



# ASP.NET MVC

Cours basé sur la technologie ASP.NET en utilisant le pattern MVC en vue de créer une application web

# Sommaire

- Introduction au développement web avec ASP.NET
- Les bases de l'ASP.NET MVC
- Comprendre le mécanisme MVC
- Etude du modèle
- Etude des vues et des contrôleurs
- Validation des données
- Appels asynchrones avec AJAX
- Annexes

# Introduction à ASP.NET

- ASP.NET est la plateforme de développement Microsoft permettant le développement d'application internet.
- Basé sur du C#.
- Offre deux logiques de développement :
  - WebForm
  - MVC
- C'est une application Web comme on peut en retrouver en JAVA , en PHP, etc.

# Introduction à ASP.NET WebForm

- Créé en 2002, actuellement WebForm est en version 5.
- Offre une couche d'abstraction permettant de réaliser des applications web comme si nous réalisions des applications Windows.
- Inconvénient :
  - cycle de vie de la page complexe.
  - Moins de maîtrise du HTML généré.
  - Beaucoup d'efforts pour contourner certaines limitations.
  - Etc.

# Introduction à ASP.NET MVC

- Version basée sur le patron de conception MVC apparu vers 2010.
- Plus robuste et plus récent que la version WebForm.
- Avantages :
  - Une totale liberté au niveau du HTML, CSS, JavaScript.
  - Utilisation de jQuery.
  - Utilisation d'AJAX, etc.
- ASP.NET MVC sera au cœur de ce cours! (c'est la méthode de développement offrant la meilleure qualité).

# Pourquoi ASP.NET MVC?

- ASP.NET MVC se rapproche de beaucoup de ce que vous avez déjà pu faire. Vous aurez quelques particularités spécifiques à apprendre, mais la logique sera globalement la même.
- Maîtrise totale du modèle HTTP ainsi que du code HTML généré.
- La maîtrise du HTML est un élément déterminant pour faire un site qui fonctionne sur des PC, sur des tablettes ou des smartphones.
- Permet d'intégrer le « responsive design ».
- Patron de conception robuste, fiable, testable.

# Les outils de développement

- L'IDE Visual Studio permet de développer des application internet en ASP.NET MVC.
- Version utilisée de ASP.NET MVC : 4.5.
- Possibilité d'ajouter vos extensions favorites pour faciliter le développement et les tests.
- Possibilité d'ajouter des librairies tierces comme jQuery, etc... via le Nuget Package Manager.



# Les Frameworks

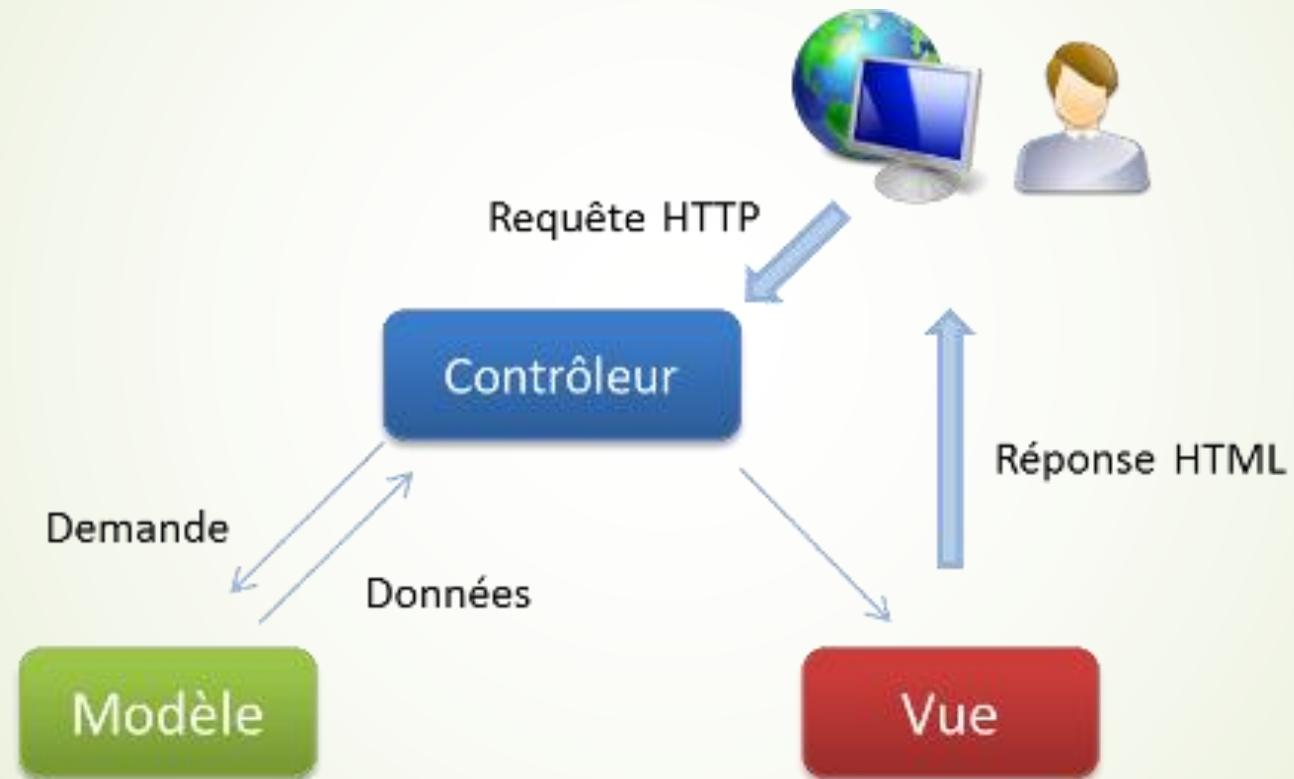
- ASP.NET MVC permet d'intégrer toutes les bibliothèques du marché, c'est un projet WEB :
  - JQuery pour la partie JS.
  - Knockout / ReactJs / etc.
  - Bootstrap pour la partie Design.
  - Kendo UI / Devexpress (outils payants proposant des composants supplémentaires).
  - Etc.



# Le pattern MVC

- MVC est un patron de conception utilisable dans de multiples technologies (PHP, JAVA, etc.) permettant de séparer l'affichage des informations, les actions des utilisateurs et l'accès aux données.
- MVC : Modèle – Vue – Contrôleur
- Avantage de la structure MVC :
  - Séparation claire et nette de chaque couche
  - Architecture robuste de plus en plus utilisée
  - Maintenance et mises à jour plus aisée
  - Permet des tests automatisés et plus aisés

# Le pattern MVC : Schéma



# M comme Modèle

- Couche composée d'un ensemble de classe permettant d'accéder à une source de données (exemple : base de données, services web, fichiers XML, etc.)
- Le modèle ne connaît ni la vue, ni le contrôleur. Il est seulement mis à disposition d'un utilisateur en lecture ou en écriture
- Intégration possible (et recommandée) d'un ORM et d'une couche d'accès aux données dans cette couche (voir cours 1)
- Testable unitairement

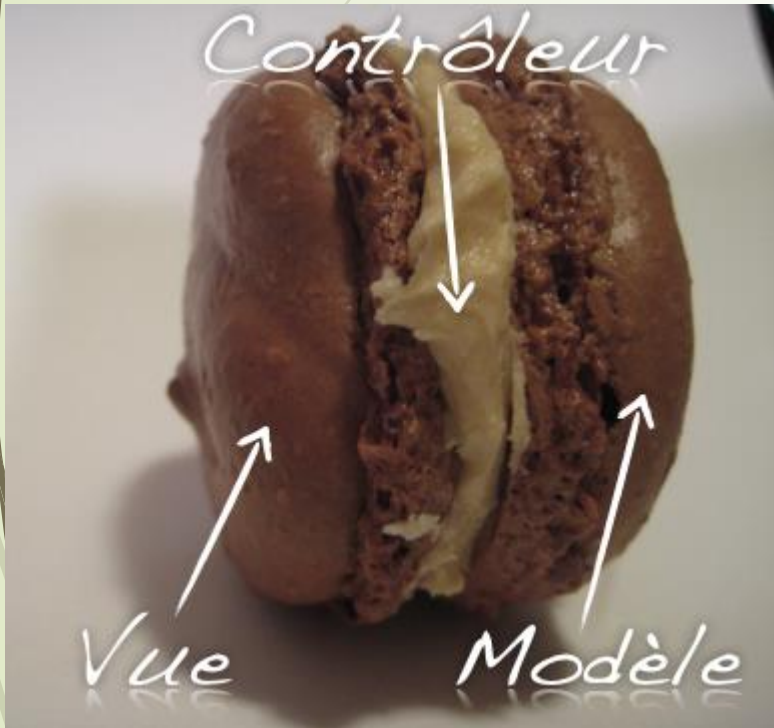
# V comme Vue

- La vue permet de présenter à l'écran le résultat de l'action initié par un utilisateur
- La vue présente les données issues du modèle
- Ecrite principalement avec du code HTML5 et CSS3
- Possibilité d'ajouter du code JavaScript (jQuery, Ajax, etc.)
- ASP.NET MVC offre une collection de contrôle utilisateur pour être plus efficace dans la création des vues (dropdownlist, combobox, Bouton d'action, champ texte, etc.)

# C comme Contrôleur

- Le contrôleur fait le lien entre la vue et le modèle. Il détermine quels traitements doivent être effectués pour une action donnée
- Le contrôleur est « le cœur » du système
- Contrôle les données venant de l'utilisateur
- Récupère les données du modèles
- Retourne une vue (réponse HTML) à l'utilisateur en utilisant le modèle

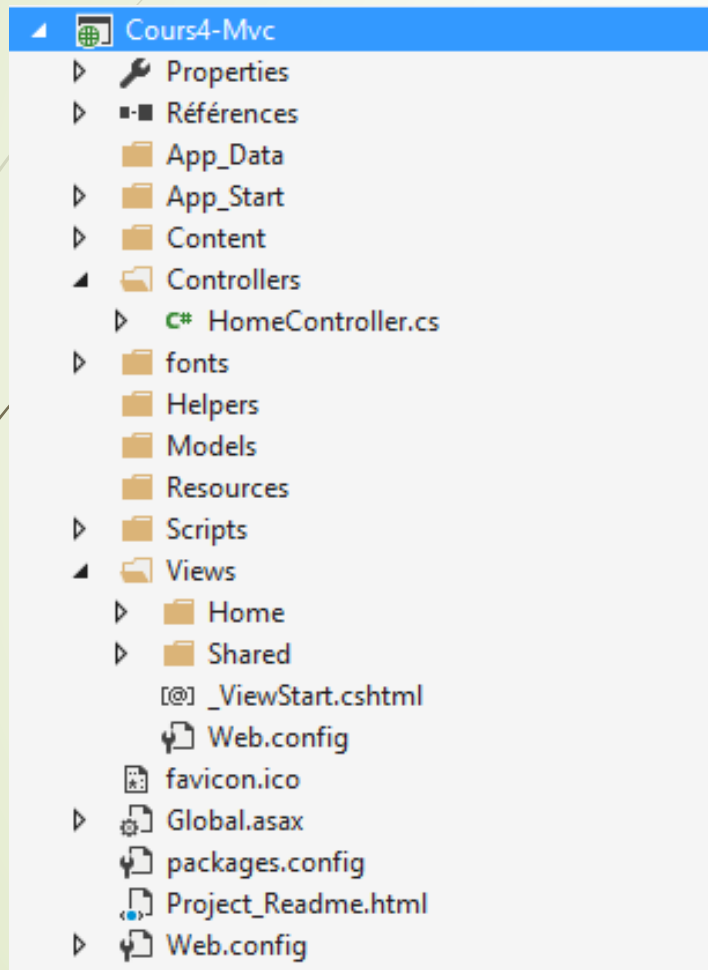
# Schéma MVC



- Le contrôleur sert à lier la vue et le modèle.
- Ne jamais mélanger des données de la vue et des données du modèle.
- Les contrôles se font dans le contrôleur.
- L'appel à la BusinessLayer (ou EntityFramework) permet d'éviter de réécrire le modèle.
- Notion de ViewModel?



# Projet de type MVC : Arborescence



- Architecture type d'un projet web ASP.NET MVC
- Respecter les conventions de nommage (voir diapositive sur les conventions de nommage)
- Architecture auto-générée par Visual Studio (ou presque)



# Projet de type MVC : Arborescence

- App\_Start : stocke les classes de configuration du projet (routes, bundles, etc.)
- Content : stocke les différentes ressources pour le site web (images, css, etc.)
- Controllers : stocke les contrôleurs MVC
- Fonts : stocke les polices spécifiques (si besoin) du site web
- Helpers : stocke les différents Helper MVC et les extensions
- Models : stocke les ViewModels (attention, différents du modèle du MVC)
- Resources : stocke les fichiers de localisation (application multi langue)
- Scripts : stocke les différents fichier JavaScript (exemple : jQuery)
- Views : stocke les vues MVC et les vues partielles
- Racine : stocke les fichiers de configuration et de démarrage de l'application (exemple : global.asax)

# Conventions ASP.NET MVC

Quelques bonnes pratiques à respecter :

- Les contrôleurs se trouvent par défaut dans le répertoire Controllers donc inutile de configurer ou de spécifier ce point
- Un contrôleur est une classe suffixée du mot Controller (exemple : la classe **HomeController** est le contrôleur **Home**)
- Les vues en rapport avec un contrôleurs se trouvent dans le répertoire Views, dans un sous-répertoire du même nom que le contrôleur.
- La vue par défaut d'une action s'appelle Index.cshtml. L'action du contrôleur porte le même nom c'est-à-dire Index
- On peut réutiliser des vues à condition de les placer dans le répertoire **Shared**

# Conventions ASP.NET MVC

Le respect des conventions précédentes permet :

- Simplifier et accélérer le développement
- Réduire la taille et la complexité des fichiers de configuration
- Obtenir une solution « standard »

# Le principe du routing (1/4)

- Notion de routing (ou routage en français) : Une route permet de transformer une URL en une action du contrôleur.
- **Route** = point d'entrée de l'application.
- Les routes sont définies dans le répertoire App\_Start

## Exemple de routes du fichier RouteConfig.cs :

```
1 référence
public class RouteConfig
{
    1 référence
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}
```

## Le principe du routing (2/4)

```
routes.MapRoute(  
    name: "Default",  
    url: "{controller}/{action}/{id}",  
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }  
);
```

- La route s'appelle « Default »
- Elle s'applique pour les requêtes de type /xxx/yyy/zzz.
- Oninstanciera le contrôleur qui aura pour nom **xxxController**.
- On appellera la méthode **yyy()**.
- On pourra passer à la méthode le paramètre **zzz** de type **optionnel**.
- **NB : IgnoreRoute** permet d'ignorer les routes ayant pour extension axd (une ressource).

## Le principe du routing (3/4)

- On peut définir des valeurs par défaut grâce à la propriété **defaults**.

```
routes.MapRoute(  
    name: "Default",  
    url: "{controller}/{action}/{id}",  
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }  
);
```

- Exemple : Si la valeur du contrôleur est absente, le contrôleur utilisé sera HomeController.
- Attention, une seule route par défaut et par application (propriété **name** = Defaults).



# Le principe du routing (4/4)

## ➤ Exercice sur le routing

Ci-joint la route suivante :

```
routes.MapRoute(  
    name: "MesProduits",  
    url: "{controller}/{action}/{id}",  
    defaults: new { controller = "Produit", action = "Index", id = UrlParameter.Optional }  
);
```

Donner l'URL pour :

- Atteindre le contrôleur Produit.
- Appeler la méthode GetProduit du contrôleur Produit avec l'ID 12.

Quelle seront les éléments appelés si je fais :

- /
- /Produit/ListerProduit



# Créer et générer des routes (1/3)

- On peut évidemment ajouter autant de route que nécessaire.
- L'ordre d'ajout des routes est important.
- Exemple : la route `/Calculatrice/Soustraire/10/6` est une route avec deux paramètres qui renverra 4.
- On peut séparer les fragments d'URL par le caractère de son choix (exemple un `-` au lieu d'un `/`) mais c'est déconseillé.

## Créer et générer des routes (2/3)

- On peut définir plusieurs routes dans un fichier de configuration.
- Il faut respecter la règle suivante : On commence toujours par ajouter les routes les **plus spécifiques** pour finir par la route la **plus générale** (idéalement, la **route par défaut**).
- Lorsque le moteur MVC transforme une URL en une action du contrôleur, il parcourt les routes dans l'ordre où elles sont définies et s'arrête dès qu'il trouve une route qui correspond à l'URL.

# Créer et générer des routes (3/3)

- **Avantage du mécanisme de routing**
  - Générer facilement des URL grâce au nom du contrôleur et au nom d'une action.
  - Permettre la navigation entre écran.
  - On peut placer l'URL tout simplement dans une balise `<a></a>` côté client pour naviguer d'un écran à un autre.
  - MVC dispose de **Helpers** pour générer une URL et une balise de type `<a></a>` (voir chapitre sur les vues).
  - MVC permet la génération de route côté contrôleur (exemple : redirection).

# Paramétrer les routes

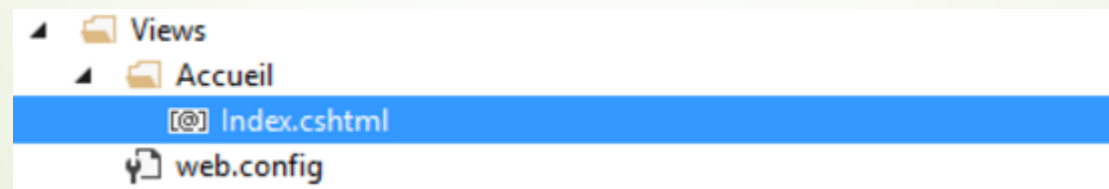
- Il est possible d'ajouter des contraintes sur des éléments de route via la propriété **constraints**.
- Utilisation des expressions régulières (vaste choix de contrôle!).
- Si l'expression régulière n'est pas respectée, la route sera ignorée et donc la méthode du contrôleur ne sera pas appelée.

## Exemple de contraintes :

```
routes.MapRoute(  
    name: "MesProduits",  
    url: "{controller}/{action}/{id}",  
    defaults: new { controller = "Produit", action = "Index", id = UrlParameter.Optional },  
    constraints: new { id = @"\d+" }  
);
```

# Afficher les vues

- Se place dans le répertoire /Views.
- Se place dans un sous-répertoire correspondant au nom du contrôleur à laquelle elle est liée.
- Par convention, elle porte le nom de l'action du contrôleur.
- **Exemple** : une vue retournée par l'action Index du contrôleur Home se place dans le répertoire /Views/Home et s'appellera Index.cshtml.



# Afficher les vues (suite)

- L'action du contrôleur retourne la vue.

```
public ActionResult Index()  
{  
    return View("Index");  
}
```

- **NB** : Retourner une vue impose les règles suivantes :
  - Par défaut une vue Index retourne une vue Index sinon il faut le préciser dans le « return ».
  - Respect de la hiérarchie des vues dans le répertoire Views, le nom du contrôleur DOIT matcher.
  - Vérifier ses routes.

# Les vues « fortement » typées

- Il existe une surcharge de la méthode **View()** qui permet de passer le modèle directement à la vue.
- RAPPEL : La vue met à disposition de l'utilisateur des données du modèle.
- 1) Définir le modèle de la vue à l'aide du **@model** au début de la vue (voir exemple diapo suivante).
- 2) Pensez à ajouter le modèle dans votre contrôleur et notamment dans le « return ».

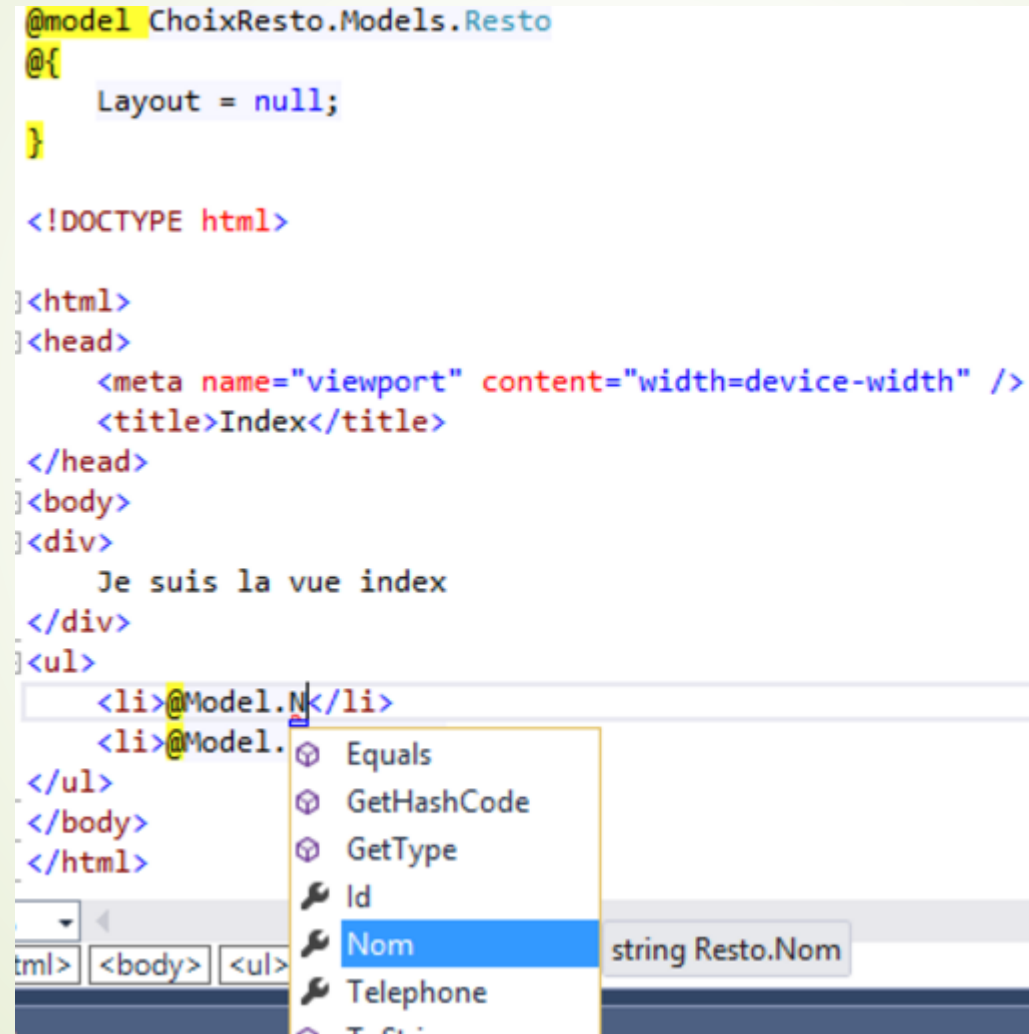


# Les vues « fortement » typées

```
@model ChoixResto.Models.Resto
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
<div>
    Je suis la vue index
</div>
<ul>
    <li>@Model.N</li>
    <li>@Model.N</li>
</ul>
</body>
</html>
```



# Les moteurs de vues

- Il existe deux moteurs de vues en ASP.NET MVC pour développer nos vues HTML :
  - Le moteur de vue ASPX (utilisé principalement en WebForm).
  - Le moteur de vue Razor.

# Le moteur de vue ASPX

- Verbeux.
- Lisibilité réduite, maintenance compliqué.
- Extension des vues : aspx.
- Se base sur des <% et des %> (comme les balises <? En PHP).
- De moins en moins utilisé car c'est plus pour les utilisateurs de WebForm.
- **NB** : à partir de VisualStudio 2013, il semblerait même qu'il n'es tplus possible de créer une vue ASPX.

# Le moteur de vue Razor

- Intelligent, syntaxe légère.
- Lisibilité optimisée.
- Tout est fait pour que le développeur ait l'impression d'écrire dans un seul et unique langage.
- Extension des vues : cshtml.
- Se base sur des @.
- S'utilise en ASP.NET MVC (vues typées => puissance).

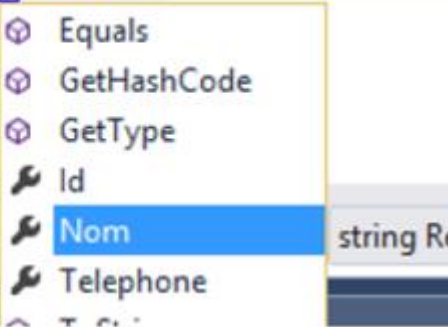
# La syntaxe Razor

- Prends tout son sens lorsque l'on utilise des vues fortement typées.
- Permet d'afficher les propriétés du modèle.
- Permet l'utilisation d'instruction C#.

```
@model ChoixResto.Models.Resto  
@{  
    Layout = null;  
}
```

On peut mettre du code ici!

```
<ul>  
    <li>@Model.N</li>  
    <li>@Model.  
</ul>  
</body>  
</html>
```



Facile pour afficher du texte par exemple

# Mixer HTML / C# : c'est possible!

- Exemple d'utilisation d'un **foreach** dans une vue CSHTML :
  - Le modèle de la vue : **@model List<Resto>** (OUI on peut mettre une collection comme objet typé à votre vue!).
  - Utilisation d'un **foreach** classique.
  - N'oubliez pas l'**include**.

```
1 <table>
2   <tr>
3     <th>Nom</th>
4     <th>Téléphone</th>
5   </tr>
6   @foreach (var resto in Model)
7   {
8     <tr>
9       <td>@resto.Nom</td>
10      <td>@resto.Telephone</td>
11    </tr>
12  }
13 </table>
```

# Autres instructions Razor

- Ne se limite pas au **foreach** :
  - Possibilité de faire des **if / else** pour un affichage conditionnel avec un test sur une propriété du modèle par exemple.
  - Possibilité d'instancier des **variables** pour les réutiliser.
  - Gestion de la sécurité.
  - Etc.

## Exemple IF / ELSE :

```
@if (Model.IsEnabled) {  
    <p>Le produit est activé !</p>  
}  
else {  
    <p>Le produit n'est pas actif.</p>  
}
```

## Exemple instruction entre @{ }:

```
@{  
    /* Bloc de code sur plusieurs lignes */  
    string uneVariable = "valeur initiale";  
    uneVariable += "un autre morceau";  
}  
  
<p>Une variable vaut : @uneVariable.</p>
```



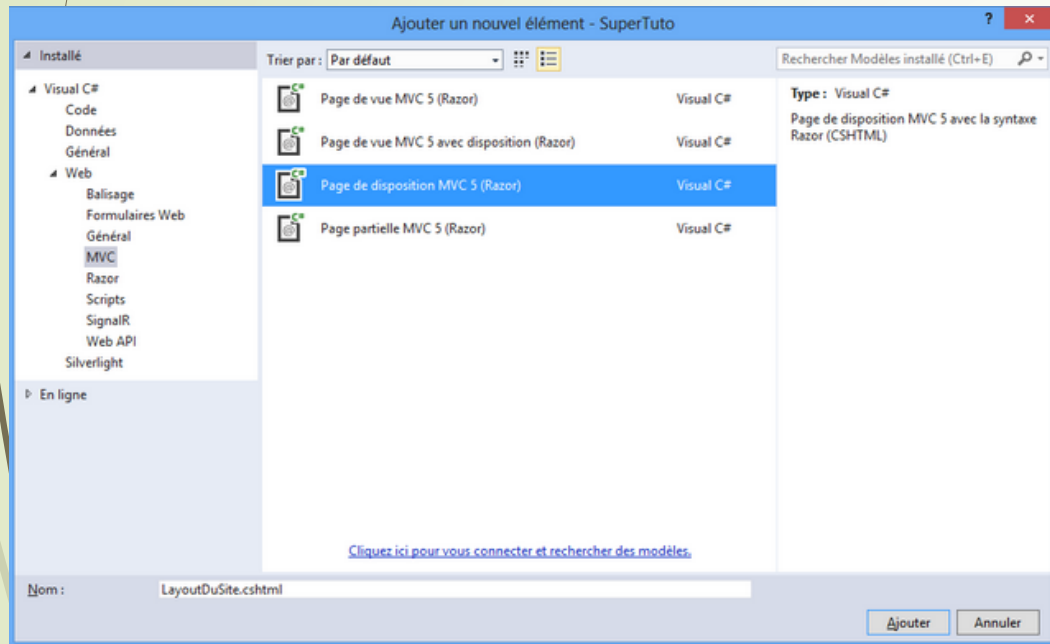
# Quelques principes sur les vues

- Malgré l'utilisation d'un moteur de vue, une vue ça reste du HTML, du CSS, du JavaScript, etc.
- L'utilisation de Razor est plus que **préconisée** (voire obligatoire) maintenant.
- Mixer du HTML et du C# peut faciliter l'affichage mais attention à ne pas placer des contrôleurs logiques (métiers) dans vos vues!
- **RAPPEL** : Une vue c'est une présentation des données du modèle en lecture ou en écriture.

# Les Layouts

- Problématique : est-ce que nous sommes suffisamment fous pour répéter à la main tous ces éléments à travers toutes nos vues ?
- Solution : **Les Layouts**.
- Se place dans le répertoire **Shared** du répertoire Views.
- Utile pour les éléments « statiques » d'une page Web :
  - Header
  - Footer / Copyright
  - Menu
  - Etc.

# Les Layouts (suite)



- Un Layout est une page spéciale.
- Notion de Section avec le **RenderBody()**.
- On peut changer le titre en dynamique avec la syntaxe Razor.
- On l'utilise dans n'importe quelle vue de la façon suivante:

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
</head>
<body>
  <div>
    @RenderBody()
  </div>
</body>
</html>
```

```
@{
    Layout = "~/Views/Shared/LayoutDuSite.cshtml";
    ViewBag.Title = "Accueil";
}
```

# Les helpers HTML

- Définition simple : Permet d'écrire moins de code tout en gardant un contrôle fort sur le HTML (généré).
- **Génère** du code HTML à partir d'une instruction Razor.
- Permet de substituer tous les contrôles standards du Web :
  - TextBox
  - RadioButton, CheckBox
  - Combo
  - Formulaire
  - Label
  - Etc.

# Les helpers HTML - Exemple

- En haut l'utilisation d'un **Helper**.
- En bas le code HTML généré par le **Helper**.

```
@Html.TextBox("tbxNom", "ADRIEN")
```

```
<input id="tbxNom" name="tbxNom" value="ADRIEN" type="text"></input>
```

- Gardez le contrôle du HTML, ajouter vos élément de génération avec les **HtmlAttributes** :

```
@Html.TextBox("tbxNom", "ADRIEN", new { @class = "maClasseCss", style = "color: red" })
```

```
<input id="tbxNom" class="maClasseCss" name="tbxNom" style="color: red" value="ADRIEN" type="text"></input>
```

# Les Helpers - Complément

- Il est possible de créer ses propres Helpers pour créer ses propres contrôles réutilisables (exemple : Framework).
- Certaines librairies offrent une large gamme de Helpers (Devexpress, Telerik, Kendo, etc.).
- Consulter la doc pour retrouver tous les Helpers disponibles... :
  - <https://msdn.microsoft.com/en-us/library/system.web.mvc.htmlhelper%28v=vs.118%29.aspx>

# Les helpers fortement typés

- Fonctionne sur le même principe que les Helpers.
- S'utilise sur des vues fortement typées à l'aide de la propriété **@model** comme vu précédemment.
  - On suffixe le HtmlHelper par un « For ».
  - On utilise une **expression Lambda** pour binder la propriété du modèle au Helper.

## Utilisation d'un Helper fortement typé :

```
@model WebApplication1.Models.MonBeauModele
@{
    ViewBag.Title = "Contact";
}

<div>
    @Html.TextBoxFor(m => m.Nom)
</div>
```



# Les Helpers HTML – Récapitulatif

Helper	Description
<code>Html.ActionLink</code>	Génère un lien vers une méthode (action) de contrôleur avec la balise <code>&lt;a href&gt;</code>
<code>Html.BeginForm</code>	Génère une balise de formulaire <code>&lt;form&gt;</code>
<code>Html.CheckBox</code>	Génère une case à cocher <code>&lt;input&gt;</code> de type "checkbox "
<code>Html.DropDownList</code>	Génère une liste déroulante de type <code>&lt;select&gt;</code> , <code>&lt;option&gt;</code>
<code>Html.Hidden</code>	Génère un champ caché <code>&lt;input&gt;</code> de type "hidden "
<code>Html.ListBox</code>	Génère une liste déroulante multi-sélectionnable de type <code>&lt;select&gt;</code> , <code>&lt;option&gt;</code>
<code>Html.Password</code>	Génère une zone de texte permettant de saisir un mot de passe <code>&lt;input&gt;</code> de type "password "
<code>Html.RadioButton</code>	Génère un bouton radio avec une balise <code>&lt;input&gt;</code> de type "radio "
<code>Html.RouteLink</code>	Fait la même chose que <code>Html.ActionLink</code> à part qu'il utilise une route en entrée
<code>Html.TextArea</code>	Génère une zone de texte modifiable sur plusieurs lignes avec la balise <code>&lt;textarea&gt;</code>
<code>Html.TextBox</code>	Génère une zone de texte modifiable sur une seule ligne avec la balise <code>&lt;input&gt;</code> de type "text "

Helper	Description
<code>Url.Action</code>	Génère une URL en utilisant le mécanisme de routing
<code>Url.Content</code>	Génère une URL absolue à partir d'une URL relative
<code>Url.RouteUrl</code>	Fait la même chose que <code>Url.Action</code> à part qu'il utilise une route en entrée

- Privilégier l'utilisation des Helpers et des Helpers fortement typées!
- Tous les Helpers ne peuvent pas être utilisés dans leurs versions « typées ».

# Les vues partielles

- Permet la **réutilisation** d'un « morceau de vue » à plusieurs endroits.
- Similaire aux UserControls vus en WPF.
- S'inclue à l'aide de l'instruction **@Html.Partial**(« »).
- Peut être fortement typée.
- Le contrôleur peut retourner des vues partielles (exemple : rechargement en AJAX d'un morceau de vue).

Ajouter une vue

Nom de la vue :  
Connexion

Modèle :  
Empty (sans modèle)

Classe de modèle :

Classe de contexte de données :

Options de vue :  
☒ Créer en tant que vue partielle  
☐ Bibliothèques de scripts de référence  
☐ Utiliser une page de disposition :

# Créer et utiliser un contrôleur

- Tous les contrôleurs héritent de la classe **Controller** du Framework .NET.
- Comporte des actions.
- Retourne des vues ou des vues partielles.

```
public ActionResult Index()  
{  
    return View();  
}
```

```
[ActionName("Index")]  
public ActionResult LeNomQueJeVeux()  
{  
    return View();  
}
```

# Contrôleur – Accéder au modèle

- Comment accéder au modèle et à la base ?
  - 1) On récupère un « ordre » de la part de l'utilisateur via une méthode du contrôleur.
  - 2) On utilise la BusinessLayer qui s'occupe de gérer EF et la base de données.
  - 3) On récupère le résultat de la part de la BusinessLayer dans un try / catch.
  - 4) On affiche le résultat de l'action à l'utilisateur.
  
- **NB** : Sans BusinessLayer, vous utilisez un contexte EF dans la partie applicative et vous créez une dépendance inutile!

# Le ModelBinding

- Principe de base d'ASP.NET MVC au niveau du contrôleur appelé « **Binding de modèle** ».
- S'utilise à partir d'une vue fortement typée.
- Grace aux différents champs qui sont passés à la requête, le contrôleur peut reconstruire notre modèle.
- ASP.NET MVC reconnaît bien que les propriétés ont les mêmes noms que les champs de formulaire. Il fait alors une liaison de données entre les deux, permettant ainsi de créer un objet.

# Le ModelBinding - Exemple

```
@using (Html.BeginForm()) {  
    @Html.ValidationSummary(true, "Échec de la modification du mot de passe")  
    <div>  
        <fieldset>  
            <legend>Informations de compte</legend>  
  
            <div class="editor-label">  
                @Html.LabelFor(m => m.OldPassword)  
            </div>  
            <div class="editor-field">  
                @Html.PasswordFor(m => m.OldPassword)  
                @Html.ValidationMessageFor(m => m.OldPassword)  
            </div>  
  
            <div class="editor-label">  
                @Html.LabelFor(m => m.NewPassword)  
            </div>  
            <div class="editor-field">  
                @Html.PasswordFor(m => m.NewPassword)  
                @Html.ValidationMessageFor(m => m.NewPassword)  
            </div>  
  
            <div class="editor-label">  
                @Html.LabelFor(m => m.ConfirmPassword)  
            </div>  
            <div class="editor-field">  
                @Html.PasswordFor(m => m.ConfirmPassword)  
                @Html.ValidationMessageFor(m => m.ConfirmPassword)  
            </div>  
  
            <p>  
                <input type="submit" value="Modifier le mot de passe" />  
            </p>  
        </fieldset>  
    </div>  
}
```



```
[HttpPost]  
public ActionResult ChangePassword(ChangePasswordModel model)  
{  
    if (ModelState.IsValid)  
    {  
        // Si nous sommes arrivés là, quelque chose a échoué, réafficher le formulaire  
        return View(model);  
    }  
}
```



# Principe de la validation

- Concept MVC découpés en 3 prédicats :
  - Utilisation des **DataAnnotation** au niveau des entités du modèle (ou des ViewModel).
    - Required, StringLength, RegularExpression, etc. (Attention à ne pas confondre avec les DataAnnotations d'EntityFramework...)
  - **Côté serveur**
    - Dans le ou les contrôleurs.
  - **Côté client**
    - Dans la ou les vues en JQuery par exemple.
- NB : La validation est capitale avant de lancer une opération de type Create / Update / Delete.



# Validation du modèle (côté serveur)

- Validation indispensable au moment du POST
- Utilisation l'objet **ModelState**.
- Récupérer un dictionnaire d'erreurs pour informer l'utilisateur en retour.

## Exemple d'utilisation de ModelState :

```
if ((vm == null) || (!ModelState.IsValid))  
{  
    return Json(new { success = false, errors = ModelState.Keys.SelectMany(key => ModelState[key].Errors).Select(mse => mse.ErrorMessage) });  
}
```

# Validation côté client

- La validation côté client **ne remplace pas** la validation côté serveur.
- Plus « User-Friendly », meilleure réactivité.
- Utilisation de JQuery.Validate.Unobtrusive.
- Utilisation du **@Html.ValidationMessageFor()**.

**Modifier un restaurant**

Nom

Téléphone  × Le numéro de téléphone est incorrect

# Les Bundles

- Un Bundle est un **regroupement** de CSS ou JS sous une même clé.
- Optimise le chargement des pages (moins de requêtes).
- Facilite l'inclusion dans la balise HEAD.
- Simplifie la gestion du cache côté navigateur (validité du cache).
- Possibilité de « **minifier** » le contenu bundle (optimisation du poids des fichiers qui transitent).

# Les Bundles - Utilisation

## Déclaration du Bundle:

```
bundles.Add(new ScriptBundle("~/bundles/jquery").Include(  
    "~/Scripts/jquery-{version}.js",  
    "~/Scripts/jquery-{version}.min.js"  
));
```

## Utilisation du Bundle:

```
@Scripts.Render("~/bundles/jquery")
```

- La déclaration du Bundle se fait dans le fichier **BundleConfig** (répertoire **App\_Start**).
- L'utilisation du Bundle se fait dans n'importe quelle vue entre des balises de type **<head></head>**.
- Certains bundles sont automatiquement ajoutés à la création d'un projet MVC.

# Les Areas (Zones)

- Cloisonnement de vues / routes dans un sous « dossier »
  - Gestion d'une partie front dans un dossier « Front »
  - Gestion d'une partie back (administration) dans un dossier « Back »
- AreaRegistration : indique que l'on n'a plusieurs zones dans le fichier Global.asax.
- Configuration spécifique par zone (using utilisés dans les vues, etc.).
- Gestion de routes spécifiques à chaque zone.
- ViewStart pour définir le Layout à utiliser.
- Possibilité de créer un Layout par zone.

# Un peu d'AJAX dans tout ça

- Ajax permet d'effectuer des requêtes asynchrones pour récupérer des données, mettre à jour des portions de pages, soumettre des formulaires, etc.
- Ajax fonctionne particulièrement bien avec le JSON.
- Ajax s'utilise avec jQuery.



Microsoft jQuery Unobtrusive Ajax  
jQuery plugin that unobtrusively sets up  
jQuery Ajax.

Installer

```
<script type="text/javascript" src="~/Scripts/jquery.unobtrusive-ajax.js"></script>
```

# Exemple de requête AJAX

## Exemple de requête AJAX / JQuery:

```
function ChargeVuePartielle() {  
    $.ajax({  
        url: '@Url.Action("AfficheTableau", new {id = ViewBag.Id })',  
        type: 'GET',  
        dataType: 'html',  
        success: function (result) {  
            $('#tableauResultat').html(result);  
        }  
    });  
}
```



# AJAX : Pour aller plus loin...

- On peut tout faire avec AJAX, il faut tirer parti de toute cette puissance pour avoir une application contemporaine :
  - Formulaire asynchrone avec `@Ajax.BeginForm`.
  - Rafraichir une source de données (exemple : liste déroulante).
  - Gérer les erreurs.
  - Rafraichissement périodiques.

# Templates MVC (1/2)

- Un Template MVC est un « modèle d'affichage » :
  - Définir la mise en page pour afficher un type ou une classe.
  - Définir la mise en page pour modifier un type ou une classe.
- Réutilisation de ces modèles dans différents écrans de l'application (**Don't Repeat Yourself**).
- Arborescence à respecter (répertoire **Shared**).

# Templates MVC (2/2)

- On distingue deux types de templates

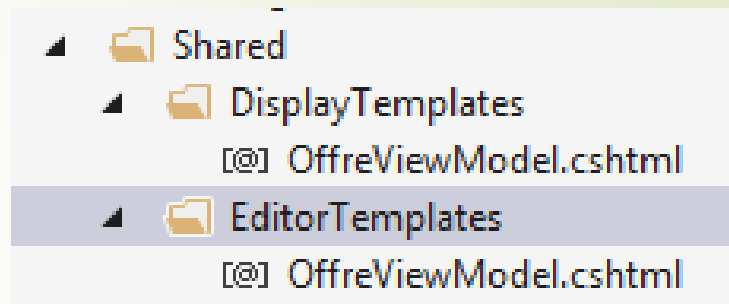
- **DisplayTemplates**

- `Html.DisplayFor()`
    - `Html.DisplayForModel()`

- **EditorTemplates**

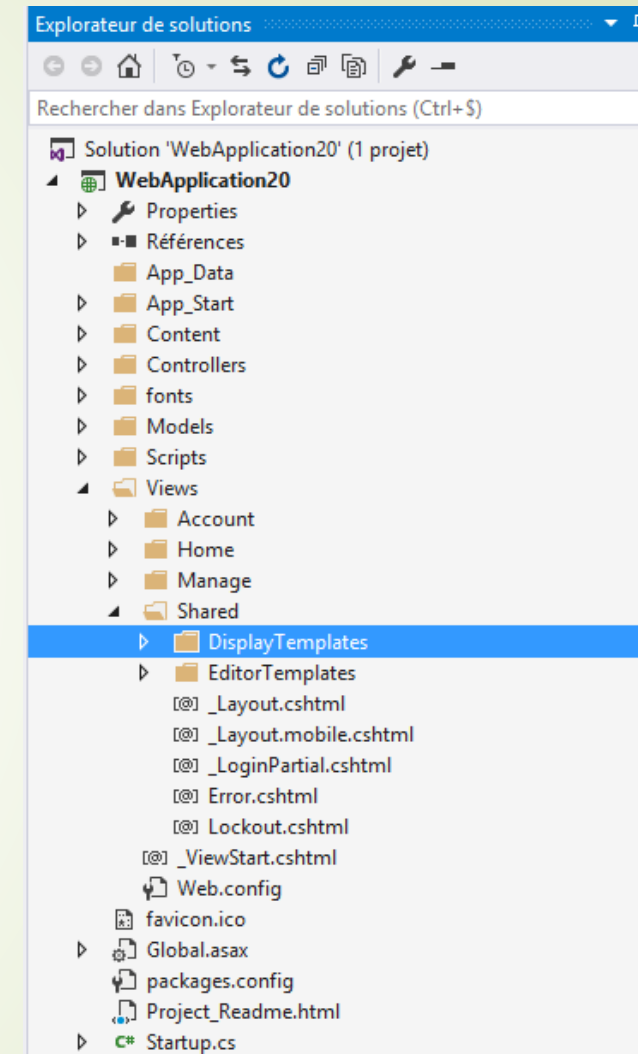
- `Html.EditorFor()`
    - `Html.EditorForModel()`

- Idéalement, le nom de la vue => Le nom de l'élément que l'on souhaite afficher / modifier.



# Vues « mobiles »

- Permet de personnaliser l'affichage sur des terminaux mobiles :
  - Terminaux mobiles (smartphones).
  - Tablettes.
  - Différents types de téléphone (exemple : différencier un affichage sous Android et sous Windows Phone).
- Se base sur le UserAgent.
- Switch de manière automatique
  - Reste configurable à l'aide du DisplayModeProvider.
  - Attention à rester responsive, c'est plus simple que de dupliquer les vues pour de l'affichage mobile.



# Vues « mobiles » = applications mobiles?

- Utilisation de Apache Cordova :
  - Embarquer un site ASP.NET dans une application mobile
  - Bénéficier des layouts mobiles.
- Quand c'est bien fait, on ne voit pas que ce n'est pas du natif !



APACHE  
**CORDOVA**<sup>TM</sup>

## TD / Exercices

- Création d'un projet de base, valider l'architecture de ce dernier, lancer l'application Web.
- Créations de vues, tests, helpers, etc.
- Créations de contrôleurs.
- Validation côté serveur.
- Etc.

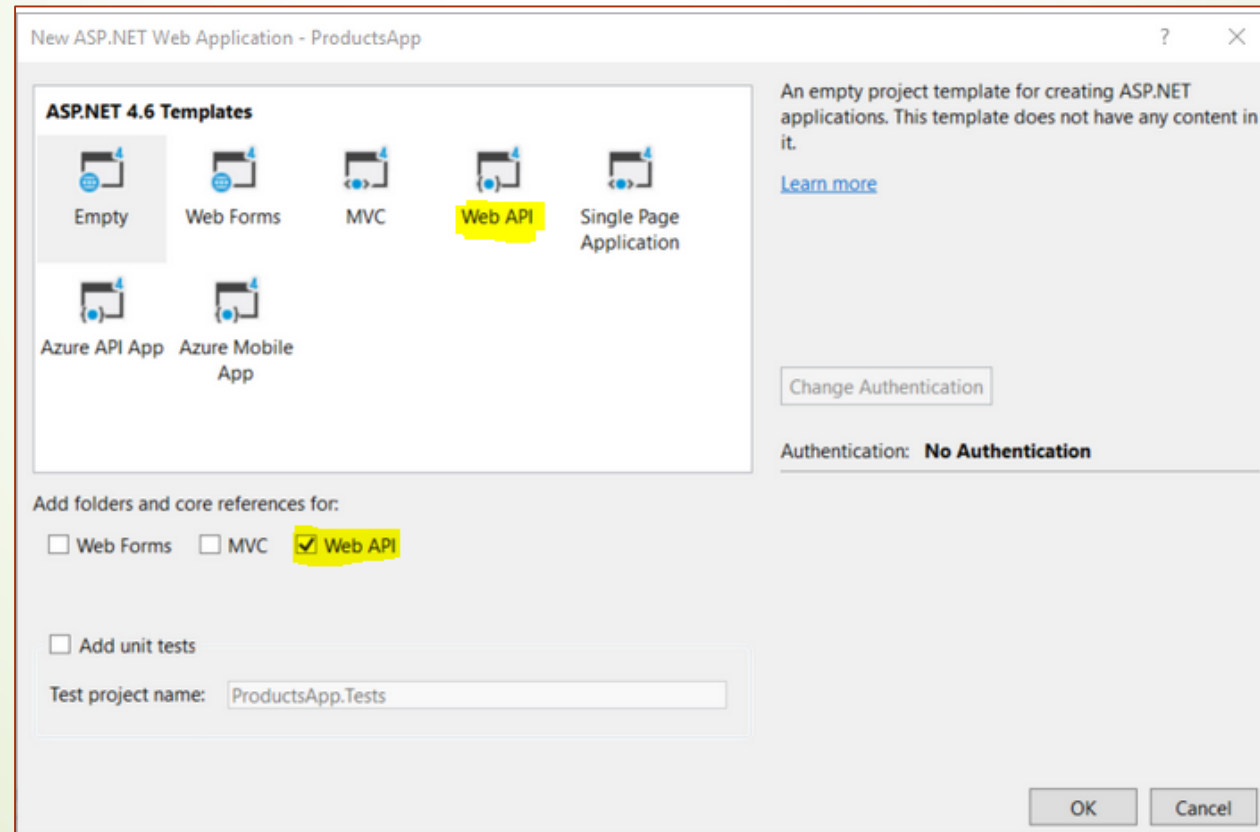
# ASP.NET MVC : C'est pas fini...

- Ceci n'est qu'une approche de ASP.NET MVC, il existe encore de nombreuses notions permettant de réaliser des applications encore plus complexes et toujours plus performantes :
  - Sécurité, authentification, droits d'accès, etc.
  - Déploiement sur IIS, pool d'application, etc.
  - Filtres, custom ModelBinders, ActionResult, configuration.
  - Utilisation des contrôleurs en asynchrones.
  - IOC, Unity.
  - RazorGenerator.
  - Etc.



# ASP.NET WEB API 2

- Nouveau type de projet : Permet d'initialiser la structure de base d'une Web API
- Différent d'un projet WCF



# Créer un contrôleur

- Il faut créer un contrôleur de type Web API 2
- Il doit hériter du type de base : **ApiController**
- Il propose plusieurs actions
  - On respecte le REST (PUT, DELETE, POST...)
- Chaque action peut prendre N paramètres

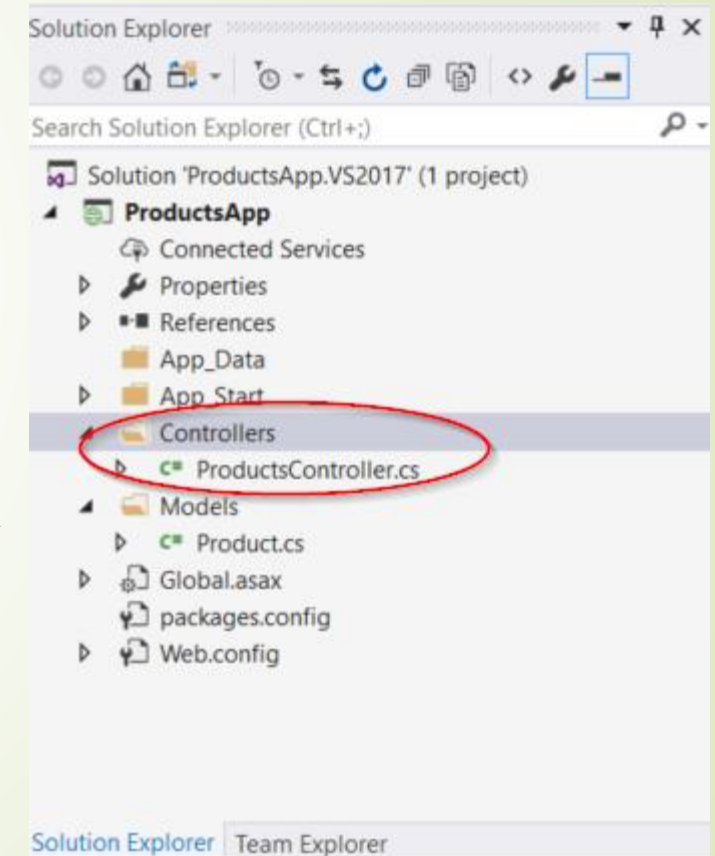
```
public class ProductsController : ApiController
{
    Product[] products = new Product[]
    {
        new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1.99 },
        new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.99 },
        new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 9.99 }
    };

    public IEnumerable<Product> GetAllProducts()
    {
        return products;
    }

    public IHttpActionResult GetProduct(int id)
    {
        var product = products.FirstOrDefault((p) => p.Id == id);
        if (product == null)
        {
            return NotFound();
        }
        return Ok(product);
    }
}
```

# Architecture type - Intégration

- Architecture type d'un projet Web API 2
  - On retrouve le App\_Start et la notion de route
  - Un contrôleur par Web API
  - Eventuellement des modèles pour ne pas exposer des classes du modèle métier
- Directement connecté à notre architecture SOA
  - Se connecter à la base SQL
  - Exploiter notre métier



# Au final ?

- La structure du projet et du cours ressemblerait à ça :

