoutant le nom de la migration en argument de notre commande d'update, comme ceci : **Update-Database <Nom de migration>**



# Exercice – API de Répertoire Téléphonique

Vous devrez réaliser une API servant à un utilisateur d'accéder à un répertoire téléphonique. La personne devra pouvoir opérer un CRUD de base (Ajout, Lecture, Edition et Suppression de contacts depuis l'API). Pour ce faire, vous devrez utiliser un contrôleur sécurisé permettant de réaliser ce CRUD (qui héritera de **BaseController**).

L'API devra communiquer avec l'utilisateur via des JSON et être compatible avec Postman (La communication se fera alors au format **raw** de type **application/json**). La base de données sera réalisée avec Entity Framework Core et injectée dans le contrôleur pour lui en permettre l'accès (pensez à vérifier votre cycle de vie de service lors de l'injection de dépendance), et les méthodes du CRUD se feront via ce service qui servira au contrôleur de contexte de données.

Vous devrez donc à terme avoir un **ContactsController** qui gèrera toutes les fonctionnalités énoncées à partir d'une classe de contexte de données possédant la **collection** des contacts ainsi que des utilisateurs. Les contacts auront comme champs leur **nom**, leur **prénom**, leur **nom complet**, leur **date de naissance**, leur **âge**, leur **sexe**, leur **email**, leur **numéro de téléphone** et leur **avatar** sous la forme d'une URL. Les utilisateurs auront comme caractéristiques leur **nom d'utilisateur**, leur **email**, leur **mot de passe** et leur **date d'inscription**. Seul un utilisateur logué aura accès au répertoire, qui ne sera donc pas publique ! Un utilisateur de type **administrateur** pourra quant à lui modifier les autres utilisateurs.

# Les relations de Cardinalité

Lorsque l'on utilise une base de données, nous avons souvent besoin d'avoir recours à de la cardinalité. Cette cardinalité permet l'association et la liaison des tables entre elles dans le cadre d'une base de données SQL. Il en existe plusieurs :

- Le One-to-One qui lie un élément à un autre (Une personne possèdera une carte d'identité)
- Le One-to-Many qui lie un élément à plusieurs (Une personne peut posséder plusieurs enfants)
- Le Many-to-One qui lie plusieurs élément à un seul (Plusieurs produits peuvent être dans le panier de l'utilisateur)
- Le Many-to-Many qui lie plusieurs éléments à plusieurs (Plusieurs téléphones peuvent être achetés par plusieurs personnes)

Pour réaliser ces associations de cardinalités via EF Core, il suffit de modifier les modèles pour prendre en compte ces associations. Par exemple ici pour une application de gestions de film, nous avons une classe d'emprunt qui doit lier à une même date un utilisateur et plusieurs films (une personne emprunte plusieurs film d'un coup) :

On peut voir ici que la représentation d'une cardinalité 1-1 est simplement l'utilisation d'une classe en propriété d'une autre.

Il est également (et recommandé) d'ajouter aux cardinalités des variables pour gérer les clés secondaires tel que **[ForeignKey("NomVariable")]** que l'on placera au dessus de la classe de liaison pour éviter les soucis de tracking d'ID dans EF Core en cas d'édition

```
3 références
public class Dog
{
    1 référence
    public int Id { get; set; }
    2 références
    public string Name { get; set; }
    2 références
    public string Breed { get; set; }

    0 références
    public int MasterId { get; set; }

    [ForeignKey("MasterId")]
    0 références
    public virtual Master Master { get; set; }
}
```



# Le Many to Many

Stopios Consulting

Dans le cas où l'on veut travailler avec des relations de type One-to-Many ou Many-to-Many, il nous faut utiliser une variable de type collection pour créer cette association. Il fut un temps où l'on avait besoin de créer une classe servant de lien entre ces deux entités via l'utilisation d'une liste de la classe 1 et d'une liste de la classe 2 dans une classe de liaison 3, mais désormais EntityFramework est suffisamment malin pour comprendre que l'on désire alors une relation de ce type par lui-même. EF Core créera de lui-même les clés secondaires ainsi que les colonnes StrangerID en se basant sur le nom des propriétés de navigation que l'on aura mit dans notre classe. Il nous suffit donc d'utiliser des propriétés de ce type dans une classe pour obtenir une relation de type One-To-Many, Dans l'exemple ci-dessous, MasterId aurait par exemple été créé pour nous en base de données. Cependant, sa création manuelle rend son utilisation possible dans notre code métier.

```
0 références  
public virtual List<Dog> Dogs { get; set; }  
}
```

Master.cs

```
0 références  
public int MasterId { get; set; }  
  
[ForeignKey("MasterId")]  
0 références  
public virtual Master Master { get; set; }  
}
```

Dog.cs

Le mot-clé « **virtual** » sert à EF Core dans le but de surcharger les propriétés. Il est facultatif mais vivement recommandé.

Pour réaliser du Many-to-Many, il suffit d'utiliser deux propriétés de type collection, une liste de classe B dans la classe A ainsi qu'une liste de classe A dans la classe B. EF Core se chargera encore une fois de créer pour nous une table de jointure et d'y créer les clés secondaires.

# Obtenir les jointures

Si l'on souhaite utiliser les propriétés de navigation, c'est avant tout dans le but d'avoir accès aux jointures. Par exemple, si l'on veut modifier ou afficher un maître ainsi que ses chiens, il nous faut remplir les valeurs de ses chiens, sous peine de nous voir offrir une exception de type **NullReferenceException** car les chiens de la personne sont tous nuls lors de leur édition.

Pour éviter ce genre de problème, il existe deux méthodes très utiles pour alimenter les objets à partir des propriétés. Il s'agit tout d'abord de la méthode **Include()** qui prend en paramètre une lambda permettant de cibler une des propriétés de navigation de la classe A pour obtenir son ou ses objets de classe B. Par exemple, on peut ainsi avoir les chiens d'un maître lors de la récupération de tous les maîtres :

```
1 référence
public ICollection<Master> GetAll()
{
    var masters = context.Masters.Include(x => x.Dogs).ToList();

    return masters;
}
```

Si l'on a besoin d'avoir en plus de cela d'alimenter les objets alimentés de base, alors on peut ajouter à la suite de la méthode précédente la méthode **ThenInclude()** qui fonctionne comme sa sœur, en lui passant une lambda permettant de cibler les propriétés de navigation de la classe B :

```
1 référence
public ICollection<Master> GetAll()
{
    var masters = context.Masters.Include(x => x.Dogs).ThenInclude(x => x.Address)
        .ToList();
}
```

# Exercice – API de Déménagement

Vous devrez réaliser une application permettant à des **utilisateurs** (dont les informations de login seront leur **email** et un **mot de passe**) logués de l'API d'obtenir le **référentiel** d'une compagnie de déménagement. Ce référentiel leur permettra d'avoir à leur disposition une liste de **clients** ainsi qu'une liste de **propriétés**. Ces clients contiendront comme caractéristiques leur **nom**, leur **prénom**, leur **adresse email** et leur **numéro de téléphone**. Les propriétés contiendront comme champs leur adresse composée d'un **numéro** et **intitulé de rue**, d'un **code postal** ainsi que d'une **commune**.

Via l'utilisation de ces deux classes, et en plus de permettre un **CRUD** de base pour celles-ci, vous devrez faire en sorte de permettre aux utilisateurs de l'API de faire **emménager** ou **déménager** les personnes dans les propriétés, et de savoir lors du visionnage des deux classes le **nombre de logements possédés par une personne** ou le **nombre d'habitants dans un logement**.

# Le Repository Pattern

Le Repository Pattern est un architecture Pattern utilisé dans le but d'augmenter encore une fois la scalabilité et la maintenabilité de notre application. De plus, il est utile dans le sens où il évitera les répétition de la part du programmeur de nombreuses méthodes **Include()** et/ ou **ThenInclude()** en plus d'offrir une sorte d'encapsulation du DataContext. Le contexte devient alors l'élément central du Repository Pattern, mais n'est plus directement accédé par les contrôleurs, qui se voient à la place injecter des « **repositories** » qui contiennent chacun une partie du contexte total.

Pour réaliser un repository pattern optimal, il convient de créer une classe abstraite de base **BaseRepository.cs** qui va avoir comme seules lignes de codes la construction et l'injection du DataContext dans une variable protégée. Viennent ensuite l'interface **IRepository<T>** qui va offrir les méthodes permettant le CRUD de base pour chaque modèle de notre API. Enfin, des classes de type **DogsRepository.cs** vont implémenter et hériter des deux précédentes dans le but d'offrir les méthodes réelles de manipulation des modèles. On obtient donc par exemple :

```
3 références
public abstract class BaseRepository
{
    protected readonly ApplicationDbContext context;

    1 référence
    public BaseRepository(ApplicationDbContext context)
    {
        this.context = context;
    }
}
```

```
0 références
public interface IRepository<T> where T : class
{
    0 références
    public T GetById(int id);
    0 références
    public ICollection<T> GetAll();
    0 références
    public T Add(T entity);
    0 références
    public bool DeleteById(int id);
    0 références
    public T Update (T entity);
    0 références
    public T AddOrUpdate(T entity);
}
```

# Exercice – API de Pizzas

@Utopias Consulting

Vous devrez réaliser une application permettant à des utilisateurs de l'API d'obtenir le menu d'une pizzeria sous la forme d'un **JSON** qui contiendra une **liste des pizzas disponibles**. Ces pizzas auront comme particularités un **nom**, un **prix**, une **date de création**, leur caractère **piquante** ou **végétarienne**, une **image** ainsi qu'une **liste d'ingrédients**.

Dans le cas où un **utilisateur** souhaiterait se loguer, il fournirait son email et son mot de passe et pourrait à son tour commander une ou plusieurs pizzas en spécifiant l'**heure de livraison** (celle-ci étant par défaut 30 min après la commande), son **numéro de téléphone** ainsi que l'**adresse**.

S'il s'agit d'un **administrateur** qui se logue, il disposera alors de la **liste des commandes ordonnées par priorité et urgence**, ainsi que les capacités de **modifier** le menu à sa guise (édition, suppression, ajout et visionnage des pizzas) ou de voir et d'**interagir avec la liste des utilisateurs** (par exemple accéder au meilleur client dans le but de lui envoyer une promotion ?).

*@ Utopios Consulting*

# **Introduction à Docker**



# Qu'est-ce que Docker ?

Docker est un service permettant d'éviter et de palier à l'utilisation de machines virtuelles lors du déploiement et du débbuging d'applications. Le problème majeur que Docker cherche ainsi à retirer de la chaine de production est le besoin d'installer et de gérer plusieurs environnements de développement qu'un administrateur réseau devait il y a encore de cela quelques années réaliser.



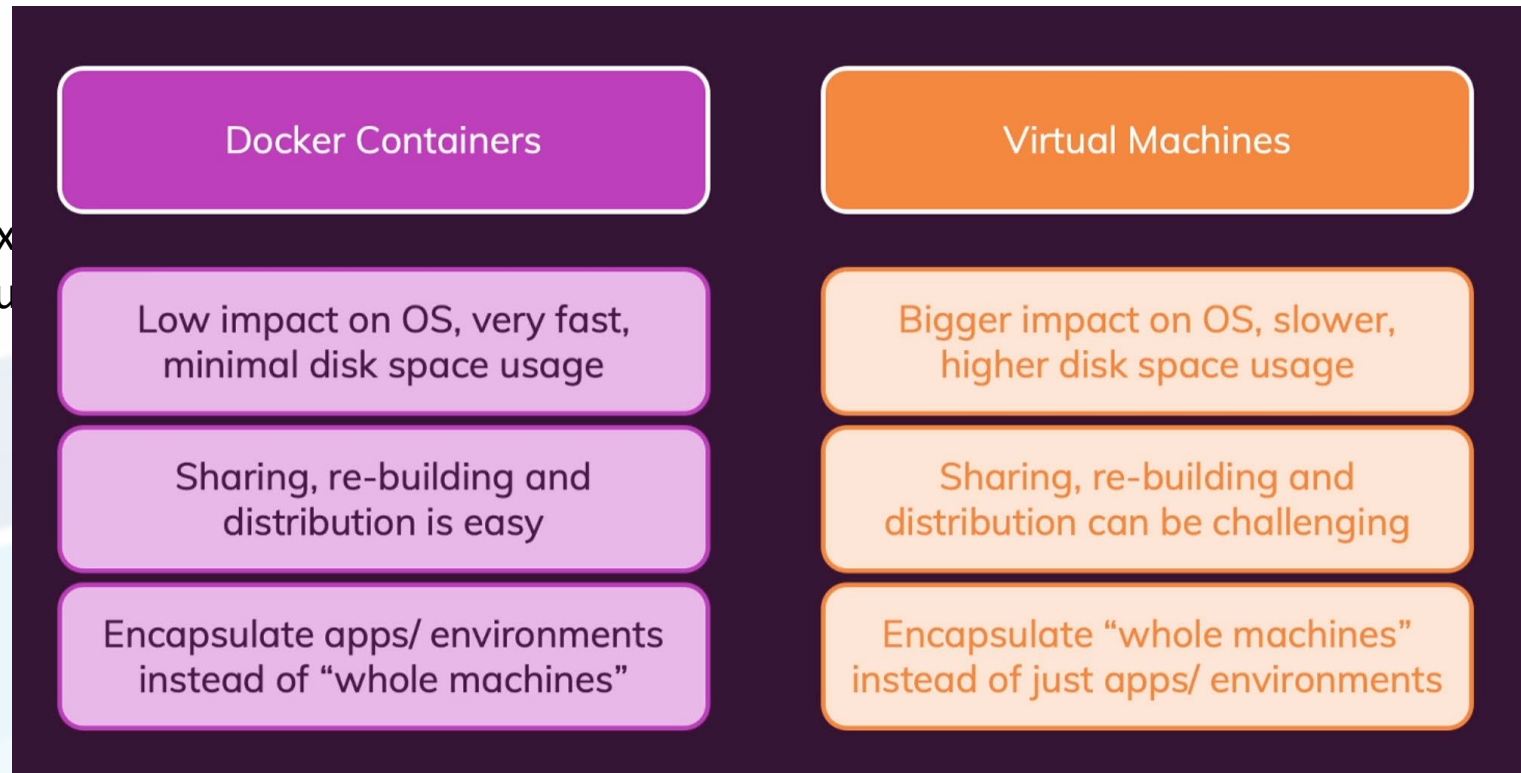
# Le Container

@ Utopios Consulting

Via Docker, on est désormais capable d'embarquer dans un « conteneur » toutes les dépendances et l'environnement de lancement de notre application, ce qui permet de travailler sur une même machine avec par exemple plusieurs versions de Linux et / ou d'ASP.NET en fonction de l'environnement de lancement cible. De plus, via Docker et un fichier Dockerfile (fichier de configuration du conteneur et de son build), il n'y a plus besoin de se soucier des dépendances NuGet, qui seront importées avec le conteneur via la commande appropriée.

De plus il est possible d'avoir autant de conteneur que l'on le veut sur une machine (virtuelle ou non). Grâce à cela, nous pouvons aisément tester notre application sur différents systèmes dans des soucis d'optimisation ou de benchmarking.

Docker règle donc de nombreux problèmes que pouvaient avoir les productions se basant sur des machines virtuelles, et ce sans utiliser autant de place dans le disque dur. Il existe par exemple de MB alors que l'installation d'une machine virtuelle nécessite plusieurs GB de mémoire disque.



# Installer Docker

topios Consulting

Pour installer Docker, il n'y a rien de plus simple. Il suffit de se rendre sur le site : <https://docs.docker.com/get-docker/>

Une fois fait, il faut sélectionner l'option qui nous concerne. Ici, nous prendrons l'option Docker Desktop for Windows, ce qui nous installera l'outil de gestion de Docker sur notre ordinateur.

Lorsque l'on veut déployer via Docker et Visual Studio, il est recommandé d'avoir installé Docker de cette manière

## Get Docker

### Update to the Docker Desktop terms

Commercial use of Docker Desktop in larger enterprises (more than 250 employees OR more than \$10 million USD in annual revenue) now requires a paid subscription. The grace period for those that will require a paid subscription ends on January 31, 2022. [Learn more.](#)

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

You can download and install Docker on multiple platforms. Refer to the following section and choose the best installation path for you.



### Docker Desktop for Mac

A native application using the macOS sandbox security model which delivers all Docker tools to your Mac.



### Docker Desktop for Windows

A native Windows application which delivers all Docker tools to your Windows computer.



### Docker for Linux

Install Docker on a computer which already has a Linux distribution installed.

# Les commandes de la CLI Docker

Docker se manipule principalement via le terminal, que cela soit sur Windows ou autre.

Pour apprendre la liste des commandes et obtenir des explication, il suffit de taper **docker --help** ou encore **docker <commande> --help**

Par exemple, pour observer la liste de nos processus de conteneurs en train de fonctionner sur notre machine, on peut faire **docker ps [-a pour All]** afin de voir l'ensemble de nos conteneurs, même ceux éteints.

```
$ docker ps --help

Usage:  docker ps [OPTIONS]

List containers

Options:
  -a, --all          Show all containers (default shows just running)
  -f, --filter filter Filter output based on conditions provided
  --format string    Pretty-print containers using a Go template
  -n, --last int     Show n last created containers (includes all states) (default -1)
  -l, --latest       Show the latest created container (includes all states)
  --no-trunc         Don't truncate output
  -q, --quiet        Only display container IDs
  -s, --size         Display total file sizes
```

# Les images Docker

A côté des conteneurs, Docker se base également sur ce que l'on appelle des « images ». Les Images sont en réalité les « plans » du code et des dépendances de notre application, qui seront lancées dans des conteneurs. Les conteneurs et les images agissent donc de concert pour faire fonctionner une application via Docker.

Les images sont des « packages » partageables qui peuvent être utilisées pour lancer toutes les applications que l'on veut. Il est également possible de prendre comme point de départ une image, de la compléter, et de sauvegarder ce changement pour avoir une nouvelle images plus complexe comme futur point de départ de notre développement.

En réalité, les images sont les moyens les plus évident de sauvegarder les changements dans notre application, ces changements étant en effet perdus lorsque l'on stoppe notre conteneur (à moins de l'avoir lié à un volume, que nous verrons plus tard dans ce cours).

Via la commande du CLI **docker run <nom d'image>**, on peut dès à présent lancer des images dans un conteneur (qui sera créé dans le but de supporter ladite image).

Pour observer l'ensemble de nos images, il suffit de taper la commande **docker images** pour nous voir offrir une liste de nos images sauvegardées sur ce système :

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
gharrowcontainerstest.azurecr.io:443/gharrowapilinux	latest	0028240f8b26	7 hours ago	211MB
gharrowapilinux	latest	0028240f8b26	7 hours ago	211MB
testapi	dev	55c2fba64fc7	2 days ago	208MB
gharrowapp	dev	733f748f679e	2 days ago	208MB
gharrowapilinux	dev	c20332b621c3	2 days ago	208MB



# Le mode interactif

Dans le cas où l'on souhaite pouvoir interagir avec nos conteneurs, il existe la possibilité de les lancer de façon interactive. Pour ce faire, il n'y a rien de plus simple, il suffit d'ajouter le paramètre « **-it** » à la commande de lancement, comme ceci :

```
PS C:\Users\Mat-S-Aint> docker run -it debian  
root@3bd13a73bab2:/#
```

De la sorte il est possible d'avoir facilement un terminal linux dans le but de préparer des futures images que l'on pourrait par la suite sauvegarder et / ou publier. Il est également bon de savoir qu'après la commande ci-dessus, nous aurions pu rajouter un autre point d'entrée de notre application (par défaut, le point d'entrée de cette image est le lancement de /bin/sh, mais il est possible d'en changer en ajoutant quel point d'entrée nous voulons atteindre de la sorte : **docker run -it debian <point d'entrée>**)

Pour sortir de notre conteneur de debian, il nous suffit d'entrer la commande linux **exit** dans le but de quitter la session root

Dans le cas où l'on aurait lancé notre conteneur sans l'avoir mis en mode interactif, ou alors si l'on veut passer d'un conteneur à l'autre, il est possible d'entrer dans une version interactive d'un conteneur via la commande **docker exec -it <conteneur> <commande>** pour lancer la commande interactivement et nous permettre d'accéder au conteneur.



# Dockerfile

@ Utopios Consulting

Lorsque l'on souhaite créer une image, le moyen le plus simple est d'avoir recours à un type de fichier particulier : le « **Dockerfile** ». Ce fichier doit idéalement se trouver à la racine de l'emplacement que l'on souhaite transformer en image dans un soucis de faciliter son utilisation. De plus, on peut y adjoindre un autre fichier, nommé « .dockerignore » et qui fonctionne de la même façon que son homologue de Git.

Pour écrire dans un fichier Dockerfile, ce n'est pas très compliqué. Par exemple, un fichier Dockerfile permettant le lancement d'une application front-end React JS basée sur une image de node se présentera comme ci-contre :

```
# pull official base image
FROM node:13.12.0-alpine

# set working directory
WORKDIR /app

# add `/app/node_modules/.bin` to $PATH
ENV PATH /app/node_modules/.bin:$PATH

# install app dependencies
COPY package.json ./
COPY package-lock.json ./
RUN npm install --silent
RUN npm install react-scripts@3.4.1 -g --silent

# add app
COPY . ./

# start app
CMD ["npm", "start"]
```

# Le Contenu du Dockerfile

Dans un fichier Dockerfile, il y a plusieurs instructions. Nous allons les voir une par une parmi les plus utilisées :

- **FROM** sert à définir l'image de base servant à la construction de l'image que l'on cherche à créer. Il est possible d'ajouter à la suite de cette instruction un alias pour notre section du Dockerfile (Un dockerfile peut être constitué d'un assemblage d'images et d'instructions avant de fournir une image finale de sortie) Pour ce faire, il suffit d'utiliser le mot clé **AS**
- **WORKDIR** sert à définir le dossier qui servira de base hiérarchique une fois notre image créée. Dans le cas où la hiérarchie contiendrait des espaces ou des caractères spéciaux, alors une syntaxe sous la forme d'un tableau de string est possible
- **ENV** permet de définir des variables d'environnement système que notre image pourra par la suite utiliser. Une syntaxe avec un signe d'égalité est aussi possible en cas de préférences personnelles.
- **COPY** sert à copier des dossiers ou des fichiers depuis un dossier source vers l'un des dossiers de l'image
- **RUN** permet de lancer des commandes lors de la création de l'image dans le but de l'alimenter
- **CMD** permet de spécifier une des commandes de base pour définir le point d'entrée (modifiable) de notre image
- **ENTRYPOINT** sert à spécifier une commande de base pour le point d'entrée (fixe ou presque) de notre image
- **EXPOSE** sert dans un but didactique afin de permettre aux gens lisant notre Dockerfile de comprendre que notre application a pour but de se servir d'un port en particulier (par exemple une application front-end React aurait besoin d'exposer le port 3000 pour l'HTTP)

# Docker Build

@ Utopios Consulting

Une fois notre image proprement configurée, il ne nous reste plus qu'à utiliser une nouvelle commande de la CLI : **docker build <emplacement de l'image>** pour créer une image nommée automatiquement pour nous via une ID.

Il est également possible d'ajouter un nom personnalisé à notre image via un paramètre de la commande, par exemple : **docker build -t bonjour** . va créer une nouvelle image basée sur notre Dockerfile et la nommer « bonjour » pour plus de clarté lorsque l'on parcourra la liste des images dockers (via la commande **docker images** par exemple)

Une fois notre image créée, nous pouvons la lancer via la commande **docker run <nom ou ID de l'image>**. Cependant, dans le cas où l'on a besoin d'accéder à des fonctionnalités web de notre application, il nous faudra également ajouter un paramètre à cette commande, comme dans l'exemple ci-contre :

```
docker run -p 80:3000 testapi
```

# Le développement avec Docker

Après avoir un peu expérimenté avec Docker dans le cadre du développement d'une application, on se rend rapidement compte d'un problème majeur : Les images sont des fichiers de type « read only ». Une fois l'image créée et lancée, la modification du code dans les fichiers de base ne se fera pas dans le conteneur, ce qui rend le développement et les tests d'une application conteneurisée plus difficile. Pour résoudre ce problème, il est possible de synchroniser les fichiers de notre ordinateurs avec ceux de l'image.

De plus, les images Docker sont construites à partir de couches. En effet, lors du build d'une image, Docker utilise un cache permettant d'éviter la répétition d'instruction et de compression de l'image. A chaque build, Docker va donc mettre en tampon chaque instruction de notre Dockerfile. La création de l'image est donc de type Read-Only car elle se fait à partir de la suite de ces couches permettant d'atteindre le résultat final.

Le lancement de l'image dans un conteneur ajoute par définition une nouvelle couche à cette succession d'instruction dans le but de la rendre interactive, mais ne peut pas changer les modifications précédentes.

Si l'on modifie des fichiers copiés dans une image, alors Docker va détecter une différence entre le cache et les fichiers à copier, ce qui va demander une re-copie de tous les fichiers spécifiés dans l'instruction, pouvant ainsi ralentir le build d'une image drastiquement. Il est donc intéressant de séparer les instructions dans le but d'améliorer la vitesse de build de notre image pour accélérer le développement sous Docker.

Par exemple, dans le cadre d'une application à dépendance, l'ajout du fichier de dépendance et l'installation des dépendances devrait être différenciée de la copie des fichiers de code pour accélérer le build de l'image Docker

# Récap' de la CLI @ Utopios Consulting

Les commandes de bases de la CLI de Docker sont nombreuses, et il ne faut pas oublier à utiliser le paramètre `--help` dans le but de se voir offrir un descriptif de ladite commande. Voici cependant un bref récapitulatif des commandes les plus utilisées :

- Pour lister les images présentes sur notre machine, on peut utiliser la commande **docker images**
- Pour analyser une image, on peut utiliser la commande **docker image inspect**
- Pour retirer une image (attention à ce qu'elle ne soit pas utilisée dans un conteneur, sous peine de devoir ajouter le paramètre `-f`), on peut utiliser **docker rmi**, ou **docker prune** dans le but de toutes les retirer
- Pour créer une image, il faut réaliser un build basé sur un Dockerfile via la commande **docker build**
- Pour taguer une image, il suffit d'ajouter le paramètre `-t` lors du build de l'image, afin de lui donner un nom
- Pour lancer un conteneur de façon interactive, il faut ajouter le paramètre `-i`, de même, pour le lancer de façon détachée, il faut le paramètre `-d`
- Pour suivre les messages consoles générés par un conteneur, on peut utiliser la commande **docker logs**
- Les conteneurs peuvent être nommés via l'utilisation du paramètre `--name` suivi du nom du conteneur que l'on souhaite
- Les conteneurs peuvent être listés via l'utilisation de la commande **docker ps** (n'oubliez pas le paramètre `-a` pour aussi voir les conteneurs à l'arrêt)
- Les conteneurs peuvent être supprimés via l'utilisation de la commande **docker rm**
- Un conteneur peut être stoppé via l'utilisation de la commande **docker stop** ou relancé via la commande **docker start** pour ne pas avoir à recréer un nouveau conteneur via la commande **docker run**
- Pour réaliser un mappage des ports lors du lancement d'un conteneur, il faut utiliser le paramètre `-p` suivi du port de notre machine puis du port du conteneur (Par exemple `-p 80:3000` pour mapper le port de base HTTP avec le port 3000 du conteneur)

# Les tags d'images

Il est fréquent que les images Docker disposent de tags, ce qui rallonge leur nom en les suffixant de leur tag.

Par exemple, l'image de node **node:12** est différente de l'image **node** de base, qui se basera en réalité sur l'image suffixée de latest (**node:latest**).

Quand on ajoute le paramètre **-t** lors du build d'une image, on y ajoute un tag, qui peut être un nombre ou une chaîne de caractère. Par convention, les tags d'images sont soit des mots du style **buster** ou **latest**, soit des numéros de version sous la forme **X.Y.Z** ou **XXX**.

En naviguant dans notre DockerHub, et en visionnant nos images, on peut ainsi observer les différents tags disponibles pour notre création.

Sort by

Newest

Filter Tags

TAG

latest

Log4Shell CVE not detected

Last pushed 5 days ago by dojanky

DIGEST

73c1ce254098

1ad58d4a1091

fa363e1bb577

+2 more...

OS/ARCH

linux/amd64

linux/arm/v7

linux/arm64/v8

COMPRESSED SIZE

352.44 MB

311.98 MB

344.11 MB

docker pull node:latest

TAG

slim

Log4Shell CVE not detected

Last pushed 5 days ago by dojanky

DIGEST

c4e9c92ba08d

78f7d5712de6

7a95ba494884

+2 more...

OS/ARCH

linux/amd64

linux/arm/v7

linux/arm64/v8

COMPRESSED SIZE

75.97 MB

68.51 MB

74.75 MB

docker pull node:slim

TAG

current-slim

Log4Shell CVE not detected

Last pushed 5 days ago by dojanky

DIGEST

c4e9c92ba08d

78f7d5712de6

7a95ba494884

+2 more...

OS/ARCH

linux/amd64

linux/arm/v7

linux/arm64/v8

COMPRESSED SIZE

75.97 MB

68.51 MB

74.75 MB

docker pull node:current-slim



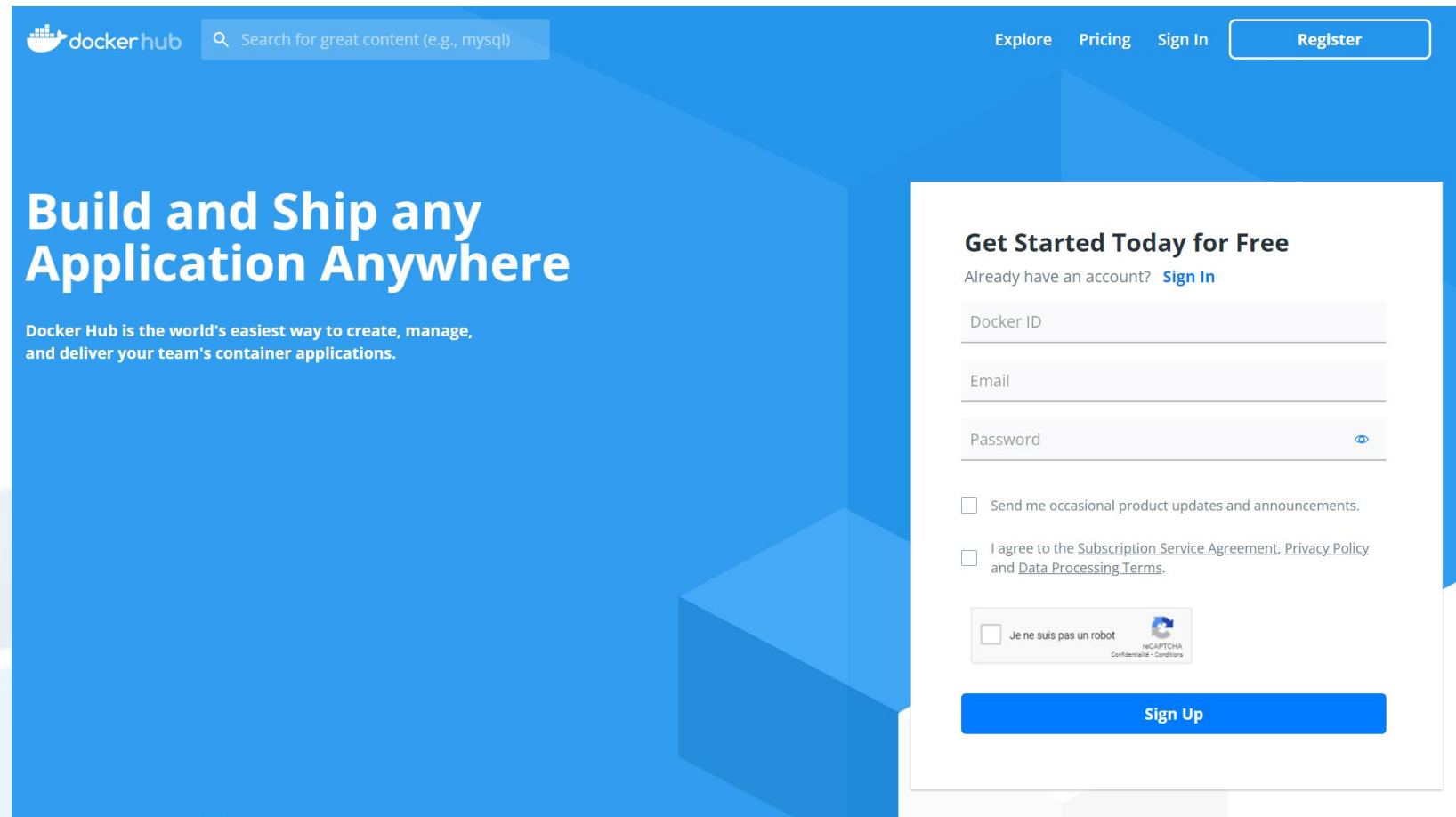
# DockerHub

@ Utopios Consulting

Lorsque l'on utilise Docker, il existe un emplacement dans le réseau Internet servant à stocker nos images Docker dans le but de pouvoir les partager avec d'autres développeurs ou les conserver pour un usage ultérieur.

Pour accéder et utiliser DockerHub, il suffit de se créer un compte (et accessoirement de synchroniser le logiciel Docker Desktop avec notre compte utilisateur).

Lien : <https://hub.docker.com/>



The image shows the Docker Hub homepage with a blue background. The main heading is "Build and Ship any Application Anywhere". Below it, a subheading states: "Docker Hub is the world's easiest way to create, manage, and deliver your team's container applications." The top navigation bar includes the Docker Hub logo, a search bar with the placeholder "Search for great content (e.g., mysql)", and links for "Explore", "Pricing", "Sign In", and a "Register" button. A registration overlay is positioned on the right side of the page.

**Get Started Today for Free**

Already have an account? [Sign In](#)


Docker ID

Email

Password

☐ Send me occasional product updates and announcements.

☐ I agree to the [Subscription Service Agreement](#), [Privacy Policy](#) and [Data Processing Terms](#).

☐ Je ne suis pas un robot  reCAPTCHA  
Confidentialité - Conditions

**Sign Up**

# Utiliser des images de DockerHub

Pour récupérer ou partager des images via ou vers DockerHub, il nous faut apprendre quelques commandes supplémentaires :

Dans le but de publier nos images sur DockerHub, il faut déjà les avoir construites via la commande « **docker build** » puis il faudra la publier dans des registres de conteneurs. Nous allons ici utiliser DockerHub, mais il existe d'autres endroits, comme par exemple Azure ou AWS, qui proposent des registres de conteneurs privés destinés au indépendants ou aux entreprises.

En nous servant de DockerHub, notre image deviendra publique, et sera disponible pour toute personne voulant l'utiliser. Attention donc à nos données sensibles, car une personne mal attentionnée pourrait s'en servir contre nous.

Pour publier vers DockerHub, il suffit d'utiliser la commande **docker push <nom de l'image>**

Pour récupérer de DockerHub, il suffit d'utiliser la commande **docker pull <nom de l'image>**

Ces commandes sont assez simples à retenir. Il est également fréquent de préfixer le nom des images que l'on compte publier de notre pseudonyme, de sorte à ce que le nom de notre image ne soit plus **nom-image** mais **pseudo/nom-image**

Il est possible de pusher une image non existante pour créer un repository vide. Une fois fait, la création d'une image portant ce nom, soit à partir de zéro et d'un **docker build**, soit via le clonage (**docker tag <ancien> <nouveau>**) permettra d'y pousser notre image finale.

Avant de pouvoir réaliser ces manipulations, il peut être nécessaire de se loguer via **docker login**

# Exercice – Somme et Moyenne en Python

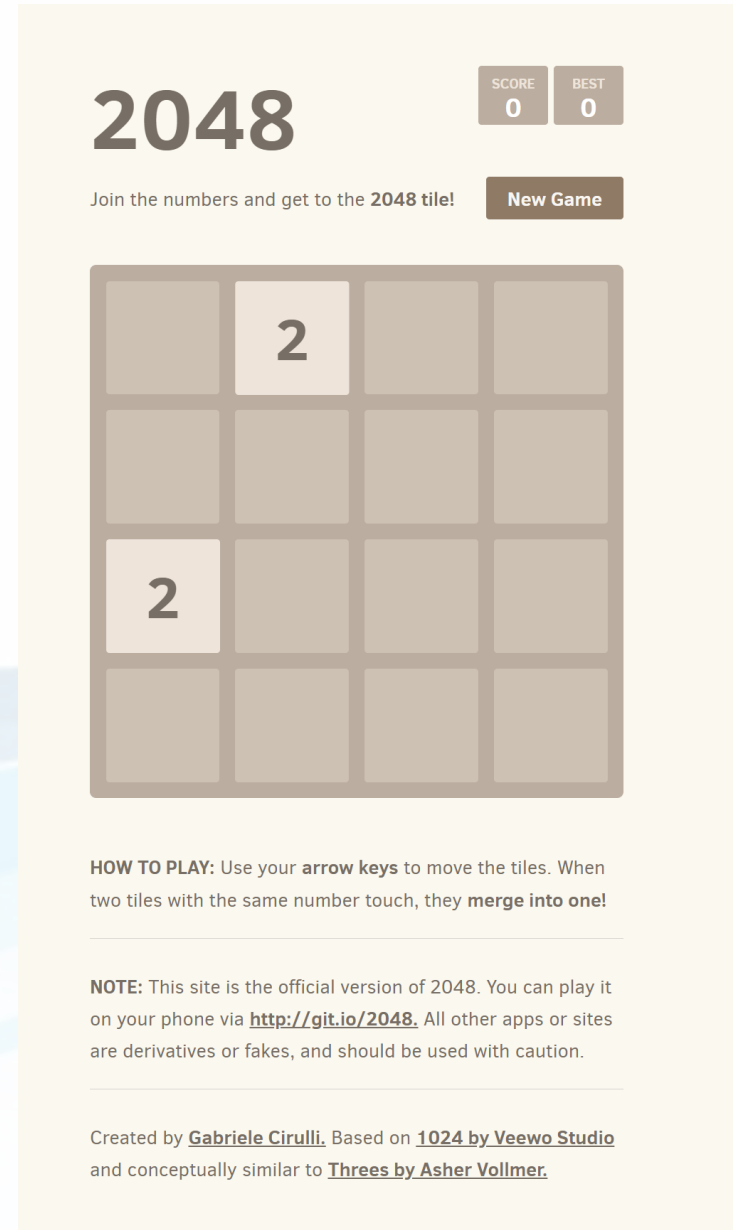
En créant un Dockerfile adapté au lancement d'un script Python fourni, vous devrez réussir à faire fonctionner une application console Python servant à un utilisateur pour entrer une série de notes et de se voir offrir la somme, la moyenne, la meilleure note et la moins bonne note de la liste entrée par l'utilisateur.

Vous devrez rechercher via l'utilisation du site DockerHub une image adaptée au lancement de ce script, et la faire tourner localement sur votre ordinateur de la façon adaptée à la manipulation du programme Python.

# Exercice – Application 2048

Via l'utilisation d'un Dockerfile, faites en sorte de créer sur votre machine une image Docker permettant de lancer le jeu 2048 via votre navigateur (pour ce faire, vous utiliserez le mappage de ports de sorte à y accéder via l'utilisation de <http://localhost>).

Vous trouverez aisément plusieurs possibilités d'images Docker correspondant à cette application sur DockerHub.



# La persistance avec les Volumes

Jusqu'à maintenant, nos images et nos conteneurs Docker avaient un problème majeur dans le cadre où l'on souhaitait par exemple faire tourner des applications professionnelles : elles ne disposaient pas de **persistance de données** ! En effet, lors de l'arrêt du conteneur, les modifications que l'on aurait pu effectuer à l'image sont automatiquement supprimées. Il est possible de pallier à ce problème en créant une nouvelle image à partir des changements, mais cette manipulation devient rapidement impossible lors de l'utilisation réelle d'un conteneur pour une base de données par exemple !

Application (Code + Environment)	Temporary App Data (e.g. entered user input)	Permanent App Data (e.g. user accounts)
Written & provided by you (= the developer)	Fetches / Produced in running container	Fetches / Produced in running container
Added to image and container in build phase	Stored in memory or temporary files	Stored in files or a database
"Fixed": Can't be changed once image is built	Dynamic and changing, but cleared regularly	Must not be lost if container stops / restarts
Read-only, hence stored in <u>Images</u>	Read + write, temporary, hence stored in <u>Containers</u>	Read + write, permanent, stored with <u>Containers</u> & <u>Volumes</u>

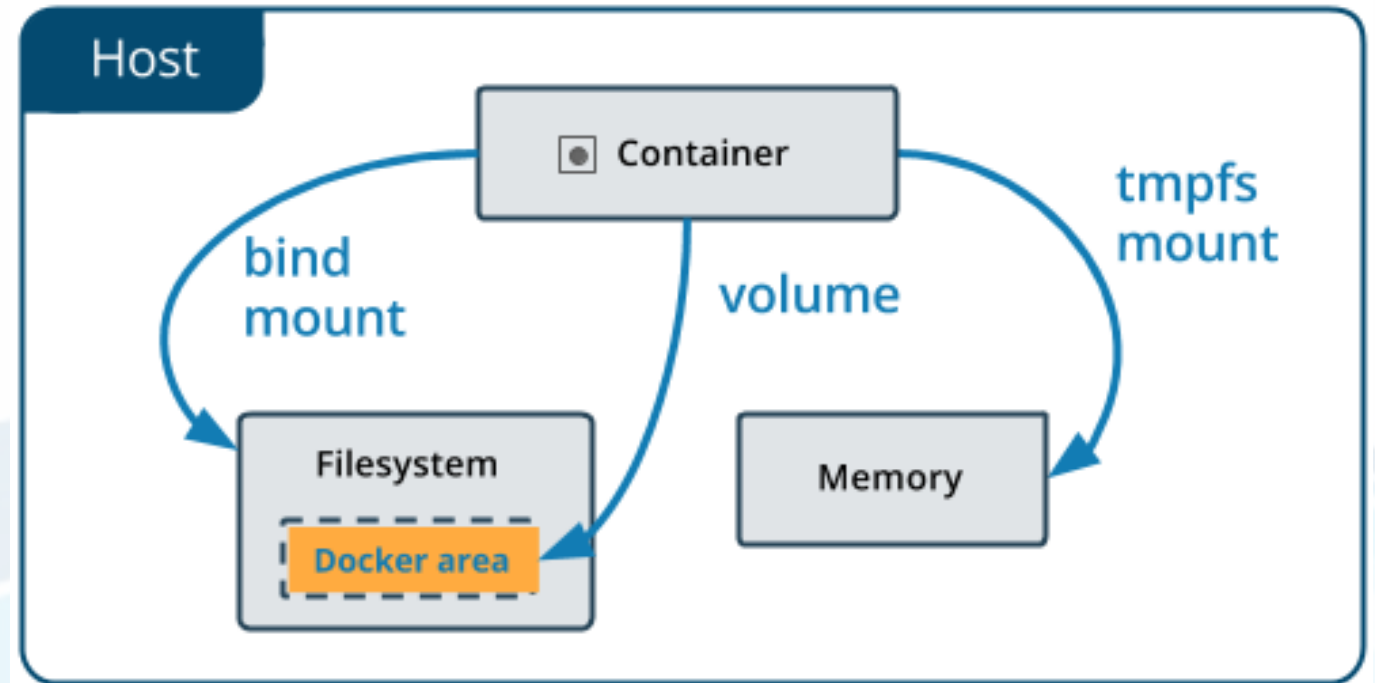
# Comprendre le problème

Les volumes sont en réalité des dossiers se trouvant sur la machine hôte. Ils sont par la suite montés lors de l'ajout du support d'un volume lors de l'utilisation de la commande **docker run** se servant du paramètre de volume **-v**. Il est exactement possible de synchroniser les fichiers avec ceux du conteneur dans un souci de rapidité de développement. Il n'y a donc plus besoin de relancer le build d'une image à chaque fois que l'on modifie notre application et notre code, comme par exemple dans une application de type React JS censé pouvoir offrir des changements immédiatement visibles dans le navigateur.

Dans un fichier Dockerfile, il est possible d'ajouter une instruction pour ajouter un volume à notre image : **VOLUME**

Cette instruction prendra en paramètre un tableau de string pour créer le volume.

Par exemple **VOLUME ["/app/data"]** va créer un volume rendant persistant la hiérarchie demandée. L'application va donc pouvoir travailler sur ce dossier et ne pas en perdre les changements en cas d'arrêt du conteneur, comme cela pourrait être le cas si une erreur survient.





# Nommer les Volumes

Il existe dans le monde des volumes Docker deux grands types de volumes : les volumes nommés et les volumes anonymes.

Pour accéder les volumes anonymes, le seul moyen de le faire est d'accéder à la liste des volumes via la commande **docker volumes ls** qui aura un nom généré automatiquement. Dans le cas où l'on stoppe l'application, alors le volume disparaîtra de la liste des volumes (il sera supprimé automatiquement si vous avez ajouté l'option **--rm** lors de votre **docker run**). Si l'on veut supprimer les volumes anonymes, il faut passer par des commandes du CLI : **docker volume rm <nom du volume>** ou **docker volume prune** pour tous les supprimer en cas de non utilisation.

Pour remédier à cela, on peut utiliser des volumes nommés dans le but d'accéder en permanence au chemin pointant vers l'emplacement du volume qui est créé par Docker pour nous. De plus, les volumes nommés resteront en cas de soucis techniques ou d'arrêt du conteneur. Ils sont essentiels lorsque l'on travaille avec des bases de données par exemple. Pour travailler avec des volumes nommés, il faut ajouter le paramètre **-v** lorsque l'on lance le conteneur.

La syntaxe est de la sorte : **-v <nom volume>:<chemin du dossier du volume>**

Les volumes nommés, à contrario des anonymes, ne sont pas attachés à leur conteneur, et peuvent travailler indépendamment de ces derniers. Il est possible du coup de travailler à partir de plusieurs conteneurs sur un même volume.

# Bind Mounts

@ Utopios Consulting

À côté de la commande permettant de créer des volumes stockés dans la hiérarchie de fichiers de Docker, qui ne nous est pas accessible, nous pouvons lier des dossiers de notre ordinateur avec ceux du conteneur dans un soucis d'update automatique de nos changements dans une application. Pour ceci, nous avons recours au « **Bind Mounts** ». La différence majeure avec les volumes est donc la capacité pour le développeur de gérer lui-même l'emplacement du dossier servant à la persistance des données du conteneur.

Pour réaliser un binding, il faut ajouter un paramètre supplémentaire lors de l'utilisation de la commande **docker run**. La syntaxe se présente de la sorte :

**docker run -v <hiérarchie absolue de dossier ou fichier>:<hiérarchie fichier ou dossier du conteneur>**

Pour obtenir rapidement la hiérarchie actuelle de notre dossier et éviter d'avoir à la réécrire, il existe deux commandes faciles d'utilisation qui sont pour Mac et Linux : **-v \$(pwd):/app** et pour Windows : **-v "%cd%":/app**

Si l'on réalise un binding, il faut faire attention aux dépendances, comme cela pourrait être le cas dans un projet Node JS. En effet, le dossier bindé ne possède par forcément, comme cela serait le cas du dossier de l'image, le dossier **node\_modules** ainsi que son contenu. Ce problème provient du fait que le dossier bindé va supplanter le dossier de l'image. Il est possible de spécifier des parties de notre hiérarchie qui seront ignorées par cette mécanique. Pour ce faire, on réalise un autre volume (anonyme) qui servira à stocker les dépendances : **docker run -v \$(pwd):/app -v /app/node\_modules** va par exemple créer un nouveau volume anonyme qui va contenir les packages node et qui va pallier au manque de ces derniers sur notre propre système de fichiers.

*@ Utopios Consulting*

# **Conteneuriser une API**

The bottom of the slide features several overlapping, wavy lines in shades of light blue and white, creating a modern, fluid background element.

# Création d'un projet avec Docker

Lors du lancement d'un nouveau projet, il est possible d'ajouter directement Docker dans l'équation au niveau de la troisième fenêtre de génération du projet. Pour cela, il suffit de cocher « Activer Docker » et de spécifier le type de système d'exploitation ciblé : Linux ou Windows :

A screenshot of a software configuration window with a dark theme. It features a checkbox labeled 'Activer Docker' which is checked. Below it is a label 'OS Docker' followed by a dropdown menu currently showing 'Linux'. The dropdown menu is open, showing 'Linux' and 'Windows' as options. At the bottom, there is another checked checkbox labeled 'Activer la prise en charge d'OpenAPI'.

Une fois fait, nous pouvons observer une configuration supplémentaire de lancement de notre projet dans le fichier **launchSettings.json** :

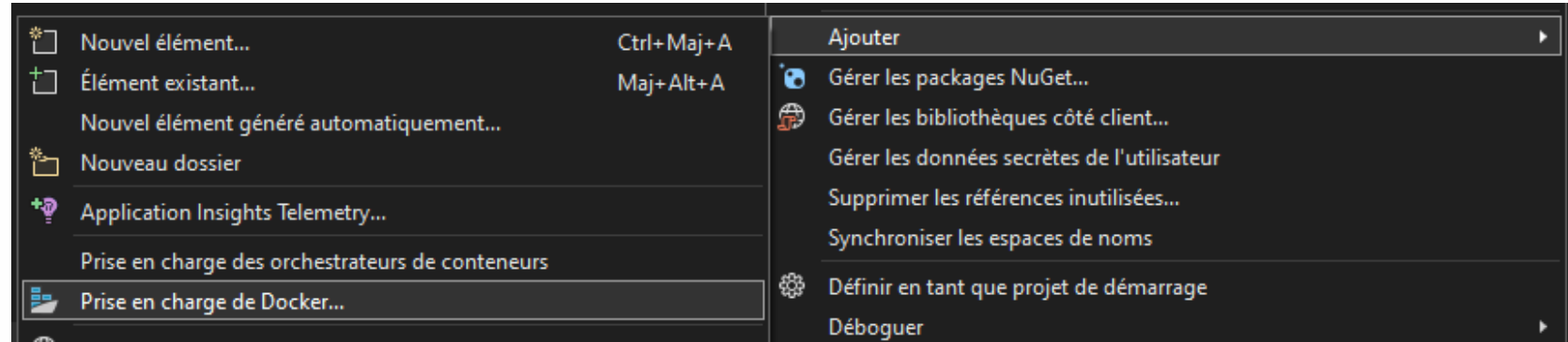
```
},
  "Docker": {
    "commandName": "Docker",
    "launchBrowser": true,
    "launchUrl": "{Scheme}://{ServiceHost}:{ServicePort}/swagger",
    "publishAllPorts": true,
    "useSSL": true
  }
}
```

Via l'utilisation de ce paramètre de lancement, nous pouvons tester et développer notre API depuis un conteneur stocké sur notre ordinateur.

# Création d'un Dockerfile d'API

Dans le cas où l'on disposerait d'un projet n'ayant pas été créé via l'option d'activation de Docker au démarrage, il est possible d'y remédier aisément en générant un Dockerfile via Visual Studio. Pour cela, il suffit de faire un clic droit sur le projet et de sélectionner les options

**Ajouter > Prise en charge de Docker...**  
puis de sélectionner le système  
d'exploitation visé :



Une fois fait, nous disposons donc d'un Dockerfile qui servira au développement ou à la publication de notre application dans un Conteneur :

Son fonctionnement est similaire à un Dockerfile qui se baserait sur node pour faire tourner une application front. En effet, la ligne de commande **dotnet restore** va servir dans le but d'ajouter toutes les dépendance de package à notre projet tandis que les seuls fichiers copiés seront les fichiers de code et les assets de la solution.

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["ApplicationDemo/ApplicationDemo.csproj", "ApplicationDemo/"]
RUN dotnet restore "ApplicationDemo/ApplicationDemo.csproj"
COPY . .
WORKDIR "/src/ApplicationDemo"
RUN dotnet build "ApplicationDemo.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "ApplicationDemo.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "ApplicationDemo.dll"]
```

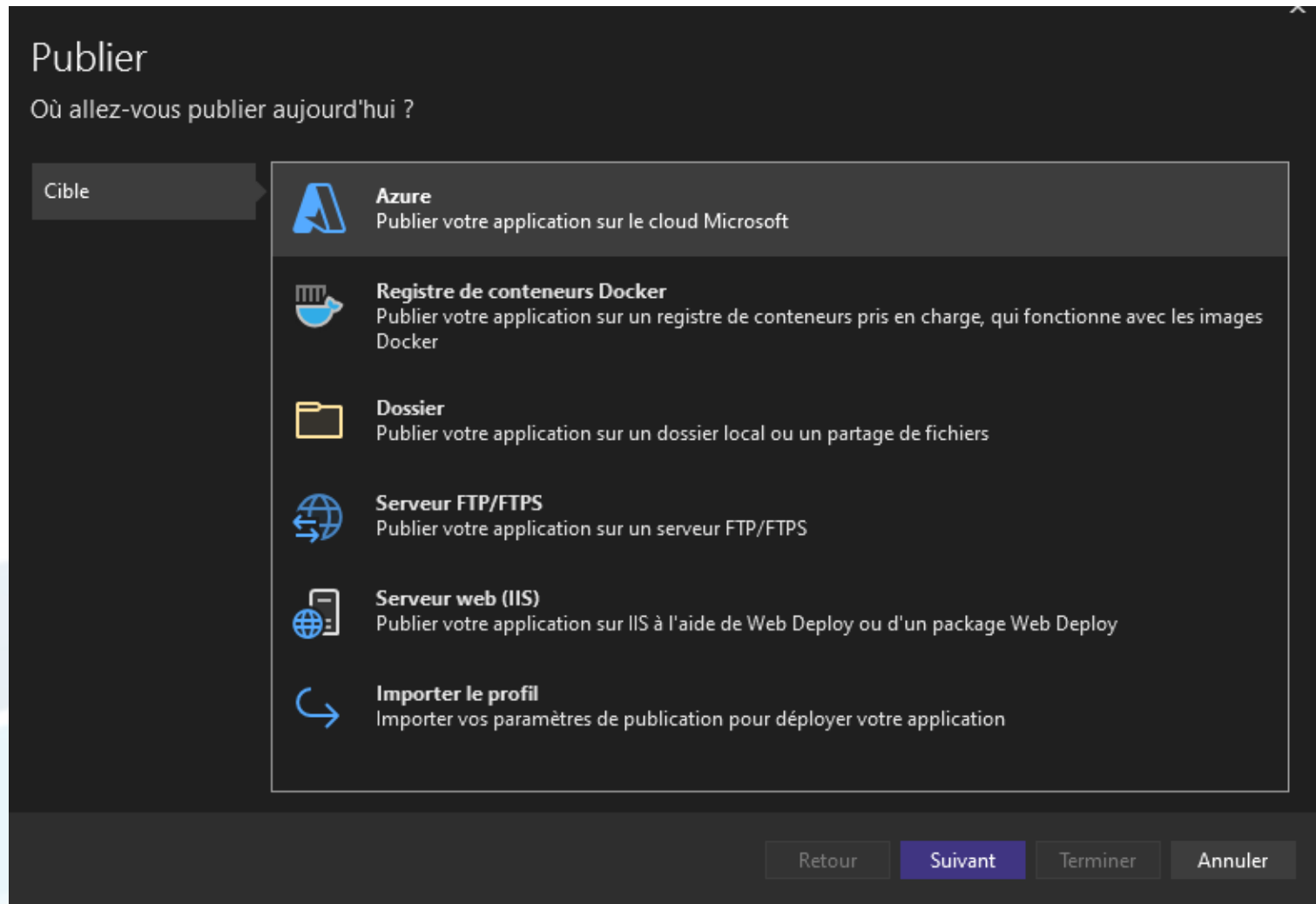
# Publication de l'API dockerisée

Une fois notre API finalisée et prête à être partagée, il nous faudra la publier. Pour cela, il suffit simplement de faire un clic droit sur le projet de sélectionner l'option **Publier...** qui nous ouvrira une nouvelle fenêtre :

Dans le cadre de notre API, les options les plus évidentes sont sans doute la publication sur Azure ou la publication dans un registre de conteneurs Docker.

Nous allons commencer en sélectionnant la publication dans le registre de conteneurs Docker en vue de l'ajout de notre image sur notre DockerHub.

On vous demandera ensuite votre identifiant et votre mot de passe de DockerHub avant de vous afficher la page suivante permettant de réaliser la publication finale.





# Ajout de l'API sur DockerHub

Une fois dans la fenêtre suivante (cette dernière servant de récapitulatif à vos différentes options de publication d'application réalisées avec Visual Studio), vous n'avez plus qu'à appuyer sur Publier et laisser votre ordinateur faire le travail.

Il est à noter qu'il est possible de modifier le numéro de version de l'application en appuyant sur le logo à droite de ce dernier dans le but d'éviter d'avoir le tag « **latest** ». N'oublier pas que pour Docker, les convention de nommage des version sont soit au format **X.X.X**, soit des noms comme « **buster** » ou des nombres entiers tels que **12**

Une fois la publication effectuée, un message d'alerte de succès viendra vous confirmer de sa réussite :

