

Formation ASP.NET Core

Présentation

Gaëtan GOUHIER

- Ingénieur en informatique.
- Spécialisé dans le développement .NET
- Certifié Microsoft (70-483, 70-486, AZ-203) / Azure Developer Associate
- Développeur et formateur indépendant
- <https://www.gaetan-gouhier.net/>

Et vous ?

- Votre connaissance du sujet ? (Développement, C#, ASP.NET, MVC)
- Vos attentes ?

Infos

- Horaires: 9h-12h30, 13h30-17h
- Emargement

Sommaire

1. [Introduction](#)
2. [Présentation du modèle MVC](#)
3. [Création d'une application MVC](#)
4. [Gestion des vues](#)
5. [Application de styles dans les applications](#)
6. [Les vues partielles](#)
7. [View components](#)
8. [Injection de dépendance](#)
9. [Middleware](#)
10. [Les filtres](#)
11. [Internationalisation des applications](#)
11. [Mise en œuvre d'AJAX](#)
12. [Contrôle de validation avec JavaScript et jQuery](#)
13. [Entity Framework Core](#)
14. [Aspects avancés](#)
15. [Optimisation](#)
16. [Web API](#)
17. [Tester son application](#)
18. [Gestion de la sécurité](#)
19. [Hébergements](#)
20. [Razor Pages](#)
21. [Blazor](#)

Introduction

Le Framework .NET

- Le Framework .NET est l'environnement de développement Microsoft pour la création d'application.
- Il inclut un ensemble de technologies :
 - Un modèle de programmation orienté objet complet pour la réalisation d'application.
 - Des composants visuels sophistiqués offrant des interfaces riches et conviviales.
 - Des outils de déploiements.
 - Un modèle d'exécution efficace et sécurisé.
- L'objectif principal est de faciliter vos développements, en intégrant un ensemble d'outils visant à réduire et à optimiser le code.

Le framework .NET core

- .NET Core est le nouveau framework open source et multiplateforme de Microsoft
- Le framework .NET 4.8 restera maintenu encore de nombreuses années
- Ces principales caractéristiques sont :
 - **Multiplateforme** : s'exécute sur Windows, macOS et Linux ; peut être porté sur d'autres systèmes d'exploitation.
 - **Open source**
 - **Indépendant de l'éditeur**: vous n'êtes ni obligé d'utiliser Visual Studio pour le développement ni IIS pour l'hébergement.
 - N'importe quel éditeur associé avec dotnet Command Line Interface permet de créer, générer et exécuter votre application.
 - n'importe quel serveur compatible OWIN peut héberger votre application.
 - **Souplesse de déploiement** : peut être inclus dans votre application ou installé côte à côte à l'échelle d'un utilisateur ou de l'ordinateur.
 - **Compatibilité** : .NET Core est compatible avec le .NET Framework, Xamarin et Mono

Le framework .NET core

- Ce Framework est devenu vraiment intéressant à partir de la version 2.1
- Des outils comme NET Portability Analyzer pourront permettre de voir si le projet est compatible .NET Core
 - L'installer en tant qu'extensions VS 2019
 - Puis le lancer depuis l'onglet Analyse
- Pour migrer du .NET Framework vers .NET Core (ce qui est déconseillé) il faut le faire à la main ou modifier des fichiers de la solution (déconseillé aussi)
- Rappel : le Framework est différent du langage, par défaut le .NET Core 2.X est sur du C# 7.3 mais on peut utiliser une version en dessous
- Le .NET Core 3.X est lui par défaut sur C# 8

.NET

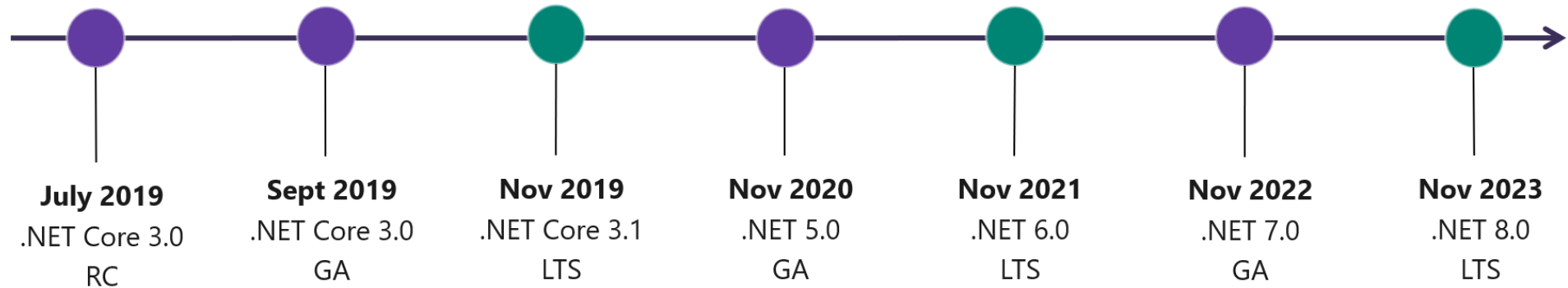
- .NET Core version 3 apporte le support des applications lourdes WinForm & WPF. Cette version est la dernière de .NET Core
- .NET Framework est en version 4.8. Cette version sera la dernière du .NET Framework
- On est maintenant en .NET 7 (Novembre 2022)
- Tous les ans une nouvelle version du .NET arrivera

.NET – A unified platform



La suite

.NET Schedule



- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed

VB

C++

C#

F#

CLS (Common Language Specification)

ASP.NET (Web)

UWP (Universal Windows Platform)

Entity Framework Core

Framework Base Class Library

CLR (Common Language Runtime)

Operating System

Le protocole HTTP

- HTTP a été inventé par Tim Berners-Lee avec les adresses Web (URL) et le langage HTML pour créer le World Wide Web.
- Le protocole HTTP (Hypertext Transfer Protocol) est un protocole de communication client-serveur conçu pour le Web utilisant généralement le port 80 par défaut.
- HTTP repose sur un mécanisme de requête/réponse :
 - Pour toute requête émise par le client vers le serveur, le serveur doit renvoyer une réponse.
 - C'est pour cela que lorsque vous demandez une page qui n'existe pas sur le serveur, celui-ci vous retourne tout de même une réponse vous indiquant que la ressource demandée n'existe pas.

Les requêtes HTTP

- Dans le protocole HTTP, une méthode est une commande spécifiant un type de requête, c'est-à-dire qu'elle demande au serveur d'effectuer une action.
- En général, l'action concerne une ressource identifiée par l'URL qui suit le nom de la méthode.
 - GET : méthode la plus courante pour demander une ressource sans effet sur la ressource
 - HEAD : méthode qui demande des informations sur la ressource, sans demander la ressource elle-même
 - POST : méthode qui doit être utilisée pour ajouter une nouvelle ressource (un message sur un forum ou un article dans un site)
 - OPTIONS : méthode qui permet d'obtenir les options de communication d'une ressource ou du serveur en général
 - CONNECT : méthode qui permet d'utiliser un proxy comme un tunnel de communication
 - TRACE : méthode qui demande au serveur de retourner ce qu'il a reçu, dans le but de tester et d'effectuer un diagnostic sur la connexion
 - PUT : méthode qui permet de remplacer ou d'ajouter une ressource sur le serveur
 - DELETE : méthode qui permet de supprimer une ressource du serveur

Différentes architectures du .NET Core

- ASP.NET Core MVC
 - Architecture historique d'un projet ASP.NET
 - Standard d'une majorité de framework web
- ASP.NET Core Razor Pages
 - Nouvelle façon de faire du ASP.NET
 - Basée sur l'architecture MVVM

.NET Core CLI

- On va pouvoir très simplement utiliser des lignes de commandes pour toutes sortes d'actions
 - Créer un nouveau projet
 - Gérer les dépendances
 - Lancer le programmes
 - Etc...
- `dotnet new mvc`
- `dotnet build`
- `dotnet run`

.NET Core IDE

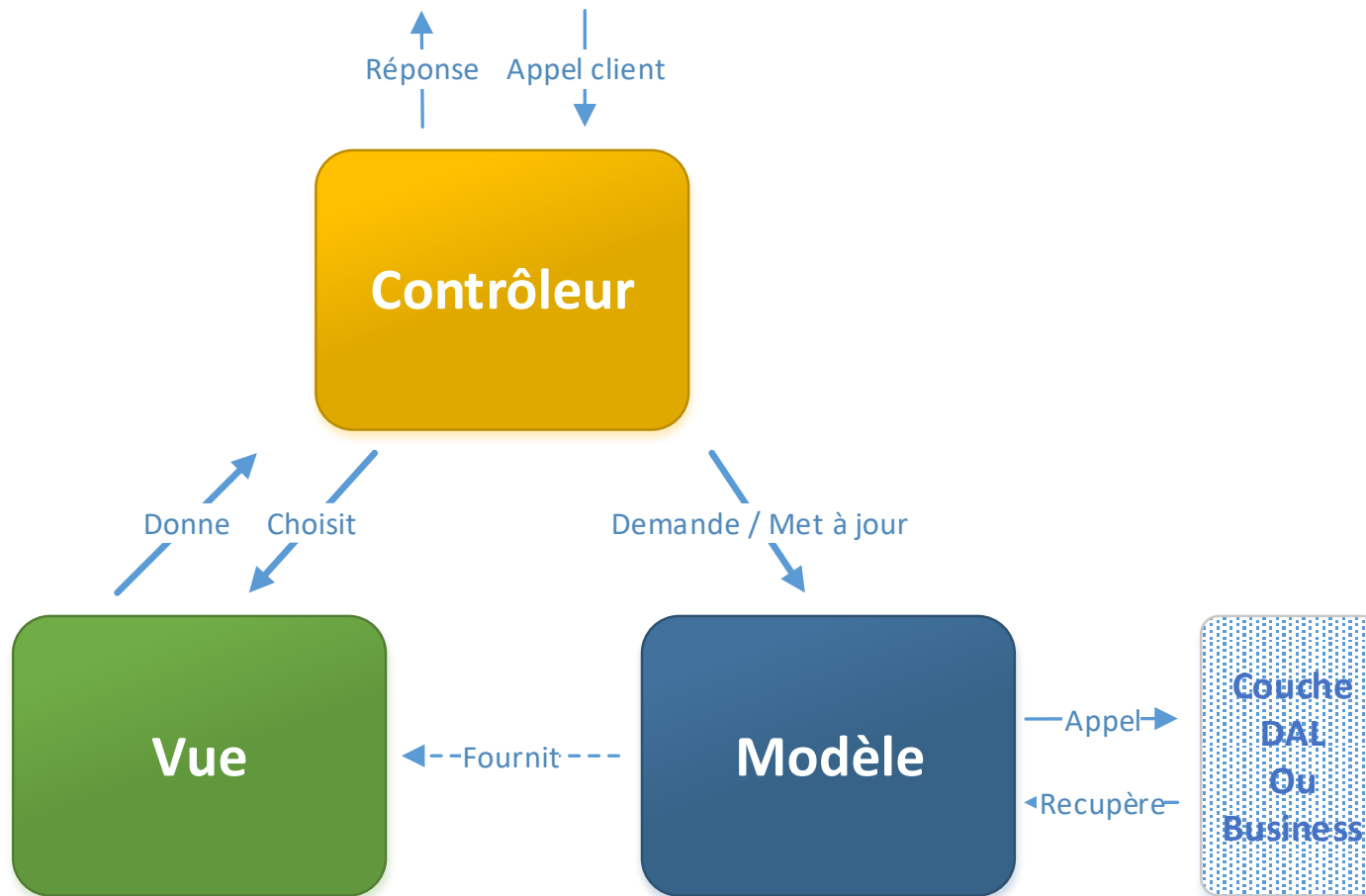
- On pourra toujours utiliser Visual Studio de façon classique ou utiliser d'autres IDE ou éditeur de texte grâce à des plugins ou la CLI

Présentation du modèle MVC

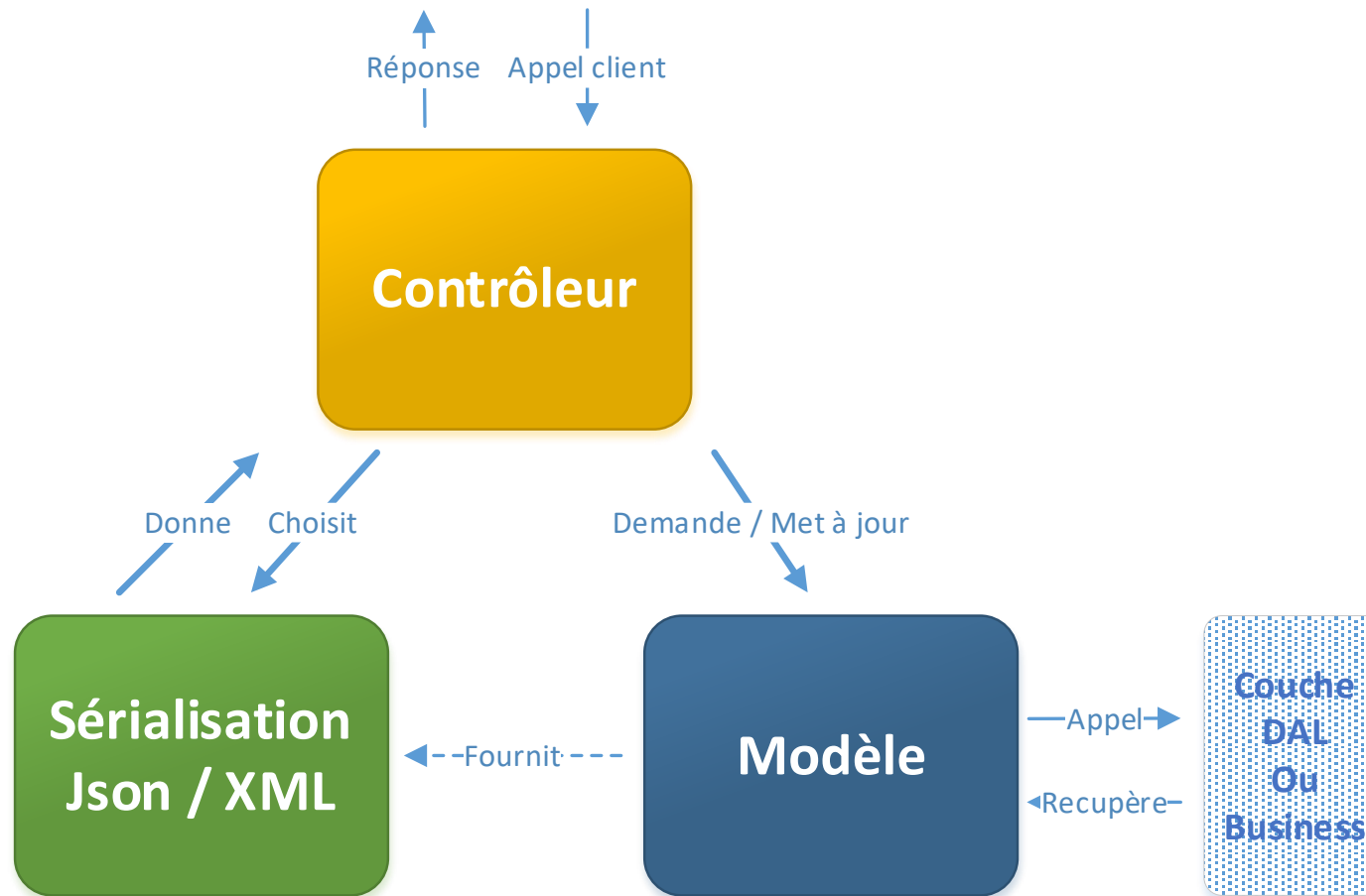
Le Design Pattern MVC

- Le MVC est un Design Pattern qui permet de structurer une application en couches :
 - M (Model) : Il représente les données de l'application à savoir les composants métiers et définit les interactions avec les supports de stockage (base de données, fichiers XML, etc.).
 - V (View) : Elle représente l'interface utilisateur. Elle n'effectue aucun traitement et se contente simplement d'afficher les données que lui fournit le modèle. Il peut y avoir plusieurs vues qui présentent les données d'un même modèle.
 - C (Controller) : Il gère l'interface entre le modèle et le client. Il va interpréter la requête de ce dernier pour lui envoyer la vue correspondante. Il effectue la synchronisation entre le modèle et les vues

Le Design Pattern MVC



Le Design Pattern MVC – mode API

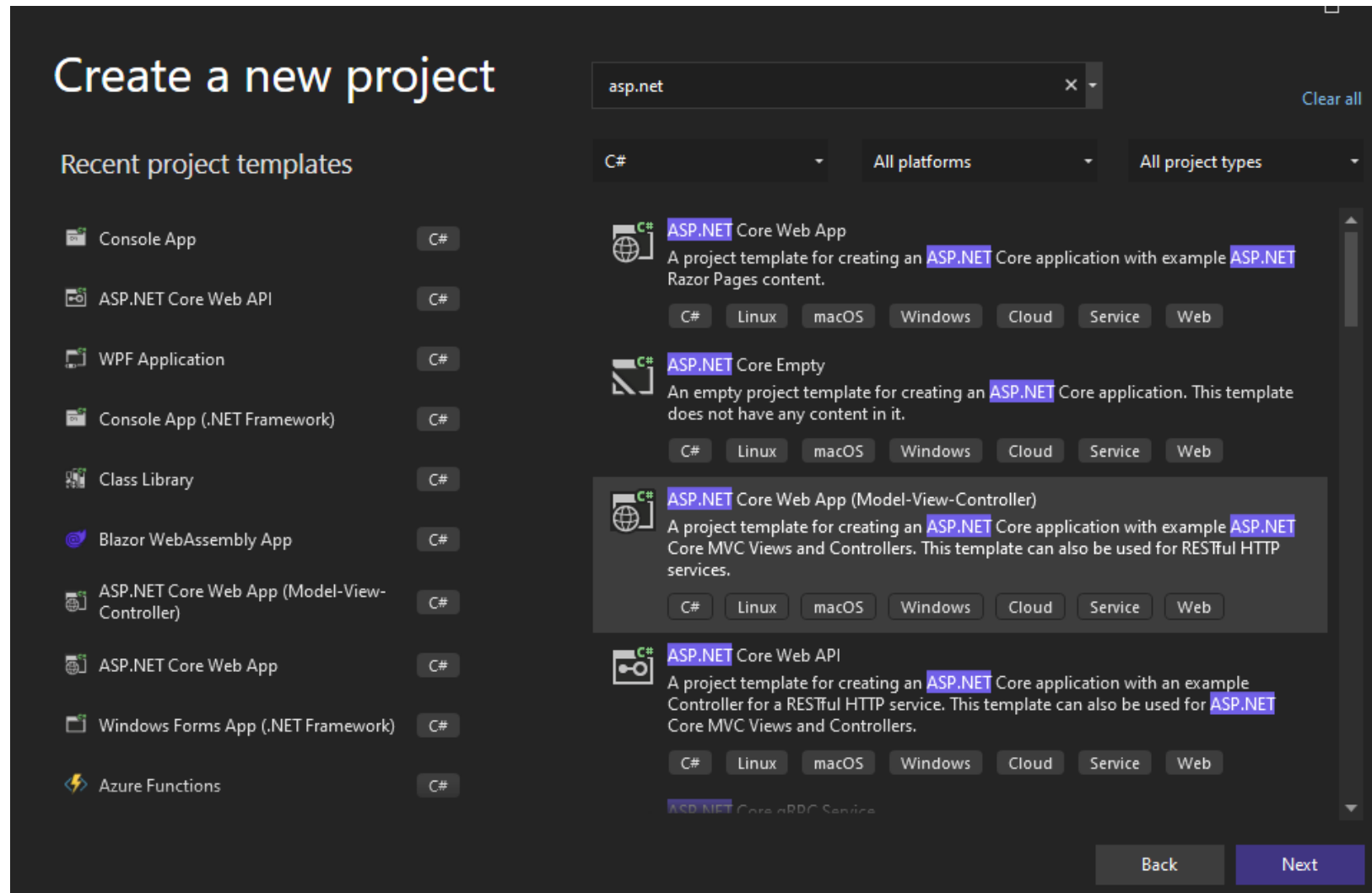


Les différentes versions du modèle MVC

- Les nouveautés d'ASP.NET Core :
 - L'injection de dépendance : Directement intégrée dans le moteur ce qui permet d'éviter les couplages forts.
 - Les Middleware : Les middlewares sont des composants (des classes C#) qui seront chaînés et qui vont intervenir chacun leur tour sur le traitement de la requête HTTP.
 - Les Tag Helpers : Nouveaux tags pour la gestion de la présentation qui remplacent les helpers.
 - Les View Components : Possibilité de créer ces propres composants avec une partie traitement.
 - Unification des contrôleurs MVC et WebAPI.

Création d'une application MVC

Création d'un projet Web MVC



Création d'un projet Web MVC

Framework ⓘ

.NET 6.0 (Long-term support) ▼

Authentication type ⓘ

None ▼

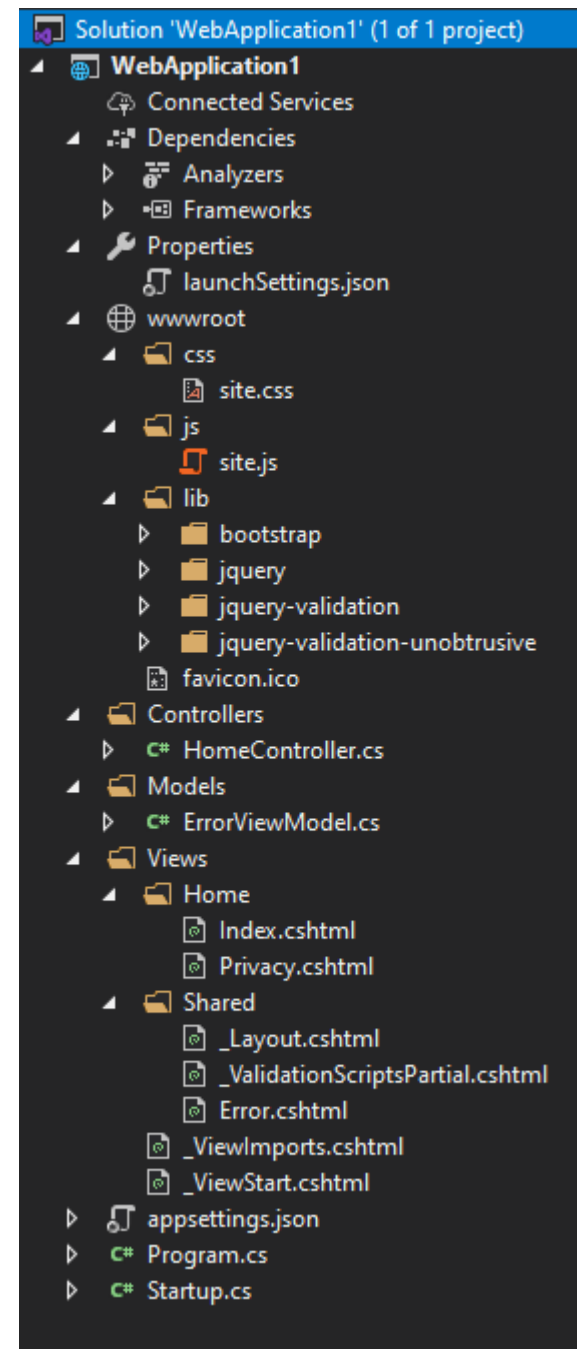
☒ Configure for HTTPS ⓘ

☐ Enable Docker ⓘ

Docker OS ⓘ

Linux ▼

Création d'un projet Web MVC



Le routage

- Le routage ASP.NET permet d'utiliser des URL qui n'ont pas à être mappées à des fichiers dans un site Web.
- Vous pouvez utiliser des URL qui décrivent l'action de l'utilisateur et qui soient par conséquent plus compréhensibles par l'utilisateur.
- Lorsque vous demandez cette URL `/Home/Index/3`, le code suivant est exécuté :
 - `HomeController.Index(3)`

<http://www.unSite.com/Home/Index/3>

Controller

Action

ID

Le routage

- Un itinéraire est un type d'URL qui est mappé à un gestionnaire.
- Vous n'avez généralement pas à écrire de code pour ajouter des itinéraires dans une application MVC mais on peut en rajouter si besoin.
- Les modèles de projets Visual Studio pour MVC incluent des itinéraires d'URL préconfigurés.
- La route par défaut effectue le traitement suivant :
 - Le premier segment d'une URL est lié au nom d'un contrôleur.
 - Le second segment de l'URL est lié à une action d'un contrôleur.
 - Le troisième paramètre est lié à un paramètre nommé id.

Création du contrôleur

- Un contrôleur est chargé de contrôler la façon dont un utilisateur interagit avec une application ASP.NET MVC.
- La création se réalise de la façon suivante :
 - Faites un clic droit sur le dossier **Controllers**.
 - Dans la boîte de dialogue **Add Controller** (Ajouter Contrôleur), entrez le nom de votre contrôleur. Le contrôleur doit toujours posséder un nom qui finit par Controller (HomeController)
 - Cliquez sur le bouton **Add** (Ajouter) pour ajouter le nouveau contrôleur à votre projet.

Le contrôleur

- Il doit réaliser le traitement et déléguer la présentation à la vue

```
public class HomeController : Controller
{ // => Hérite de la classe Controller
    public ActionResult Index()
    {
        // Traitement à réaliser
        // Délégation de l'affichage à la vue
        return View(); // => La vue appelée est la page Index (correspond au
nom de la méthode)
    }
} // => Elle doit se trouver dans le répertoire Views/Home (nom du contrôleur)
```

Structure d'une application MVC

- À chaque contrôleur doit correspondre un répertoire du nom du contrôleur dans le répertoire Views
- Controller
 - HomeController
 - Index (méthode du contrôleur)
- Views
 - Home
 - Index.cshtml
- **Convention Over Configuration**

Appel d'une vue

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View("nomAutreVue");
    }
}
```

Appel d'une vue

- Il est possible d'effectuer une redirection vers une autre Action

```
public ActionResult Index()  
{  
    return RedirectToAction("AutreAction");  
}
```

- La méthode RedirectToAction fait une nouvelle demande et l'URL dans la barre d'adresse du navigateur est mise à jour avec l'URL générée par MVC

Rôle des vues

- Les vues sont conçues exclusivement pour l'encapsulation de la logique de présentation.
- Elles ne doivent pas contenir de logique d'application ni de code de récupération de base de données.
- Une vue restitue l'interface utilisateur appropriée au moyen des données qui lui sont passées par le contrôleur.

Le moteur Razor

- Razor est un moteur de vues qui a été intégré pour ASP.NET MVC 3 et qui est devenu le moteur par défaut.
- L'objectif est de rendre le design des vues plus facile.
 - Minimiser le nombre de caractères requis à l'écriture d'un fichier de vues.
 - Introduire une nouvelle manière de coder qui est fluide et rapide.

@instructions

- La création se réalise de la façon suivante :
 - Faire un clic droit sur le dossier Views
 - Ajouter Vue

Gestion des vues

```
@{ ViewBag.Title = "SimplePage"; }  
<div>  
    <h2>Bienvenue en MVC</h2>  
</div>
```

Communication entre Contrôleur et Vue

- Le contrôleur doit transmettre à la vue les données à présenter.
- Le Framework met à disposition un objet prédéfini ViewData:

```
ViewData["message"] = "Bienvenue au cours MVC";  
return View();
```

```
<h2>@ViewData["message"]</h2>
```

Le ViewBag

- ViewBag est une nouveauté MVC3 qui permet d'échanger des données entre le contrôleur et la vue.
- C'est un "emballage" pour rendre plus simple le stockage et la récupération des paires clé/valeur dans le magasin de stockage utilisé par "ViewData".

```
ViewBag.message = "Bienvenue dans le monde MVC"; // Dans le  
contrôleur  
@ViewBag.message @* dans la vue *@
```

- ViewData et ViewBag représentent les mêmes données.
- ViewBag est simplement un wrapper dynamique autour des mêmes données, vous permettant d'y accéder d'une manière plus commode.
 - Avantage : c'est que nous n'avons pas besoin d'effectuer des "conversions".
 - Inconvénient : Il n'y a pas de prise en charge "IntelliSense" car c'est un type dynamique.

L'objet TempData

- L'objet TempData permet également la transmission de données entre le contrôleur et la vue.
- Il permet de conserver les données lors d'une redirection sur une action. La durée de vie des données est donc plus importante que pour un ViewBag.

```
TempData["message"] = "Bienvenue au cours MVC.net";
```

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

- Vue index appelée par l'action index :

```
<p>@TempData["message"]</p>
```

- La valeur du message sera affichée, ce qui ne serait pas le cas avec un "ViewData" ou un "ViewBag".

Rappel

`//ViewData['SimpleValue'] = ""; => replaced by ViewBag`

`ViewBag.SimpleValue = "Simple value ViewBag";`

- Disponible sur la vue retournée

`TempData["SimpleValue"] = "Simple value TempData";`

- Disponible sur la vue retournée ou une redirection serveur

Démo + Exo 1

Gestion des vues

Présentation de Razor

- Razor est le moteur de rendu utilisé par défaut à partir des applications MVC3.
- Le moteur de vue Razor offre les avantages suivants :
 - Il permet de limiter le nombre de caractères séparant le code serveur du code HTML. Il est compact, expressif et fluide et réduit le code nécessaire à écrire et en facilite la compréhension.
 - Il est facile à apprendre, car nécessite peu de concepts à maîtriser.
 - Il améliore également l'IntelliSense avec Visual Studio.
- Les pages utilisant le moteur Razor ont une extension de type ".cshtml".
- Lorsque l'utilisateur accède à une page Web Razor, le serveur exécute ce code et génère dynamiquement à la volée du code purement HTML qui est envoyé vers le serveur

La syntaxe de base

- Toute instruction Razor commence avec le caractère @.
- Vous pouvez mélanger du texte et des expressions Razor

<p>Nous sommes le @DateTime.Now.ToShortDateString()</p>

- Il est possible de distinguer deux grands groupes de types de syntaxe utilisables avec Razor.
 - Les instructions uniques : instructions qui sont évaluées et directement rendues en plein cœur du contenu statique. : @Instructions
 - Les blocs de codes : Il s'agit de sections entières qui ne peuvent contenir que du code et aucun contenu statique. @ { instructions ; }

Les blocs de code

- Le code qui est écrit à l'intérieur d'un bloc de code doit respecter toutes les règles du langage cible du gabarit Razor courant, c'est-à-dire C# ou Visual Basic .NET.
- Dans le cas d'un gabarit C#, les différentes instructions doivent donc obligatoirement se terminer par un point-virgule.
- À l'inverse des instructions isolées, elles n'impliquent pas forcément de sortie qui se retrouverait dans la classe générée par Razor.

```
@{
```

```
int compteur = 1; string nom = "test"; var Message = "Hello";
```

```
}
```

```
Message : @Message
```

- Les commentaires en Razor se présentent sous la forme suivante :

```
@* Commentaires *
```

Les structures de contrôle

- Razor supporte la plupart des structures conditionnelles qui sont utilisables en C# ou en Visual.
- La structure if permet d'effectuer un traitement en fonction de l'évaluation d'une condition.

```
@if (compteur == 0)
{
    <p> Pas d'utilisateur connecté </p>
}
else
{
    <p> @compteur </p>
}
```

Les structures de contrôle

- La structure switch existe comme dans le langage C#

```
@switch (compteur)
{
    case 0:<p>pas d'utilisateur connecte </p> break;
    case 1: <p>un utilisateur connecté </p> break;
    default: <p>plusieurs utilisateurs connectés </p>break;
}
```


Les structures de contrôle

- Razor met à disposition un ensemble de structures itératives permettant de parcourir les collections :
 - La boucle foreach :

```
@foreach (string mess in messages)
{
    <p>@mess</p>
}
```

- Les boucles "while" et "do while".

Fonctionnement de Razor

- L'analyseur de syntaxe de Razor arrive à faire la distinction entre le code et le contenu statique:
 - lorsque celui-ci est de type XML ou d'un de ses dérivés.
- À l'intérieur des structures de contrôles, on peut trouver autant de code que de contenu statique

```
@if ( ... ) { <p>Bonjour</p> }  
@if ( ... ) { var contenu = "bonjour"; @contenu }
```

Fonctionnement de Razor

- Attention, le scénario suivant ne fonctionne pas car il n'y a pas de signe distinctif permettant à Razor de détecter que le contenu du bloc if est un contenu statique et non du code.

```
@if ( ... ) { Bonjour }
```

- Il faut utiliser

```
<text>Bonjour</text>
```

```
<span>Bonjour</span>
```

Les Helpers

- Pour la création des vues, le framework propose des méthodes d'assistance qui offrent un moyen simple pour la génération du rendu HTML.
- Ces méthodes sont fournies par la classe `HtmlHelper`.
- Toutes les méthodes `HtmlHelper` génèrent du code HTML et retournent le résultat sous forme de chaîne.
- Les méthodes d'extension pour la classe `HtmlHelper` se trouvent dans l'espace de noms `System.Web.Mvc.Html`.
- Les méthodes d'assistance et d'extension sont appelées à l'aide de la propriété `Html` de la vue, qui est une instance de la classe `HtmlHelper`.

Fonctionnement des Helpers

- Une fois compilée, chaque vue MVC hérite en fait d'une classe générique `WebViewPage` qui possède la propriété suivante :

```
public HtmlHelper<TModel> Html { get; set; }
```

- C'est pourquoi les méthodes de la classe `HtmlHelper` sont directement disponibles dans une vue, via la propriété `Html`.
- Utiliser les méthodes de la classe `HtmlHelper` ne correspond à rien d'autre qu'à simplement renvoyer une chaîne de caractères qui sera interprétée par le moteur de vue de façon à écrire cette chaîne dans le flux de la réponse au navigateur.

Création de lien en Razor

- La réalisation de lien se fera à l'aide de la méthode

```
public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,  
string linkText, string actionName, string controllerName);
```

- Exemple :

```
@Html.ActionLink("Info Message", "GetInfoMessage", "Message")
```

- Le code généré sera sous la forme suivante :

```
<a href="/Message/GetInfoMessage">Info Message</a>
```

Création de liens avec paramètres

- La méthode `ActionLink` permet de passer facilement des paramètres à l'action du contrôleur

```
@Html.ActionLink("Action avec paramètre", "ActionAvecParametre",  
"Home", new { id = 5 + 5, value = "ma valeur" }, null)
```

```
<a href="/Home/ActionAvecParametre/10?value=ma%20valeur">Action avec  
paramètre</a>
```

```
public ActionResult ActionAvecParametre(int id, string value)
```

Les principales méthodes de la classe Helper

- Le framework possède d'autres méthodes d'extension :
 - BeginForm : marque le début d'un formulaire et établit un lien vers la méthode d'action qui restitue le formulaire.
 - CheckBox : case à cocher.
 - DropDownList: liste déroulante.
 - ListBox : restitue une zone de liste.
 - Password : zone de texte pour l'entrée d'un mot de passe.
 - RadioButton: case d'option.
 - TextArea : zone de texte multiligne.
 - TextBox : zone de texte.

La nouveauté ASP.NET Core: les tags helpers

- Microsoft a développé une nouvelle fonctionnalité dont le but est de simplifier et d'alléger l'écriture de vues avec Razor : Les Tag Helpers.
- Ces Tag Helpers permettent d'exécuter du code côté serveur afin de générer du HTML, mais en gardant le standard HTML
 - Là où, avec le HTML Helper, le développeur utilisait des méthodes C#, les Tag Helpers permettent d'intégrer directement de la logique métier dans les éléments HTML.
- La notion de TagHelper est directement implémentée au niveau de Razor

```
<a asp-area="" asp-controller="Home" asp-action="ActionAvecParametre"  
asp-route-id="10" asp-route-value="ma valeur">Action avec  
paramètre</a>
```

Développement de Helpers

```
public class EmailTagHelper : TagHelper {  
    private const string EmailDomain = "contoso.com";  
  
    // Can be passed via <email mail-to="..." />.  
    // PascalCase gets translated into kebab-case.  
  
    public string MailTo { get; set; }  
  
    public override void Process(TagHelperContext context, TagHelperOutput output) {  
        output.TagName = "a";    // Replaces <email> with <a> tag  
  
        var address = MailTo + "@" + EmailDomain;  
  
        output.Attributes.SetAttribute("href", "mailto:" + address);  
        output.Content.SetContent(address);  
    }  
}
```

Développement de Helpers

- Chargement dans `_ViewImports`

```
@addTagHelper *, WebApplication1
```

- Utilisation dans la vue

```
<Email mail-to="test"></Email>
```

- Génèrera

```
<a href="mailto:test@contoso.com">test@contoso.com</a>
```

Exemple d'un model

```
public class Message
{
    public string Emetteur { get; set; }
    public string Contenu { get; set; }
    public DateTime Date { get; set; }
}
```

Exemple

```
<form asp-action="Index">
    <div>
        <label for="Emetteur">Emetteur :</label>
        <input type="text" name="Emetteur" />
    </div>
    <div>
        <label for="Contenu">Contenu :</label>
        <input type="text" name="Contenu" />
    </div>
    <button>Valider</button>
</form>
```

[HttpPost]

```
public ActionResult Index(string Emetteur, string Contenu)
```

Exemple

```
<form asp-action="Index">
    <div>
        <label for="Emetteur">Emetteur :</label>
        <input type="text" name="Emetteur" />
    </div>
    <div>
        <label for="Contenu">Contenu :</label>
        <input type="text" name="Contenu" />
    </div>
    <button>Valider</button>
</form>
```

[HttpPost]

```
public ActionResult Index(Message message)
```

La directive model

- Il existe une directive pour indiquer que vous souhaitez utiliser les classes du modèle fortement typées dans vos fichiers de votre vue.
- Cette ligne se met dans le fichier cshtml, généralement tout en haut
- On ne peut avoir qu'un seul model par page

`@model` MonProjet.Models.Message

Les helpers génériques

```
<input asp-for="Contenu"/>
```

```
<input id="Contenu" type="text" name="Contenu">
```


Les champs cachés

```
<input asp-for="Id" type="hidden" value="10" />
```

Gestion des labels

- Il est possible de lier le nom d'un label au nom de la propriété de la classe Métier avec la méthode :

```
<label asp-for="Theme"></label>
```

```
<input asp-for="Theme"/>
```

- Il est possible de paramétrer le libellé dans le classe métier à l'aide de l'annotation Display :

```
[Display(Name="Contenu à afficher")]
```

```
public string Contenu { get; set; }
```

Réalisation de formulaire avec HtmlHelper Générique

```
@model MvcApp1.Models.Message
<form asp-action="Index">
    <div>

        <label asp-for="Emetteur"></label>

        <input asp-for="Emetteur"/>
    </div>
    <div>

        <label asp-for="Contenu"></label>

        <input asp-for="Contenu"/>
    </div>
    <button>Valider</button>
</form>

[HttpPost]

public ActionResult Index(Message m)
```

Les helpers génériques

- Les extensions génériques sont à préférer lorsque la vue est fortement typée, pour deux raisons principales :
 - Lorsqu'une propriété change de nom, des erreurs peuvent alors être détectées dans la vue à la compilation
 - Pour l'utilisation de type complexe, on peut accéder directement à l'élément associé :

```
<input asp-for="Emetteur.Nom" class="styleMessage"/>
```

Affichage de modèle

- Grâce au `@model` on pourra aussi afficher des informations du contrôleur vers la vue fortement typés

```
var message = new Message { Contenu = "content1" };  
  
return View(message);
```

```
@model SupportCours.Models.Message  
<div>  
    @Model.Contenu  
</div>
```

Affichage de modèle

- On peut aussi typer notre vue en liste

```
var messages = new List<Message>();  
return View(messages);
```

```
@model IEnumerable<SupportCours.Models.Message>  
<div>  
    @foreach (var message in Model)  
    {  
        <p>@message.Contenu</p>  
    }  
</div>
```

Gestion des listes

- Il existe des Helpers pour la gestion des listes.
- On peut associer une collections d'objets métier à une liste en utilisant la classe SelectList.

```
<select asp-for="Emetteur"  
        asp-items="@ (new  
        SelectList(ViewBag.Emetteurs))">  
    </select>
```

Gestion des listes

```
<select name="Emetteur">  
    @foreach (WebApplication1.Models.Emetteur emetteur in  
        ViewBag.ListeEmetteurs)  
    {  
        <option value="@item.Id">@item.Name</option>  
    }  
</select>
```


Réalisation des vues

- Visual Studio permet de créer des vues en s'appuyant sur un modèle à l'aide des vues fortement typées

Nom de la vue : View

Modèle : Create

Classe de modèle : Message (SupportCours.Models)

Classe de contexte de données :

Options :

☐ Créer en tant que vue partielle

☐ Bibliothèques de scripts de référence

☒ Utiliser une page de disposition :

(Laissez vide s'il est défini dans un fichier Razor _viewstart)

Ajouter Annuler

Les vues fortement typées

- Elles proposent différents modes de présentation :
 - List : permet d'afficher une liste d'objets
 - Details : affiche les valeurs d'un objet unique
 - Delete : permet de supprimer un élément de la liste
 - Edit : permet la modification des valeurs de l'objet
 - Create : pour la création d'un objet

Passer une liste en post

- Pour passer une liste dans un post il suffit d'avoir plusieurs input du même name

```
<form asp-action="Index">
    <label>value 1</label>
    <input type="text" name="vals" value="" />
    <br />
    <label>value 2</label>
    <input type="text" name="vals" value="" />
    <br />
    <button>Valider</button>
</form>
```

```
public ActionResult ActionSimpleAvecListeFormulaire(List<string>
vals)
```

Passer une liste d'objets en post

- Pour passer une liste dans un post il suffit d'avoir plusieurs input du même name

```
<form asp-action="Index">
    @for (int i = 0; i < 3; i++)
    {
        <span>Message @i</span>
        <br />
        <input type="text" name="messages[@i].Contenu" />
        <input type="text" name="messages[@i].Emetteur" />
        <br />
    }
    <br />
    <button>Valider</button>
</form>

public ActionResult ActionSimpleAvecListeFormulairePlusModel
(List<Message> messages)
```

Résumé : Utilisation des bons paramètres

- Il faut toujours bien regarder la surcharge des méthodes appelées (survol de la souris dans VS)
- Quand un paramètre attendu est une `RouteValues`, vous devrez avoir quelque chose comme `new { MonParam = MaValeur }` avec `MonParam` qui est aussi un param de l'action ciblé (le nom doit être le même)
- Quand un paramètre est un `Model`, vous devez lui envoyer une variable (`article`, `article.Commentaire`, etc...) du type du model de la vue qui est ciblée

Résumé : Passage d'informations du contrôleur vers la vue

- **ViewBag.KEY** : à utiliser dans une action qui effectue un `return View();`
 - Utilisation dans la vue : `@ViewBag.Key`
- **TempData["KEY"]** : à utiliser dans une action qui fait une redirection `RedirectToAction("AutreAction")`
 - Utilisation dans la vue : `@TempData["KEY"]`
- **@model** : Passage d'un objet à la vue via le Model
 - Dans le contrôleur: `return View(MonObjet);`
 - Dans la vue: `@model MonProjet.Models.Message`
 - Utilisation dans la vue : `@Model` (qui correspond à l'objet passer en paramètre)
 - `@Model.Contenu`

Résumé : Passage d'informations de la vue vers le contrôleur

- Formulaire simple avec des inputs dont le nom correspondra au nom du paramètre attendu dans l'action du contrôleur

```
<input type="text" name="value1" value="" />
```

```
public ActionResult MonAction(string value1)
```

- Formulaire avec model avec des inputs génériques `<input asp-for="Theme" />` et l'action du contrôleur récupèrera directement l'objet

```
public ActionResult ActionAvecModel(Message message)
```

- Avec des liens HTML : `<a asp-controller="Home" asp-action="ActionAvecParametre" asp-route-id="10" asp-route-value="ma valeur">Action avec paramètre`: éléments dont le nom correspondra au nom du paramètre attendu dans l'action du contrôleur

```
public ActionResult ActionAvecParametre(int id, string value)
```

Résumé : GET vs POST

- GET
 - Les requêtes sont envoyées via des liens href
 - Par défaut les actions des contrôleurs sont en GET
 - les paramètres sont passés dans l'URL
 - L'URL étant limité en longueur on passe généralement que des paramètres de type primitifs (string, int, bool, etc...)
 - <http://localhost:61983/Home/ActionAvecParametre/42?value=ma%20valeur>
 - <https://www.google.com/search?q=hello+world&ie=utf-8&oe=utf-8&client=firefox-b-ab>

Résumé : GET vs POST

- POST
 - Les requêtes sont envoyées via des formulaires
 - Pour que les actions des contrôleurs soient en POST on ajoute l'attribut [HttpPost]
 - les paramètres sont passés en « cachés » dans la requête
 - Aucune limite de taille donc on peut passer des objets complexes (Message, Article, etc...)

Contenu :

Emetteur :

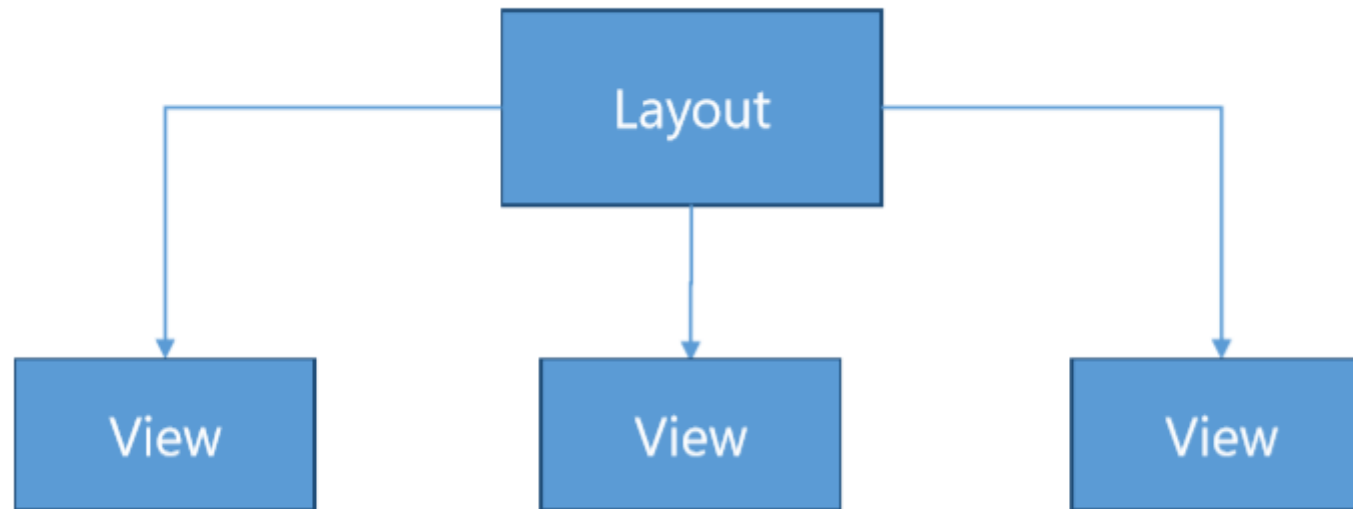
Date :

Démo + Exo 2

Application de styles dans les applications

Les Layouts

- Le moteur de Razor permet de créer des modèles de présentation pour une application web.
- Un des intérêts est de pouvoir appliquer un même modèle à différentes pages.



Gestion de modèles

- Quand vous créez un projet web, un layout standard est proposé dans le répertoire shared : _Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    ...
</head>
<body>

    <div class="container">

        @RenderBody()

    </div>

    <footer class="border-top footer text-muted">

        &copy; 2022 - WebApplication1 - <a asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>

    </footer>

</html>
```

Gestion des layouts

- La mise en œuvre d'un layout se réalise à l'aide de la balise :

`@RenderBody()`

- Cette méthode va permettre de déterminer l'endroit où le code de la page utilisant le layout va être rendu.
- Cette méthode ne peut être appelée qu'une seule fois.
- Il est possible de définir plusieurs zones de rendu dynamiques avec les sections.
- Une section est une zone de la page qui va être définie ou redéfinie au niveau de la vue qui appelle le layout.

Exemple des sections

```
@* LAYOUT *@  
<body>  
    @RenderBody()  
    <footer> @RenderSection("footer", required: false) </footer>  
    </html>  
</body>
```

```
@* PAGE *@  
<h2>Exemple de page</h2>  
@section footer {  
    <p>@DateTime.Now.Year - My ASP.NET Application</p>  
}
```

Gestion des sections

- Il est possible de tester la présence d'une section permettant ainsi d'avoir une partie statique ou une partie renseignée dynamiquement

```
@RenderBody()  
@if (IsSectionDefined("footer"))  
{  
    @RenderSection("footer")  
}  
else  
{  
    <div><p>@DateTime.Now.Year - My ASP.NET Application</p></div>  
}
```


La vue _ViewStart

- Dans chaque vue, vous devez indiquer le fichier layout à utiliser

```
@{
```

```
    Layout = "~/Views/Shared/MonLayout.cshtml";
```

```
}
```

- Razor propose un mécanisme supplémentaire, la vue _ViewStart, pour gérer l'association des pages avec un layout.
- ViewStart est en effet appelée avant chaque rendu d'une vue se trouvant dans son répertoire ou un de ses sous-répertoires.

```
@{
```

```
    Layout = "~/Views/Shared/_Layout.cshtml";
```

```
}
```

Gestion de la partie cliente

- La gestion de la présentation se réalise à l'aide des CSS.
- Les pages peuvent également contenir du code JavaScript pour la partie dynamique.
- Le lien avec la/les feuilles de style et les fichiers javascript se fera depuis le layout.
- L'utilisation des framework (Bootstrap, JQuery, ...) entraine l'intégration d'une multitude de fichiers en générant des sous-requêtes qui peuvent impacter le temps de chargement.
- Avant ASP.NET Core, on utilisait la classe BundleConfig pour minifier et concaténer les fichiers

Gestion de la partie cliente

- ASP.NET Core intègre par défaut 2 outils simples
 - Libman.json: Permet de télécharger très facilement des librairies externes
 - Sur votre projet, clic droit, Add, Client Side Library (disponible aussi en ligne de commande)
 - bundleconfig.json: Permet de minifier et concaténer des fichiers JS / CSS
 - Nécessite l'extension Visual Studio Bundler & Minifier
 - Sur le fichier à minifier, clic droit, Bundler & Minifier

Gestion de la partie cliente

- D'autres outils pourront être aussi utilisés:
 - NuGet : Recommandé pour les librairies serveur uniquement
 - Bower : N'est plus pris en charge par son concepteur
 - Npm : Un des plus utilisé
 - Yarn : Un nouveau concurrent rapide et sécurisé
 - Gulp : On peut convertir un bundleconfig en Grunt via Visual studio (clic droit sur le fichier json), Grunt sera utile si en plus de la minification et de la concaténation on veut de la compilation de fichier SASS ou LESS (Bootstrap) par exemple

Gestion de la partie cliente

- On ajoutera ensuite sur notre layout le chargement des librairies générées

```
<environment include="Development">
```

```
    <script src="~/js/site.js" asp-append-version="true"></script>
```

```
</environment>
```

```
<environment exclude="Development">
```

```
    <script src="~/js/site.min.js" asp-append-version="true"></script>
```

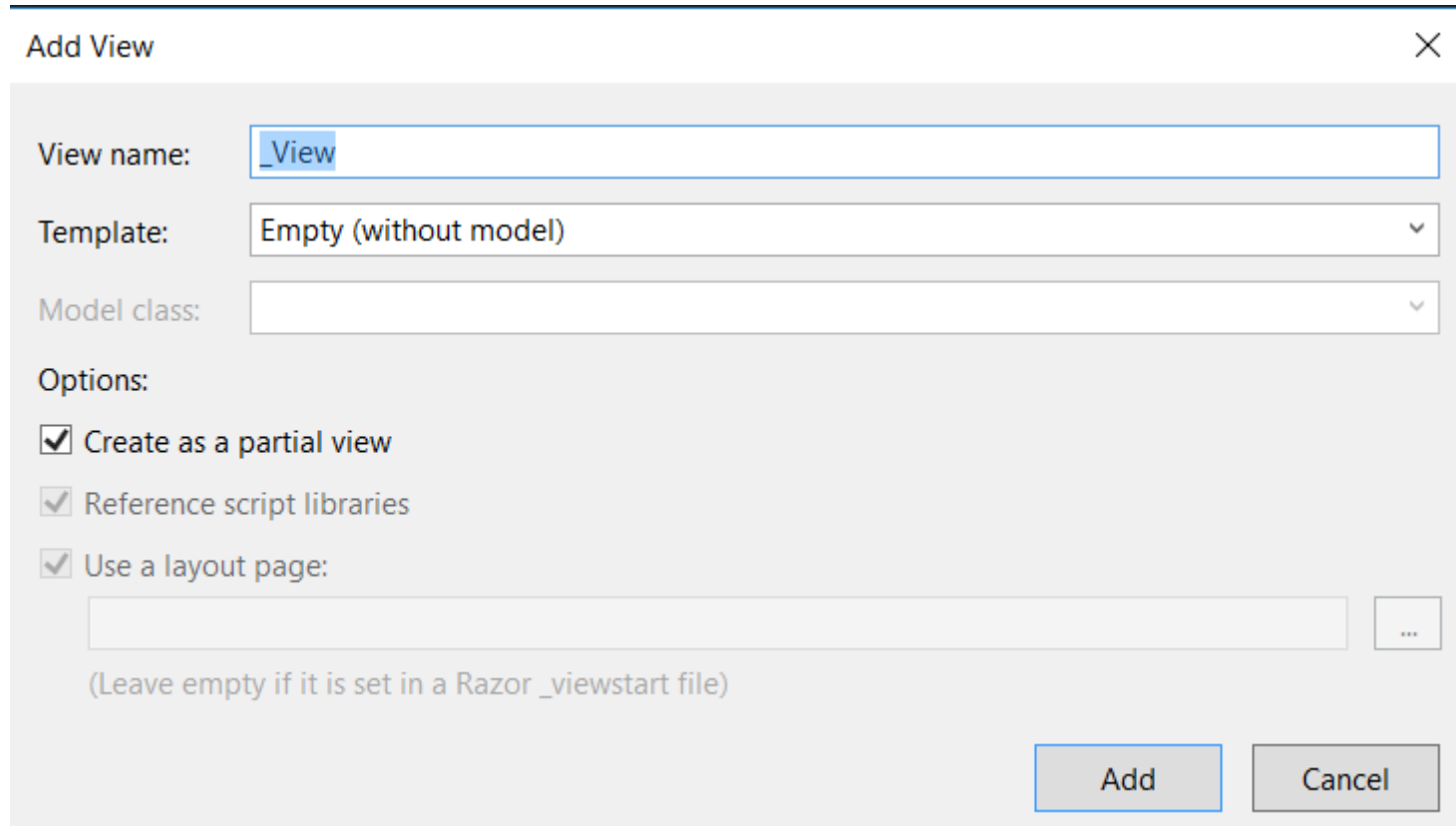
```
</environment>
```

- La gestion de l'environnement est faite dans le fichier launchSettings, sous Properties

Les vues partielles

Les vues partielles

- Le moteur MVC nous permet de réaliser des composants visuels réutilisables :
 - les vues partielles



The screenshot shows the 'Add View' dialog box with the following fields and options:

- View name:**
- Template:**
- Model class:**
- Options:**
 - ☒ Create as a partial view
 - ☒ Reference script libraries
 - ☒ Use a layout page:
 ...

(Leave empty if it is set in a Razor _viewstart file)

Buttons: **Add** and **Cancel**

Les vues partielles

- Une vue partielle est une portion de vue que l'on pourra intégrer dans une autre vue
- Les vues partielles vont être réutilisées à plusieurs endroits de votre application
- Elle devront donc être créées dans le dossier Shared, ce dossier étant accessible par toutes les autres vues
- Par convention de nommage on mettra un `_` en préfixe (`_AddCommentaire`, `_AfficherListeCommentaire`)
- La vue partielle peut avoir un modèle différent du modèle de la vue qui charge la vue partielle

Les vues partielles

- Une vue partielle est une portion de vue que l'on pourra intégrer dans une autre vue :
 - un helper qui permet de renvoyer une vue sous la forme d'une chaîne de caractères. Cette chaîne peut ainsi être utilisée à l'intérieur d'une autre vue pour produire du HTML.

```
@Html.Partial("_CreationMessage")  
@Html.Partial("_CreationMessage", Model)  
@Html.Partial("_CreationMessage", Model.ListeCommentaires)  
@Html.Partial("_CreationMessage", new Comment{ ArticleId = Model.Id })
```

```
<partial name="_CreationMessage" model="Model" />
```

- Il est possible de passer un modèle à une vue partielle

View components

View components

- Un view component est un composant visuel comme une vue partielle mais qui contient en plus sa propre partie métier
- Une vue partielle charge directement un cshtml
- Un view component appelle une classe C# qui pourra faire des traitements et ensuite retourne un cshtml (sans layout comme avec la vue partielle)
- Lors de l'appel du view component on pourra envoyer des paramètres
- Il intègre un lien Vue Contrôleur découpé en composants
- Il est remplacé de façon plus standardisé le @Html.Action

View components

```
public class MyViewComponent : ViewComponent
{
    public async Task<IViewComponentResult> InvokeAsync()
    {
        // return View("NomVue")
        // return View(monModel)
        return View();
    }
}
```

View components

- Les vues se trouveront dans le répertoire Views/Shared/Components/My/Default.cshtml
- Default est le nom de la vue par défaut, on pourra changer ce nom ou avoir plusieurs vues et retourner le nom de la vue en paramètre

```
<span>Hello world</span>
```

View components

- Pour utiliser le ViewComponent on pourra passer par un helper C#, un TagHelper ou dans un contrôleur
 - Pour le tag helper il faudra rajouter au ViewImports:
 - `@addTagHelper *, NomDuProjet`

// Dans une vue

```
@await Component.InvokeAsync("Random", new { Min = 1, Max = 100 })
```

```
<vc:random min="10" max="20"></vc:Random>
```

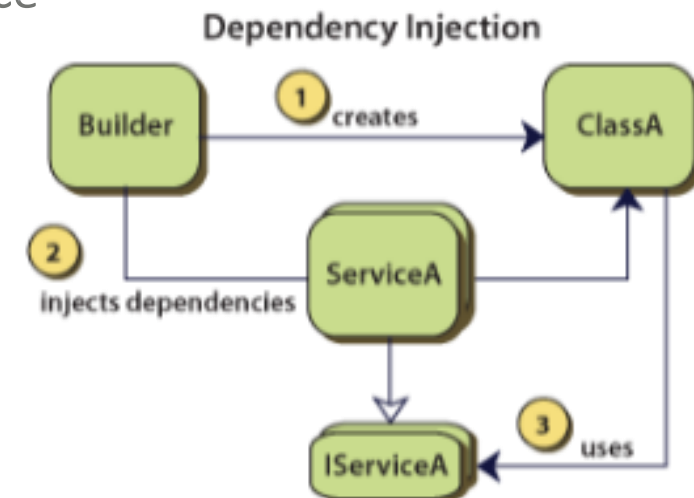
// Dans une action (pour l'appel du component par de l'ajax par exemple

```
return ViewComponent("Random", new { Min = 1, Max = 100 });
```

Injection de dépendance

Inversion Of Control (IOC)

- L'inversion de contrôle est un patron d'architecture commun à tous les frameworks
- L'objectif va être de limiter le couplage fort entre 2 classes
- L'IOC peut se faire sous 2 façon
 - L'instance locator
 - L'injection de dépendance
- ASP.NET Core propose nativement un système d'injection de dépendance



Les services

- Les services ASP.NET sont des classes qui nous rendent services
- Mis à part les modèles et les contrôleurs toutes les autres classes doivent être considéré comme des services
- Les services vont pouvoir être injectés très facilement dans les contrôleurs ou dans les autres services

Les services

```
public class RandomService
{
    readonly Random random = new Random();
    public int GetRandom(int min = 0, int max = 100)
    {
        return random.Next(min, max);
    }
}
```

Déclaration du service

- Chaque service doit être déclaré au démarrage de l'application dans la partie de configuration des services dans la classe Program.cs (l'ordre n'est pas important)

```
// Add services to the container.
```

```
builder.Services.AddScoped<RandomService>();
```

```
// Specifies that a single instance of the service will be created.
```

Singleton

```
// instance of the service will be created for each scope, a scope is created around  
each server request.
```

Scoped

```
// that a new instance of the service will be created every time it is requested.
```

Transient

Injection du service

- Pour utiliser ce service dans un contrôleur ou dans un autre service on va le rajouter en tant que paramètre du constructeur

```
readonly RandomService randomService;  
public TestController(RandomService randomService)  
{  
    this.randomService = randomService;  
}
```

// Utilisation dans les méthodes

```
randomService.GetRandom(min, max)
```

Couplage fort

- Dans l'exemple précédent le contrôleur et le service sont fortement couplé
- Si on change l'implémentation du service le contrôleur risque de changer aussi
- Pour permettre une plus grand évolutivité de notre code nous allons passer par une interface intermédiaire

```
public interface IRandomService
{
    int GetRandom(int min, int max);
}
```

Couplage fort

- Il faudra changer la configuration du service
- `builder.Services.AddScoped<IRandomService, RandomService>();`
- Il faudra changer l'injection dans le constructeur
- `public TestController(IRandomService randomService)`

Couplage fort

- Si notre RandomService doit changer le contrôleur n'aura pas à changer

```
public class CryptoRandomService : IRandomService
{
    public int GetRandom(int min, int max)
    {
        return RandomNumberGenerator.GetInt32(min, max);
    }
}
```

Couplage fort

- Changement de l'implémentation dans toute l'application

```
//builder.Services.AddScoped<IRandomService, RandomService>();  
builder.Services.AddScoped<IRandomService, CryptoRandomService>();
```


Services intégrés

```
private readonly ILogger<HomeController> _logger;  
public HomeController(ILogger<HomeController> logger)  
{  
    _logger = logger;  
}  
public IActionResult Index()  
{  
    _logger.LogDebug("ERROR");  
    return View();  
}
```

Injection de dépendance

- Injection dans la vue

```
@inject IRandomService RandomService
```

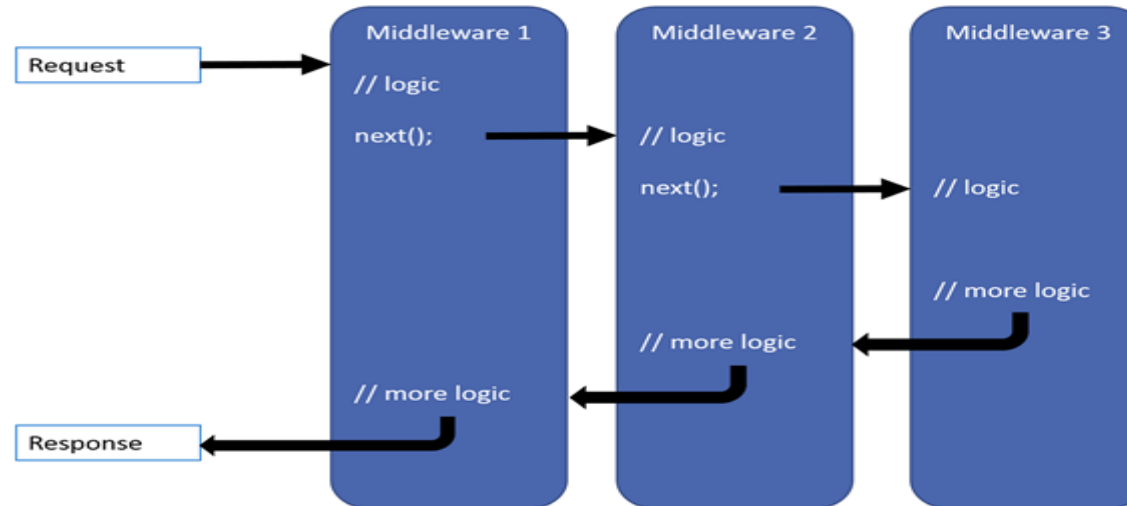
Middleware

Middleware

- Les middlewares sont un nouveau concept introduit avec ASP.NET Core
- Les middlewares sont des composants qui seront chaînés et qui vont intervenir chacun leur tour sur la requête HTTP
 - Chaque middleware va ainsi décider s'il laisse passer ou non la requête au middleware suivant en fonction de l'action qu'il vient d'effectuer sur la requête
 - Le middleware pourra aussi modifier la réponse
- Dans le cadre d'une application ASP.NET Core, les middlewares composent la chaîne de traitement (le pipeline) des requêtes HTTP. Ils interagissent sur la requête et la réponse HTTP.

Middleware

- La réponse HTTP se construit uniquement lors du passage de la requête dans cette suite de middleware.



- L'intérêt du court-circuitage de la requête par les middlewares est d'éviter des traitements inutiles.
 - Si l'utilisateur veut accéder à une page qui demande une authentification et si le middleware détecte que cet utilisateur ne l'est pas, il va automatiquement rediriger l'utilisateur vers la page de connexion sans passer la main aux middlewares suivants.

Middleware

```
// L'ordre est important car il définit l'ordre d'appel
if (!app.Environment.IsDevelopment()){
    app.UseExceptionHandler("/Home/Error");

    // The default HSTS value is 30 days. You may want to change this for production scenarios, see
    https://aka.ms/aspnetcore-hsts.

    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Middleware

- On peut lancer nos propres middleware
- Il existe 3 méthodes d'appels
 - Use: permet d'utiliser un middleware et d'appeler les suivants (ou pas si l'on veut arrêter la requête)
 - Run: définira le middleware final, pas d'appel au suivant
 - Map: redirige vers un autre pipeline

Middleware

```
app.Use(async (context, next) => {  
    await context.Response.WriteAsync("Avant 1er middleware, ");  
    await next.Invoke();  
    await context.Response.WriteAsync("Après 1er middleware, ");  
});  
  
app.Map("/Map", (map) => {  
    map.Run(async (context) => {  
        await context.Response.WriteAsync("Avant Map, "); // Utilisé avec l'url / Map Avant 1er  
        middleware, Avant Map, Après 1er middleware,  
    });  
});  
  
app.Run(async (context) => { await context.Response.WriteAsync("2eme middleware, "); });  
  
// Sans /Map: Avant 1er middleware, 2eme middleware, Après 1er middleware,
```


Exemple

```
public class Redirect404Middleware {  
    private readonly RequestDelegate _next;  
  
    public Redirect404Middleware(RequestDelegate next) { _next = next; }  
  
    public async Task InvokeAsync(HttpContext context) {  
        await _next(context);  
  
        if (context.Response.StatusCode == 404)  
            context.Response.Redirect("/Home/NotFoundCustom?urlRequest=" +  
context.Request.Host.Value + context.Request.Path, false) ;  
    }  
}
```

Exemple

```
public static class Redirect404MiddlewareExtension {  
    public static IApplicationBuilder UseRedirect404(  
        this IApplicationBuilder builder) {  
        return builder.UseMiddleware<Redirect404Middleware>();  
    }  
}
```

// Dans le Startup.cs

```
app.UseRedirect404();
```

Les filtres

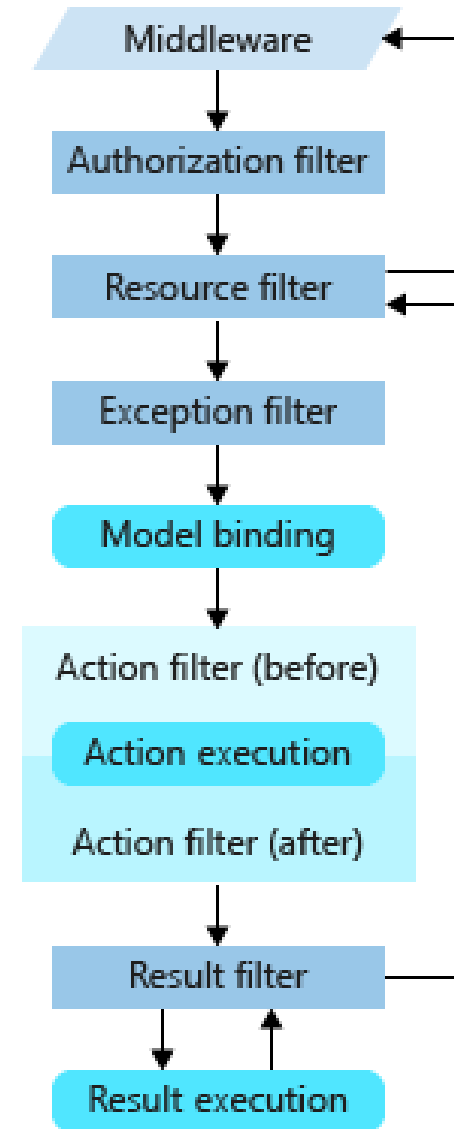
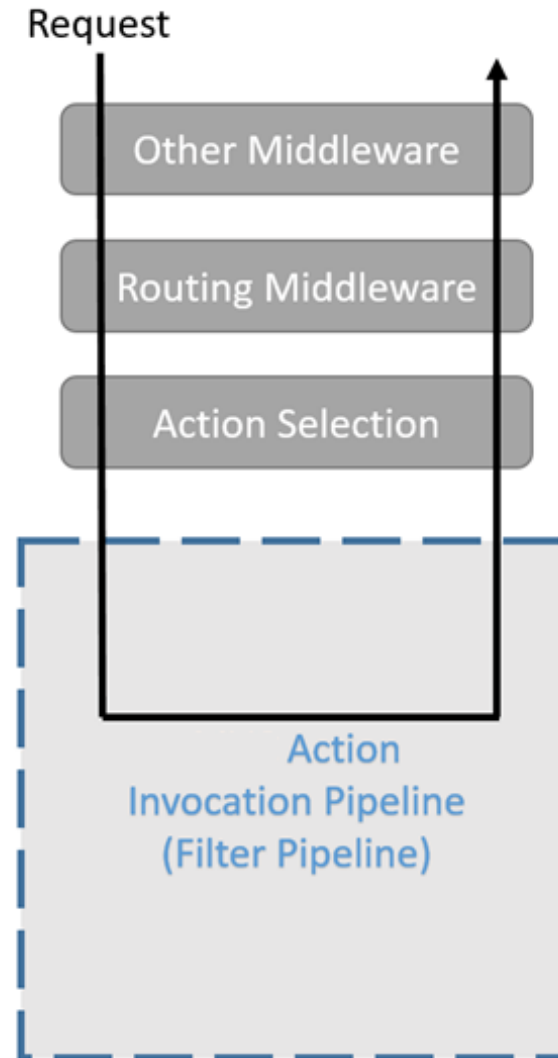
Les filtres

- Les filtres sont des traitements à réaliser avant ou après une action
- 2 grandes différences avec les middlewares
 - On choisit quand appliquer le filtre
 - Globale
 - Contrôleur
 - Action
 - Le filtre est après tous les middlewares donc on a accès à plus d'éléments : model binding, model state, route, etc.

Les filtres

- Ils peuvent être utilisés pour :
 - Authentifier les clients
 - Gérer des logs
 - Mettre en place du cache
 - Gérer les erreurs
 - Faire de la validation
 - Refactoriser du code qui doit être dupliqué dans plusieurs actions
 - Etc.

Les filtres



Application d'un filtre

- L'application d'un filtre se réalise à l'aide des annotations.
- Il peut s'appliquer sur une classe :

```
[MonFiltre]
```

```
public class HomeController : Controller { ... }
```

- Il peut s'appliquer sur une méthode

```
public class HomeController : Controller
```

```
{
```

```
    [MonFiltre]
```

```
    public ActionResult Index() { ... }
```

```
}
```

Authorization Filter

- Permet de créer son propre filtre pour un authentification custom

```
internal class CheckAdminAttribute : AuthorizeAttribute,
IAuthorizationFilter {

    public void OnAuthorization(AuthorizationFilterContext context)
    {

        var hasClaim = context.HttpContext.User.Claims.Any(c => c.Type ==
ClaimTypes.Role && c.Value == "admin");

        if (!hasClaim)
        {

            context.Result = new ForbidResult();

        }
    }
}
```


Authorization Filter

[CheckAdmin]

```
public class HomeController : Controller
```

Resource Filter

Permet de faire une action avant le model binding, utile pour du cache au plus tôt par exemple

```
public class GetCacheForIdFilter : Attribute, IResourceFilter {  
    public async void OnResourceExecuting(ResourceExecutingContext context) {  
        var key = context.HttpContext.Request.Query["Id"].ToString();  
        if (!string.IsNullOrEmpty(key)) {  
            // get cache data  
            context.Result = new JsonResult($"cache from {key}");  
        }  
    }  
  
    public void OnResourceExecuted(ResourceExecutedContext context) { } }  
}
```

Resource Filter

[GetCacheForIdFilter]

```
public IActionResult GetById(int? id)
{
    return Json("PAS DE CACHE");
}

// /Home/GetById => "PAS DE CACHE"
// /Home/GetById?id=10 => "cache from {id}"
```

Action Filter

- Permet de faire une action après le model binding et la validation, utile pour rendre la vérification du ModelState réutilisable

```
public class CheckModelStateFilter : Attribute, IActionFilter {  
    public void OnActionExecuting(ActionExecutingContext context) {  
        if (!context.ModelState.IsValid)  
        {  
            context.Result = new BadRequestObjectResult(context.ModelState);  
        }  
    }  
    public void OnActionExecuted(ActionExecutedContext context) { } } }
```

Action Filter

```
[CheckModelStateFilter]
```

```
public IActionResult GetById(int? id)
```

```
{
```

```
    return Json("OK");
```

```
}
```

```
// /Home/GetById/10 => "OK"
```

```
// /Home/GetById/Hello => "{\"id\":[\"The value 'hello' is not valid.\"]}
```

Création d'un filtre

- Le framework ASP.NET MVC fournit la classe "[ActionFilterAttribute](#)" qui implémente les interfaces "[IActionFilter](#)" et "[IResultFilter](#)" et hérite de la classe `Filter`.
- La réalisation d'un filtre s'effectue donc par héritage de la classe "[ActionFilterAttribute](#)" et en redéfinissant les méthodes :
- "[OnActionExecuting](#)" : Cette méthode est appelée avant qu'une action ne soit exécutée
- "[OnActionExecuted](#)" : Cette méthode est appelée après qu'une action soit exécutée
- "[OnResultExecuting](#)" : Cette méthode est appelée avant qu'un résultat d'action ne soit exécuté
- "[OnResultExecuted](#)" : Cette méthode est appelée après qu'un résultat d'action soit exécuté

Création d'un filtre

```
internal class LogFilterAttribute : ActionFilterAttribute {  
    ILogger<LogFilterAttribute> _logger;  
    public LogFilterAttribute(ILogger<LogFilterAttribute> logger) {  
        _logger = logger;  
    }  
    public override void OnActionExecuting(ActionExecutingContext context) {  
        _logger.LogInformation("From filter : " + context.HttpContext.Request.Path);  
  
        base.OnActionExecuting(context);  
    }  
}
```

Création d'un filtre

```
[ServiceFilter(typeof(LogFilter))]  
public IActionResult Privacy()  
{  
    return View();  
}
```


Création d'un filtre

- Un filtre est un service, il faudra donc le référencer dans les services

```
builder.Services.AddScoped<LogFilterAttribute>();
```

Démo + Exo 3

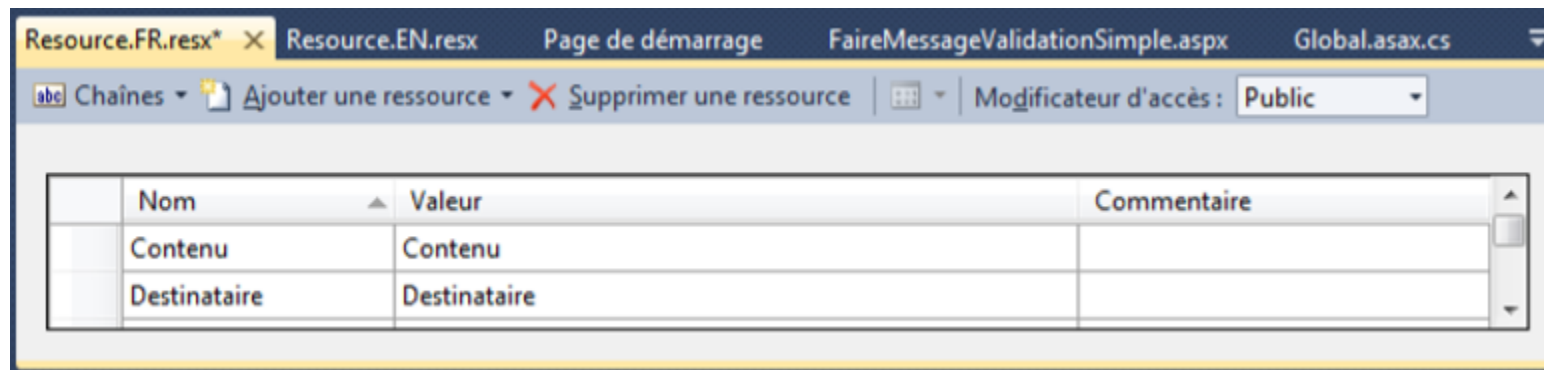
Internationalisation des applications

Internationalisation des applications

- Il est possible d'externaliser les libellés de présentation dans des fichiers de ressources.
- Ce mécanisme permet de rendre plus souple l'évolution d'une application et de gérer l'internationalisation.
- Le principe consiste à créer un fichier de ressources par langue en respectant l'extension associée à la langue :
 - ResourceView.resx
 - ResourceView.fr.resx
 - ResourceView.en.resx

Création d'un fichier de ressources

- La création d'un fichier de ressources se fait de la façon suivante :
- Ajouter -> Nouvel élément -> Fichier de ressources



Accès aux données d'un fichier de ressources

- Toutes les ressources sont compilées vers une DLL.
- Le nom du fichier est utilisé comme nom de la classe et le nom donné au libellé est considéré comme celui de la propriété de la classe.
- La valeur est accessible comme une propriété de classe.

`@SupportCours.Resources.ResourceView.Contenu`

Utilisation des annotations

- La valeur du Display peut être externalisée dans un fichier de ressources

```
[Display(Name = "Contenu", ResourceType = typeof(ResourceView))]
```

```
public string Contenu { get; set; }
```

Mise en œuvre d'AJAX

Échanges client / serveur avec AJAX

- Faire une requête du client vers le serveur et récupérer des informations sans changer de page

Exemple en jquery

```
<form id="formCreateComment">  
    <input type="hidden" id="id" />  
    <input type="text" id="value" />  
</form>  
  
<div id='commentZone'>  
    <partial name="_Comments" model="Model.Comments" />  
</div>
```

Exemple en jquery

```
$(function () {  
    $("#formCreateComment").submit(function (e) {  
        e.preventDefault();  
        $.post('/Comment/AddComment', {  
            Id: $('#id').val(),  
            Value: $('#value').val()  
        }, function (data) {  
            $('#commentZone').html(data);  
        });  
    });  
});
```

Exemple en jquery

[HttpPost]

```
public ActionResult AddComment(UnModel model)
{
    // Traitements
    return PartialView("_Comments", commentsViaRequest);
}
```

Démo + Exo 4

Contrôle de validation avec JavaScript et jQuery

Validation avec les annotations

- Il est possible de réaliser la validation en utilisant les annotations.
- Elles sont basées sur de simples déclarations et donnent la possibilité d'ajouter des règles de validation aux objets et aux propriétés avec un minimum de code.
- La classe `DataAnnotation` propose un ensemble de règles de validation :
 - `StringLength`: indique la longueur maximale de la chaîne autorisée,
 - `Required` : indique que le champ est obligatoire,
 - `RegularExpression` : permet de spécifier une expression régulière,
 - `Range` : permet de définir un intervalle de valeurs numériques,
 - `DataType` : permet de spécifier une des valeurs de type Enumérée (`EmailAddress`, `Url` ou `password`).
 - `Compare` : Comparer avec un autre champ
 - `Remote` : permet de faire une validation coté serveur

Exemple

```
public class Message
{
    [DisplayName("Contenu du message")]
    [Required(ErrorMessage = "Contenu requis")]
    [StringLength(200, ErrorMessage = "Longueur Max=200")]
    public string Contenu { get; set; }

    ...
}
```


Validation de formulaires

- Le framework fournit une classe `ModelState` pour la validation de formulaires.
- On va pouvoir tester si le formulaire est valide ou non à l'aide de la propriété `IsValid` et revenir à la page d'origine en affichant les messages d'erreurs si la page n'est pas valide.

Exemple

```
[HttpPost]
public ActionResult CreationMessageControle(Message model)
{
    if (!ModelState.IsValid) return View(model); // ERREUR
    else
    {
        // Traitements
        return RedirectToAction("Index", new { id = model.id }); // SUCCESS
    }
}
```

Exemple

- La vue doit aller chercher les messages d'erreurs définis à l'aide des annotations dans la classe métier.

```
<label asp-for="Theme"></label>
```

```
<input asp-for="Theme"/>
```

```
<span asp-validation-for="Theme" class="text-danger"></span>
```

```
<div asp-validation-summary="All">
```

```
</div>
```

Validation coté JS

- Il est possible de faire de la validation directement coté navigateur via jquery
- Grâce à cette validation coté client le contrôleur ne va pas recevoir de requête qui ne valide pas le model
 - /!\ sauf si l'utilisateur change le contenu HTML ou crée lui-même une requête HTTP
 - Il faut donc toujours mettre la validation coté serveur en plus de celle coté client
- Pour activer cette validation une vue partielle est prévu avec les 2 fichiers JS nécessaires
- Il faudra rajouter ce code dans chacune des pages où il y a besoin d'une validation coté client (ou directement dans le layout si on veut les charger tout le temps)

@section Scripts

{

<partial name="_ValidationScriptsPartial" />

}

Validation asynchrone

- Il est possible de faire de la validation asynchrone avec l'attribut Remote.
- Sur un formulaire d'inscription, il est fréquent de vouloir valider de manière asynchrone qu'un nom d'utilisateur / e-mail n'existe pas déjà en base utilisateurs.
- L'action du contrôleur doit retourner un boolean sérialisé en JSON.

Validation asynchrone

```
[Remote("Verifier", "nomControlleur", ErrorMessage = "Error")]
```

... property

```
public ActionResult Verifier(string Titre)
{
    bool res = false; //traitement de contrôle
    return Json(res);
}
```

Validation d'une vue partielle

- Si on charge une vue partielle sur un clic d'un bouton en AJAX, la validation ne sera pas gérée par JQuery
- Pour le mettre en place il faut rajouter un script en plus et un traitement après l'arrivée de la vue partielle dans le dom

```
$('#container').html(data);  
$.validator.unobtrusive.parse("form");
```

Démo + exo 5

Entity Framework Core

Entity Framework Core

- Pour rappel, ces packages nugget devront être installés
 - Microsoft.EntityFrameworkCore
 - Microsoft.EntityFrameworkCore.Tools - pour les outils en ligne de commande
 - Microsoft.EntityFrameworkCore.SqlServer - pour base de donnée SQL Server
 - Microsoft.EntityFrameworkCore.Proxies – pour le Lazy loading si nécessaire

Le Context EF Core

- Le contexte EF est un service dans ASP.NET Core
- La connexion à la BDD peut être faite soit dans le contexte soit dans la déclaration du service

Le Context EF Core - Constructeur

```
public class TestContext : DbContext
{
    public TestContext(DbContextOptions DbContextOptions) :
base(DbContextOptions)
    {

    }
}
```

Le Context EF Core - DbSet

```
public DbSet<Client> Clients { get; set; }
```

Le Context EF Core - OnConfiguring

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
    // Si non Configuré dans la création du service, on passe en
    in memory par exemple
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseInMemoryDatabase("ExoDbInMemory");
        optionsBuilder.LogTo(Console.WriteLine);
    }
}
```

Le Context EF Core - OnModelCreating

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {  
  
    var c1 = new Client { Id = 1, FirstName = "FirstName 1", LastName =  
"LastName 1" };  
  
    var c2 = new Client { Id = 2, FirstName = "FirstName 2", LastName =  
"LastName 2" };  
  
    modelBuilder.Entity<Client>().HasData(new List<Client> { c1, c2 });  
  
    base.OnModelCreating(modelBuilder);  
}
```

Le Context EF Core – Configuration du service

```
builder.Services.AddDbContext<ExoDbContext>(o => {  
  
    //o.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Exos_API_DB;Trusted_Connection=True;");  
  
    o.UseSqlServer(builder.Configuration.GetConnectionString("ExoDbContextCS"));  
}  
);
```


Entity Framework Core

- La configuration sera à mettre dans le appsettings.json

```
"ConnectionStrings": {  
    "ExoDbContextCS":  
    "Server=(localdb)\\mssqllocaldb;Database=ExoDb;Trusted_Connection=True  
    ;"  
}
```

Entity Framework Core

- Le contrôleur n'aura plus qu'à demander le contexte en service

```
private readonly WikyCoreDbContext _wikyCoreDbContext;
```

```
public HomeController(WikyCoreDbContext wikyCoreDbContext)  
{  
    _wikyCoreDbContext = wikyCoreDbContext;  
}
```

Entity Framework Core

- Pour mettre à jour la base de données on va passer par les migrations EF
- Grâce à la ligne de commande NuGet (Tools > NuGet Package Manager > Package Manager Console) on va pouvoir lancer une mise à jour de base
- A chaque changement de nos models on va créer une migration
 - `Add-Migration NOM_MIGRATION`
- Pour mettre à jour la base
 - `Update-Database`
- EF va regarder l'état de la base et passer toute les migrations qui n'ont pas encore été faites

Entity Framework Core

- Si on veut que les migrations s'appliquent automatiquement on peut le faire au démarrage de l'application (ou dans une action spécifique)

```
var app = builder.Build();  
using (var serviceScope = app.Services.CreateScope())  
{  
    var services = serviceScope.ServiceProvider;  
    var appDbContext = services.GetRequiredService<ExoDbContext>();  
    appDbContext.Database.Migrate();  
}
```

Entity Framework Core

- Si, pour des phases de debug, on veut que la base se crée et se supprime à chaque lancement on peut le faire aussi au démarrage de l'application

```
var app = builder.Build();

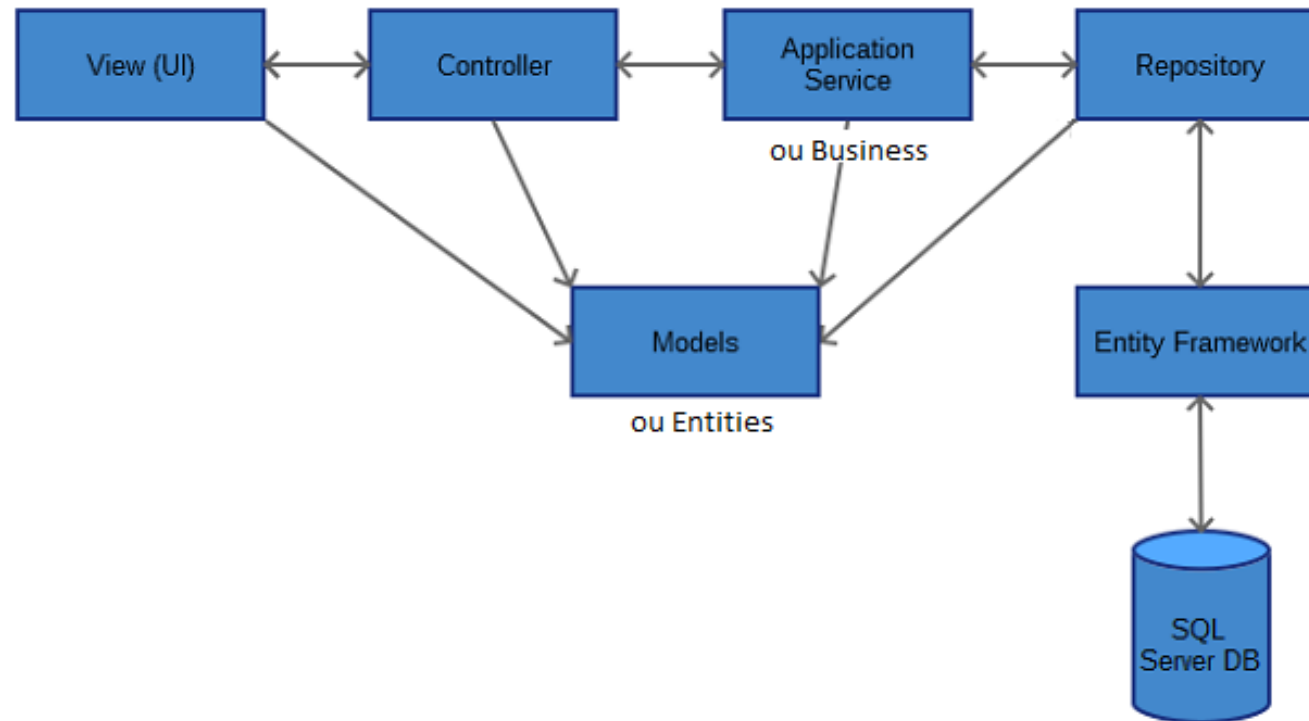
using (var serviceScope = app.Services.CreateScope())
{
    var services = serviceScope.ServiceProvider;

    var appDbContext = services.GetRequiredService<ExoDbContext>();
    appDbContext.Database.EnsureDeleted();
    appDbContext.Database.EnsureCreated();
}
```

L'architecture MVC avancé

- Le problème de l'architecture MVC classique est que le contrôleur en fait trop
 - Il gère les données envoyées et récupérées de la vue
 - Il gère la partie de gestion métier
 - Il gère l'accès à la base de données
- La solution est de redécouper les couches pour que chacune n'est qu'une responsabilité
 - View : la partie vue qui ne change pas
 - Controlleur : Le contrôleur qui gère les données envoyées et récupérées de la vue
 - Business (Application Service) : La partie qui gère l'aspect métier / fonctionnel (Un électeur ne peut pas avoir moins de 18 ans)
 - Repository : Il gère la persistance des données en BDD
 - Entities (Models) : Contient toutes les classes métiers de l'application

L'architecture MVC avancé



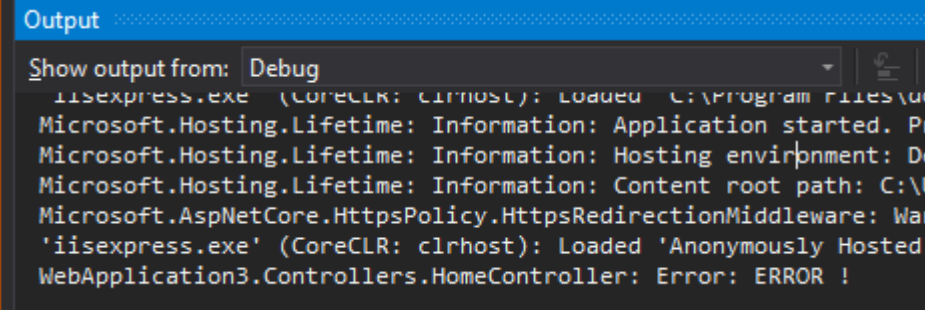
Démo + exo 6

Aspects avancés

Tracer des informations

- ASP.NET Core intègre une gestion des logs très simples
- Par défaut les logs sont dans la console mais cela pourra être modifié pour aller en base ou en fichier

```
_logger.LogError("ERROR !");
```



```
Output
Show output from: Debug
iisexpress.exe (CoreCLR: clrhost): Loaded C:\Program Files\IIS Express\iisexpress.exe
Microsoft.Hosting.Lifetime: Information: Application started. Press Ctrl+C to shut down.
Microsoft.Hosting.Lifetime: Information: Hosting environment: Development
Microsoft.Hosting.Lifetime: Information: Content root path: C:\Users\user\source\repos\WebApplication3\WebApplication3
Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware: Warning: A non-HTTPS URL is being redirected to a HTTPS URL.
'iisexpress.exe' (CoreCLR: clrhost): Loaded 'Anonymously Hosted Dynamic Intermediate Representation'
WebApplication3.Controllers.HomeController: Error: ERROR !
```

Tracer des informations

- Quand vous écrivez dans un journal, vous devez spécifier son « LogLevel » qui indique le degré de gravité
- L'API prévoit 6 niveaux de gravité :
 - **Trace** : Simples traces d'informations pour le débogage ;
 - **Debug** : Informations à court terme pour le débogage ;
 - **Information** : Informations sur le fonctionnement de l'application ;
 - **Warning** : Événements anormaux qui ne provoquent pas l'arrêt de l'application ;
 - **Error** : Erreurs et exceptions qui ne peuvent être gérées et qui entraînent un échec de l'application ;
 - **Critical** : Défaillances qui nécessitent une attention particulière.

Gestion des Areas

- Plus votre application web va grandir et plus elle va contenir des vues et des contrôleurs.
- Au bout d'un moment, cela peut vite devenir complexe à gérer
- Pour résoudre ce problème il est possible de découper son application en zones, que l'on appelle « Areas ». Cela permet notamment de regrouper des éléments par domaine fonctionnel.
- Une zone va donc regrouper un ensemble de contrôleurs et de vues dans un dossier de votre projet
- Pour créer une Area, click droit sur le projet, add, new scaffolded item, MVC Area
- Une Area crée une nouvelle route avec comme premier paramètre le nom de l'Area
 - Pour l'Area Client, il faut utiliser l'URL: `site.com/Client/Home/Index`

L'asynchrone

- Pour plus de performance on peut retourner une tâche asynchrone

```
public async Task<IActionResult> Details(int? id) {  
    if (id == null) return NotFound();  
  
    var post = await _context.Posts.FindAsync(id);  
  
    if (post == null) return NotFound();  
  
    return View(post);  
}
```

Gestion des erreurs

- ASP.NET Core intègre une gestion simple d'erreur
- Si l'application crash en développement elle retournera l'erreur complète sinon elle ira sur l'action Error qui retourne une erreur plus user friendly

```
if (env.IsDevelopment())  
{  
    app.UseDeveloperExceptionPage();  
}  
else  
{  
    app.UseExceptionHandler("/Home/Error");  
}
```

Gestion de la config

- La configuration se fait dans le fichier appsettings.json
- On peut écraser certaines config par environnement en rajoutant le nom de l'environnement au nom du fichier : appsettings.Development.json
- Il sera déconseillé d'utiliser ces fichiers pour des informations sensibles comme des chaines de connexion

Gestion de la config

- Les configurations sont chargées par ASP.NET en variables d'environnement
- .NET inclus un outil qui permet d'injecter des variables d'environnement dans notre code
- L'objectif est d'éviter d'écrire des éléments sensibles dans le code
- Les variables d'environnements sont persistées dans le fichier secrets.json de notre dossier APPDATA
 - Ce fichier n'est pas chiffré, il n'est donc pas sécurisé
- Dans les autres environnements autre que dev, il faudra injecter les bonnes variables d'environnement de façon sécurisé (KeyVault par exemple)

Secret Manager

- Le secret manager en .NET est un outil simple pour injecter des variables d'environnement
- La commande init va permettre d'initialiser le secret manager dans le projet et générer un id unique
 - `dotnet user-secrets init`
- On pourra ensuite définir nos clés
 - `dotnet user-secrets set "ConnectionStrings:CS" "CS from env"`
 - `dotnet user-secrets set "Env" "PREPROD"`
- On pourra lister les secrets de notre projet
 - `dotnet user-secrets list`
- On pourra supprimer un secret
 - `dotnet user-secrets remove "Env"`
 - `dotnet user-secrets clear`

Optimisation

Suivi de session : motivation

- Le protocole HTTP présente un gros inconvénient, c'est un protocole sans état :
 - Lorsque le client émet une requête, une connexion est ouverte avec le serveur.
 - Le serveur répond en envoyant la ressource attendue ou un code d'erreur.
 - Une fois le document arrivé, la connexion est fermée.
 - Le serveur ne conserve aucune information sur le client.

Stockage côté serveur

- Il existe un ensemble de mécanismes qui permettent de conserver des données côté client
 - Cookies
 - Query String
 - Champs cachés
- Il existe un ensemble de mécanismes qui permettent de conserver des données côté serveur
 - Session : données propres à chaque utilisateur
 - TempData : données propres à chaque utilisateur
 - Cache : contexte lié à l'application.

Optimisation des ressources avec la gestion du cache

- Il est possible de mettre des informations en cache ce qui va permettre d'améliorer les performances d'une application web en réduisant le temps nécessaire pour traiter une page Web.
- La mise en cache va permettre :
 - D'éviter de récupérer à plusieurs reprises la même information à partir de la base de données.
 - D'améliorer les performances d'une application Web, en réduisant la charge sur les serveurs.
- La mise en cache n'est pas adaptée aux applications qui incluent des changements de contenu fréquent.

Cache Tag Helper

<cache>

```
for (int i = 0; i < DateTime.Now.Second; i++)  
{  
    <div>@i</div>  
}
```

</cache>

(20 minutes par défaut)

Cache Tag Helper

- Enable: activé ou désactivé (Peut être renseigné dynamiquement via un ViewBag par exemple)
- expires-on : Expiration avec une DateTime
- expires-after : Expiration avec un TimeSpan
- expires-sliding : Expiration avec un TimeSpan par rapport au dernier appel
- vary-by : Permet d'avoir un cache différent selon un élément (query, cookie, route, user, header)
- Priority : Définit la priorité du cache, si la mémoire du serveur est surchargée les éléments avec une priorité faible seront vidés en premier

Cache Tag Helper

```
<cache expires-on="@new DateTime(2025, 12, 31, 23, 59, 59)">
```

```
</cache>
```

```
<cache expires-after="TimeSpan.FromSeconds(60)">
```

```
</cache>
```

```
<cache expires-sliding="TimeSpan.FromSeconds(5)">
```

```
</cache>
```


Cache Tag Helper

```
<cache vary-by="@Model.Id">  
    <div>  
        @Model.Name  
    </div>  
    <div>  
        @Model.Price  
    </div>  
</cache>
```

Cache Tag Helper

`<cache priority=`

`"Microsoft.Extensions.Caching.Memory.CacheItemPriority.NeverRemove">`

`</cache>`

Low = 0,

Normal = 1,

High = 2,

NeverRemove = 3

Cache côté contrôleur

- On peut aussi utiliser du cache dans notre contrôleur
- On passera cette fois par le service IMemoryCache

Cache côté - middleware

```
builder.Services.AddMemoryCache();
```

Cache - contrôleur

```
private IMemoryCache _memoryCache;  
public HomeController(IMemoryCache memoryCache)  
{  
    _memoryCache = memoryCache;  
}
```

Cache - contrôleur

```
string s;  
if (!_memoryCache.TryGetValue("key", out s))  
{  
    s = DateTime.Now.Millisecond.ToString();  
    _memoryCache.Set("key", s);  
}  
string s2 = _memoryCache.GetOrCreate("key", entry => {  
    return DateTime.Now.Millisecond.ToString();  
});
```

Cache - contrôleur

```
MemoryCacheEntryOptions cacheOptions = new MemoryCacheEntryOptions();  
cacheOptions.SetPriority(CacheItemPriority.Low);  
cacheOptions.SetSlidingExpiration(TimeSpan.FromSeconds(5));  
_memoryCache.Set("key", s, cacheOptions);
```

Les sessions

- La gestion de session se réalise de la façon suivante :
 - La première fois qu'un utilisateur fait une demande au serveur, il génère un SessionID.
 - Un SessionID est un nombre produit par un algorithme complexe, qui identifie de manière unique la session de chaque utilisateur.
 - Le serveur enregistre le SessionID sous forme de cookie dans le navigateur Web de l'utilisateur.
 - Chaque fois que le serveur est sollicité pour une page, un cookie SessionID est recherché dans l'en-tête de requête HTTP.

Configuration d'une session

```
builder.Services.AddDistributedMemoryCache();  
builder.Services.AddSession(options =>  
{  
    options.IdleTimeout = TimeSpan.FromSeconds(10);  
    options.Cookie.HttpOnly = true;  
    options.Cookie.IsEssential = true;  
});  
  
app.UseSession();
```

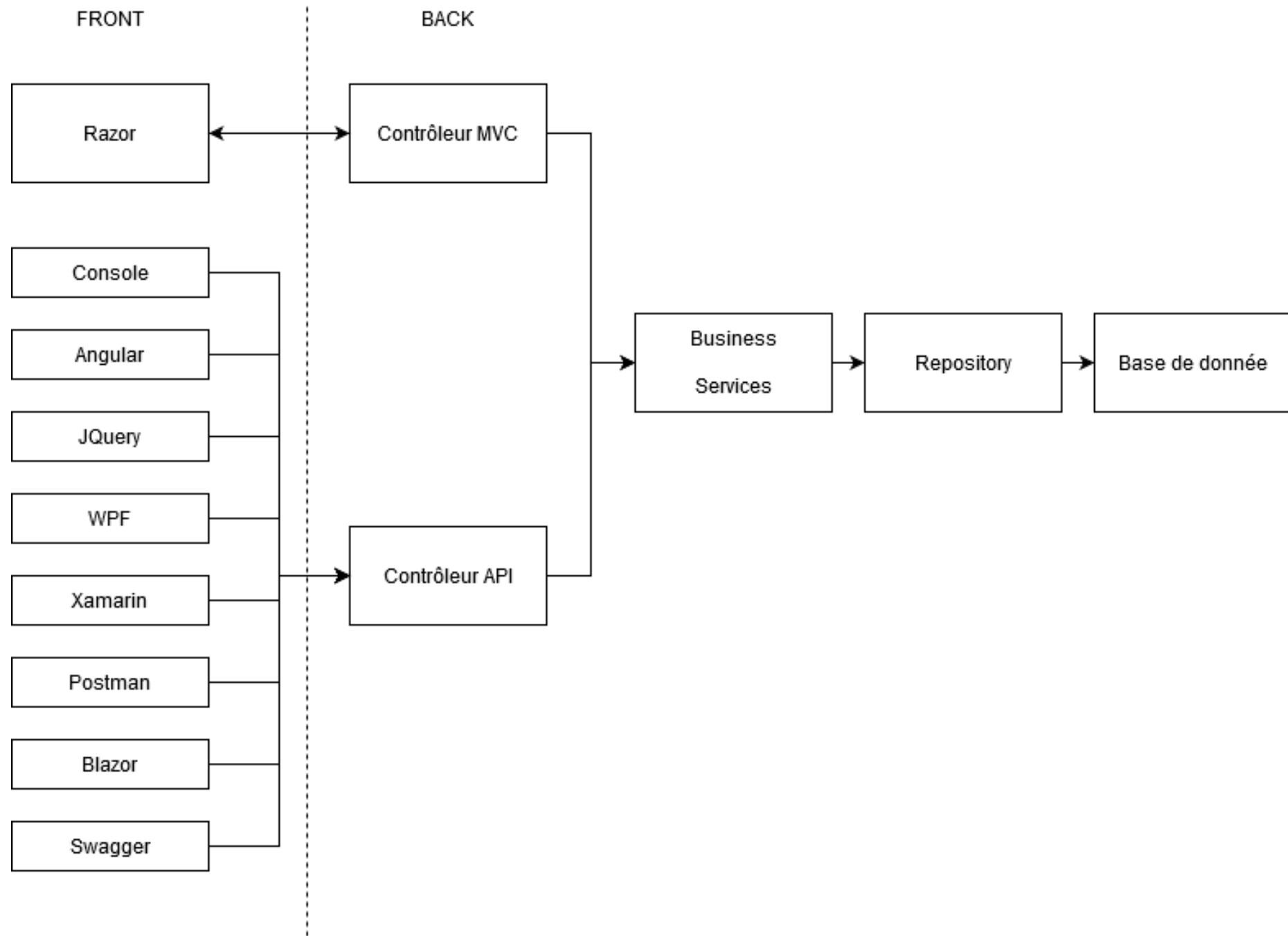
Gestion des données en session

```
HttpContext.Session.SetString("key", "value");  
var v = HttpContext.Session.GetString("key");  
HttpContext.Session.Clear();
```

Objets en session

```
public static class SessionExtensions    {  
    public static void Set<T>(this ISession session, string key, T value) {  
        session.Set(key, JsonSerializer.SerializeToUtf8Bytes(value));  
    }  
    public static T Get<T>(this ISession session, string key) {  
        var value = session.Get(key);  
  
        return value == null ? default(T) :  
            JsonSerializer.Deserialize<T>(value);  
    }  
}
```

Web API





ASP.NET Core Web App

A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

[C#](#)[Linux](#)[macOS](#)[Windows](#)[Cloud](#)[Service](#)[Web](#)

ASP.NET Core Web API

A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

[C#](#)[Linux](#)[macOS](#)[Windows](#)[Cloud](#)[Service](#)[Web](#)[WebAPI](#)

ASP.NET Core Empty

An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

[C#](#)[Linux](#)[macOS](#)[Windows](#)[Cloud](#)[Service](#)[Web](#)

ASP.NET Core Web App (Model-View-Controller)

A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP

Présentation

- Web API permet de créer des services Web RestFull qui peuvent être consommés à partir de différents types clients (réseaux sociaux, navigateurs, terminaux mobiles, etc.).
- Web API fournit les fonctionnalités suivantes :
 - un modèle de programmation HTTP moderne : Web API permet d'accéder directement et de manipuler les requêtes et réponses HTTP à l'aide d'un nouveau modèle objet HTTP fortement typé.
 - négociation de contenu : le client et le serveur peuvent travailler ensemble afin de déterminer le bon format pour les données retournées à partir d'une API (XML, Json, ...)
 - prise en charge complète des routes : l'API Web prend désormais en charge l'ensemble des fonctionnalités de routage.
 - composition de requêtes : Web API permet de prendre rapidement en charge l'interrogation par l'intermédiaire des conventions URL « Odata » .
 - validation des modèles : le modèle fournit un moyen facile d'extraire des données de diverses parties d'une requête HTTP et convertit celles-ci en objets .NET. Web API offre le même support qu'ASP.NET MVC pour la liaison des modèles et la même infrastructure de validation.

Exemple XML

```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```


Exemple JSON

```
{"employees":[  
  { "firstName":"John", "lastName":"Doe" },  
  { "firstName":"Anna", "lastName":"Smith" },  
  { "firstName":"Peter", "lastName":"Jones" }  
]}
```

Le routage

- Le système de routage utilisé pour une API Web est légèrement différent de celui d'une application ASP.NET MVC.
 - vous n'avez pas besoin de renseigner la méthode d'action.
 - L'API Web utilise la méthode HTTP et non le chemin de l'URI pour sélectionner la méthode d'action.
- L'utilisation de HTTP permet donc à Web API d'identifier automatiquement la méthode du contrôleur qui sera exécutée en fonction du verbe HTTP qui est effectué (GET, POST, etc.).

Exposition des services

```
[ApiController]
```

```
[Route("api/[controller]")]
```

```
public class TestController : ControllerBase {
```

```
}
```

Exposition des services

```
[ApiController]
[Route("api/[controller]")]
public class TestController : ControllerBase {
    [HttpGet]
    public IActionResult Get() {
        return Ok("ok!");
    }
}
```

Exposition des services

```
[ApiController]
[Route("api/[controller]")]
public class TestController : ControllerBase {
    [HttpGet]
    public IActionResult Get() {
        return Ok("ok!");
    }
    [HttpGet]
    public IActionResult OtherGet() { return Ok("ok!"); }
} ➔ ERROR
```

Exposition des services

```
[ApiController]
[Route("api/[controller]")]
public class TestController : ControllerBase {
    [HttpGet]
    public IActionResult Get() {
        return Ok("ok!");
    }
    [HttpGet("{id}")]
    public IActionResult Get(int id) { return Ok("ok!" + id); }
}
```

Exposition des services

```
[ApiController]
[Route("api/[controller]")]
public class TestController : ControllerBase {
    [HttpGet] // /api/Test
    public IActionResult Get() {
        return Ok("ok!");
    }
    [HttpGet("OtherGet")] // /api/Test/OtherGet
    public IActionResult OtherGet() { return Ok("ok!"); }
}
```

Exposition des services

```
[ApiController]
[Route("api/[controller]/[action]")]
public class TestController : ControllerBase {
    [HttpGet] // /api/Test/Get
    public IActionResult Get() {
        return Ok("ok!");
    }
    [HttpGet] // /api/Test/OtherGet
    public IActionResult OtherGet() { return Ok("ok!"); }
}
```


Exposition des services

```
[ApiController]
[Route("api/[controller]")] // [Route("[controller]")]
// [Route("api/[controller]/[action]")]
public class TestController : ControllerBase {
    [HttpGet]
    public IActionResult Get() {
        return Ok("ok!");
    }
    [HttpPost]
    public IActionResult Post(Test obj) { return Ok(obj); }
}
```

Exposition des services

```
// /api/[controller]/Get3/10/test
[HttpGet("Get3/{id}/{name}")]
public IActionResult Get3(int id, string name) {
    return Ok($"id: {id}, name: {name} via Get3");
}

// /api/[controller]/Get4?id=10&name=test
[HttpGet("Get4")]
public IActionResult Get4(int id, string name) {
    return Ok($"id: {id}, name: {name} via Get4");
}
```

Types de retours les plus utilisés

- Il existe des méthodes dans ControllerBase permettant de créer un retour et un code de statut HTTP
 - Retour de type HttpStatusCodeResult
 - `return Ok();`
 - `return NoContent();`
 - `return NotFound();`
 - `return BadRequest();`
 - `return Forbid();`
 - `return StatusCode(500);`
 - Une donnée peut cependant être retournée directement
 - `return db.Students;`

Consommation de services

- Vous pouvez tester l'utilisation des Web API :
 - Lancer l'application
 - Renseigner l'url avec la valeur suivante : <http://localhost::numPort/api/MessageService>
- Selon le navigateur vous obtiendrez le résultat au format Json ou XML
- Utilisez un outil comme POSTMAN pour simuler plus facilement l'envoi de requête

Consommation de services

- La consommation de services peut se réaliser de différentes façons :
 - lancement direct depuis le navigateur
 - Lancement côté client dans une page html ou en JavaScript/Jquery
 - Utilisation de la classe HttpClient dans n'importe quel type d'application (Console, WEB, ...).

```
HttpClient client = new HttpClient();

client.DefaultRequestHeaders.Accept.Add( new
MediaTypeWithQualityHeaderValue("application/json"));

var response =
client.GetAsync("http://localhost:59570/api/MessageService/").Result;

if (response.IsSuccessStatusCode) {

Message mess = response.Content.ReadAsAsync<Message>().Result;
```

OpenAPI et Swagger

- L'OpenAPI est un standard qui permet de décrire une API REST
- Swagger offre des outils permettant de générer cette OpenAPI pour son API Web
- Il offre également une interface (qui se base sur l'OpenAPI) qui permet d'explorer et tester les différentes méthodes offertes par le service
- En ASP.NET MVC on utilisera le package nugget Swashbuckle.AspNetCore, qui est un wrapper à Swagger
- Cette configuration est présente par défaut dans ASP.NET Core API 6
- Il suffit de lancer le projet et d'aller à l'url /swagger pour voir l'interface de test de swagger

Ajouter Swagger

```
// in ConfigureServices
```

```
services.AddSwaggerGen(c => {  
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1" });  
});
```

```
// In Configure
```

```
app.UseSwagger();  
app.UseSwaggerUI(c => {  
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");  
});
```

Référence circulaire

- On risque une référence circulaire au niveau de la sérialisation JSON si
 - Nos classes EF se référencent entre elles (propriété User d'un côté et propriété Liste de messages de l'autre par exemple)
 - On active le LazyLoading ou qu'on charge les relations via des includes
- Pour gérer ce cas il faudra activer l'option dans la configuration du service de sérialisation JSON

Référence circulaire - .NET 6

- Pour les version avant .NET 6

```
services.AddControllers().AddJsonOptions(  
    options => options.JsonSerializerOptions.ReferenceHandler =  
    ReferenceHandler.Preserve  
);
```

- A partir de la version 6 une nouvelle syntaxe est disponible, elle génère un json plus simple, plus standard mais sans référence

```
builder.Services.AddControllers().AddJsonOptions(x =>  
x.JsonSerializerOptions.ReferenceHandler =  
ReferenceHandler.IgnoreCycles);
```

Swagger – Génération XML

- Afin de compléter notre OpenAPI, on va pouvoir facilement injecter nos /// de nos méthodes à Swagger
- Changer le csproj pour activer la génération de documentation (la 2^{ème} ligne sert à enlever les warnings si les /// ne sont pas présent)

<PropertyGroup>

...

<GenerateDocumentationFile>true</GenerateDocumentationFile>

<NoWarn>\$(NoWarn);1591</NoWarn>

</PropertyGroup>

Swagger – Génération XML

- Dans la configuration de Swagger dans ConfigureServices

```
services.AddSwaggerGen(c => {  
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1" });  
    // Set the comments path for the Swagger JSON and UI.  
    var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";  
    var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);  
    c.IncludeXmlComments(xmlPath);  
});
```

Swagger – OpenAPI

- Afin d'avoir une OpenAPI plus complète on va devoir renseigner certains éléments dans nos contrôleur (les outils côté client pourront générer un code plus précis)
- On pourra prévenir quelle type de réponse le client va pouvoir récupérer

`[ProducesResponseType(200)]`

`[ProducesResponseType(404)]`

- On pourra aussi dire quel type d'objet on retourne en json

`public ActionResult<WeatherForecast> Index2()`

Versioning

- La gestion des versions de notre API pourra se faire avec la package nuget
 - Nuget Versioning Microsoft.AspNetCore.Mvc.Versioning
- Il faudra simplement configurer le service de versioning et d'indiquer la version disponible sur les actions du contrôleur

Versioning - service

```
builder.Services.AddApiVersioning(o =>
{
    //o.AssumeDefaultVersionWhenUnspecified = true;
    //o.DefaultApiVersion = new ApiVersion(1, 0);
    //o.ReportApiVersions = true;
    o.ApiVersionReader = new QueryStringApiVersionReader("version");
    //o.ApiVersionReader = new MediaTypeApiVersionReader("version");
    //o.ApiVersionReader = new HeaderApiVersionReader("api-version");
});
```

Versioning - contrôleur

```
// https://localhost:7281/V2/Test/Get  
// [Route("v{version:apiVersion}/[controller]")]  
[ApiVersion("1.0")]  
[ApiVersion("2.0")]  
public class TestController : ControllerBase
```

Versioning - actions

```
// https://localhost:7281/test/Get?version=1
[HttpGet("Get")]
[MapToApiVersion("1.0")]
public IActionResult Get() {
    return Ok("V1 !");
}

// https://localhost:7281/test/Get?version=2
[HttpGet("Get")]
[MapToApiVersion("2.0")]
public IActionResult GetV2() {
    return Ok("V2 !");
}
```


API minimaliste

- Les APIs minimalistes sont des APIs avec très peu de dépendances et de codes
- Pour des APIs avec très peu de route cela permet d'avoir une API beaucoup plus légère
- Il n'y aura pas de contrôleur, juste la gestion des routes via les middlewares

API minimaliste – Program.cs

```
var builder = WebApplication.CreateBuilder(args);  
var app = builder.Build();  
app.MapGet("/", () => "Hello World!");  
app.MapGet("/test", () => "Hello Test!");  
app.MapGet("/test/{name}", (string name) => $"Hello {name} !");  
//app.MapPost("/todoitems", async (Todo todo, TodoDb db) => { ... });  
app.Run();
```

CORS

- Dans le cas d'une API le client est souvent sur un nom de domaine différent
- Par défaut les requêtes ayant une origine différente sont bloquées
- CORS (**C**ross **O**rigin **R**esource **S**haring) est un standard permettant de passer outre cette sécurité
- En ASP.NET Core il faudra créer un service et un middleware pour autoriser les requêtes d'une autre domaine

CORS - service

```
builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(
        builder =>
        {
            builder.WithOrigins("*");
            //builder.WithOrigins("https://localhost:XXX",
"http://localhost:XXX")
            //.AllowAnyHeader()
            //.AllowAnyMethod();
        });
});
```

CORS - middleware

// Après le Swagger

```
app.UseCors();
```

Démo + exo 7

Tester son application

Différent types de test

- Tests unitaires: vérifie unitairement une méthode et vérifier son comportement
- Tests d'intégration: vérifie le comportement de plusieurs composant qui interagisse entre eux
- Tests de validation: test niveau métier de la fonctionnalité

Intérêt des tests unitaires

- On veut des preuves que notre code fonctionne correctement
- Si on a un bug, on veut la preuve qu'il ne se reproduira plus
- Si on change du code, on veut la preuve que rien n'a été cassé
- Si on casse quelque chose, on veut le savoir le plus rapidement possible
- Si on casse quelque chose, on veut savoir exactement quoi

Intégration continu

- Il faut executer les tests le plus souvent possible
- Dans l'ideal à chaque compilation
- Au moins à chaque commit
- Les tests doivent être rapide (il y en aura des centaines)
- 1 seconde pour un test est trop long
- Tests autonomes, aucune interaction n'est possible pendant les tests

AAA

- Arrange
 - Initialise les objets
 - Spécifie les valeurs que l'on veut passer à la méthode que l'on veut tester
- Act
 - On appelle la méthode que l'on veut tester avec les paramètres
- Assert
 - On vérifie que le comportement et le résultat obtenu est bien ce que l'on attendait

Création

- Génération automatique
 - A partir de l'éditeur window, click droit sur une méthode et "Créer les tests unitaires"
 - Suivre l'assistant
- Génération manuelle
 - Nouveau projet de type MsTest
 - Référencer le projet à tester
 - Ecrire nos classes de test
- Mettre un timeout

[Timeout(2000)]

Utilisation

- Exécuter les tests
 - Afficher la fenêtre Test Explorer (disponible dans le menu Test)
 - Run all
- Exécuter les tests après chaque build n'est disponible que dans la version Entreprise

Test Driven Development

- Ajouter une nouvelle fonctionnalité
 - Ecrire une version vide (pas de code)
 - Ecrire un failing test pour cette méthode
 - Faire en sorte que le test passe
- Corriger un bug
 - Ecrire un failing test qui démontre le bug
 - Corriger le bug
 - Le test sera toujours là pour être sûr que ce bug ne reviendra pas
- Changement d'implémentation
 - Faire les changements
 - Tester que les tests sont toujours bon
 - On est presque sûr qu'on n'a pas créé de bug

Exemple

```
public class CalculTaxe
{
    public double CalculTVA(double tarifHT, double pourcentageTVA)
    {
        return 0;
    }
}
```

Exemple

```
[TestMethod()]  
  
public void CalculTVATest()  
{  
    // Arrange  
  
    var prixHT = 10;  
  
    var pourcentageTVA = 20;  
  
    var prixTTC = 12;  
  
    var calculTaxe = new CalculTaxe();  
  
    // Act  
  
    var result = calculTaxe.CalculTVA(prixHT, pourcentageTVA);  
  
    // Assert  
  
    Assert.AreEqual(prixTTC, result);  
}
```


Exemple

```
[TestMethod()]
[ExpectedException(typeof(ArgumentException))]
public void CalculTVATestException() {
    // Arrange
    var prixHT = 10;
    var pourcentageTVA = -10;
    var calculTaxe = new CalculTaxe();
    // Act
    var result = calculTaxe.CalculTVA(prixHT, pourcentageTVA);
    // Assert => ExpectedException
}
```

Exemple

```
public double CalculTVA(double tarifHT, double pourcentageTVA)
{
    if(pourcentageTVA < 0)
        throw new ArgumentException(nameof(pourcentageTVA));

    return tarifHT + tarifHT * pourcentageTVA / 100;
}
```

Bouchonnage

- Un des problèmes récurrent dans les tests unitaires est le fait qu'une méthode testée a besoin d'un ou plusieurs autres services pour fonctionner
- Dans ce cas le test unitaire ne sera pas possible car le test va tester la méthode et en plus les autres services
- L'idée est de mettre un bouchon (stub) qui va remplacer les autres services par des implémentations fixes
- Plusieurs termes existe pour ce type de solution
 - Dummy : classe créée qui ne fait rien, elle sera utilisée juste pour être renseignée dans des paramètres non utilisés
 - Fake : Element qui a une implémentation qui fonctionne mais qui est seulement pour le test unitaire (base de données en mémoire par exemple)
 - Stubs : Element qui va répondre de la même façon mais qui prendra en compte les paramètres reçus
 - Mock : Element qui va répondre toujours de la même façon sans se préoccuper des paramètres reçus

Bouchonnage

- Les interfaces seront obligatoires pour réussir nos tests
- En effet si un serviceA référence directement un serviceB (couplage fort) alors il ne sera pas possible de créer un bouchon pendant le test unitaire
- Il faudra que le serviceA référence une interface IServiceB (couplage faible), le test unitaire pourra alors passer le bouchon pendant le test unitaire
- L'injection de dépendances (IoC) nous permettra de faire des bouchons encore plus facilement car le couplage est réduit au minimum

Example

```
public enum TypesTVA  
{  
    Normal,  
    Intermediaire,  
    Reduit  
}
```

Example

```
public class CalculPourcentageTVA {  
    public double PourcentageTVAViaTypeTVA(TypesTVA typeTVA) {  
        switch (typeTVA) {  
            case TypesTVA.Normal: return 20;  
            case TypesTVA.Intermediaire: return 10;  
            case TypesTVA.Reduit: return 5;  
            default: return 20;  
        }  
    }  
}
```

Example

```
public double CalculTVAViaTypeTVA(double tarifHT, TypesTVA typeTVA)
{
    var calculPourcentageTVA = new CalculPourcentageTVA();
    var pourcentageTVA =
calculPourcentageTVA.PourcentageTVAViaTypeTVA(typeTVA);

    if (pourcentageTVA < 0)
        throw new ArgumentException(nameof(pourcentageTVA));

    return tarifHT + tarifHT * pourcentageTVA / 100;
}
```

Example

```
public interface ICalculPourcentageTVA
{
    double PourcentageTVAViaTypeTVA(TypesTVA typeTVA);
}
```


Example

```
public class CalculTaxe
{
    ICalculPourcentageTVA calculPourcentageTVA;

    public CalculTaxe(ICalculPourcentageTVA calculPourcentageTVA)
    {
        this.calculPourcentageTVA = calculPourcentageTVA;
    }
}
```

Example

```
public double CalculTVAViaTypeTVA(double tarifHT, TypesTVA typeTVA)
{
    var pourcentageTVA =
    calculPourcentageTVA.PourcentageTVAViaTypeTVA(typeTVA);

    if (pourcentageTVA < 0) throw new
    ArgumentException(nameof(pourcentageTVA));

    return tarifHT + tarifHT * pourcentageTVA / 100;
}
```

Example

```
public class CalculPourcentageMock : ICalculPourcentageTVA
{
    public double PourcentageTVAViaTypeTVA(TypesTVA typeTVA)
    {
        return 20;
    }
}
```

Example

```
[TestMethod()]
public void CalculTVAViaTypeTVATest() {
    // Arrange
    var prixHT = 10;
    var typeTVA = TypesTVA.Normal;
    var prixTTC = 12;
    var calculTaxe = new CalculTaxe(new CalculPourcentageMock());
    // Act
    var result = calculTaxe.CalculTVAViaTypeTVA(prixHT, typeTVA);
    // Assert
    Assert.AreEqual(prixTTC, result);
}
```

Moq

- La création de classe de mock, stubs, fakes, etc... peut être fastidieuse
- Pour aller plus vite on peut utiliser des framework de mocking (ou stubs, fakes, etc...)
- Moq est un des framework de générateur de mock les plus utilisé en .NET

Moq

```
[TestMethod()]  
  
public void CalculTVAViaTypeTVATestMoq() {  
    var mock = new Mock<ICalculPourcentageTVA>();  
  
    mock.Setup(library => library.PourcentageTVAViaTypeTVA(TypesTVA.Normal)).Returns(20);  
  
    // Arrange  
  
    var prixHT = 10;  
  
    var typeTVA = TypesTVA.Normal;  
  
    var prixTTC = 12;  
  
    var calculTaxe = new CalculTaxe(mock.Object);  
  
    // Act  
  
    var result = calculTaxe.CalculTVAViaTypeTVA(prixHT, typeTVA);  
  
    // Assert  
  
    Assert.AreEqual(prixTTC, result); }
```

Test du repository - Fake – ASP.NET Core – EF Core

```
[TestMethod()]  
public void TestRepoOk () {  
    // Créer une BDD en mémoire dédié au test  
    // Nécessite le package Microsoft.EntityFrameworkCore.InMemory  
    var builder = new DbContextOptionsBuilder<WikyContext>()  
        .UseInMemoryDatabase("Wiky");  
    var context = new WikyContext(builder.Options);  
    context.Database.EnsureDeleted(); // Supprime la BDD entre chaque test  
    // Act + Assert  
}
```

Gestion de la sécurité

Présentation

- Les échanges entre un client et un serveur nécessitent d'être sécurisés selon plusieurs aspects :
 - **Authentification** : le client veut être sûr qu'il dialogue avec le serveur sélectionné et inversement. L'utilisateur ne pourra donc se connecter que si son login et son mot de passe sont reconnus.
 - **Autorisation** : on peut limiter l'accès aux ressources d'un serveur à un groupe d'utilisateurs.
 - **Confidentialité** : les informations qui transitent entre le client et le serveur ne doivent pas être lisibles par d'autres.
 - **Intégrité** : les informations transmises ne doivent pas être modifiées avant d'être reçues.

Cross-Site Request Forgery - XSRF

- L'objet de cette attaque est de transmettre à un utilisateur authentifié une requête HTTP falsifiée qui pointe sur une action interne au site, afin qu'il l'exécute sans en avoir conscience et en utilisant ses propres droits.
- L'utilisateur devient donc complice d'une attaque sans même s'en rendre compte. L'attaque étant actionnée par l'utilisateur, un grand nombre de systèmes d'authentification sont contournés.

Cross-Site Request Forgery - XSRF

```

```

```
<form id="theForm" action="https://myBank.com/DoTransfer"
method="post">
    <input type="hidden" name="toAcct" value="67890" />
    <input type="hidden" name="amount" value="250.00" />
</form>
<script type="text/javascript">
    document.getElementById('theForm').submit();
</script>
```

Cross-Site Request Forgery - XSRF

- Il faudra rajouter l'attribut `[ValidateAntiForgeryToken]` sur nos actions POST
- ASP.NET ajoutera automatiquement le jeton anticontrefaçon à nos formulaire (on peut aussi le forcer à la main avec `@Html.AntiForgeryToken()`)

`[HttpPost]`

`[ValidateAntiForgeryToken]`

```
public ActionResult Index(Message message)
```

- *The required anti-forgery cookie "__RequestVerificationToken" is not present*

Mécanisme d'authentification par formulaire

- Le client envoie une requête HTTP au serveur, pour accéder à une ressource protégée.
- Le processus ASP.NET regarde si cette requête contient un cookie d'identification valide
 - S'il est valide, l'accès à la ressource est autorisé.
 - Sinon, l'utilisateur est automatiquement redirigé vers la page de connexion de l'application.
- L'utilisateur tente de se connecter en fournissant un nom d'utilisateur et un mot de passe :
 - Si l'authentification réussit, alors l'accès à la ressource est autorisé et l'utilisateur est automatiquement redirigé vers l'application.
 - Si l'authentification échoue, alors l'accès est refusé.

Présentation de ASP.NET Core Identity

- Le ASP.NET MVC5 propose un mécanisme d'authentification : Asp.net Identity
- ASP.NET Core propose un mécanisme similaire : ASP.NET Core Identity
- Gère le schéma des informations utilisateur et de profil, tout en restant personnalisable
- Persistance des données d'authentification dans une base que l'on attaque avec EF (code first) :
 - Il est possible de choisir tout type de conteneur: SharePoint, Windows Azure Storage Table, NoSql database...
- Livré avec des fournisseurs spécifiques pour divers réseaux sociaux:
 - Microsoft Account, Facebook, Twitter, Google...
- Intégration OWIN :
 - Plus de dépendance avec System.Web et donc avec IIS

ASP.NET Core Identity

- ASP.NET Core Identity est distribué via NuGet
- Lorsque vous créez un projet ASP.NET Core dans Visual Studio, un mécanisme d'authentification par défaut est mis en place :
 - authentification en Individual User Account
- De base toutes les vues relatives à la gestion des utilisateurs sont incluse dans le package `Microsoft.AspNetCore.Identity.UI`
- On peut override les vues que l'on veut en effectuant une génération des vues de Identity (new Scaffolded, Identity, généré dans une area)
- Cette génération montre un exemple de l'utilisation de cette librairie

Gestion des autorisations

- La gestion des autorisations est réalisée par annotations :
 - `[Authorize]` : sécurise l'accès à la page en redirigeant l'utilisateur vers la page de login.
 - `[Authorize (Roles="nomRole")]` : restreint l'accès à la page aux utilisateurs ayant le rôle donné.
 - `[AllowAnonymous]` : autorise un accès à tous les utilisateurs. Il est utilisé quand la classe est sécurisée et que l'on veut autoriser l'accès uniquement sur certaines méthodes.
- Les annotations peuvent être appliquées sur les méthodes des contrôleurs ou directement sur la classe.

Gestion des utilisateurs et des rôles

```
private readonly UserManager<ApplicationUser> _userManager;  
private readonly SignInManager<ApplicationUser> _signInManager;  
private readonly RoleManager<IdentityRole> _roleManager;  
  
public HomeController(UserManager<ApplicationUser> userManager,  
RoleManager<IdentityRole> roleManager, SignInManager<ApplicationUser>  
signInManager) {  
    _userManager = userManager;  
    _roleManager = roleManager;  
    _signInManager = signInManager;  
}
```

Création d'un utilisateur personnalisé

// Il faudra changer toutes les référence à IdentityUser vers ApplicationUser et changer l'héritage de IdentityDbContext

```
public class ApplicationUser : IdentityUser
{
    public string CustomTag { get; set; }
}
```

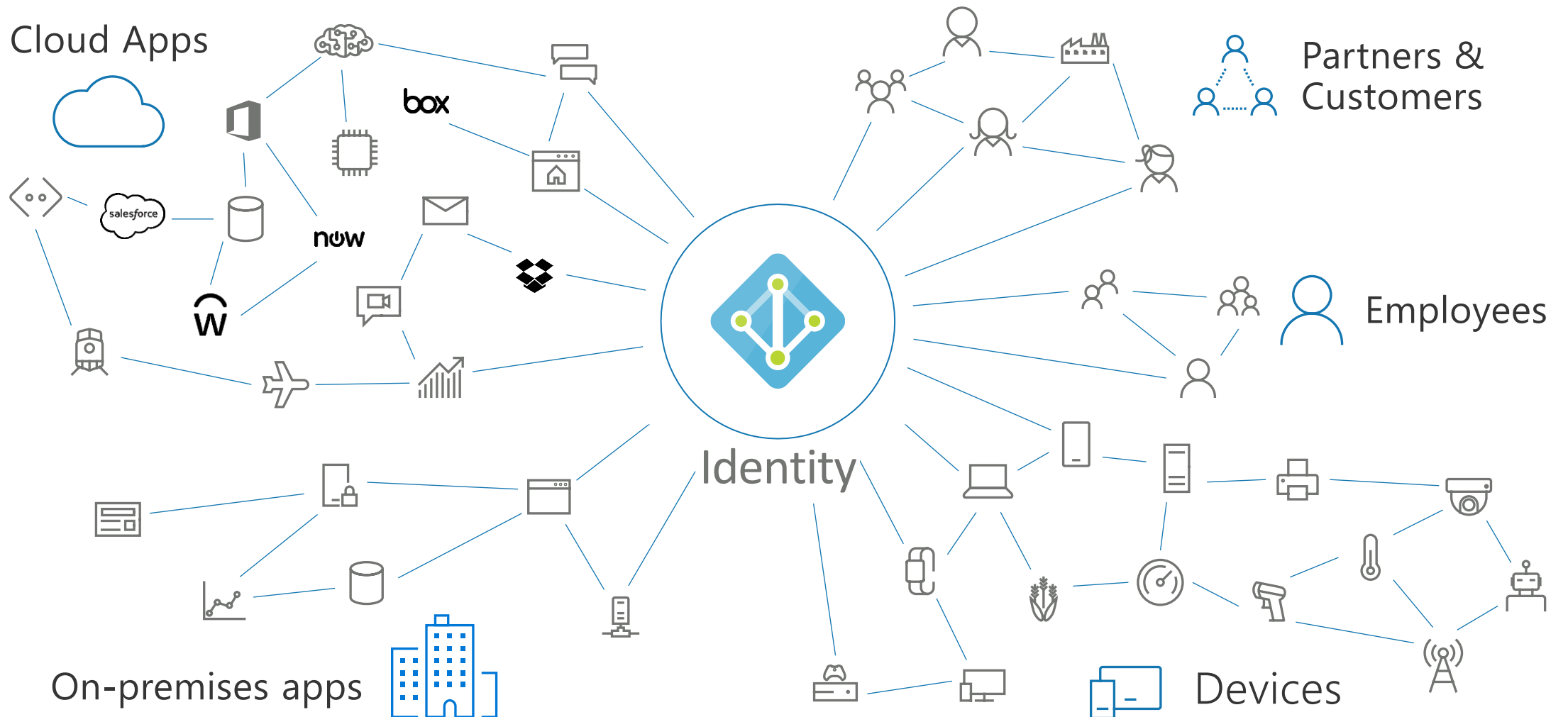
```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
```

<https://docs.microsoft.com/fr-fr/aspnet/core/security/authentication/customize-identity-model?view=aspnetcore-3.1#the-identity-model>

Serveur d'identité

- La gestion des utilisateurs dans la base de données de l'application n'est pas recommandée
- Il sera recommandé de déporter la gestion des utilisateurs sur un serveur dédié
- Pour une sécurité encore plus forte on pourra gérer nos utilisateurs sur un Active Directory

L'identity au centre de interactions



Standard

- Au fil du temps de nombreux standards on émergés pour la gestion de l'identité (authentification & autorisation)
 - OAuth
 - OIDC
 - WS-Federation
 - SAML
 - ...



OAuth / OIDC

- OAuth (Open Authorization) est un standard de délégation d'accès
 - Très utilisé dans le web
 - Permet l'authentification via un service tiers
 - Conçu pour fonctionner avec le protocole HTTP
 - Utilise un système d'échange de jetons
- OIDC (OpenID Connect) est une couche d'authentification
 - Basée sur le protocole OAuth
 - Permet de vérifier l'identité d'un utilisateur

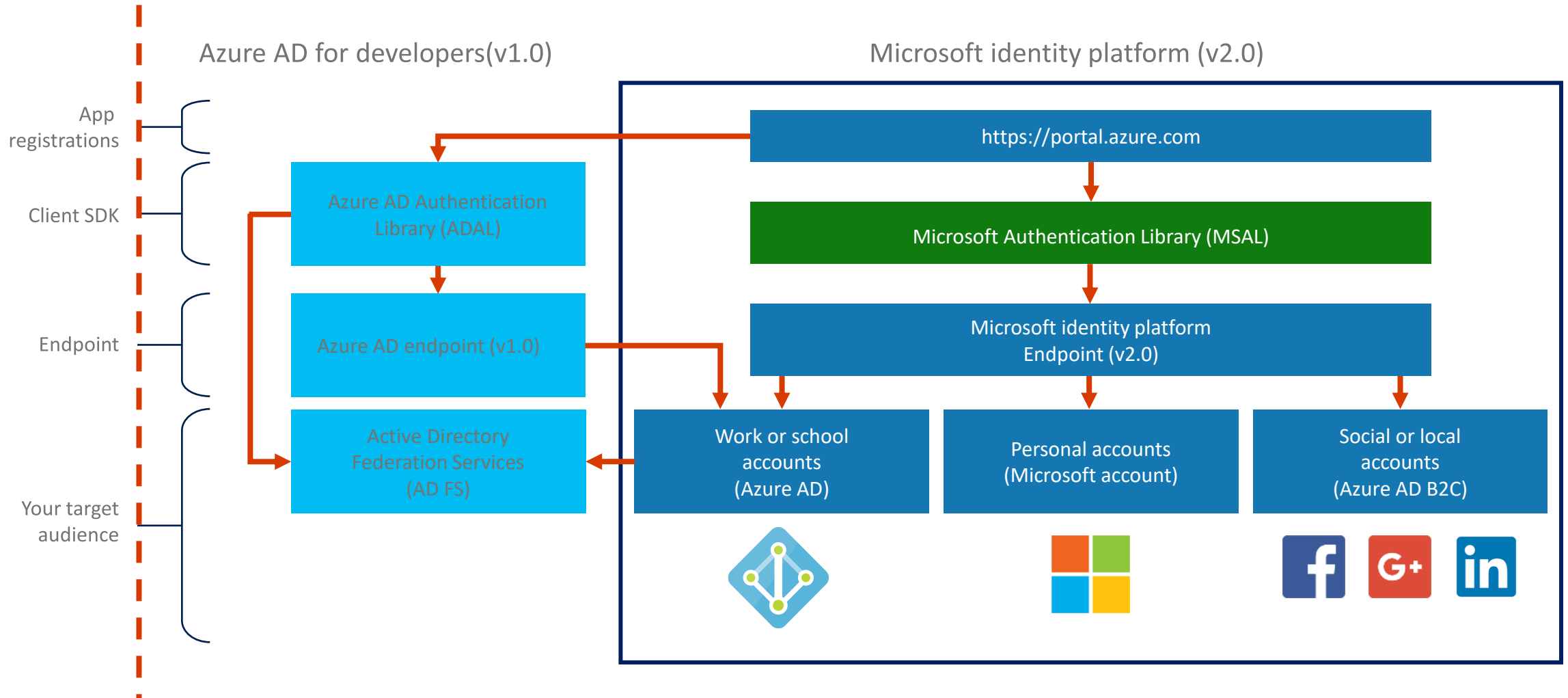
Fournisseur d'identité

- Il existe de nombreux fournisseurs d'identité (Identity Provider) utilisé dans le web (site web et api)
 - Des services qui permettent
 - d'identifier un client
 - de fournir un jeton (token) au client pour qu'il puisse accéder à une ressource
 - Parmi eux :
 - Azure Active Directory
 - Duende IdentityServer (anciennement IdentityServer4)
 - Microsoft
 - Facebook
 - Google
 - Twitter
 - ...

Microsoft Identity Platform

- Microsoft Identity Platform
 - C'est un meta-service de fournisseurs d'identité
 - Fournis un point de terminaison (endpoint)
 - Connectable à de nombreux fournisseurs d'identité
 - Accessible via la plateforme Azure
 - Supporte de nombreux flux d'authentification

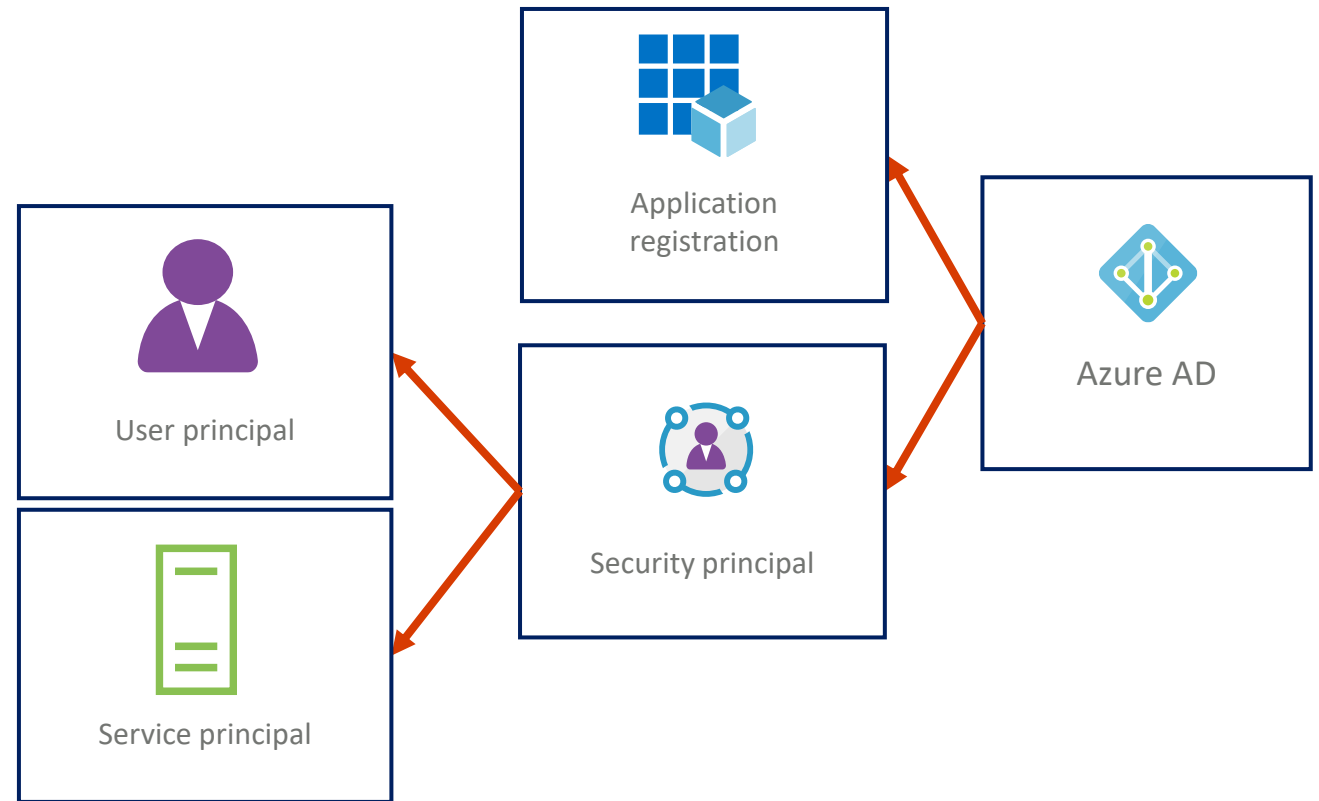
Microsoft identity platform



Azure AD

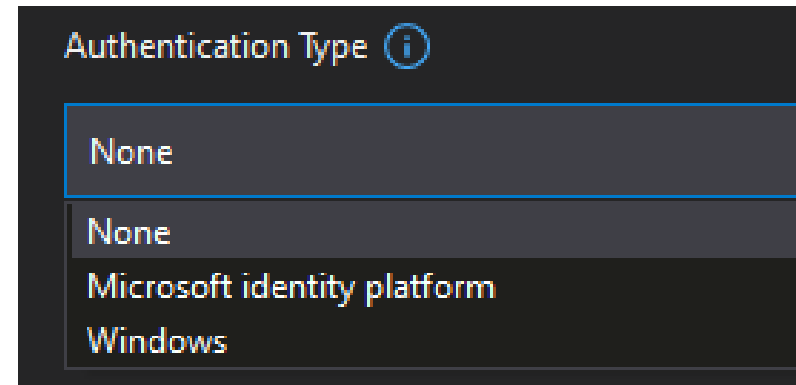
Azure AD a deux types d'objets:

- Les applications enregistrées
- Les “principal”
 - User principal
 - Service principal



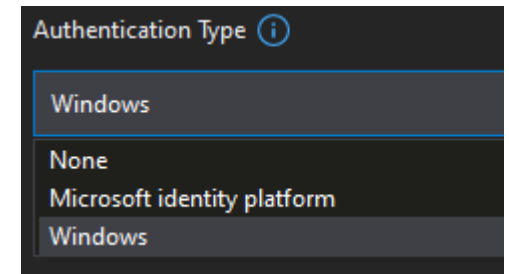
Type d'authentification de l'API

- Lors de la création d'un projet, Visual Studio nous propose deux type d'authentification :
 - Microsoft Identity Platform
 - Windows

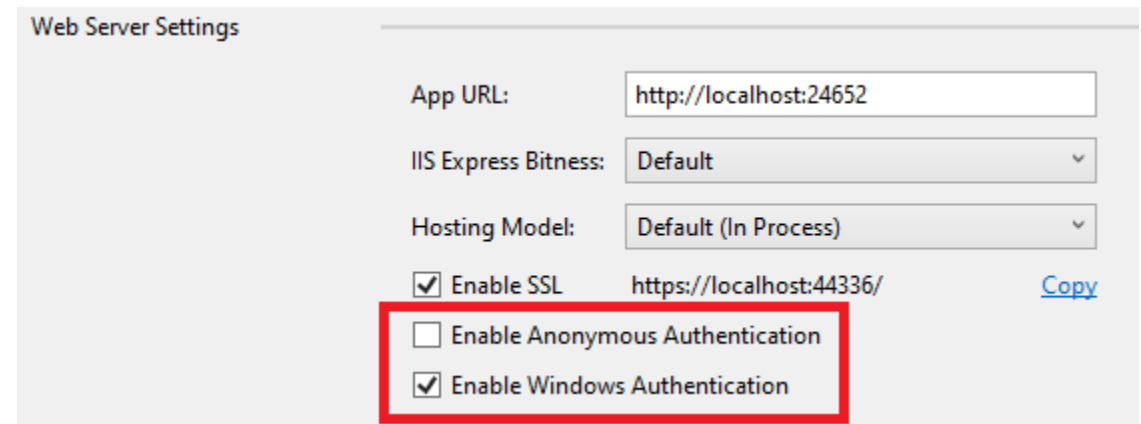


Windows Authentication

- En choisissant Windows Authentication, le web server est configuré pour authentifier l'utilisateur par
 - Windows Active Directory
 - Le compte utilisateur Windows local de la machine

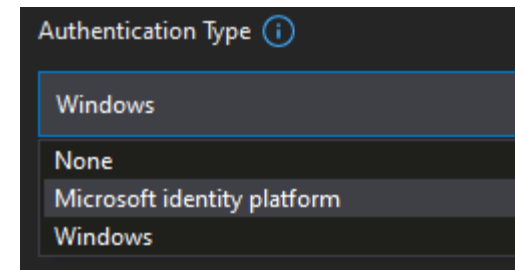


- Hormis la configuration dans les propriétés du projet, le code du projet est inchangé



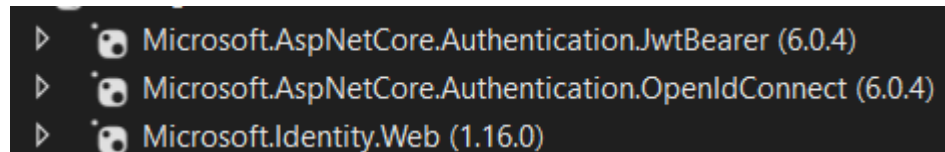
Microsoft Identity Platform

- En choisissant Microsoft Identity Platform, plusieurs modifications ont automatiquement lieu sur le projet :
 - Des packages sont ajoutés
 - Un service est ajouté dans ConfigureServices (startup.cs)
 - Un middleware est ajouté dans Configure (startup.cs)
 - Le fichier de configuration est modifié



Microsoft Identity Platform

- Les packages pour JWT, OIDC et les éléments d'Identity Platform



- Le service
 - Avec l'utilisation du JWT token par défaut
 - L'appel au fichier de configuration où se trouve les informations de l'Azure AD

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)  
                .AddMicrosoftIdentityWebApi(builder.Configuration.GetSection("AzureAd"));
```

Microsoft Identity Platform

- Le middleware est aussi modifié pour supporter l'authentification et l'autorisation

```
app.UseHttpsRedirection();
```

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

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

```
app.MapControllers();
```

Microsoft Identity Platform

- Le fichier appsettings.json est modifié pour ajouter les informations du tenant et de l'enregistrement de l'application dans Azure AD

```
"AzureAd": {  
  "Instance": "https://login.microsoftonline.com/",  
  "Domain": "qualified.domain.name",  
  "TenantId": "22222222-2222-2222-2222-222222222222",  
  "ClientId": "11111111-1111-1111-1111111111111111",  
  
  "CallbackPath": "/signin-oidc"  
},
```

* Rappel : dans le cas d'une application multi-tenant le domaine est juste à la valeur « common »

Microsoft Identity Platform

- Après la création du projet, Visual Studio ouvre une fenêtre permettant l'enregistrement de l'application ou de retrouver l'application enregistrée dans Microsoft Identity Platform



- Cette même interface permet d'ajouter les éléments de Microsoft Identity Platform (packages/config/code) dans un projet existant

Authentification et Autorisation

- Pour la gestion de l'authentification à l'utilisateur, la Web API dispose de l'attribut [[Authorize](#)]
- Pour autoriser une action sans authentification, la Web API dispose de l'attribut [[AllowAnonymous](#)]
- Ces attributs peuvent être placés au niveau d'une méthode ou d'un contrôleur

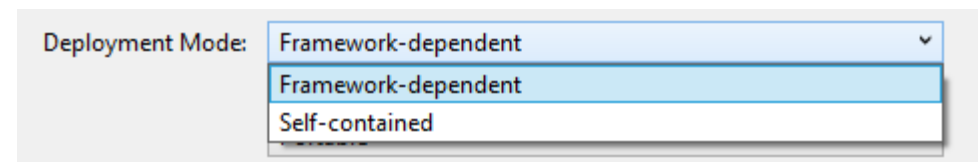
Hébergement

Publication - Prérequis

- Pour publier une application il est nécessaire de connaître quelques éléments importants sur la machine d'hébergement :
 - A-t-elle déjà .NET Core ?
 - Quelle version de .NET Core ?
 - Quel est l'OS utilisé ?
 - 32 bits ou 64 bits ?
 - Comment sera exécuté l'application ?
 - Est-ce dans le cloud ? On-premise ?

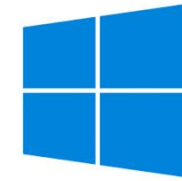
Publication - Prérequis

- Quelle version de .NET Core ?
 - Pour la production on peut viser les framework en LTS (Long Term Support)
 - Ex : .NET Core 3.1 ou .NET 6
- La machine a-t-elle déjà .NET Core ?
 - Oui ou je vais l'installer avant le déploiement
 - Mode de déploiement : **Framework-dependent**
 - Non, dans ce cas on peut déployer celui-ci avec l'application
 - Mode de déploiement : **Self-contained**



Publication - Prérequis

- Quel est l'OS utilisé ?
- 32 bits ou 64 bits ? ARM ?
 - Il faut en effet choisir le runtime le plus adapté
 - (ex : Linux-x64, Win-x64)
 - Dans le cas d'un déploiement en mode framework-dependant il est possible de choisir le runtime « **portable** » qui s'adaptera à chaque machine
- Comment sera exécutée l'application?
 - Dans un serveur web (IIS, Nginx, Apache...)
 - Dans un service Windows



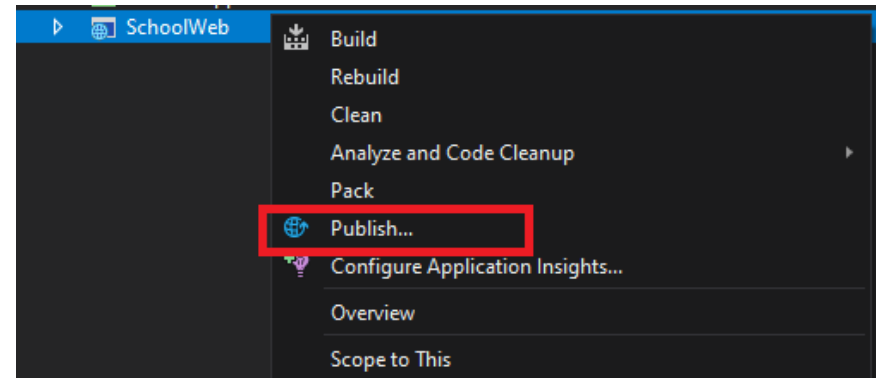
Publication - Prérequis

- Est-ce dans le cloud ? On-premise ?
 - Plusieurs possibilités existent mais ne sont pas forcément toutes disponibles dans votre entreprise
 - Publication dans un dossier de la machine ou partagé sur le réseau
 - Par FTP
 - Avec un package de déploiement
 - Directement dans IIS
 - En utilisant un container
 - A travers une Machine Virtuelle Azure
 - Dans un App Service
 - Avec Azure Container Instances

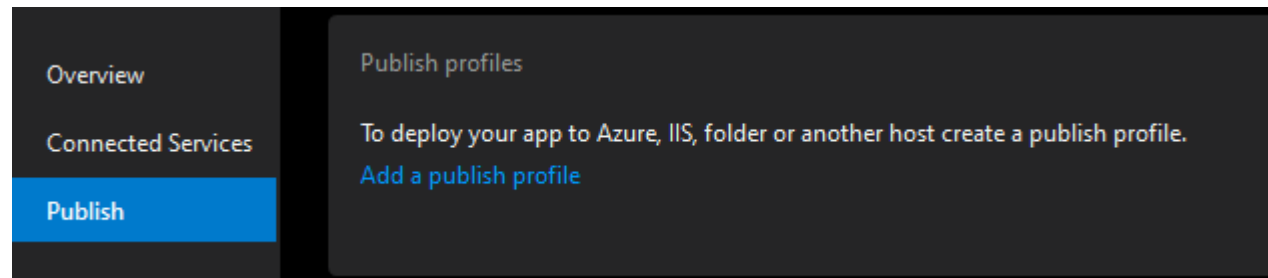


Publication avec Visual Studio

- Dans Visual Studio un bouton « publier » permet d'accéder au menu de publication

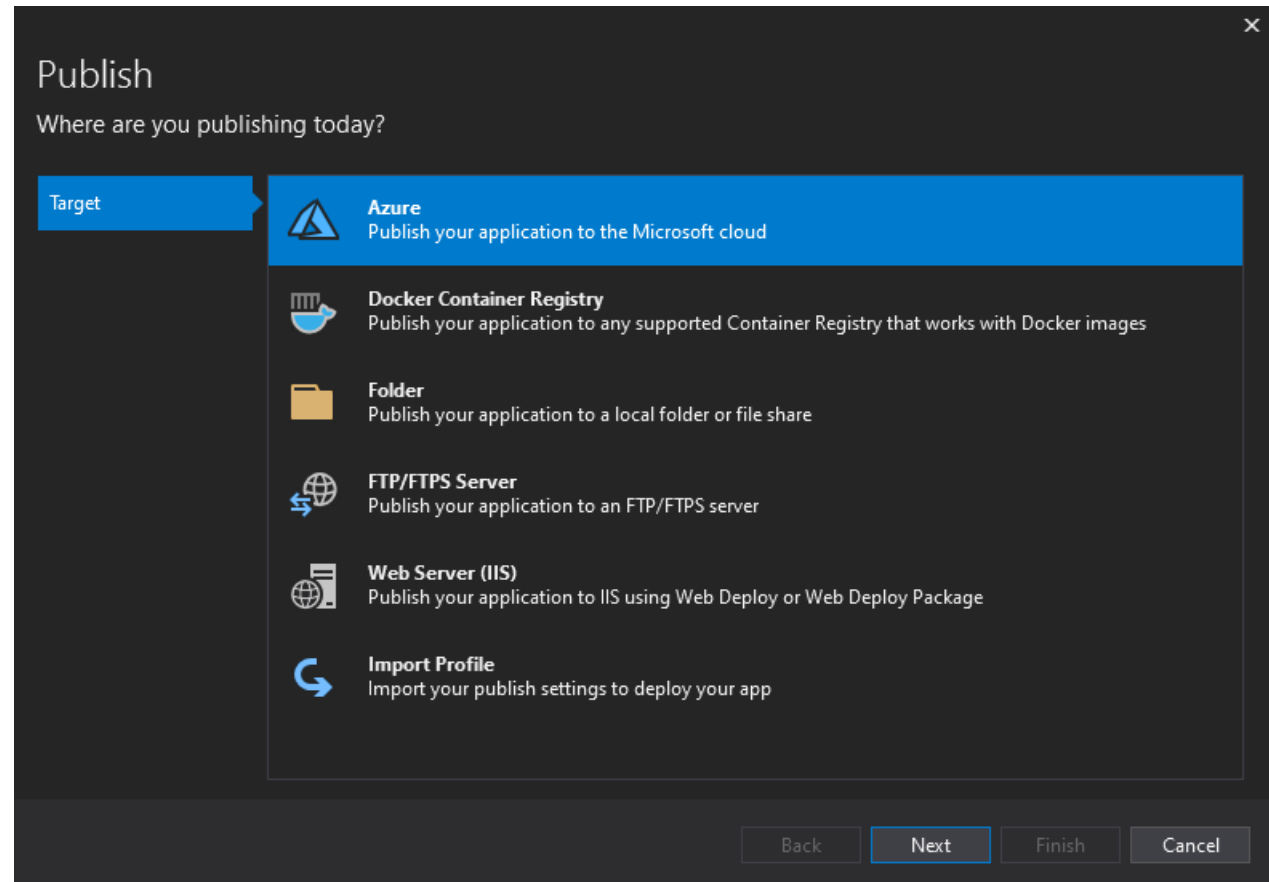


- Puis d'ajouter un profil de publication



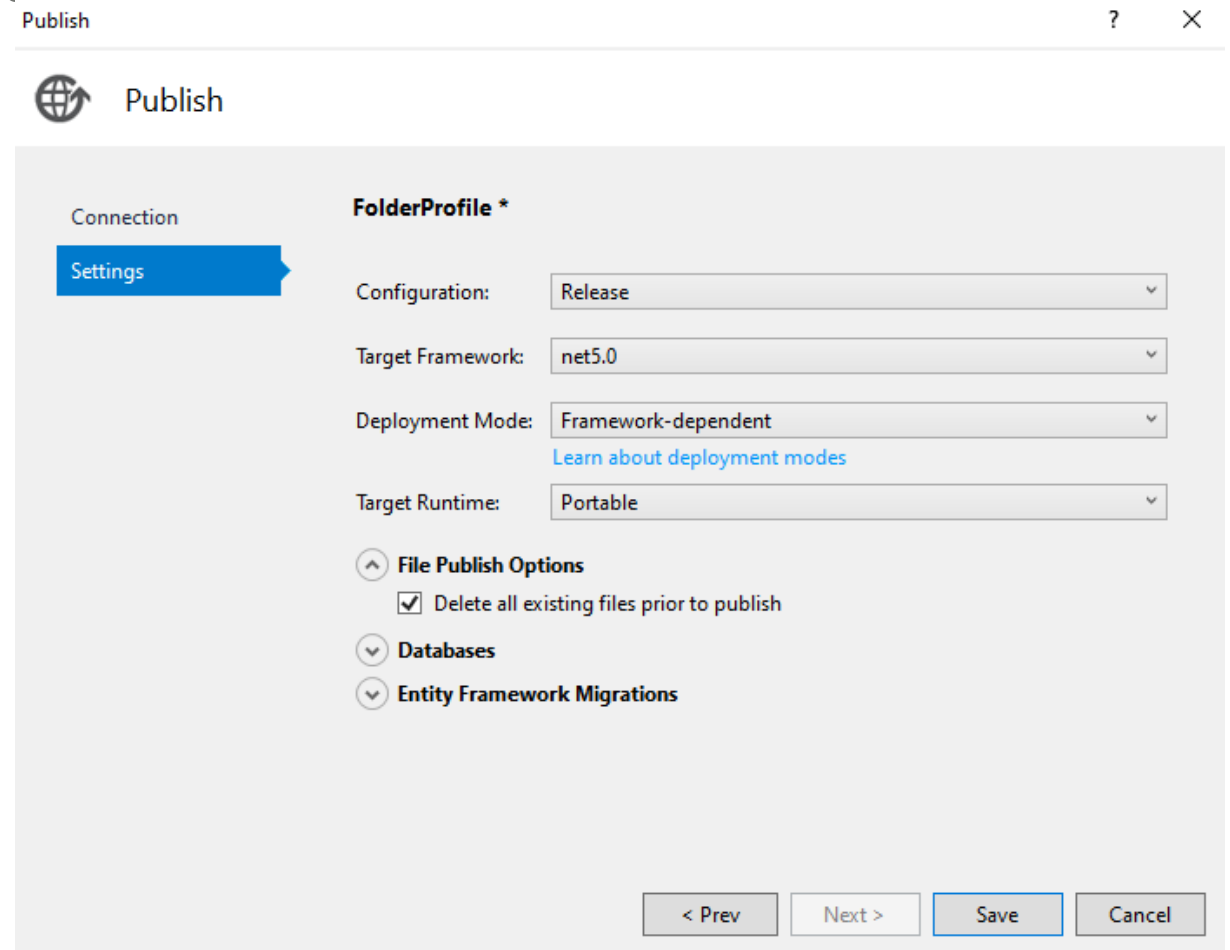
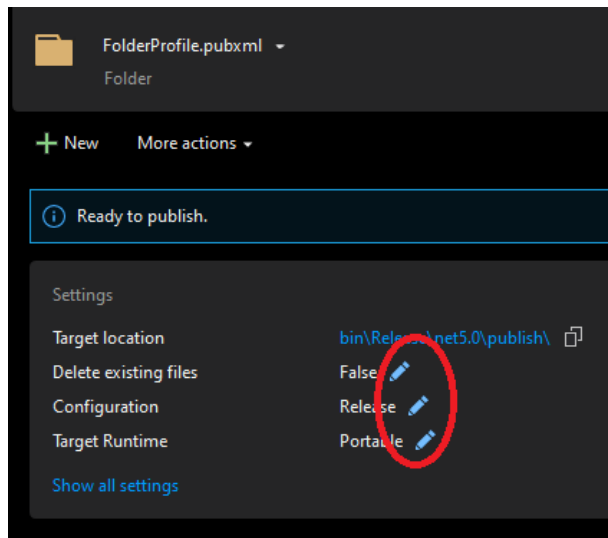
Publication avec Visual Studio

- La fenêtre nous permet alors d'ajouter un profil de publication



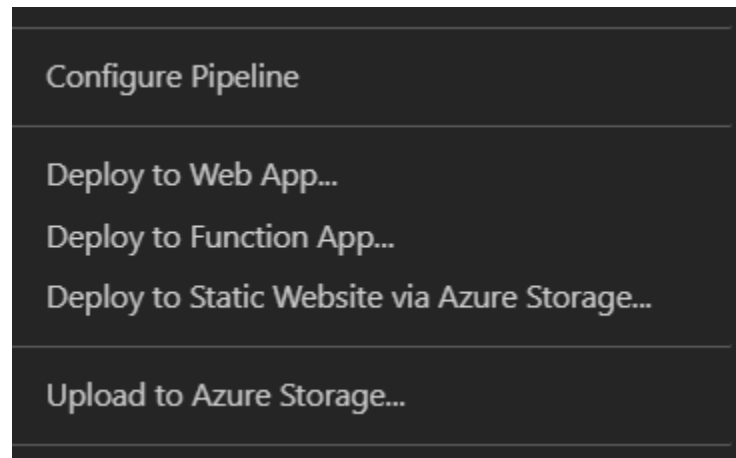
Publication avec Visual Studio

- Une fois le profil de publication sélectionné et configuré, on peut modifier les paramètres par défaut



Publication avec Visual Studio Code

- En fonction des extensions installées dans VS Code, un clic droit sur un fichier du projet peut permettre l'accès au menu de publication



* Ici l'extension Azure Tools a été ajouté

Publication avec la ligne de commande

- La ligne de commande « dotnet » ajoutée avec .NET Core permet aussi de publier

CLI .NET

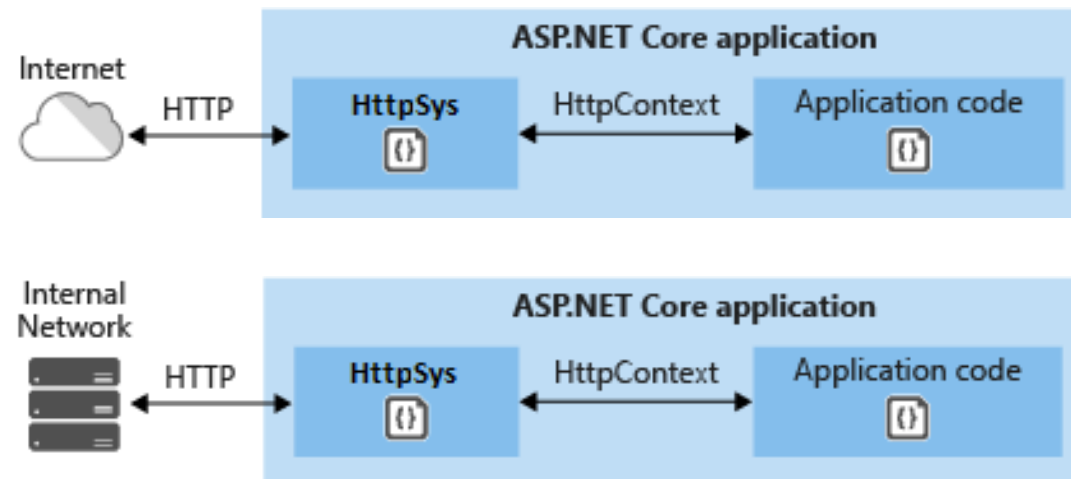
```
dotnet publish [<PROJECT>|<SOLUTION>] [-a|--arch <ARCHITECTURE>]
  [-c|--configuration <CONFIGURATION>]
  [-f|--framework <FRAMEWORK>] [--force] [--interactive]
  [--manifest <PATH_TO_MANIFEST_FILE>] [--no-build] [--no-dependencies]
  [--no-restore] [--nologo] [-o|--output <OUTPUT_DIRECTORY>]
  [--os <OS>] [-r|--runtime <RUNTIME_IDENTIFIER>]
  [--self-contained [true|false]]
  [--no-self-contained] [-v|--verbosity <LEVEL>]
  [--version-suffix <VERSION_SUFFIX>]

dotnet publish -h|--help
```

Ex : dotnet publish SchoolWeb -c Release -a win-x64 -f netcoreapp3.1

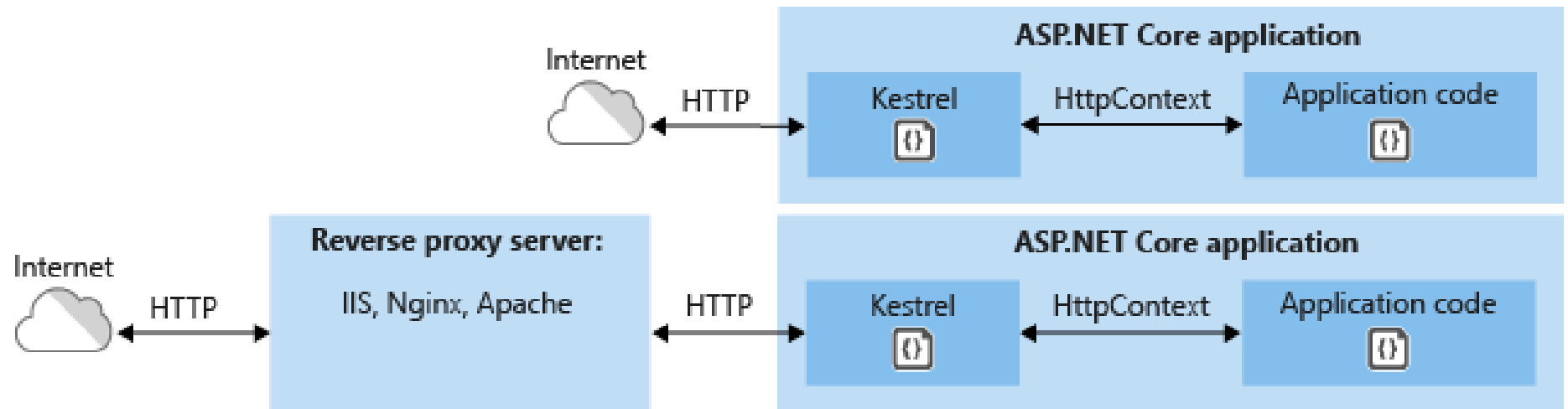
Solution HTTP.sys

- HTTP.sys est un serveur web conçu pour .NET Core
 - Alternative à Kestrel
 - Open Source
 - **Windows uniquement**
 - Compatible avec l'authentification Windows
 - Ne fonctionne pas avec IIS ou IIS Express (contrairement à Kestrel)



Solution Kestrel

- Kestrel est un serveur web conçu pour .NET Core
 - Extrêmement **performant**
 - **Multi-plateforme** (Windows, Linux, OS X)
 - Open-source
 - Pas totalement un serveur web (manque de fonctionnalités)
 - Couplage recommandé avec IIS, NGINX, APACHE...



IIS

- Pour qu'une application ASP.NET Core soit exécutée par IIS:
 - Il faut ajouter le **Windows Hosting Bundle**
 - Le bundle contient un runtime de .NET Core
 - Créer un site dans IIS
 - Configurer les chemins et les ports exposés
 - Passer .NET CLR version à « no managed code » dans l'application Pool
 - Déployer l'application
 - En choisissant Windows (x86 ou x64) ou Portable comme option de publication

Razor Pages

MODULE 19

Razor Pages

```
public class IndexModel : PageModel
{
    public string Message { get; private set; } = "PageModel in C#";
    public IndexModel()
    {
    }
    public void OnGet()
    {
        Message += $" Server time is { DateTime.Now }";
    }
}
```

Razor Pages

```
@page
```

```
@model IndexModel
```

```
<p>
```

```
    @Model.Message
```

```
</p>
```

Razor Pages

```
[BindProperty]

public Post Post { get; set; }

public async Task<IActionResult> OnPostAsync() {
    if (!ModelState.IsValid)
    {
        return Page();
    }

    // Utilisation de la propriété Post
    return RedirectToPage("./Index");
}
```

Razor Pages

```
@page
```

```
@model IndexModel
```

```
<form method="post">
```

Name:

```
<input asp-for="Post.Name" />
```

```
<input type="submit" />
```

```
</form>
```

Blazor

Blazor

- Nouvelle façon de penser le front en C#
- Le C# peut être géré côté serveur ou côté navigateur avec WebAssembly
- Permet d'avoir des UIs interactives, réutilisables en C# et HTML / CSS / JS (interopérabilité)
- WebAssembly est standardisé et exécuté comme du Javascript (sandbox, au moins aussi performant que du JS natif, etc...)
- Permet d'avoir du code C# en .NET Core partagé entre le front et le back
- De nombreux composants sont déjà disponibles en plus via Telerik, Syncfusion, etc...

Fin de la formation