

# ASP.NET MVC

# ASP.NET MVC

1. [Introduction](#)
2. [Structure d'une application MVC](#)
3. [Conventions ASP.NET MVC](#)
4. [Route et Navigation](#)
5. [Styles et Layout](#)
6. [Bundling et Minification](#)
7. [Les Contrôleurs ASP.NET MVC](#)
8. [Les Vues ASP.NET MVC](#)
9. [Les Modèles ASP.NET MVC](#)
10. [Razor](#)
11. [Formulaires et Helpers](#)
12. [Custom Helpers](#)
13. [Data Annotation et Validation](#)
14. [Custom Validation Logic](#)
15. [Display and Edit Annotation](#)

# ASP.NET MVC










- 14. [Sessions et Application](#)
- 15. [Areas](#)
- 17. [Construire des application responsive ASP.NET MVC](#)
- 18. [Sécurité](#)
- 14. [Testing](#)
- 18. [Déploiement](#)

# Introduction

ASP.NET MVC

# Introduction

Microsoft fournit les technologies nécessaires pour vous permettre de créer des applications riches web et les publier sur internet ou sur intranet

Developpement	Hébergement	Langage	
		Server	Client
 	  		  

# Petit tour de ASP.NET

Asp.net vous permet de développer des applications web tel que des portails, des blogs, des wikis, ...

## 1. Le model de programmation ASP.NET

### - But des webforms

- ✓ Cacher HTTP & HTML en modélisant des interfaces via des contrôles serveurs.
- ✓ Chaque contrôle sauvegarde son propre état grâce au *ViewState* (sérialisation des données via http Request)
- ✓ Chaque contrôle génère son propre HTML permettant l'affichage web de celui-ci et connecte les événements client avec le code serveur correspondant.



# Petit tour de ASP.NET

## 2. Les faiblesses des webforms

- Poids du ViewState

Le mécanisme du *ViewState* qui maintient l'état des contrôles entre les dialogues Client/Server entraine un transfert de larges blocs de données.

➔ Augmente le temps de réponse du server

➔ Augmente le besoin de bande passante

- Illusion de la séparation des couches

Le model *code-behind* permet de séparer la couche HTML du code applicatif.

Cependant, dans la plupart des cas, de part cette pseudo séparation, le développeur est tenté de mixer le côté *présentation* et la *logique applicative*.

- Limitation du contrôle du code HTML généré

L'HTML généré par nos contrôles serveur n'est pas nécessairement l'HTML que l'on souhaite et mettre en page via le css peut s'avérer ardu. De plus, le rendu sur certains navigateur échoue parfois de part le non respect des standards.

- Testing

Il est quasiment impossible de tester le design séparément du code applicatif et mettre en place des tests automatiques est un challenge.

# ASP.NET MVC

- **Pattern MVC**

Le pattern MVC n'est pas neuf (1978) mais il a gagné en popularité dans le monde web pour les raisons suivantes :

- **Model**

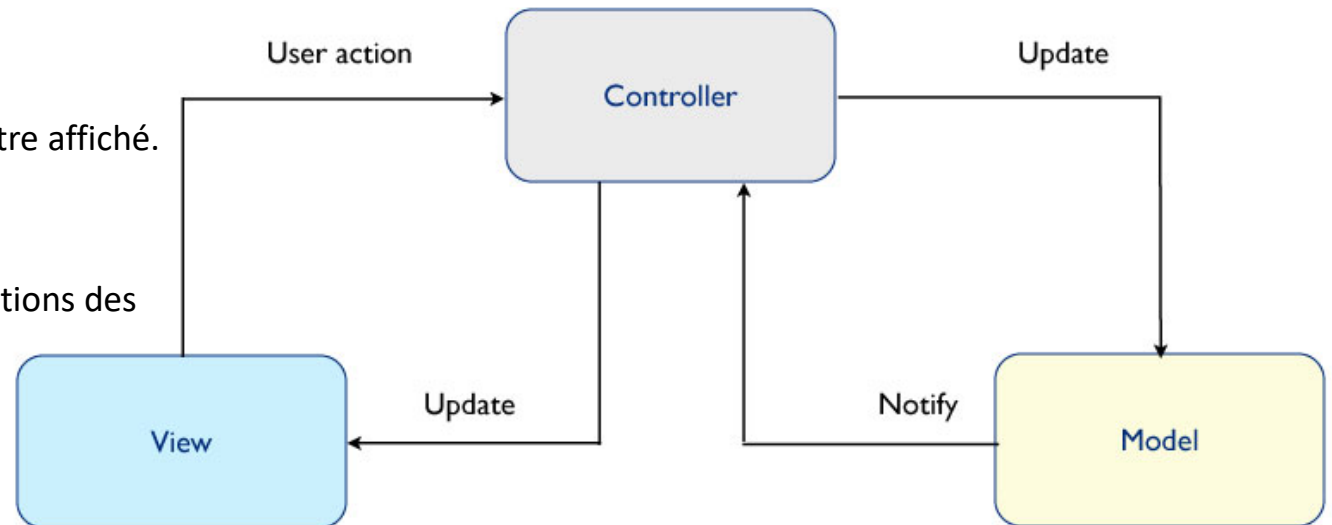
Classes qui décrivent les données sur lesquels nous travaillons. Les règles business qui définissent la façon dont les données doivent être manipulées, modifiées y sont également Implémentées.

- **View**

Définit comment l'interface utilisateur doit être affiché.

- **Controller**

Classes qui interagissent avec les communications des utilisateurs



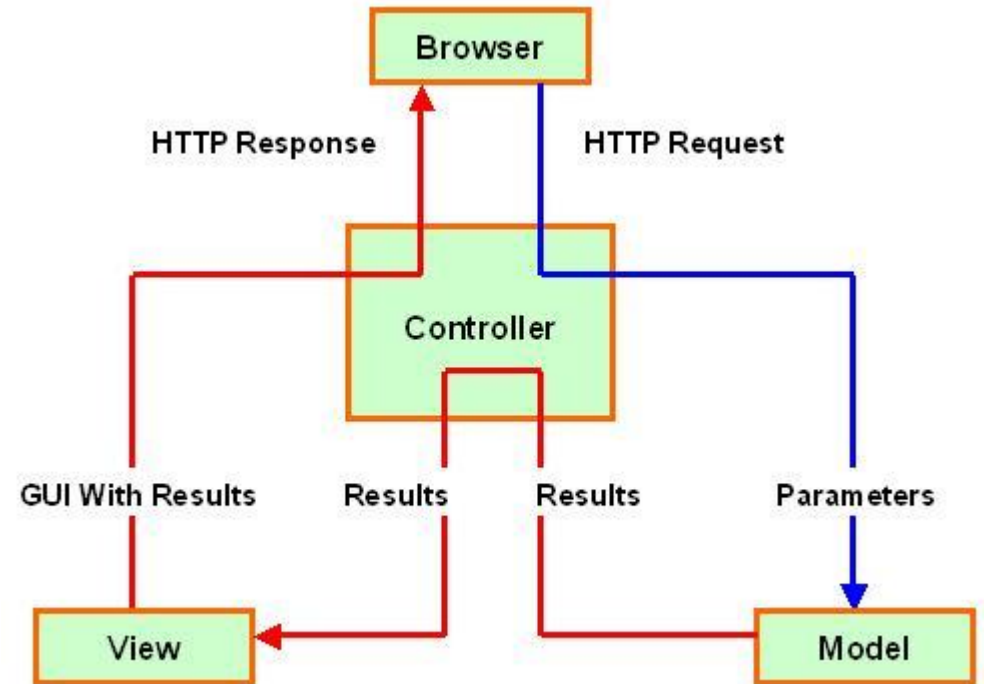


# ASP.NET MVC

## Le MVC appliqué aux frameworks Web.

ASP.NET MVC Framework implémente une version MVC spécialement repensée pour le monde web.

1. Les interactions utilisateur avec une application MVC un cycle naturel  
L'utilisateur effectue une action et l'application répond en présentant une vue mise à jour à celui-ci. Très intéressant pour une application web se basant sur des *request* et *responses*.
2. Les applications web requièrent des combinaisons de plusieurs technologies (DB, HTML, Services,...) généralement décomposées en différentes couches



# ASP.NET MVC

ASP.NET MVC contextualise le pattern mvc et traduit grossièrement les différentes couches comme suit :

- **Model**

Classes qui décrivent le domaine de notre application. Ces objets de domaine encapsulent souvent les données stockées en base de données ainsi que le code destiné à les manipuler et les règles business.

En ASP.NET, il fréquent que le modèle soit composé d'une couche de data (DAL) implémentée via EntityFramework ou Nhibernate et de code « custom » représentant les règles business.

- **View**

Il s'agit d'un template permettant la génération automatique d'HTML.

- **Controller**

Il s'agit d'une classe spéciale qui gère les relations entre les vues et le Modèle. Il répond aux entrées utilisateur et , au besoin, décide quel vue doit être générée.

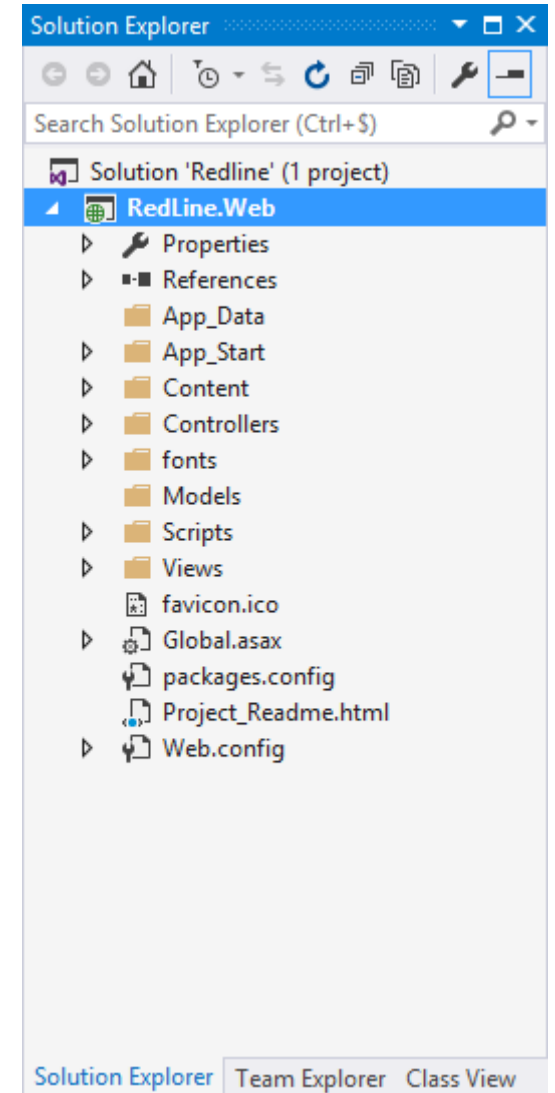
La convention veut que le nom de cette classe se termine par *Controller*.

# Structure d'une application MVC

ASP.NET MVC

# Structure d'une application MVC

Dossier	But
/Controllers	Dossier où se trouve les contrôleurs qui capturent les requêtes URL
/Models	Les classes qui représentent et manipulent les classes et les objets business
/Views	Les templates UI responsable de la génération des sorties, html
/Scripts	Scripts .js et librairies javascript
/fonts	Les polices (Fonts) de bootstrap
/Content	Pour déposer les css, images et autres contenus (pas les scripts)
/App_Data	Fichiers de données (Read/Write)
/App_Start	Fichiers de configuration pour le routing, bundling et Web API



# Conventions ASP.NET MVC

ASP.NET MVC

# Conventions ASP.NET MVC

MVC est conçu autour de valeurs par défaut sur la base de conventions qui peuvent être redéfinies au besoin.  
Ce concept est communément appelé «convention over configuration »

*“We know, by now, how to build a web application. Let’s roll that experience into the framework so we don’t have to configure absolutely everything again.”*

*Ruby On Rails*

*« Nous savons , désormais, comment construire une application web .  
Partageons cette expérience de sorte que nous ne devons pas configurer à nouveau absolument tout »*

# Conventions ASP.NET MVC

- **Conventions over Configuration**

Ce concept est illustré par la présence des trois dossiers qui font que l'application peut fonctionner :

- Controllers
- Models
- Views

Ces dossiers ne sont pas présent dans notre *Web.config* (fichier de configuration de notre web app) car ils doivent juste être là par *convention* afin que le moteur ASP.MVC puisse retrouver ces éléments.

En d'autre termes, nous ne devons pas spécifier au moteur MVC que nos vues se trouvent dans le dossier *Views* puisque par *convention*, il le sait déjà.

# Conventions ASP.NET MVC

Les conventions ASP.NET MVC sont plutôt simple :

- Chaque nom de classe de type Contrôleur, se termine par *Controller* (*HomeController, AccountController, ProductController,...*)
- Il n'a qu'un dossier *Views* pour toutes les vues de l'application
- Les vues utilisées par les contrôleurs doivent se trouver dans un sous-dossier de *Views* qui portera le nom du Contrôleur sans le suffixe. (*ProductController* aura ses vues dans */Views/Product*)
- Les interfaces utilisateurs réutilisables (*partial view,...*) se retrouvent dans le dossier *Views/Shared*



# Route et Navigation

ASP.NET MVC

# Route et Navigation

## 1. Uniform Resource Locator

Il est important pour une application web de fournir des URLs respectant les règles suivantes :

- Un nom de domaine facile à retenir et à épeler
- Elle doit être courte
- Elle doit représenter la structure du site
- Elle doit être immuable

Traditionnellement, l'url représente un fichier physique.

Exemple :

<http://example.com/albums/list.aspx> permet de déduire qu'un fichier list.aspx existe dans le dossier albums.

Il existe donc un lien directe entre l'url et les fichiers existants sur le disque.

Ce n'est pas le cas en MVC, où l'url est mappé à une méthode à exécuter plutôt qu'à fichier physique.

# Route et Navigation

## 2. Routing ou Url rewriting?

L'url rewriting consiste à mapper une url sur une autre.

C'est une technique généralement utilisée pour mapper des urls peu explicite vers des urls plus précises.

Exemple :

<http://www.shop.com/Fiche.aspx?id=45&cat=22> → <http://www.shop.com/Sport/Velo/Fiche>

Le Routing consiste quand à lui à mapper une url vers une ressource, pas nécessairement une page web.

Il définit comment la requête est « dispatché » basé sur les caractéristiques de l'URL – il n'y a donc pas de réécriture.

# Route et Navigation

## 3. Les différentes façon de mettre en place le routing

Les informations principales concernant le routing se trouve dans le dossier *App\_Start*, dans le fichier *RouteConfig.cs*.

Nous n'avons pas à écrire toutes les combinaisons possibles pour les urls.

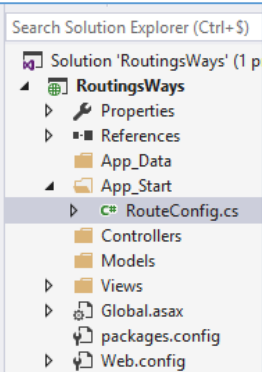
En effet, chaque route contient un Url pattern qui est comparé à l'url demandée.

Si le pattern correspond, il est utilisé pour résoudre l'url.

```
using System.Web.Routing;

namespace RoutingsWays
{
    public class RouteConfig
    {
        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 }
            );
        }
    }
}
```



# Route et Navigation

Request	Segments
http://mysite.com/Admin/Index	controller = Admin action = Index
http://mysite.com/Index/Admin	controller = Index action = Admin
http://mysite.com/Apples/Oranges	controller = Apples action = Oranges
http://mysite.com/Admin	Erreur - trop peu de segment
http://mysite.com/Admin/Index/Soccer	Erreur - trop de segment

# Route et Navigation

## Par défaut :

Mvc n'utilise le pattern que pour les urls qui correspondent exactement. Si il y a le bon nombre de segment, il utilise de manière littérale ceux-ci pour trouver le controller et l'action

## **4. Définir des valeurs par défaut**

La valeur par défaut est utilisé lorsque l'url ne match avec aucun pattern

```
routes.MapRoute(  
    "Default", // Nom d'itinéraire  
    "{controller}/{action}/{id}", // URL avec des paramètres  
    new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Paramètres par défaut  
);
```

# Route et Navigation

## Segment static

Tous les segments ne sont pas nécessairement variables.

Si nous désirons supporter l'url suivante

<http://mydomain.com/Public/Home/Index>

```
routes.MapRoute("", "Public/{controller}/{action}", new { controller = "Home", action = "Index" });
```

## Remarque :

Les routes sont appliquées dans l'ordre de leurs déclarations dans le fichier Global.asax

# Route et Navigation

## 5. L'attribut Route

Afin de définir une route, il est possible d'utiliser l'attribut route.

Celui-ci prend en argument un string appelé *Route Template*.

Il définit le mapping nécessaire pour atteindre l'action.

Si nous avons besoin de définir plusieurs route, il suffit de répéter l'attribut.

```
public class HomeController : Controller
{
    // GET: Home
    [Route("")]
    [Route("home")]
    [Route("home/index")]
    public ActionResult Index()
    {
        return View();
    }
}
```



# Route et Navigation

## 6. Contraintes personnelles

Il est également possible d'ajouter des contraintes dans les paramètres de la route.

Par exemple, le paramètre *Id* définit dans le routeconfig peut revêtir l'aspect d'un int, un double, un string,....

Nous pouvons définir une contrainte afin de forcer, par exemple, le type.

Deux choix s'offre à nous :

- Ajouter la contrainte dans le RouteConfig
- Implémenter IRouteConstraint

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new
    {
        controller = "Home",
        action = "Index",
        id = UrlParameter.Optional
    },
    //Id uniquement numérique
    constraints: new { id = @"\d+" }
);
//Pour les controlleur commençant par H,
//Seul les action INdex et COntract sont valides
constraints: new { controller="^H.*",
                    action = "^Index$|^Contact$" }
```

# Route et Navigation

```
public class IntRouteConstraint : IRouteConstraint
{
    public bool Match(HttpContextBase httpContext,
        Route route,
        string parameterName,
        RouteValueDictionary values,
        RouteDirection routeDirection)
    {
        int test = 0;
        return int.TryParse(values[parameterName].ToString(), out test);
    }
}
```

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

RouteConfig

ception Fenêtre Commande Fenêtre Exécution Sortie Automatique Variables locales Espion 1

# Styles et Layout

ASP.NET MVC

# Styles et Layout

Razor apporte, avec les layouts, une solution afin de centraliser l'aspect de notre web application.

Le principe est de regrouper, dans une page spécifique, les éléments communs de mise en page pour le site (tels que le menu, l'entête, le pied de page, ou autres scripts JavaScript utilisés par toutes les pages).

Par défaut, lors de la création d'un site Web utilisant le moteur de vue Razor le fichier de `_Layout` est créé dans `/Views/Shared`

```
_Layout.cshtml*  X
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>@ViewBag.Title - My ASP.NET Application</title>
7   @Styles.Render("~/Content/css")
8   @Scripts.Render("~/bundles/modernizr")
9
10 </head>
11 <body>
12   <div class="navbar navbar-inverse navbar-fixed-top">
13     <div class="container">
14       <div class="navbar-header">
15         <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
16           <span class="icon-bar"></span>
17           <span class="icon-bar"></span>
18           <span class="icon-bar"></span>
19         </button>
20         @Html.ActionLink("Application name", "Index", "Home", new { area = "" },
21           new { @class = "navbar-brand" })
22       </div>
23       <div class="navbar-collapse collapse">
24         <ul class="nav navbar-nav">
25           <li>@Html.ActionLink("Home", "Index", "Home")</li>
26           <li>@Html.ActionLink("About", "About", "Home")</li>
27           <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
28         </ul>
29         @Html.Partial("_LoginPartial")
30       </div>
31     </div>
32   </div>
33   <div class="container body-content">
34     @RenderBody()
35     <hr />
36     <footer>
37       <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
38     </footer>
39   </div>
40
41   @Scripts.Render("~/bundles/jquery")
42   @Scripts.Render("~/bundles/bootstrap")
43   @RenderSection("scripts", required: false)
44 </body>
45 </html>
--
```

# Styles et Layout

La vue *Layout* contient le DocType, Head et le body comme toute page classique HTML...

La seule différence et non la moindre est l'appel des méthodes *RenderBody()*, des *Renders* et de la création d'une *RenderSection*

- **RenderBody**

L'appel à cette méthode permet de définir l'emplacement où sera affiché le rendu des vues envoyées via les *Controller*

- **Renders**

- @Styles.Render(...) : permet de spécifier l'emplacement où seront insérés les styles présents dans le Bundle passé en paramètre.
- @Scripts.Render(...) : permet de spécifier l'emplacement où seront insérés les scripts présents dans le Bundle passé en paramètre.

- **RenderSection**

Un layout peut avoir plusieurs sections (header, footer, script, ...).

Afin de définir ces « zones » et permettre lors du rendu d'injecter du contenu dans notre Layout, nous utiliserons l'instruction @RenderSection.

# Bundling et Minification

ASP.NET MVC

# Bundling et Minification

Ces deux techniques sont utilisées pour optimiser le temps de chargement.

Elles permettent de diminuer le nombre de requêtes vers le server et de réduire la taille des ressources demandées (CSS et javascript)

**Bundling** : Permet de regrouper les fichiers dans un seul bundle et réduire considérablement les temps de chargement de la page.

**Minification** : La minification est l'ensemble des différentes optimisations de code js et/ou css. Cela passe par la suppression des espaces inutiles mais également d'autres techniques tels la suppression des commentaires, l'abréviation des variables, ...

# Bundling et Minification

Le système de Bundle est configurable via le fichier `/App_Start/BundleConfig.cs`.

Afin de regrouper les scripts/css ensemble, il suffit de compléter les bundles existants ou d'en créer un nouveau :

```
bundles.Add(new  
StyleBundle("VirtualPath", ["cdn"]).Include("~/PathToResource"));
```

Ou

```
bundles.Add(new  
ScriptBundle("VirtualPath", ["cdn"]).Include("~/PathToResource"));
```

```
public class BundleConfig  
{  
    // For more information on bundling, visit http://go.microsoft.com/fwlink/?LinkId=301862  
    public static void RegisterBundles(BundleCollection bundles)  
    {  
        bundles.Add(new ScriptBundle("~/bundles/jquery").Include(  
            "~/Scripts/jquery-{version}.js"));  
  
        bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(  
            "~/Scripts/jquery.validate*"));  
  
        // Use the development version of Modernizr to develop with and learn from. Then, when you're  
        // ready for production, use the build tool at http://modernizr.com to pick only the tests you need.  
        bundles.Add(new ScriptBundle("~/bundles/modernizr").Include(  
            "~/Scripts/modernizr-*"));  
  
        bundles.Add(new ScriptBundle("~/bundles/bootstrap").Include(  
            "~/Scripts/bootstrap.js",  
            "~/Scripts/respond.js"));  
  
        bundles.Add(new StyleBundle("~/Content/css").Include(  
            "~/Content/bootstrap.css",  
            "~/Content/site.css"));  
    }  
}
```

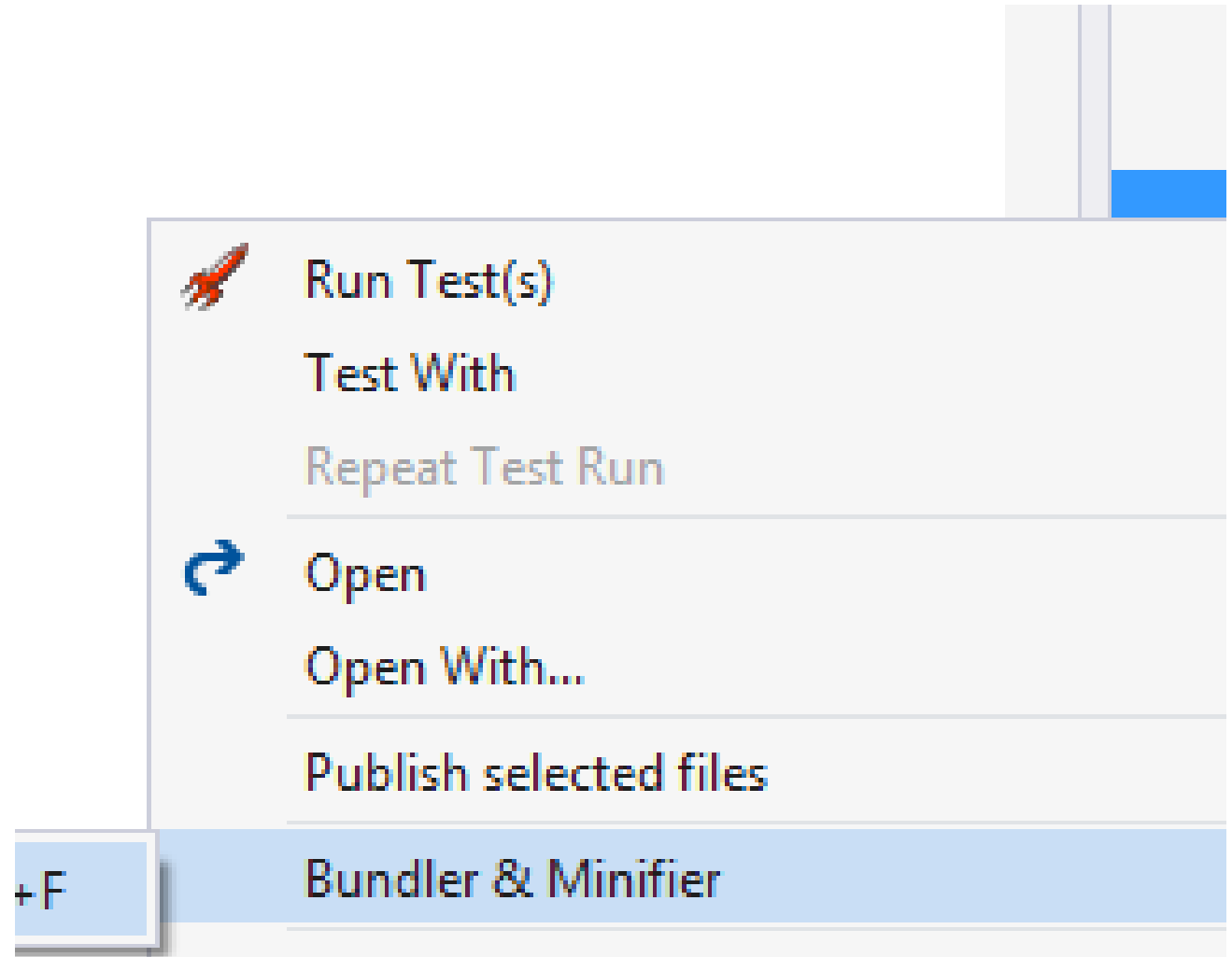


# Bundling et Minification

Une extension visual studio (« Bundler & Minifier ») permet de faciliter la mise en place de la minification et du bundling.

Plus d'information sur

<https://visualstudiogallery.msdn.microsoft.com/9ec27da7-e24b-4d56-8064-fd7e88ac1c40>



# Les contrôleurs

ASP.NET MVC

# Les Contrôleurs

- **Le rôle du Contrôleur**

Il est chargé de répondre aux entrées utilisateur et , souvent, il doit également répercuter les changements sur le Modèle initiés par les entrées utilisateur.

En résumé, il doit donc se charger de toute communication entre le Modèle et la vue et générer le cas échéant les vues de réponse.

Dans le cas d'un site web statique, l'URL pointe directement vers le fichier HTML à produire.

Dans le cas d'un site web dynamique, l'URL pointe vers des scripts qui sont chargés de générer l'HTML à la volée.

En MVC, c'est légèrement différent :

L'URL indique au moteur de *Routing\**, quel sera le contrôleur à instancier, quelle méthode à appeler dans celui-ci et fournit les arguments nécessaires à la méthode (*Action*)

L'action du contrôleur décide alors de la vue à utiliser et donc de l'HTML à générer

*\*Voir plus loin dans le cours*

# Les Contrôleurs

## Les bases d'un contrôleur

Nous allons parcourir les bases via une simple exemple : Le HomeController.

Ce HomeController est obtenu en choisissant le template de projet *ASP.NET Web Application* et en choisissant uniquement MVC.

Il est responsable de la Home Page, About Page et Contact Page.

```
public class HomeController : Controller Héríte de Controller
{
    public ActionResult Index() Home Page
    {
        return View();
    }

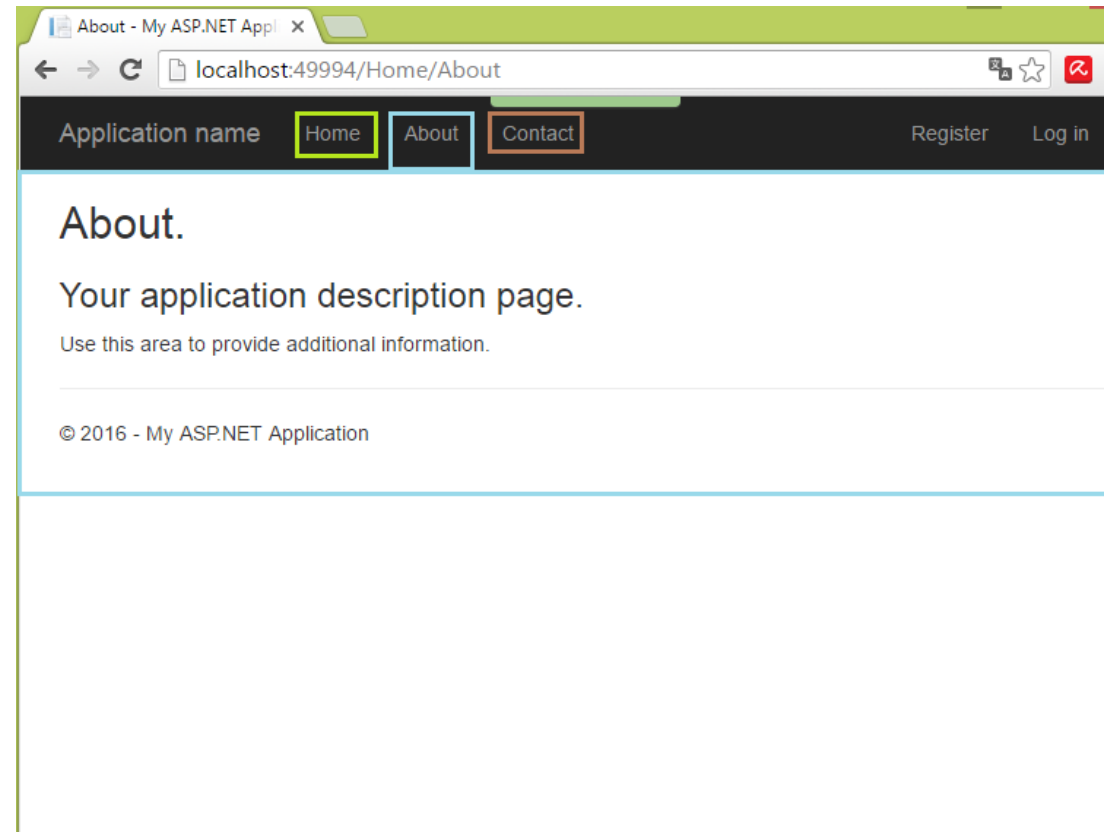
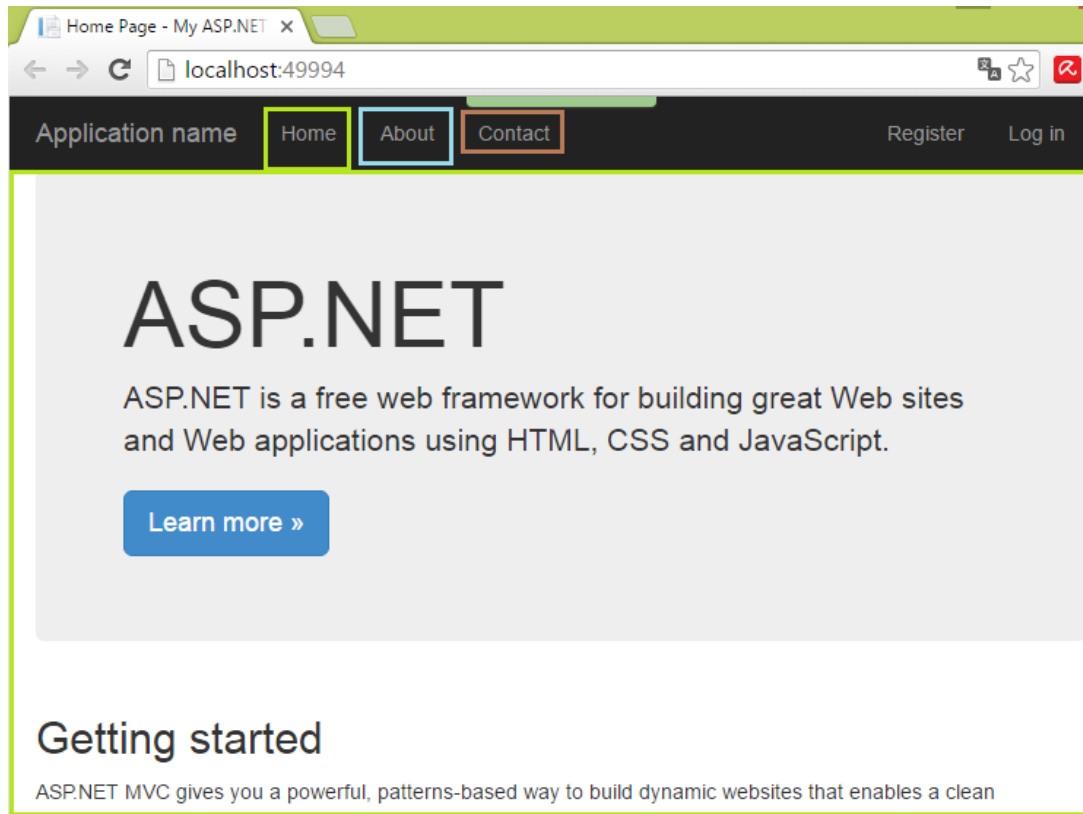
    public ActionResult About() About Page
    {
        ViewBag.Message = "Your application description page.";

        return View();
    }

    public ActionResult Contact() Contact Page
    {
        ViewBag.Message = "Your contact page.";

        return View();
    }
}
```

# Les Contrôleurs



# Les Contrôleurs

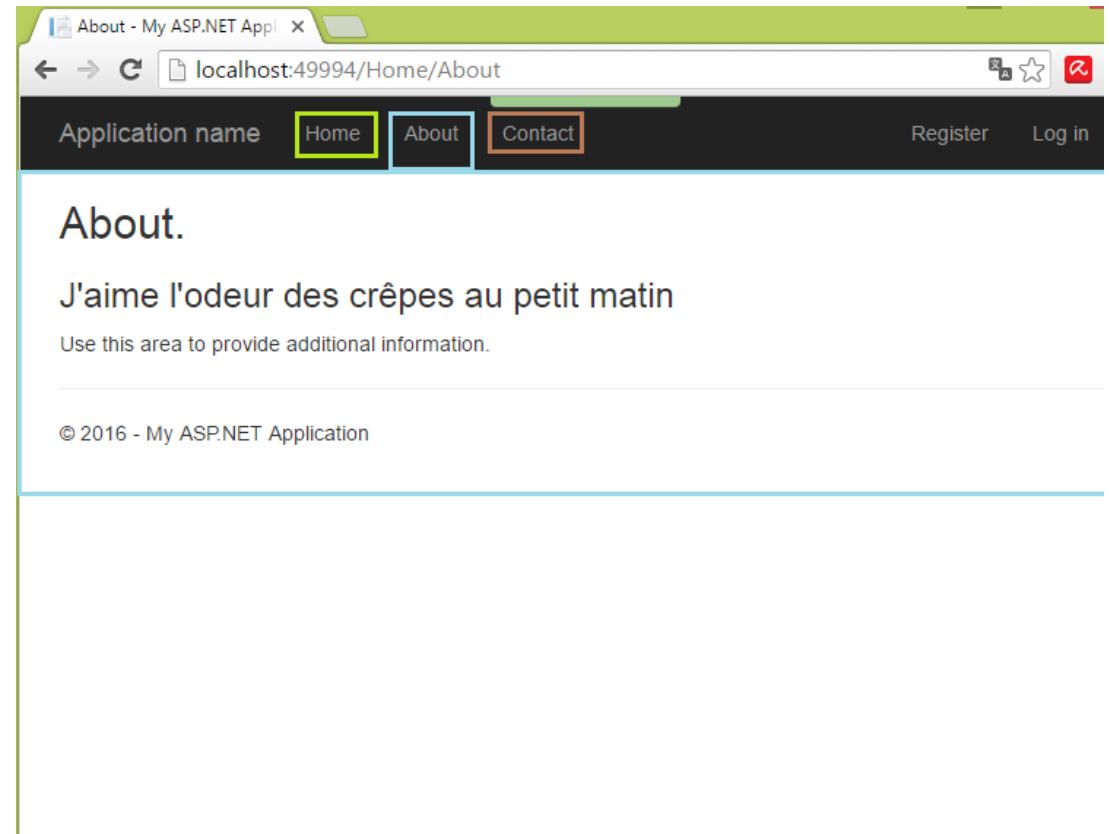
```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    public ActionResult About()
    {
        ViewBag.Message = "J'aime l'odeur des crêpes au petit matin";

        return View();
    }

    public ActionResult Contact()
    {
        ViewBag.Message = "Your contact page.";

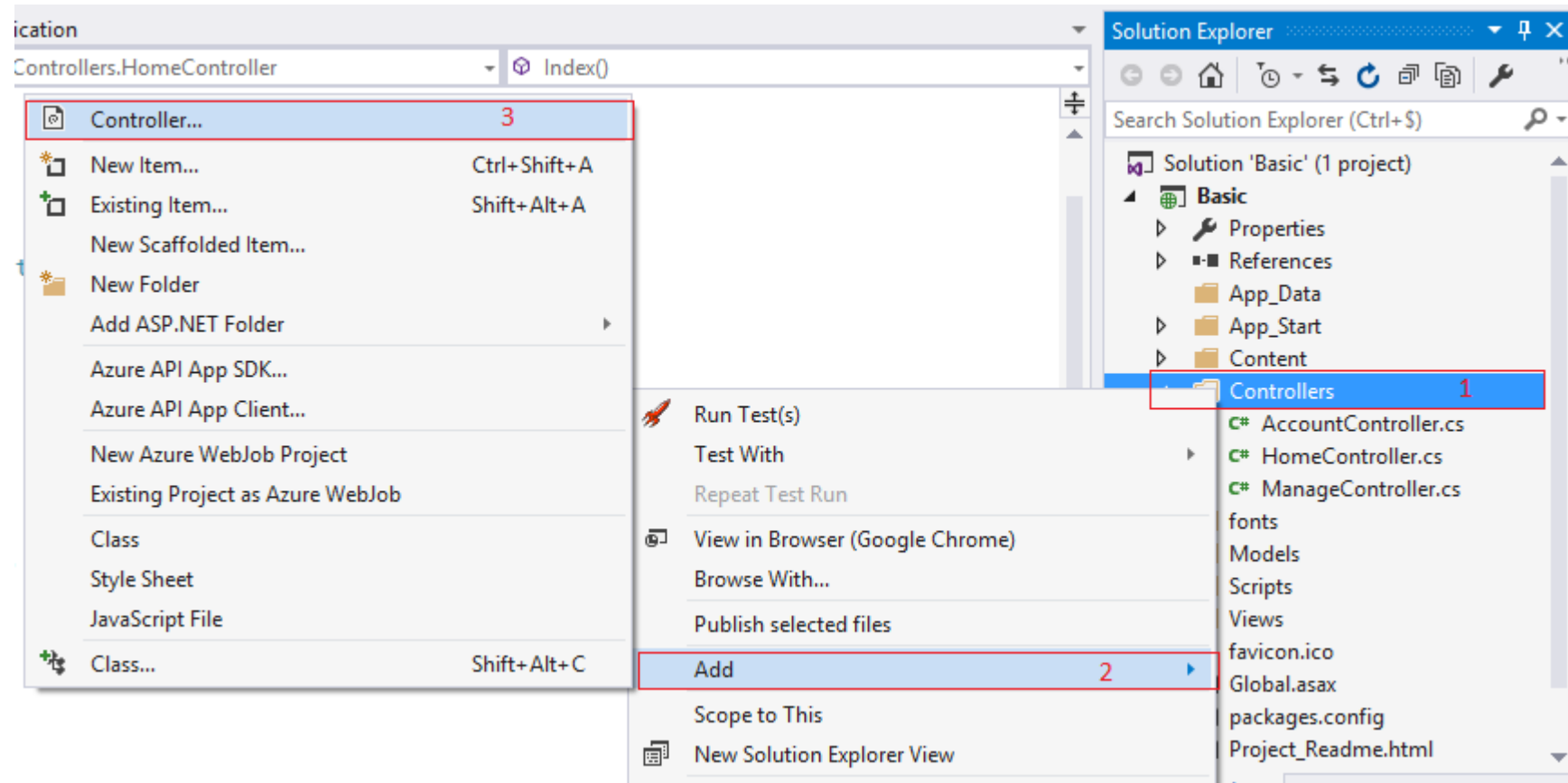
        return View();
    }
}
```



# Les Contrôleurs

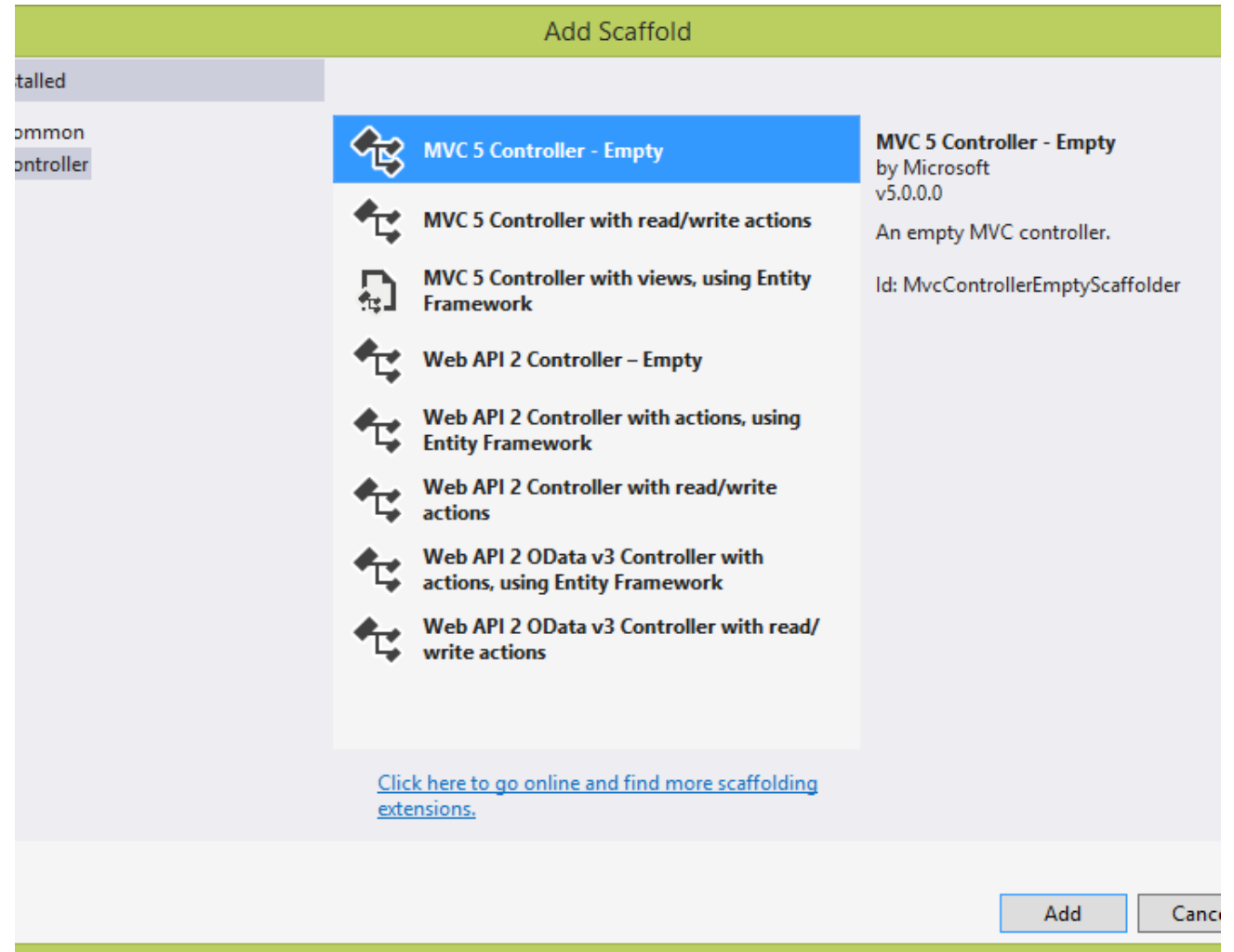
## Comment créer un contrôleur

- 1- Click droit sur le dossier Controller
- 2- Add
- 3- Controller



# Les Contrôleurs

## 4- Choisir le Empty scaffolding





# Les Contrôleurs

5- Choisir un nom pour le controller

**Remarque :**

Le suffixe Controller est déjà encodé, veuillez à ne pas l'effacer

The diagram illustrates two sequential steps in adding a controller. Each step is represented by a green header bar labeled 'Add Controller' above a light gray form area. In the first form, the text input field contains 'DefaultController' and a blue-bordered 'Add' button is on the right. A large red arrow points from this first form down to a second, identical form. In the second form, the text input field contains 'StoreController' and it also has an 'Add' button.

# Les Contrôleurs

Le contrôleur possède par défaut une action *Index*.

Nous pouvons changer le type de retour afin de par exemple renvoyer un String

```
namespace Basic.Controllers
{
    public class StoreController : Controller
    {
        // GET: Store
        public ActionResult Index()
        {
            return View();
        }
    }
}
```



```
namespace Basic.Controllers
{
    public class StoreController : Controller
    {
        // GET: Store
        public string Index()
        {
            return "Nous somme sur Store.Index";
        }
    }
}
```

# Les Contrôleurs

Nous pouvons également ajouter deux actions :

- Browse
- Details

Et ensuite lancer le projet et

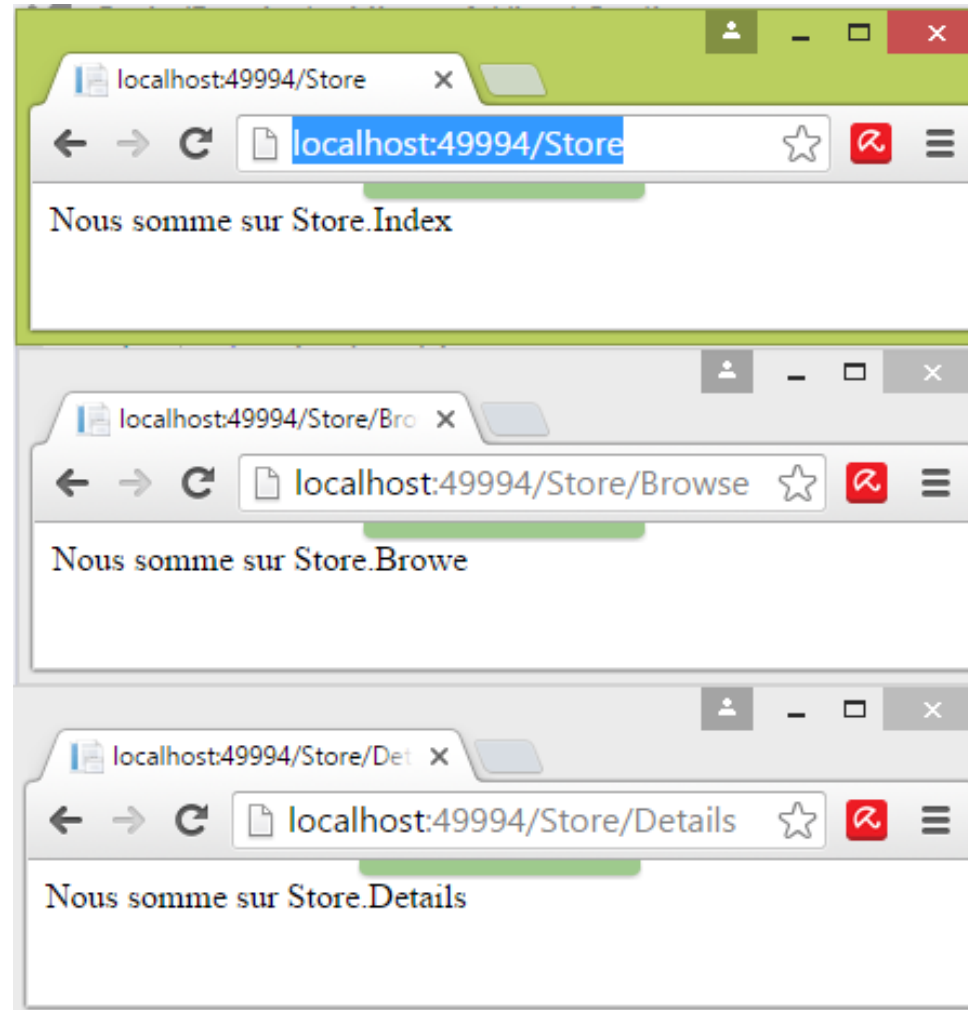
Naviguer vers les urls suivantes :

- /Store
- /Store/Browse
- /Store/Details

```
// GET: Store
public string Browse()
{
    return "Nous somme sur Store.Browse";
}

// GET: Store
public string Details()
{
    return "Nous somme sur Store.Details";
}
```

# Les Contrôleurs



# Les contrôleurs

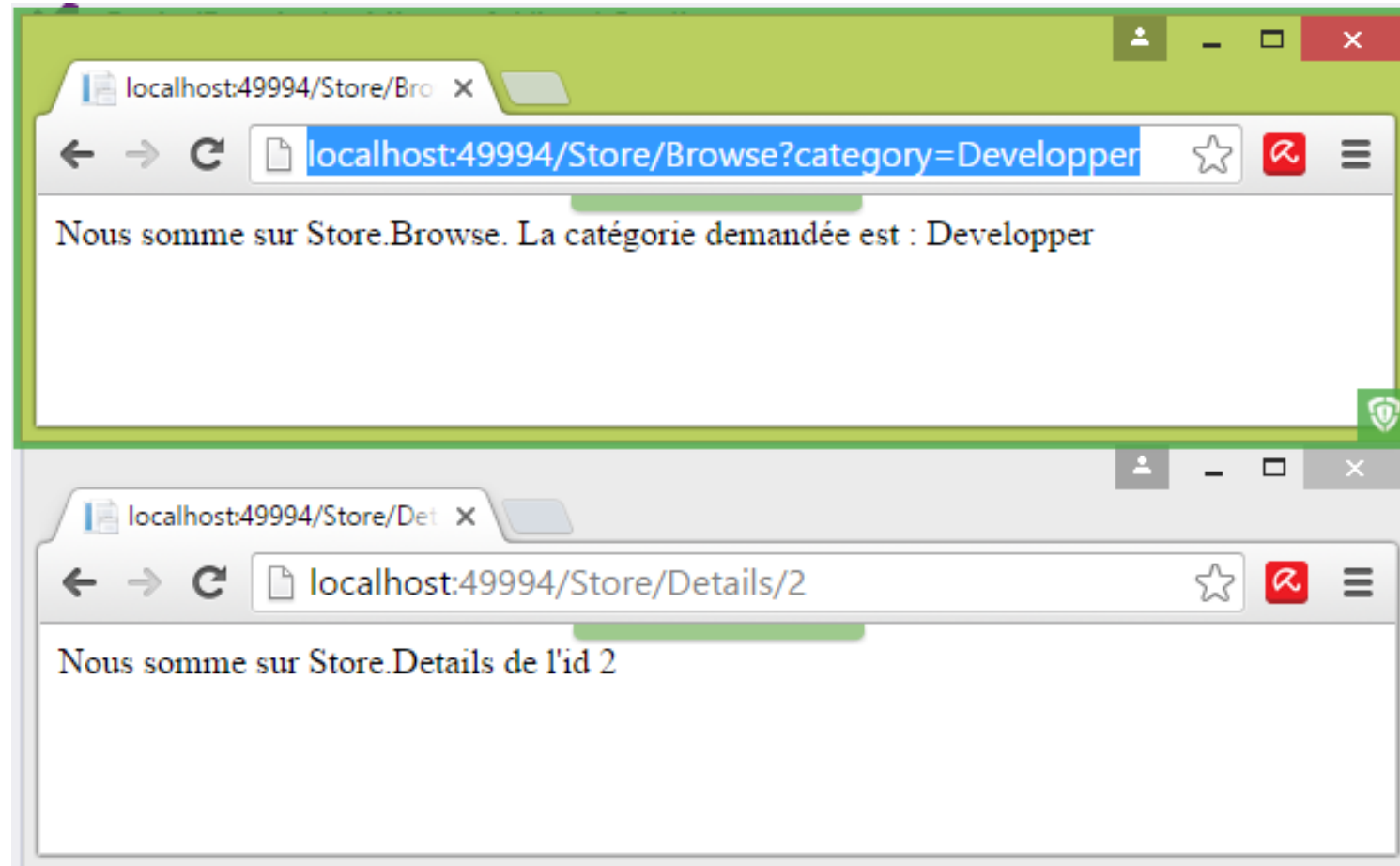
Comme toute méthode ou fonction, les Actions peuvent recevoir des paramètres.

Nous pouvons par exemple spécifier un paramètre *Category* pour l'action *Browse* et un paramètre *Id* pour l'action *Details*.

```
// GET: Store/Browse?category=Dev
public string Browse(string category)
{
    return @"Nous somme sur Store.Browse.
           La catégorie demandée est : "+ category;
}

// GET: Store/Details/5
public string Details(int id)
{
    return "Nous somme sur Store.Details de l'id "+ id;
}
```

# Les contrôleurs



# Les contrôleurs

- **La réception des données**

Un contrôleur a besoin d'accéder aux entrées de données (chaîne de caractères, formulaires, paramètres url,...)

- 1) **Extraire les données de l'objet *Context***

Si on crée un contrôleur héritant de l'objet Controller, nous avons accès aux propriétés suivantes :

- Request
- Response
- RouteData
- HttpContext
- Server

Chacune de ces propriétés nous permet d'obtenir des informations par rapport à leur contexte d'exécution.

# Les contrôleurs

Propriétés	Type	Description
Request.QueryString	NameValueCollection	Récupérer le GET
Request.Form	NameValueCollection	Récupérer le POST
Request.Cookies	HttpCookieCollection	Récupérer les cookies
Request.HttpMethod	String	Méthode utilisée (Get/POST)
Request.Headers	NameValueCollection	Le header HTTP complet
Request.Url	Uri	L'url
Request.UserHostAddress	String	L'adresse IP du client
RouteData.Route	RouteBase	L'entrée dans la table de routing de la ressource
RouteData.values	RouteValueDictionary	Les paramètres de la route
HttpContext.Application	HttpApplicationBase	La zone de stockage application
HttpContext.Cache	Cache	La zone de cache



# Les contrôleurs

Propriétés	Type	Description
HttpContext.Item	IDictionary	Stockage de la request actuelle
HttpContext.Session	HttpSessionStateBase	Récupérer la Zone session
User	IPrincipal	Information de l'utilisateur logué
TempData	TempDataDictionary	Espace de stockage temporaire pour l'utilisateur courant

```
public ActionResult RenameProduct()
{
    // Accès aux propriétés de contexte
    string userName = User.Identity.Name;
    string serverName = Server.MachineName;
    string clientIP = Request.UserHostAddress;
    DateTime dateTimeStamp = HttpContext.Timestamp;
    string oldProductName = Request.Form["Nom"];
    string newProductName = Request.Form["Prenom"];
    return View("ProductRenamed");
}
```

# Les contrôleurs

## 2) Utiliser les paramètres de la méthode Action

```
public ActionResult ShowWeatherForecast()  
{  
    string city = RouteData.Values["city"].ToString();  
    DateTime forDate = DateTime.Parse(Request.Form["forDate"]);  
}
```



```
public ActionResult ShowWeatherForecast(string city, DateTime forDate)  
{  
}
```

Nous n'avons rien d'autre à faire : le framework MVC va chercher les données dans les objets du contexte.

Remarque : Action n'autorise pas de passage via *Ref* ou *Out*

# Les contrôleurs

La classe de Base *Controller* obtient les valeurs pour les paramètres en utilisant les composants du framework MVC : le *value provider* et le *model Binder*.

## A. *Value provider*

Il représente l'ensemble des données disponibles. Il va rechercher les informations à partir de *Request.Form*, *Request.QueryString*, *Request.Files* et *RouteData.Value*.

## B. *Model Binder*

Le model Binder par défaut peut créer et peupler n'importe quel type .NET.

Nous en reparlerons plus tard.

Si le framework ne peut pas trouver de valeurs pour un paramètre, celui-ci envoie un null ➔ Exception

Pour éviter cette exception, nous pouvons utiliser des types nullable (int?, string?,...) ou encore des paramètres avec valeurs par défaut (string query=« default »,...)

# Les contrôleurs

## L'envoi des données

Un Controller , après avoir traité la requête, doit généralement produire une réponse.

Pour cela, au lieu de directement faire appel à l'objet *Response* (*Response.Write*, *Response.redirect*,....) nous renvoyons un objet dérivé de *ActionResult*.

Quand le framework reçoit un objet *Actionresult*, il appelle la méthode *ExecuteResult* de la classe *Controller*.

```
public override void ExecuteResult(ControllerContext context)
{
    string destinationUrl = UrlHelper.GenerateContentUrl(Url, context.HttpContext);
    if (Permanent)
    {
        context.HttpContext.Response.RedirectPermanent(destinationUrl,
            endResponse: false);
    }
    else
    {
        context.HttpContext.Response.Redirect(destinationUrl, endResponse: false);
    }
}
```

# Les contrôleurs

```
public class DerivedController : Controller
{
    //
    // GET: /Derived/

    public void Index()
    {
        string controller = (string)RouteData.Values["controller"];
        string action = (string)RouteData.Values["action"];
        Response.Write(string.Format("Controller: {0}, Action: {1}", controller, action));
    }

    public ActionResult Redirect()
    {
        return new RedirectResult("/Derived/Index");
    }
}
```

# Les contrôleurs

Type	Controller helper
ViewResult	View
PartialViewResult	PartialView
RedirectToRouteResult	RedirectToAction, RedirectToActionPermanent, RedirectToRoute RedirectToRoutePermanent
RedirectResult	Redirect RedirectPermanent
ContentResult	Content
FileResult	File
JsonResult	Json
JavaScriptResult	JavaScript
HttpUnauthorizedResult	None
HttpNotFoundResult	HttpNotFound
HttpStatusCodeResult	None
EmptyResult	None

# Les Vues

ASP.NET MVC

# Les vues

- **But**

La vue est responsable de fournir l'Interface Utilisateur (UI).

En bref, lorsque le Contrôleur a terminé l'exécution de la logique business, il délègue l'affichage à la vue.

**Remarque :**

Contrairement aux autres framework basé sur les fichiers, les vues ne sont pas directement accessibles.

Nous ne pouvons pas préciser une URL pour atteindre directement la vue.

La vues est TOUJOURS « rendue » via un contrôleur qui fournit les données pour l'affichage si nécessaire.



# Les vues

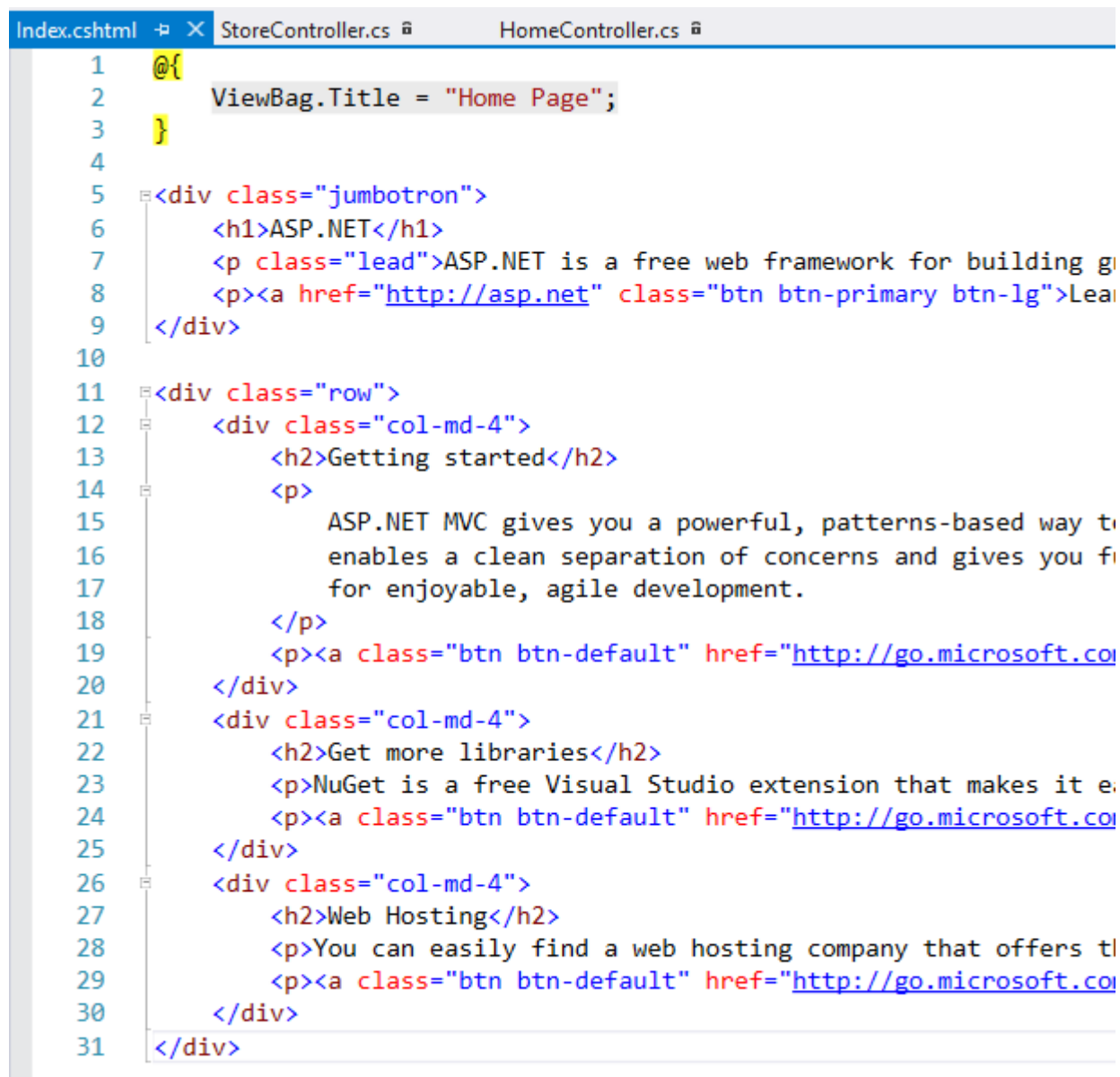
## Les bases d'une vue

La vue la plus simple disponible, est celle qui n'a pas besoin d'informations (données) venant du contrôleur.

**Exemple :** Index.cshtml

Le rendu de cette vue se fait dans le contrôleur *Home*, dans l'action *Index*

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



```
Index.cshtml X StoreController.cs HomeController.cs
1 @{
2     ViewBag.Title = "Home Page";
3 }
4
5 <div class="jumbotron">
6     <h1>ASP.NET</h1>
7     <p class="lead">ASP.NET is a free web framework for building g
8     <p><a href="http://asp.net" class="btn btn-primary btn-lg">Lea
9 </div>
10
11 <div class="row">
12     <div class="col-md-4">
13         <h2>Getting started</h2>
14         <p>
15             ASP.NET MVC gives you a powerful, patterns-based way to
16             enables a clean separation of concerns and gives you fi
17             for enjoyable, agile development.
18         </p>
19         <p><a class="btn btn-default" href="http://go.microsoft.co
20     </div>
21     <div class="col-md-4">
22         <h2>Get more libraries</h2>
23         <p>NuGet is a free Visual Studio extension that makes it e
24         <p><a class="btn btn-default" href="http://go.microsoft.co
25     </div>
26     <div class="col-md-4">
27         <h2>Web Hosting</h2>
28         <p>You can easily find a web hosting company that offers ti
29         <p><a class="btn btn-default" href="http://go.microsoft.co
30     </div>
31 </div>
```

# Les vues

Cet exemple basique nous montre l'utilisation du mot clé *ViewBag*.

Le *ViewBag* permet de définir des propriétés ou des objets qui seront disponibles sur la vue.

Ces objets dynamiques sont accessible via *Controller.ViewBag*

```
public ActionResult Index()  
{  
    ViewBag.Message = "Hello";  
    ViewBag.Date = DateTime.Now;  
    return View();  
}
```



```
<div>  
    Nous sommes le @ViewBag.Date.DayOfWeek  
    Le message est @ViewBag.Message  
</div>
```

## Remarque :

Le *ViewBag* est utilisée pour le passage de petites données.

En aucun cas, il ne faut utiliser le *ViewBag* pour passer des objets complexes.

# Les vues

- **Les conventions**

Dans le contrôleur, nous avons simplement l'instruction `return View()`

Pas besoin de donner le nom de la vue car, par convention, le moteur MVC utilise la logique suivante :

→ Dossier View → Dossier portant le nom du contrôleur → Fichier portant le nom de l'action → Extension cshtml ou aspx ou asmx.

Les deux dernière extension sont moins souvent utilisées.

.chhtml = razor view

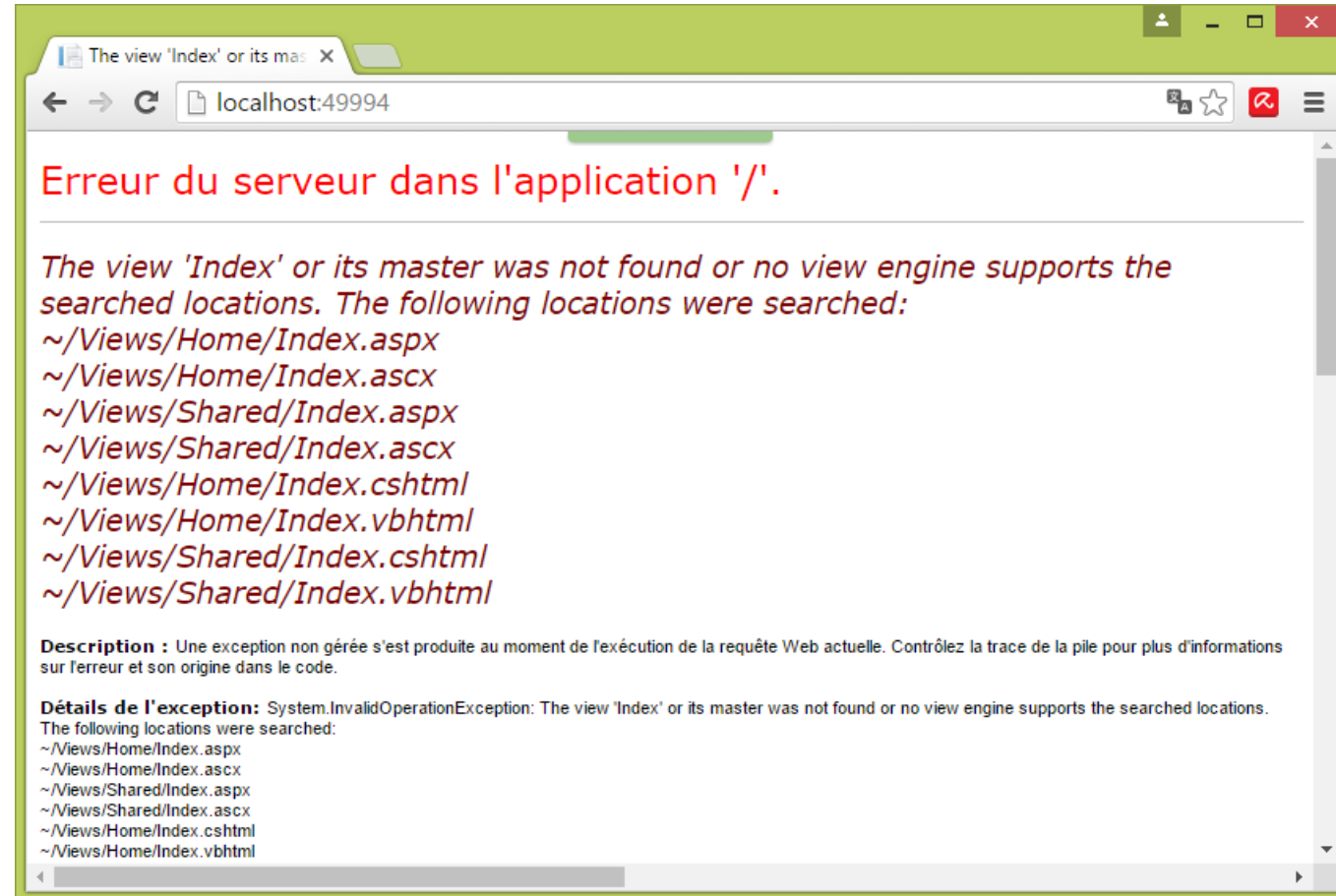
.aspx = WebForm (Old school)

.asmx = fichier webservice asp

Si le moteur MVC ne trouve pas de fichier approprié via la logique précédente, il regarde dans le dossier *View/Shared*.

Et si toujours pas de fichier correct, nous obtenons une exception.

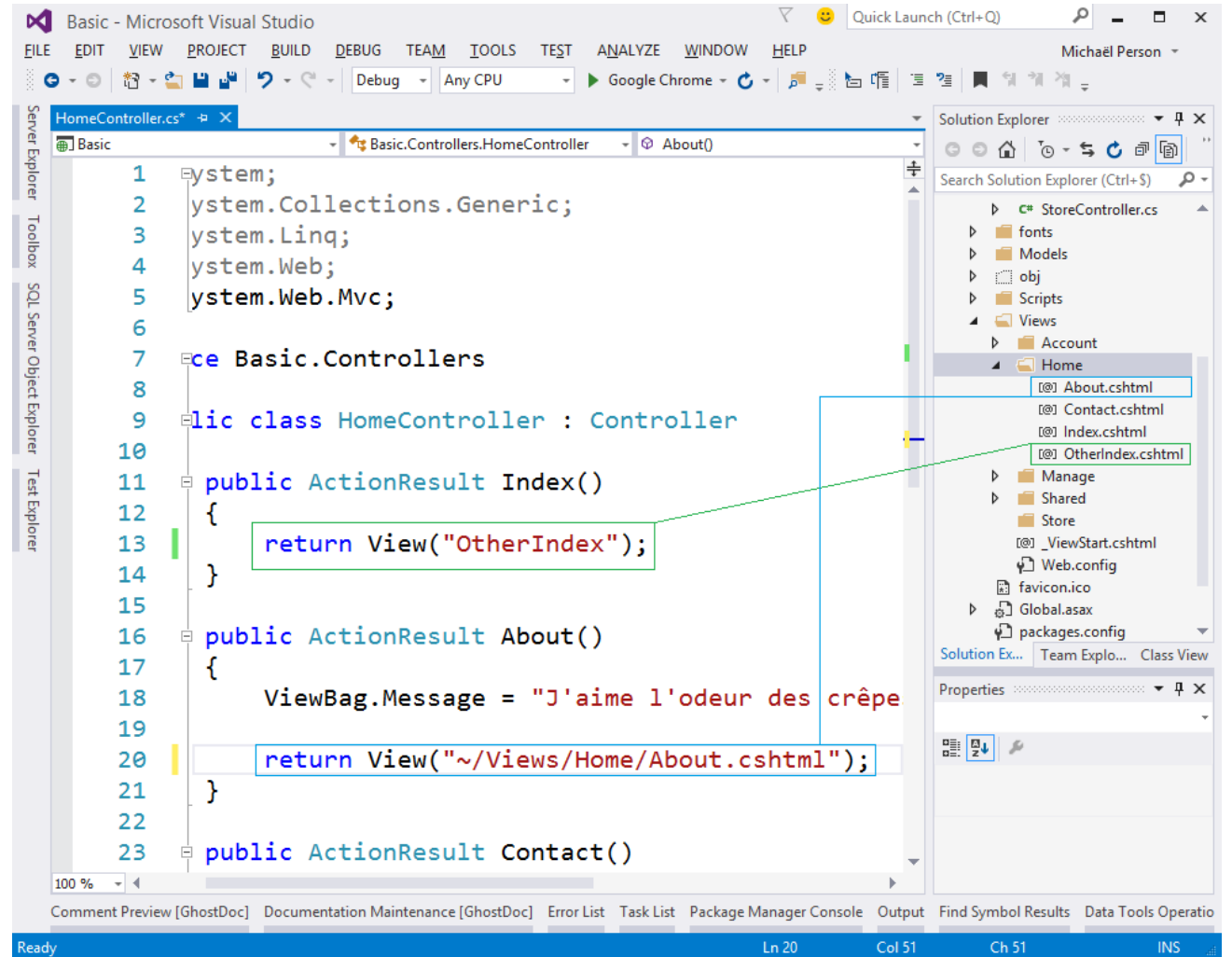
# Les vues



# Les vues

Il est cependant possible de contourner la convention.

- passant le nom de la vue
- En donnant le chemin complet vers la vue



# Les vues

## Le « ViewBag »

Les données gérées par les vues ne sont pas toujours en lien directe avec le model de domaine.

Une vue pourrait, par exemple, afficher les informations d'un produit mais également les informations de l'utilisateur courant, les catégories liées, etc...

Une façon simple, sans utiliser les modèles et les vues fortement typées que nous verrons plus tard, est d'utiliser le *ViewBag*.

Bien entendu, nous utiliserons cette technique pour uniquement des données « basique » et non pour des données complexes.

```
troller.cs -> X
Basic.Controllers.HomeController
Index()

6
7 namespace Basic.Controllers
8
9 public class HomeController : Controller
10
11 public ActionResult Index()
12 {
13     ViewBag.InfoUser = new { Nom = "Person", Prenom = "Mike" };
14     ViewBag.InfoProduit = new { Nom = "Eau", Prix = 2.14 };
15     return View("OtherIndex");
16 }
17

cshtml -> X
1
2 @{}
3     ViewBag.Title = "OtherIndex";
4 }
5
6 <h2>OtherIndex</h2>
7 <h3>Utilisateur</h3>
8 @ViewBag.InfoUser.Nom @ViewBag.ViewBag.InfoUser.Prenom
9 <hr />
10 @ViewBag.InfoProduit.Nom : @ViewBag.ViewBag.InfoProduit.Prix &euro;
11
12
```

# Les vues

## Créer une vue

La création d'une vue est grandement facilitée grâce à Visual Studio.

La façon la plus simple est de partir de l'action :

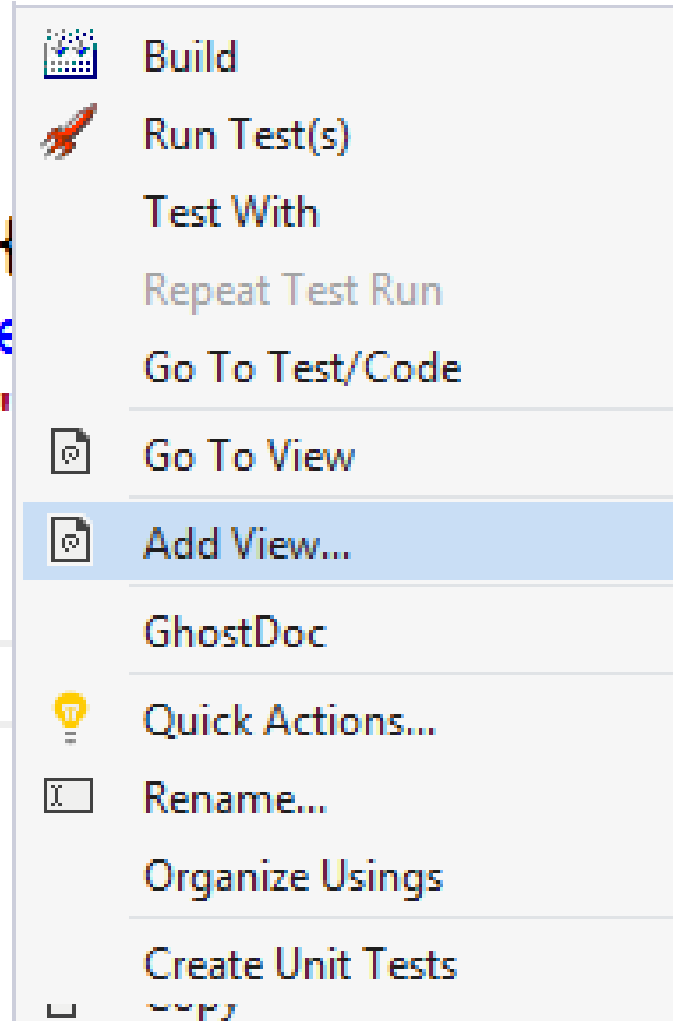
- Click droit
- Add View

```
Result Index()
```

```
InfoUser = new {  
InfoProduit = new {  
View("OtherIndex")
```

```
Result Edit()
```

```
View();
```



# Les vues

Dans l'écran suivant , nous devons choisir les options de création :

- **View Name** : nom de la vue
- **Template** : Permet d'avoir une vue avec les actions usuels par rapport à un template (List, Edit, Create,...).

C'est ce qu'on appelle le *scaffolding*

The screenshot shows the 'Add View' dialog box. The 'View name' field contains 'Edit'. The 'Template' dropdown is open, showing 'Empty (without model)' as the selected option. The 'Model class' dropdown is also open, showing a list of options: 'Create', 'Delete', 'Details', 'Edit', 'Empty', 'Empty (without model)', and 'List'. The 'Data context class' dropdown is open, showing 'Empty'. The 'Options' section has three checkboxes: 'Create as a partial' (unchecked), 'Reference script lib' (unchecked), and 'Use a layout page:' (checked). Below the 'Use a layout page:' checkbox is a text box and a button with three dots. Below the text box is the text '(Leave empty if it is set in a Razor \_viewstart file)'. At the bottom right are 'Add' and 'Cancel' buttons.



# Les vues

Scaffold	Description
Create	Une vue avec un formulaire permettant de créer une instance du Model. 1 label + 1 input pour chaque propriété
Delete	Une vue avec un formulaire permettant de supprimer une instance existante du Model. 1 label + la valeur courante de chaque propriété.
Details	Une vue affichant les valeurs de chaque propriété du Model.
Edit	Une vue avec un formulaire permettant de modifier une instance existante du Model. 1 label + la valeur courante de chaque propriété.
Empty	Vue vide. Seul le type du Model peut apparaitre.
Empty(without model)	Une vue vide. Seul template permettant de ne pas sélectionner de Model lors de la création.
List	Une vue affichant les propriétés du Type Model. Il faut lui passer un <i>Ienumerable&lt;T&gt;</i> . Cette vue contient également un lien vers les actions <i>Create/Edit/Delete</i>

# Les vues

- **Create as Partial View** : permet de créer une vue qui n'aura ni <head> ni <body> ni <html>. Elle est destinée à s'intégrer dans une autre.
- **Reference script libraries** : Permet d'intégrer les bibliothèques javascripts (jQuery,..) nécessaires à la vue.  
Pour *Create et Edit*, cette option ajoute une référence au bundle *jqueryEval* qui est nécessaire pour la validation de formulaire côté client.
- **Use a layout page** : permet de spécifier le template visuel associé à la vue. Par défaut, il s'agit de la vue partielle : /Views/Shared/\_layout.cshtml

The screenshot shows the 'Add View' dialog box. The 'View name' field contains 'Edit'. The 'Template' dropdown is set to 'Create'. The 'Model class' and 'Data context class' dropdowns are empty. Under the 'Options' section, the 'Use a layout page' checkbox is checked. Below this checkbox is a text box for specifying the layout page, followed by a browse button ('...'). A note indicates to leave it empty if it's set in a Razor \_viewstart file. The 'Add' and 'Cancel' buttons are at the bottom right.

# Les Modèles

ASP.NET MVC

# Les Modèles

La couche Modèle s'assure de la logique Business.

Il faut bien distinguer :

- Le « modèle global », est indépendant de la partie cliente et il est utilisé dans le cadre inter-applicatifs. Dans la plupart des cas, il est soit fourni par une autre équipe de développement sur un projet qui définit déjà les différentes entités, soit il est généré par un outil tel que « Entity Framework », « LINQ To SQL », etc.  
Dans ces deux cas de figure, la solution consiste à ne rien faire de particulier concernant le modèle global si ce n'est de s'adapter à lui au moment de la construction du modèle de présentation.
- Le « modèle client », également appelé modèle de présentation, est entièrement lié à l'application cliente et répond à ses problématiques.

# Les Modèles

Par convention, il est recommandé de placer les classes modèles dans le dossier Model.

Cependant, pour les projets de grande envergure, il est parfois utile de travailler via les bibliothèques de classes.

## 1. Construire le « *Modèle Global* » :

Lors de la définition du modèle global, il faut garder en tête que celui-ci peut, à terme, être utilisé au sein de plusieurs applications au sens général du terme : applications hébergées sur un serveur, applications mobiles, applications en ligne de commandes, applications riches ...

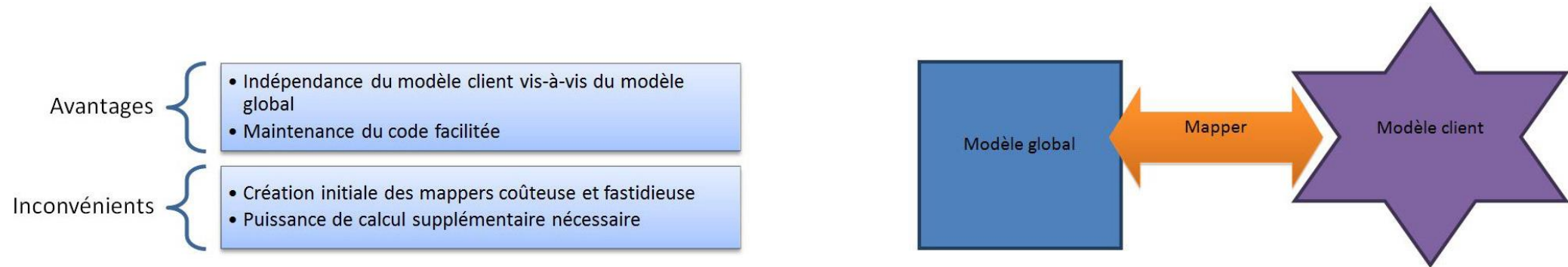
Il n'y a donc AUCUNE LOGIQUE d'interaction (réaction aux événements UI, Gestion de formulaire,...) dans ce modèle.

# Les Modèles

## 2. Construire le « *Modèle Client* » :

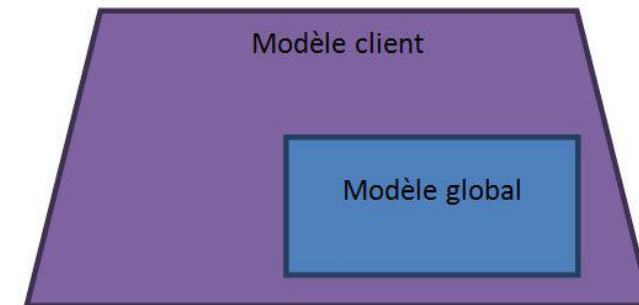
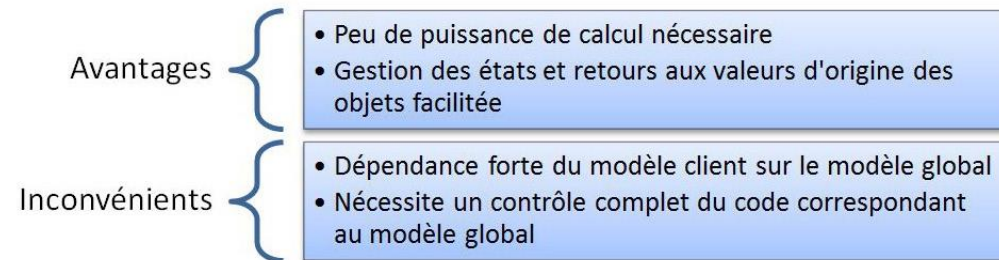
Pour cela, il existe deux techniques, le « Mapping » ou le « Wrapping ».

Le « *Mapping* » consiste à faire correspondre une ou plusieurs propriétés du modèle global avec une ou plusieurs propriétés du modèle client. Ainsi, pour transposer une entité du modèle global en une entité du modèle client, il faut remplir l'instance du modèle client à l'aide d'un objet appelé « **Mapper** » qui va se contenter de copier les valeurs d'un objet à l'autre en prenant en compte les différences de noms.



# Les Modèles

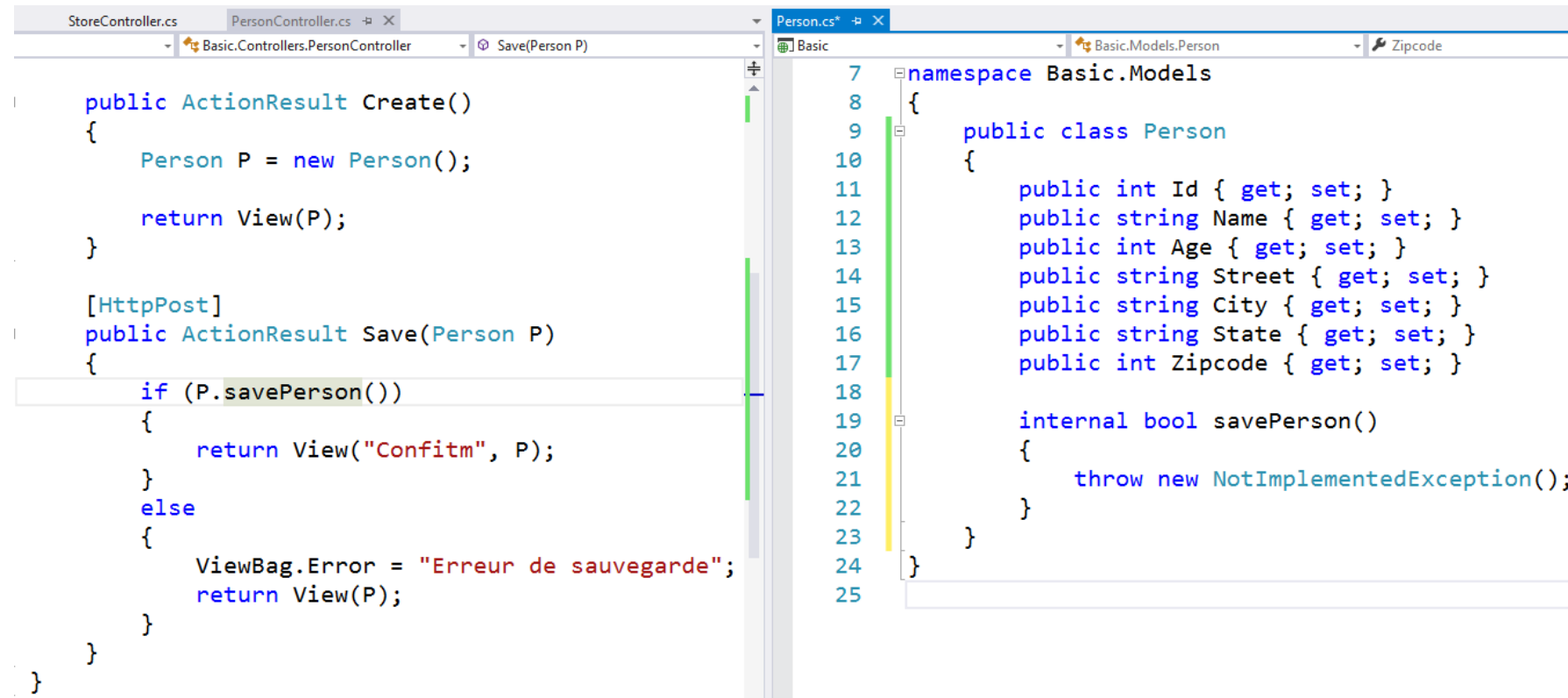
Quant au « *Wrapping* », il consiste littéralement à enrober ou encore envelopper le modèle global par le modèle client qui se comporte comme un adaptateur. Aussi, chaque objet du modèle client va contenir une instance du modèle global et l'utiliser comme conteneur de données. C'est cette technique que nous allons employer.



# Les Modèles

- Intégration du modèle dans le contrôleur

L'utilisation du modèle dans le contrôleur consiste généralement à simplement instancier celui-ci et ensuite de faire appel aux différentes méthodes, fonctions afin d'extraire les données nécessaires pour l'intégration avec les vues.



```
StoreController.cs | PersonController.cs | Person.cs*
Basic.Controllers.PersonController | Save(Person P) | Basic | Basic.Models.Person | Zipcode

public ActionResult Create()
{
    Person P = new Person();

    return View(P);
}

[HttpPost]
public ActionResult Save(Person P)
{
    if (P.savePerson())
    {
        return View("Confitm", P);
    }
    else
    {
        ViewBag.Error = "Erreur de sauvegarde";
        return View(P);
    }
}
}

namespace Basic.Models
{
    public class Person
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public int Zipcode { get; set; }

        internal bool savePerson()
        {
            throw new NotImplementedException();
        }
    }
}
```



# Les Modèles

## Scaffolding

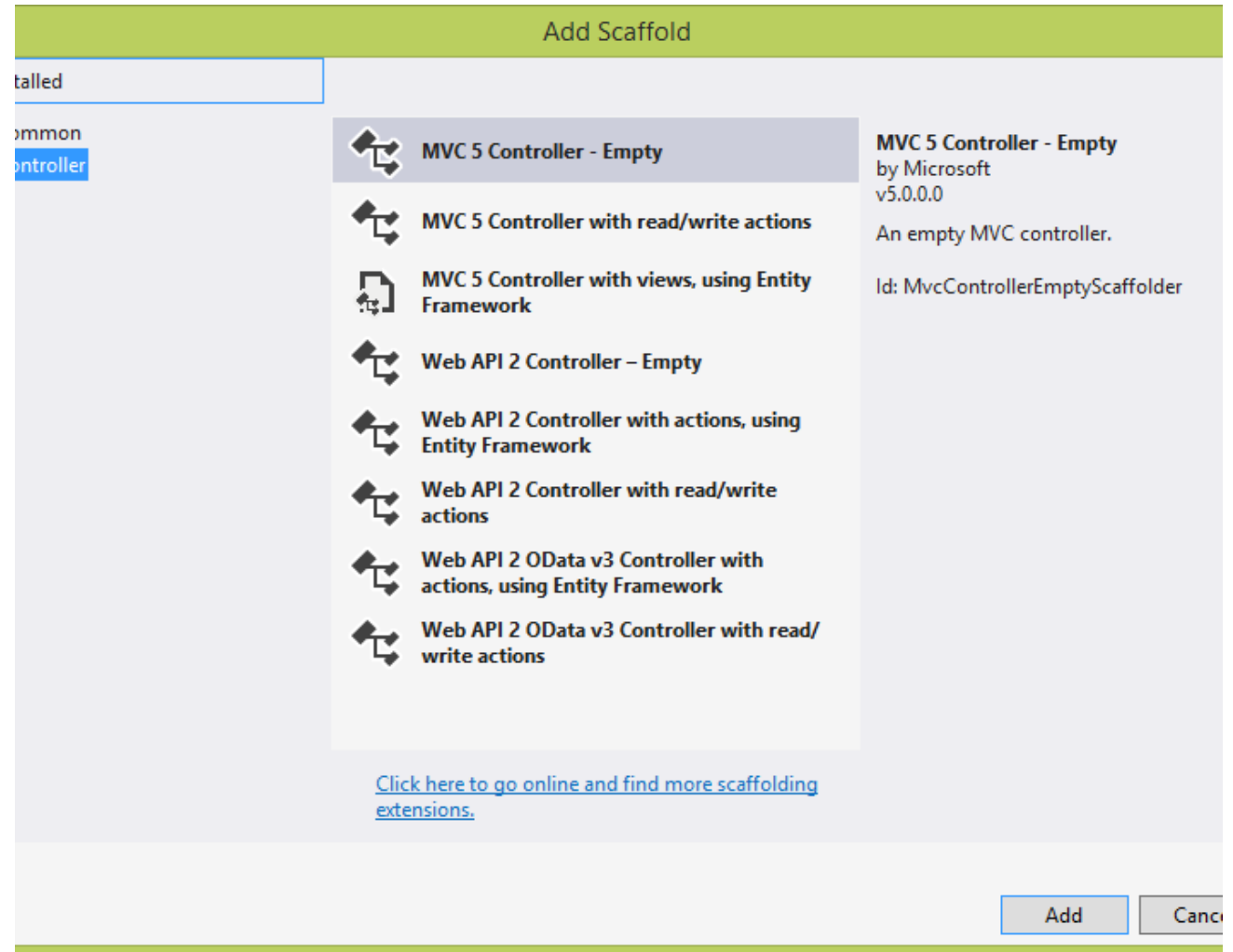
En ASP.NET, le scaffolding permet de générer le squelette CRUD (**C**reate, **R**ead, **U**psert et **D**eleter)

Il a également la possibilité, en analysant le model utilisé pour la création d'une vue fortement typé, de générer un contrôleur, les vues associées et, dans certains cas, les classes nécessaires à la manipulation des données.

### Remarques :

N'utilisez JAMAIS le scaffolding afin de générer la totalité de votre application.

Utilisez le scaffolding uniquement pour générer le code rébarbatif afin de vous concentrer sur l'essentiel de la logique de votre application.



# Les Modèles

## Empty

Une simple classe dérivée de la classe *Controller*. Pas de vue créée.

## Controller with read/write Actions

Un controller avec les actions GET et POST pour le CRUD.

Les actions ne sont pas entièrement vides mais ne contiennent

Aucun code fonctionnel.

```
}

// GET: RW/Details/5
public ActionResult Details(int id)
{
    return View();
}

// GET: RW/Create
public ActionResult Create()
{
    return View();
}

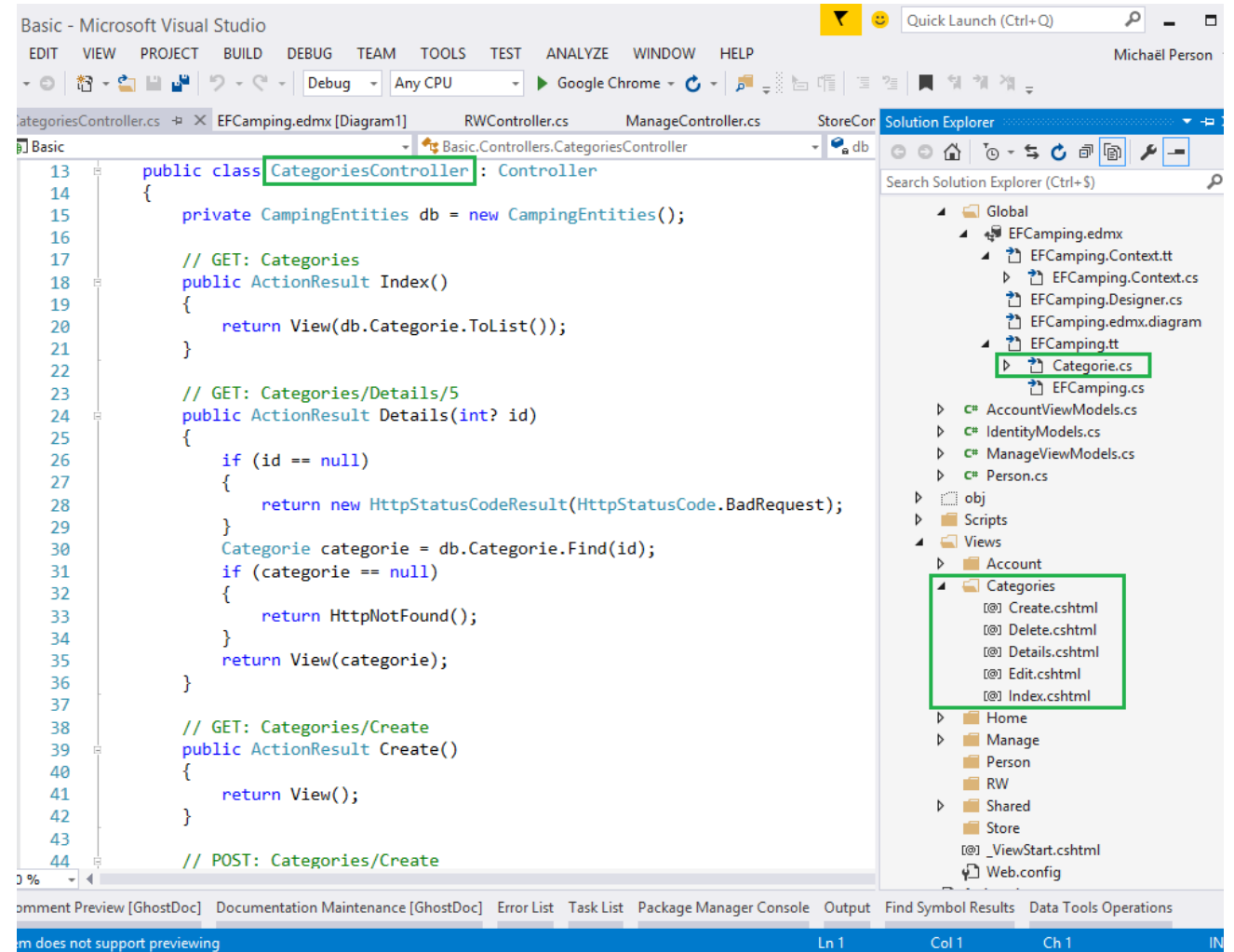
// POST: RW/Create
[HttpPost]
public ActionResult Create(FormCollection collection)
{
    try
    {
        // TODO: Add insert logic here

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

# Les Modèles

## Controller with views , using Entity Framework

Ce template génère le contrôleur contenant les actions pour le CRUD, ainsi que le code fonctionnel et les vues pour la persistance et la manipulation des données à partir d'une base de données.



# Les Modèles

## Model Binding

Sans le moteur MVC, nous sommes obligé de manipuler l'objet Request afin de récupérer par exemple les données d'un formulaire posté.

Cela peut-être long et les conversions de données peuvent entrainer des erreurs.

Grâce au moteur MVC, nous pourrons nous simplifier la vie et limiter les erreurs.

```
// POST: /Create
[HttpPost]
public ActionResult Create()
{
    Person p = new Person();
    p.Name = Request.Form["Name"];
    p.Age = int.Parse(Request.Form["Age"]);
    //etc...
    return View();
}
```

# Les Modèles

- **DefaultModelBinder**

A la place de récupérer les données de l'objet Model via *Request*, nous pouvons simplement spécifier l'objet que nous voulons récupérer.

Le moteur MVC va automatiquement instancier un objet du type attendu et mapper les valeurs venant du formulaire aux propriétés de l'instance créée.

**Remarque :**

Le model doit avoir un constructeur vide afin de pouvoir l'utiliser.

```
// POST: /Create
[HttpPost]
public ActionResult Create()
{
    Person p = new Person();
    p.Name = Request.Form["Name"];
    p.Age = int.Parse(Request.Form["Age"]);
    //etc...
    return View();
}
```



```
// POST: /Create
[HttpPost]
public ActionResult Create(Person P)
{
    return View();
}
```

# Les Modèles

Le moteur MVC ne se contente pas de récupérer les données de l' HTTP POST mais également de l'HTTP GET .

Le moteur MVC est comme un chien de recherche, lorsqu'on s'attend à avoir un paramètre, il cherche partout jusqu'à ce qu'il trouve.

## Explicit Model Binding

Il est également possible de faire appel à deux méthodes *UpdateModel* et *TryUpdateModel*.

Ces deux méthodes permettent d'effectuer une liaison explicite.

```
/ POST: /Create
HttpPost]
public ActionResult Create()

    Person P = new Person();
    try
    {
        UpdateModel<Person>(P);
        //Logique de sauvegarde DB
        //.....
        return View();
    }
    catch (Exception ex )
    {
        ViewBag.Error = ex.Message;
    }
    return View();
```

# Les Modèles

*TryUpdateModel* permet d'effectuer également le Binding mais sans lever d'exception.

## Remarque :

Il est possible de vérifier si le binding c'est effectué correctement ou si les données envoyées respectent les contraintes de données (type, nul,...)

Pour cela, il faut utiliser :

`If(ModelState.IsValid)`

```
// POST: /Create
[HttpPost]
public ActionResult Create()
{
    Person P = new Person();
    if (TryUpdateModel<Person>(P))
    {
        //Logique de sauvegarde DB
        //....
        return View();
    }
    else
    {
        ViewBag.Error = "Erreur de sauvegarde";
    }
    return View();
}
```

# Razor

ASP.NET MVC



# Razor

Nous allons faire un rapide tour de la syntaxe Razor.

Le but de Razor est de minimiser le nombre de caractères requis à l'écriture d'un fichier de vues et d'introduire une nouvelle manière de coder qui est fluide et rapide. Au contraire de ce qu'on connaît déjà, le développeur n'a pas besoin de s'interrompre de coder pour dénoter explicitement les blocs serveur dans le HTML. Le parseur de Razor est assez intelligent afin d'inférer ceci à partir du code.

- L'extension des vues Razor est .cshtml pour les vues sous C# et .vbhtml pour les vues utilisant VB.NET.
- Les fichiers qui ne peuvent pas être affichés directement par la requête (les master pages, les vues partiels, etc.) sont précédés par un caractère “\_” au début de leur noms. Donc si vous essayez d'afficher \_Layout.cshtml qui est une Master Page vous obtiendrez une erreur serveur.

# Razor

- **Code Expression**

Pour commencer à utiliser *Razor*, nous avons simplement à utiliser @.  
Ce signe permet d'effectuer la transition entre l'html et le code.

Les expressions sont évaluées directement :

```
<h1> Nombre d'éléments : @items.Lengths</h1>
```



**Nombre d'éléments : 3**

- **Code Block**

Le code de block simple et multi ligne est contenu entre les accolades précédés par un caractère “@”:

```
@{ var @nom = "JASKULA";  
    var @prenom = "Thomas";}  
<p>Bonjour @prenom. Votre nom est @nom.</p>
```

# Razor

- **Variables**

Pour déclarer les variables utilisez le mot clé “**var**” ou la syntaxe classique fortement typée. Le mot clé “**new**” est également autorisé.

```
var message = "Razor est cooooooooool";  
string title = "Syntaxe de Razor";  
int number = 1558;  
var stringBuilder = new System.Text.StringBuilder();
```

- **Chaines de caractères**

Si dans votre chaîne des caractères il y a les backslash “\” ou les guillemets, vous pouvez utiliser l’opérateur “@” pour indiquer la chaîne verbatim : *var folder = @"C:\Files\ASP\MVC";*

Pour les textes avec les guillemets, vous devez utiliser l’opérateur de chaîne verbatim est de doubler les guillemets comme dans l’exemple ci dessous : *var citation = @"Thomas a dit : ""Utilisez Razor est vous serez libérés.""";*

A l’intérieur du bloc de code, si vous voulez afficher le texte plein vous pouvez utiliser les caractères “@:” : *@{ @:Ceci est un texte plein. }*

# Razor

- **Commentaires**

Dans Razor pour indiquer les commentaires vous commencez avec les caractères “@\*” et terminez avec “\*@”.

A l’intérieur de blocks de code vous avez la possibilité d’utiliser la syntaxe de commentaire de votre langage de programmation. Quelques exemples :

```
@{  
    @* Ceci est un commentaire *@  
    int age = 85;  
    @* Tout est commenté *@  
    @* int age = 85; *@  
    @* Ceci est le commentaire multiligne  
    une autre ligne *@  
}
```

# Formulaires & Helpers

ASP.NET MVC

# Formulaires & Helpers

- **L' Action et la Méthode**

Les deux attributs les plus importants pour un formulaire sont : action et method.

L'attribut *action* spécifie l'endroit où les informations doivent être envoyées. L'url peut être relative ou absolue.

L'attribut *method* spécifie si l'envoi se fait via HTTP GET ou HTTP POST. La valeur par défaut est HTTP GET.

Lorsque le formulaire est envoyé en GET, les données du formulaire sont envoyées en clair via l'url

<http://www.exemple.com/LeaveAComment.aspx?nom=Person&Message=LeMessage>

Dans le cas d'un POST, les données sont incluses dans le Body de la requête HTTP.

## GET ou POST?

Get pour les opérations en lecture seule

POST pour les opérations de mise à jour ou d'un changement de mot de passe....

# Formulaires & Helpers

Si un formulaire classique peut, sans aucun problème, poster les données vers notre action de notre contrôleur, ASP.NET MVC nous offre des aides pour générer nos formulaires, les *Html Helpers*

## 1. Html.BeginForm()

Html Helper produisant un formulaire html dont l'action est configurée pour poster celui-ci vers la même url si aucun paramètre n'est passé.

Différentes signature permettent de paramétrer le post

The image displays two side-by-side windows illustrating the use of the `Html.BeginForm()` helper in ASP.NET MVC.

The left window shows the rendered HTML output of the form. It features a text input field labeled "Nom:" and a "Create" button. The HTML structure is as follows:

```
<h2>Index</h2>
<form action="/RW/Index" method="post">
  Nom: <input type="text" id="Nom" name="Nom" />
  <input type="submit" value="Create" />
</form>
```

The right window shows the source code of the `Index.cshtml` file. It demonstrates how the `Html.BeginForm()` helper is used to generate the form structure:

```
@model Basic.Models.Person
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
@using (Html.BeginForm())
{
    @:Nom: <input type="text" id="Nom" name="Nom" />
    @:<input type="submit" value="Create" />
}
```

# Formulaires & Helpers

Nous pouvons par exemple spécifier les paramètres suivant à l' Html.BeginForm() :

```
@using (Html.BeginForm("Save", "Person", FormMethod.Post))
```

➔ <form action="/Person/Save" method="post">

```
@using (Html.BeginForm("Save", "Person", FormMethod.Post, new { @class = "FormCss" }))
```

➔ <form action="/Person/Save" class="FormCss" method="post">



# Formulaires & Helpers

## 2. Html.ValidationSummary

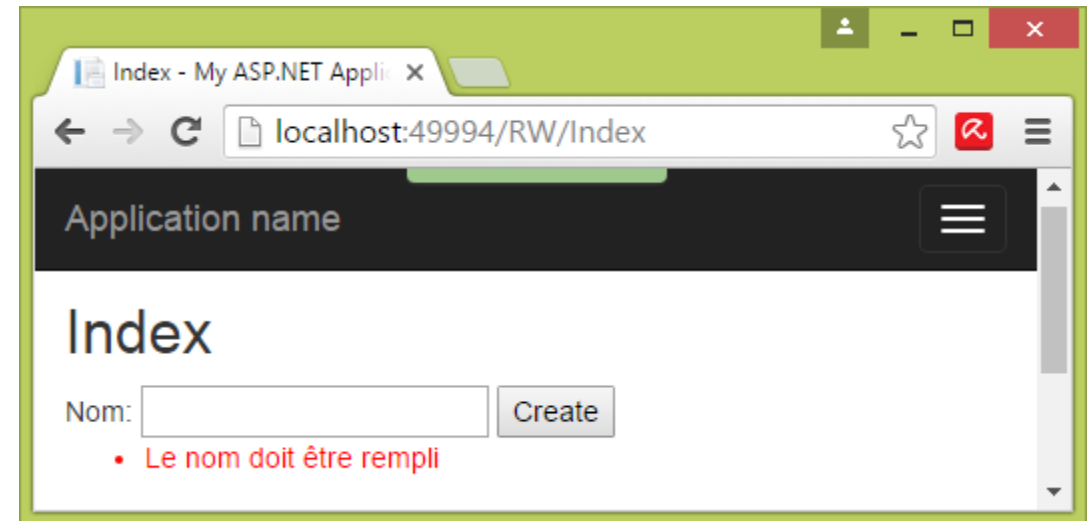
Ce helper permet d'afficher toutes les erreurs de validations du Model dans une liste.

```
<span style="Color:Red">  
    @Html.ValidationSummary();  
</span>
```

Les erreurs de validations sont récupérées du ModelState.

Si nous avons besoin d'une erreur autre que celle générées par le model, nous pouvons en ajouter dans notre c# :

```
ModelState.AddModelError("Nom",  
    "Le nom doit être rempli");
```



# Formulaires & Helpers

## 3. Html.TextBox and Html.TextArea

@Html.TextBox("Title", Model.Title)

➔ <input id="Title" name="Title" type="text" value="The Title" />

@Html.TextArea("text", "hello <br/> world")

➔ <textarea cols="20" id="text" name="text" rows="2">hello &lt;br /&gt; world</textarea>

## 4. Html.Label

@Html.Label("Texte")

➔ <label>Texte</label>

# Formulaires & Helpers

## 5. Html.DropDownList and Html.ListBox

```
@Html.DropDownList("DDColor", (IEnumerable<SelectListItem>)ViewBag.Couleurs)
```

```
<select id="DDColor" name="DDColor">  
  <option>Noir</option>  
  <option>Jaune</option>  
  <option>Rouge</option>  
</select>
```

```
@Html.ListBox("DDColor", (IEnumerable<SelectListItem>)ViewBag.Couleurs)
```

```
<select id="DDColor" multiple="multiple" name="DDColor">  
  <option>Noir</option>  
  <option>Jaune</option>  
  <option>Rouge</option>  
</select>
```

# Formulaires & Helpers

## 6. Html.ValidationMessage

Quand il existe une erreur dans le Model, nous pouvons utiliser cet helper afin d'afficher le message d'erreur associé.

```
@Html.ValidationMessage("Nom") @*Nom est la clé dans le dictionnaire errorState*@
```

```
<span class="field-validation-error" data-valmsg-for="Nom" data-valmsg-replace="true">
```

```
    Le Nom doit être rempli
```

```
</span>
```

# Formulaires & Helpers

## Les Helpers « strongly-typed »

Il existe une autre sorte d'helper qui permettent d'éviter les erreurs de frappes.

Avec les « Strongly-typed », nous utilisons une expression lambda afin de définir les propriétés à utiliser pour les inputs de notre formulaire.

Ce sont les helpers qui sont utilisés par le *scaffolding*.

Certains Helpers sont générés suivant les *DataAnnotation* vus plus loin dans ce cours

```
@model Models.Genre
@using (Html.BeginForm())
{
    @Html.ValidationSummary(excludePropertyErrors: true)
    <fieldset>
        <legend>Edit Album</legend>
        <p>
            @Html.LabelFor(m => m.GenreId)
            @Html.DropDownListFor(m => m.GenreId,
                ViewBag.Genres as SelectList)
        </p>
        <p>
            @Html.TextBoxFor(m => m.Title)
            @Html.ValidationMessageFor(m => m.Title)
        </p>
        <input type="submit" value="Save" />
    </fieldset>
}
```

# Custom Helpers

# Custom Helpers

Il est intéressant de pouvoir créer ses propres *html helper* permettant de ne pas devoir répéter les même tags razor et html...  
Nous pouvons les comparer aux *Users controls*.

Nous avons déjà vu un *html helper* nous permettant de construire un formulaire à partir d'un modèle, un autre permettre la validation et un dernier pour créer un lien.

(@Html.BeginForm(), @Html.ValidationSummary(), @Html.ActionLink )

Il existe deux type d'html helpers :

# Custom Helpers

## Création d'un *Inline Html Helper*

La façon la plus rapide de créer un helper est d'utiliser le tag @helper directement dans une vue.

```
@helper Createlist(string[] items)
{
    <ul>
        @foreach (string item in items)
        {
            <li>@item</li>
        }
    </ul>
}
<h4>InlineHelper</h4>
Days of the week: <p/>
@Createlist(ViewBag.Days)
<p />
Fruit I like: <p />
@Createlist(ViewBag.Fruits)
```

### InlineHelper

Days of the week:

- Lundi
- Mardi
- Mercredi
- Jeudi
- Vendredi
- Samedi
- Dimanche

Fruit I like:

- Pommes
- Poires
- Abricots

```
public ActionResult Index()
{
    string[] days = { "Lundi", "Mardi", "Mercredi",
                     "Jeudi", "Vendredi", "Samedi",
                     "Dimanche" };
    string[] Fruits = { "Pommes", "Poires", "Abricots" };
    ViewBag.Days = days;
    ViewBag.Fruits = Fruits;

    return View();
}
```



# Custom Helpers

## Création d'un *External Html Helper*

Le problème du type d'helper précédent est qu'il ne peut être utilisé que dans la vue où il est déclaré.

De plus, si le code augmente en complexité, cela peut devenir difficilement gérable.

L'alternative est donc de créer un helper externe en C#.

Il s'agit d'une méthode d'extension de l'objet *HtmlHelper*

```
this HtmlHelper html
```

```
public static class CustomHtmlHelpers
{
    public static MvcHtmlString List(this HtmlHelper html, string[] listItems)
    {
        TagBuilder tag = new TagBuilder("ul");
        foreach (string item in listItems)
        {
            TagBuilder itemTag = new TagBuilder("li");
            itemTag.SetInnerText(item);
            tag.InnerHtml += itemTag.ToString();
        }
        return new MvcHtmlString(tag.ToString());
    }
}
```

# Custom Helpers

Les paramètres qui suivent permettent de passer des valeurs de la vue vers l'*Helper*.

La façon la plus simple de créer de l'*HTML*, sans devoir s'inquiéter des caractères spéciaux et ceux d'échappement, est d'utiliser la classe *TagBuilder*.

```
TagBuilder tag = new TagBuilder("ul");
```

Membre	Description
InnerHtml	Permet de renvoyer directement de l'HTML contenu dans un string
SetInnerText(string)	Permet de spécifier le contenu texte d'un élément html. Le string en paramètre est automatiquement encodé pour permettre un affichage sur.
AddCssClass	Permet d'ajouter une classe css à un élément html
MergeAttribute(string, string, bool)	Ajout d'un attribut à un élément html. Le premier paramètre est le nom de l'attribut, le deuxième sa valeur, le dernier si l'attribut existant doit être remplacé

# Data Annotation et Validation

ASP.NET MVC

# Data Annotation et Validation

Les data annotations sont disponibles dans l'espace de nom : *System.ComponentModel.DataAnnotations*.

Ces attributs permettent la validation côté server et la validation côté client est supportée par le framework si les data annotations portent sur une propriété.

## 1. L'attribut Required

Il suffit de décorer les propriétés obligatoires afin que le système sache qu'elles doivent être remplies pour que le modèle soit valide.

[Required]

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

Si la propriété est NULL ou Empty, une erreur de validation est lancée.

Cela a pour effet de faire passer *ModelState.IsValid* à false. Ce qui nous permet de vérifier du côté serveur.

Et si le formulaire postant les données utilise l'helper *validationMessageFor*, un message d'erreur apparaît sur le formulaire et le formulaire n'est pas posté. (Sauf si javascript est désactivé)

# Data Annotation et Validation

## 2. StringLength

Permet de définir une taille maximale et/ou minimale pour une chaîne de caractère.

```
[StringLength(160, ErrorMessage = "Taille max: 160 et min :3", MinimumLength=10)]  
public string FirstName { get; set; }
```

## 3. RegularExpression

Permet de spécifier une expression régulière permettant de valider la valeur de la propriété.

```
[RegularExpression(@"[A-Za-z0-9._%+- ]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]  
public string Email { get; set; }
```

## 4. Range

Permet de définir les bornes pour une valeur numérique

```
[Range(7.0, 77.0, ErrorMessage = "Erreur, vous ne pouvez plus lire Tintin")]  
public int Age { get; set; }
```

# Data Annotation et Validation

## 5. Compare

Permet de spécifier que deux propriétés doivent être de valeurs égales

```
[RegularExpression(@"[A-Za-z0-9._%+- ]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]  
public string Email { get; set; }  
[Compare("Email")]  
public string EmailConfirm { get; set; }
```

## 6. Remote (System.Web.Mvc)

Permet de proposer une validation via un callback server (AJAX).

Pour cela, l'action du contrôleur permettant de valider la propriété reçoit en paramètre la valeur de l'input généré (veillez à respecter le nom de la propriété pour le nom du paramètre) et renvoie un JSON pour permettre la gestion du retour en javascript.

```
[Remote("CheckUserName", "Account")]  
public string UserName { get; set; }  
  
public JsonResult CheckUserName(string username)  
{  
    return Json(result, JsonRequestBehavior.AllowGet);  
}
```

# Custom Validation Logic

ASSP.NET MVC

# Custom Validation Logic

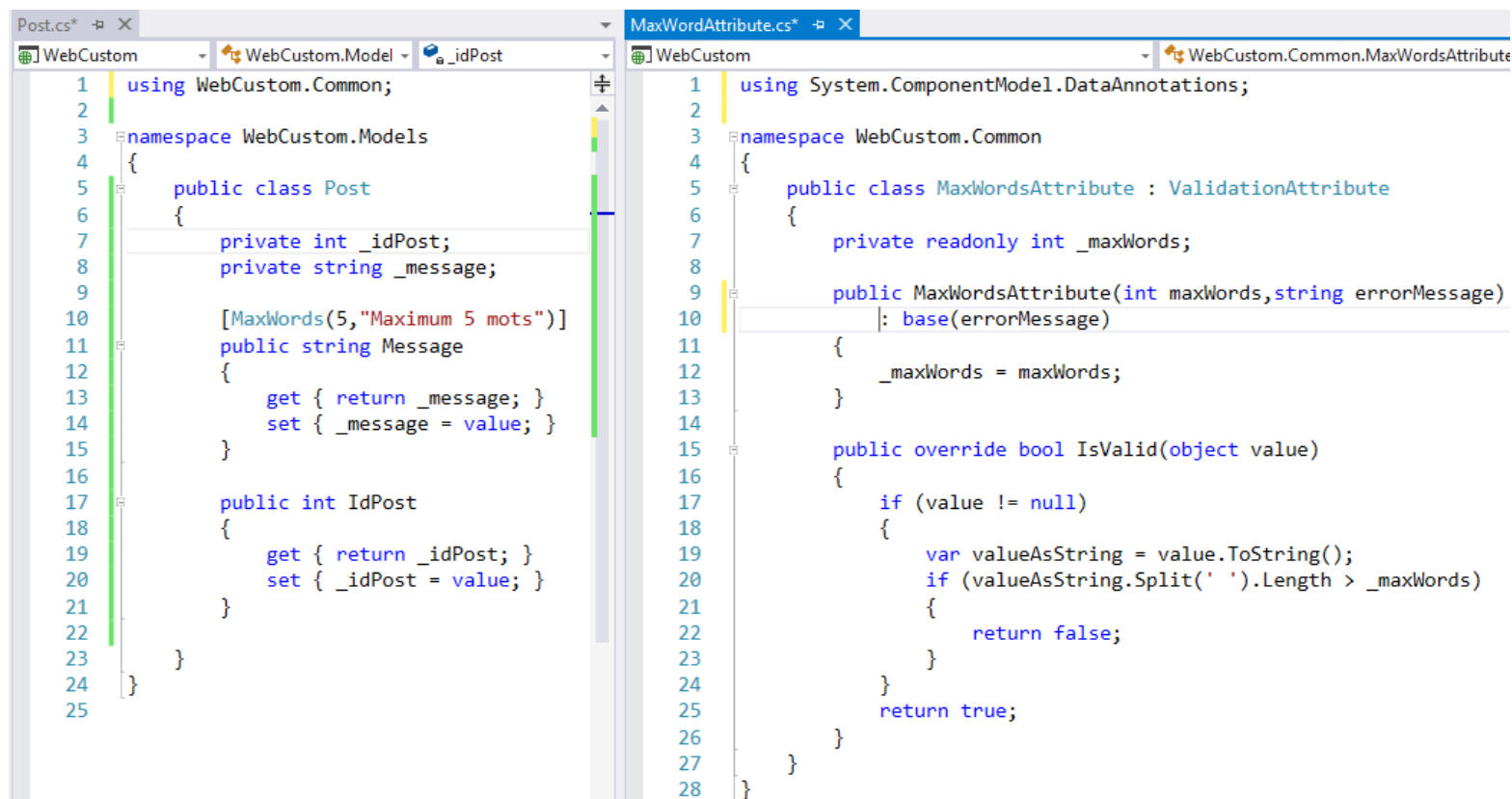
Si nous avons besoin d'ajouter des contraintes business, nous pouvons passer par les *Custom Attribute*

Toutes les annotations héritent de *ValidationAttribute*.

Nous pouvons donc utiliser cette classe abstraite pour étendre les annotations suivant nos besoins.

## Exemple:

Imaginons que nous désirons contrôler le nombre de mots maximum pour les post de notre blog.



```
1 using WebCustom.Common;
2
3 namespace WebCustom.Models
4 {
5     public class Post
6     {
7         private int _idPost;
8         private string _message;
9
10        [MaxWords(5, "Maximum 5 mots")]
11        public string Message
12        {
13            get { return _message; }
14            set { _message = value; }
15        }
16
17        public int IdPost
18        {
19            get { return _idPost; }
20            set { _idPost = value; }
21        }
22    }
23 }
24
25
```

```
1 using System.ComponentModel.DataAnnotations;
2
3 namespace WebCustom.Common
4 {
5     public class MaxWordsAttribute : ValidationAttribute
6     {
7         private readonly int _maxWords;
8
9         public MaxWordsAttribute(int maxWords, string errorMessage)
10            : base(errorMessage)
11        {
12            _maxWords = maxWords;
13        }
14
15        public override bool IsValid(object value)
16        {
17            if (value != null)
18            {
19                var valueAsString = value.ToString();
20                if (valueAsString.Split(' ').Length > _maxWords)
21                {
22                    return false;
23                }
24            }
25            return true;
26        }
27    }
28 }
```



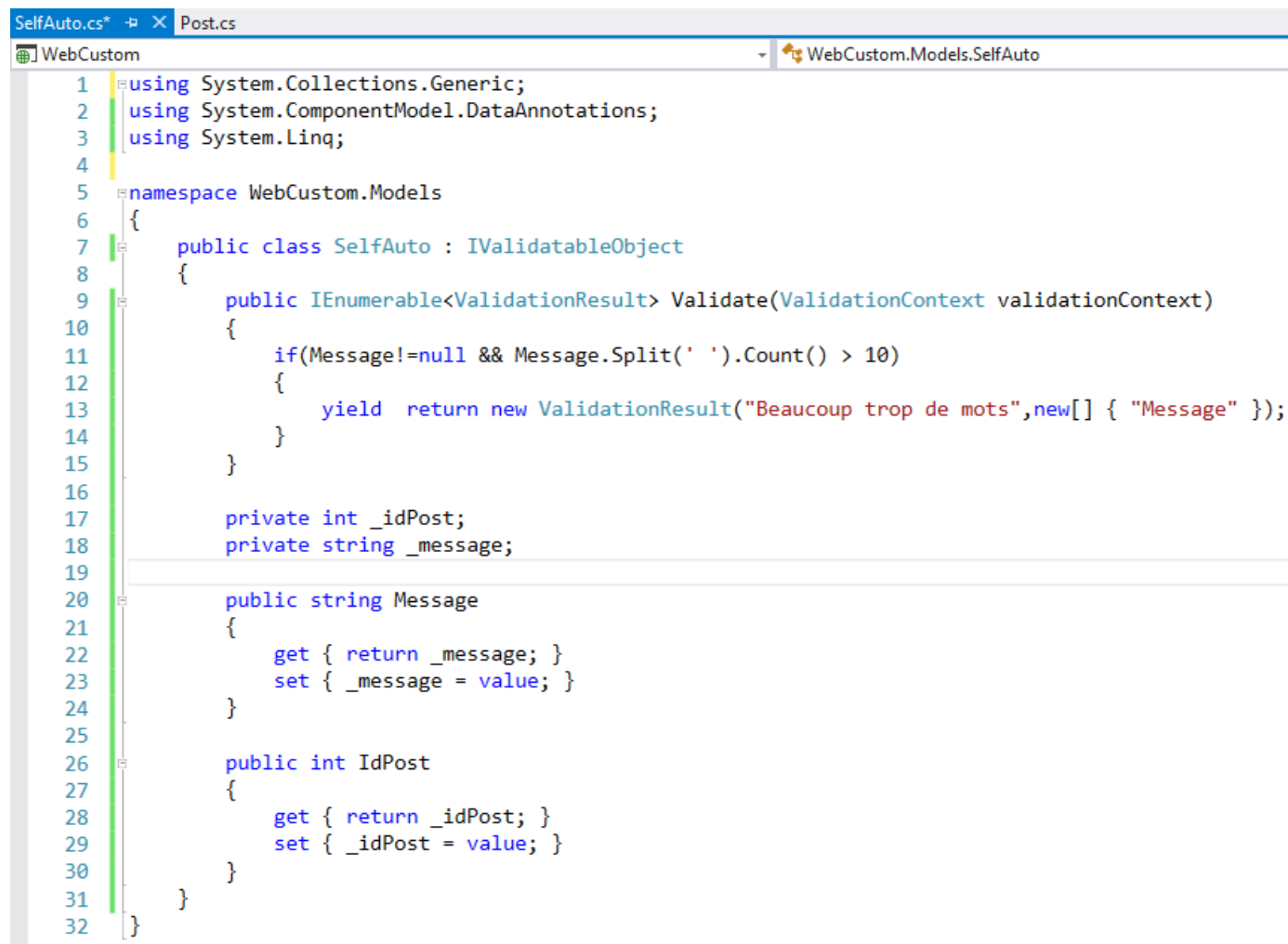
# Custom Validation Logic

Une autre possibilité est d'utiliser l'interface *IValidatableObject* afin de permettre à l'objet de se valider lui-même.

Prenons le même exemple :

En utilisant cette technique,

- nous ne devons plus utiliser *ModelState.IsValid* mais la fonction *Validate*.
- Le type de retour est une énumération et plus simplement un booléen
- Pas de paramétrage pour le message , ni pour la valeur possible



```
SelfAuto.cs* X Post.cs
WebCustom WebCustom.Models.SelfAuto

1 using System.Collections.Generic;
2 using System.ComponentModel.DataAnnotations;
3 using System.Linq;
4
5 namespace WebCustom.Models
6 {
7     public class SelfAuto : IValidatableObject
8     {
9         public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
10         {
11             if(Message!=null && Message.Split(' ').Count() > 10)
12             {
13                 yield return new ValidationResult("Beaucoup trop de mots",new[] { "Message" });
14             }
15         }
16
17         private int _idPost;
18         private string _message;
19
20         public string Message
21         {
22             get { return _message; }
23             set { _message = value; }
24         }
25
26         public int IdPost
27         {
28             get { return _idPost; }
29             set { _idPost = value; }
30         }
31     }
32 }
```

# Les annotations Display et Edit

ASP.NET MVC

# Les annotations Display et Edit

Via le scaffolding, nous gagnons un temps précieux pour la création de nos formulaires, listes, ...

Cependant, le moteur mvc utilise le nom des propriétés par défaut pour les label, nom de colonne, etc...

Ors, un nom de propriété ne peut pas avoir d'espace, d'accent, etc...

Nous pouvons remédier à ce problème via les attributs d'affichage :

## 1. L'attribut Display

Est utilisé pour permettre de donner un nom plus *user-friendly* mais aussi définir l'ordre d'apparition, si il s'agit d'une valeur auto-générée, une description pour les tooltip, ...

```
private DateTime _datePost;

[Display(Name = "Date de publication", Description = "Date de publication du Post")]

public DateTime DatePost
{
    get { return _datePost; }
    set { _datePost = value; }
}
```

# Les annotations Display et Edit

## 2. ScaffoldColumn

Cet attribut nous permet de spécifier qu'une colonne ne doit pas être prise en compte pour la génération automatique.

```
[ScaffoldColumn(false)]  
public int IdPost  
{  
    get { return _idPost; }  
    set { _idPost = value; }  
}
```

## 3. DisplayFormat

Permet de contrôler le format d'affichage mais également de savoir si l'input doit afficher le format en mode edit ou encore un texte alternatif en cas de null...

```
[DisplayFormat(ApplyFormatInEditMode =true, DataFormatString ="{0:c}", NullDisplayText ="Donnez votre eval")]  
public float Eval  
{  
    get { return _evaluation; }  
    set { _evaluation = value; }  
}
```

# Les annotations Display et Edit

## 4. ReadOnly

Permet de ne pas tenir compte de la valeur renvoyée par le formulaire. La propriété ne peut pas être modifiée.

## 5. DataType

Permet de s'assurer que le bon type de donnée sera prise lors du scaffolding et dans l'utilisation de l'helper *EditorFor<>* qui déduit le type d'input par rapport au type de donnée.

```
public string Password
{
    get { return _password; }
    set { _password = value; }
}
```

Un simple champs texte sera généré

```
[DataType(DataType.Password)]
public string Password
{
    get { return _password; }
    set { _password = value; }
}
```

Un champs html5 password sera généré

# Les annotations Display et Edit

## 6. HiddenInput (System.Web.Mvc)

Permet de générer un input type hidden

[[HiddenInput](#)]

```
public string Login
{
    get { return _login; }
    set { _login = value; }
}
```

# Sessions et Application

ASP.NET MVC

# Sessions et Application

Il existe des événements particuliers en ASP.NET permettant de détecter les lancement et les arrêt des sessions et de l'application.

Ces événements se trouvent dans le fichier Global.asax

## 1. Application

<u>Événement</u>	<u>Description</u>
Application_Start	Exécuté après le premier appel à une page du site depuis le démarrage de IIS
Application_End	Appelé lorsque l'application se termine, cela ne signifie pas que IIS s'arrête mais est d'office appelé si, pour une raison quelconque IIS est arrêté

## 2. Session

<u>Événement</u>	<u>Description</u>
Session_Start	Appelé lors de chaque nouvelle session d'un navigateur client
Session_End	Fin de session soit explicite (Session.Abandon()) ou suite à un time out

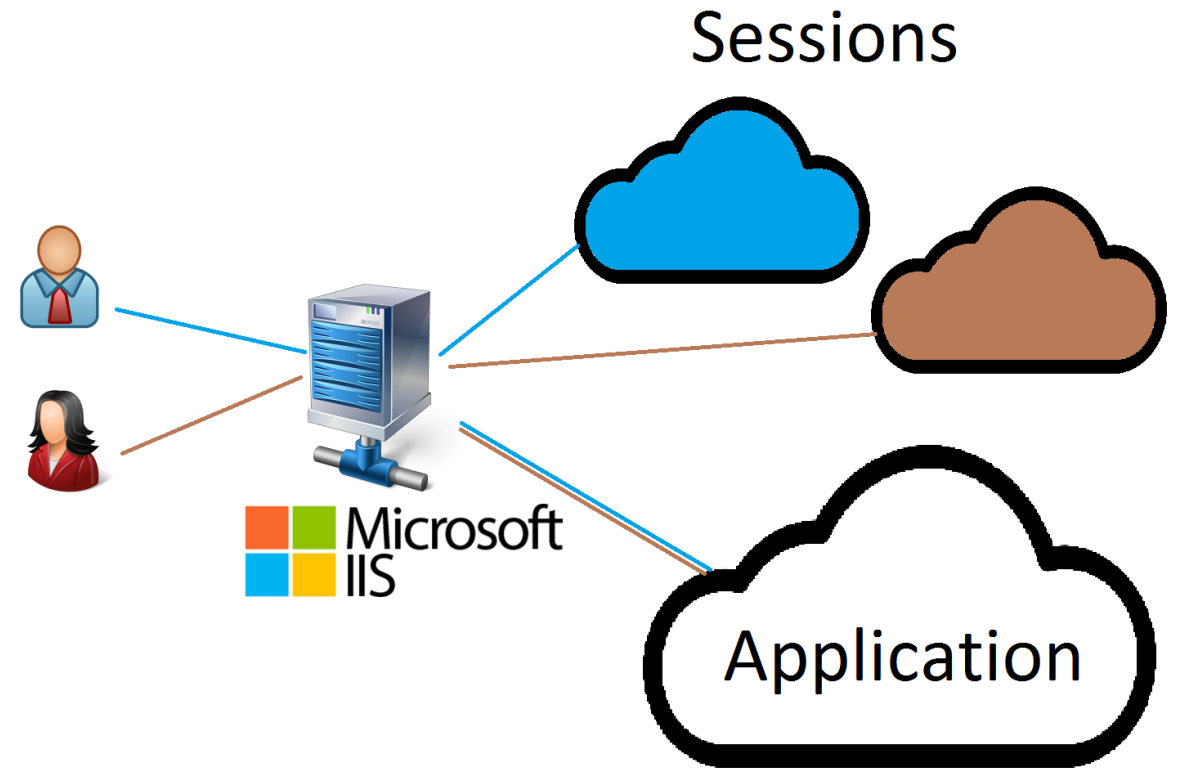


# Sessions et Application

Soit deux utilisateurs qui envoient une requête vers le serveur IIS.

Il y aura :

- **un seul objet "Application" commun** à tous les utilisateurs du site
- **deux objets "Session"** correspondant chacun à un utilisateur précis.

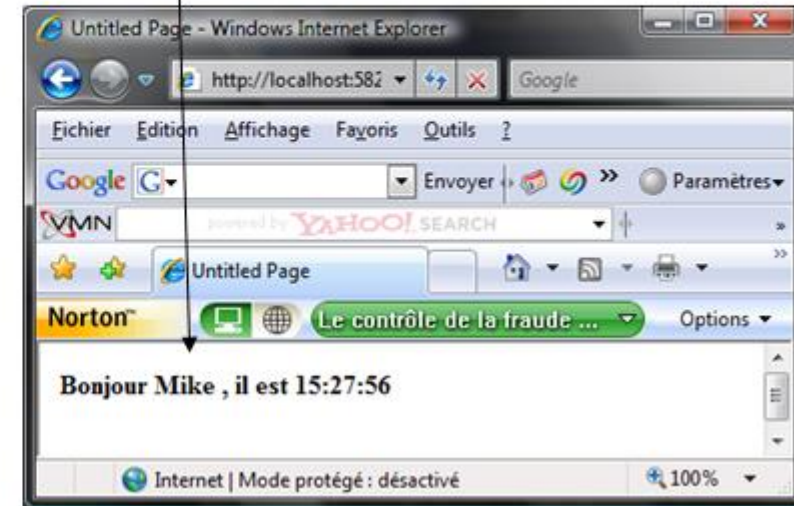
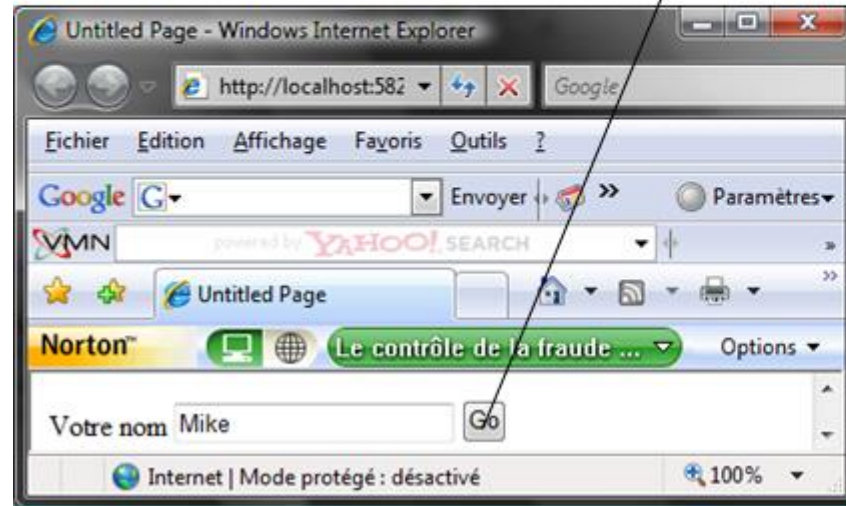


# Sessions et Application

## Pourquoi des Session ?

Les session ASP.NET vous permet de stocker et de récupérer des valeurs pour un utilisateur à mesure que ce dernier navigue dans les différentes pages ASP.NET qui composent une application Web. HTTP est un protocole sans état, ce qui signifie que votre serveur Web traite chaque demande de page HTTP comme une demande indépendante

Session



# Sessions et Application

- Comment utiliser les session ?

La session ASP.NET est activé par défaut pour toutes les applications ASP.NET. Les variables d'état de session ASP.NET sont définies et récupérées facilement à l'aide de la propriété Session qui stocke les valeurs de variables de session comme une collection indexée par nom.

Par exemple, l'exemple de code suivant crée les variables de session **FirstName** et **LastName** pour représenter le prénom et le nom d'un utilisateur et les définit sur les valeurs récupérées des contrôles TextBox.

```
Session["FirstName"] = FirstNameTextBox.Text  
Session["LastName" ] = LastNameTextBox.Text
```

ASP.NET stocke par défaut les informations sur la session dans l'espace mémoire de l'application ASP.NET. Vous pouvez stocker éventuellement les informations sur la session à l'aide d'un service autonome afin qu'elles soient conservées, en cas de redémarrage de l'application ASP.NET, dans un serveur SQL Server

# Sessions et Application

- **Comment changer l'endroit de stockage des sessions ?**

Vous pouvez spécifier le mode que l'état de session ASP.NET doit utiliser en assignant une valeur d'énumération **SessionStateMode** à l'attribut **mode** de l'élément `sessionState` dans le fichier **Web.config** de votre application.

➔ **Mode InProc**

Il s'agit du mode par défaut : les variables de sessions sont stockées en mémoire sur le serveur web

➔ **Mode Sql server**

- Le mode **SQLServer** stocke l'état de session dans une base de données SQL Server.
- L'utilisation de ce mode garantit que l'état de session est conservé en cas de redémarrage de l'application Web et qu'il est disponible pour plusieurs serveurs Web dans une batterie de serveurs Web.

```
<configuration>
  <system.web>
    <sessionState mode="SQLServer" sqlConnectionString="Integrated Security=SSPI;data source=SampleSqlServer;" />
  </system.web>
</configuration>
```

# Sessions et Application

Pour utiliser le mode SQLServer, vous devez d'abord être sûr que la base de données d'état de session ASP.NET est installée sur SQL Server.

**Remarque :**

Vous pouvez installer la base de données d'état de session ASP.NET à l'aide de l'outil Aspnet\_regsql.exe

Pour configurer une application ASP.NET afin qu'elle utilise le mode SQLServer, procédez comme suit dans le fichier Web.config de l'application :

- Définissez l'attribut mode de l'élément **sessionState** avec la valeur SQLServer.
- Définissez l'attribut sqlConnectionString avec une chaîne de connexion pour votre base de données SQL Server.

Plus d'info : <https://support.microsoft.com/en-us/help/317604/how-to-configure-sql-server-to-store-asp-net-session-state>

# Sessions et Application

- Installation de la base de données d'état de session à l'aide de l'outil Aspnet\_regsql.exe

Pour installer la base de données d'état de session sur SQL Server, exécutez l'outil

Aspnet\_regsql.exe présent dans le dossier **systemroot\Microsoft.NET\Framework\numéroversion** de votre serveur Web.

Fournissez les informations suivantes avec la commande :

- Nom de l'instance du serveur SQL Server à l'aide de l'option -S.
- Informations d'identification de connexion pour un compte ayant l'autorisation de créer une base de données sur SQL Server. Utilisez l'option -E pour utiliser l'utilisateur actuellement connecté, ou l'option -U pour spécifier un ID utilisateur avec l'option -P permettant de spécifier un mot de passe.
- Option de ligne de commande -ssadd pour ajouter la base de données d'état de session.
- Par défaut, vous ne pouvez pas utiliser l'outil Aspnet\_regsql.exe pour installer la base de données d'état de session sur SQL Server Express. Pour exécuter l'outil Aspnet\_regsql.exe et installer une base de données SQL Server Express, vous devez d'abord activer l'option SQL Server Agent XPs à l'aide des commandes T-SQL, comme illustré ci-après :

# Sessions et Application

- **Bonne pratique pour l'utilisation des sessions**

Pour placer un objet dans une Session d'un utilisateur, nous procédons ainsi :

```
Session["NomUtilisateur"] = "Richard"
```

- Il y a plusieurs inconvénients :
  - Si la variable NomUtilisateur est utilisé dans plusieurs pages, il faut bien savoir qu'elle s'appelle NomUtilisateur, il ne faut pas faire de fautes de frappes et surtout être sur que cette variable n'existe pas déjà dans une autre page avec une utilisation complètement différente. De plus, si vous voulez changer le nom de la variable en LeNomDelUtilisateur, vous devez revoir tout votre code de toutes vos pages,
  - La variable est non typé (c'est un Object dans Session) donc quand vous voulez y accéder, vous êtes obligés d'effectuer des opérations de casting :

```
String sM = (String)Session["NomUtilisateur"]
```

**Qui dit casting dit possibilité d'erreur de casting.**

# ASP.NET: les sessions – Bonne pratique

Donc, l'idée est de créer une classe tampon qui accédera aux variables sessions

```
public static class UserSession
{
    public static string NomUtilisateur
    {
        get{return HttpContext.Current.Session["NomUtilisateur"];}
        set{HttpContext.Current.Session["NomUtilisateur"] = value;}
    }
}
```

Dans nos pages nous pouvons désormais écrire :

```
String sM = UserSession.NomUtilisateur
```

**Pas de casting, un seul endroit pour gérer le nom des variables sessions, etc.**



# AuthorizeAttribute : Attributs personnalisés

Maintenant que nous savons mettre en place une session, il serait bien de protéger les accès à certaines pages tant que l'utilisateur n'est pas connecté à une session valide!

Grâce à la class `AuthorizeAttribute`, nous aurons la possibilité de redéfinir via un Attribut quelles méthodes, voir controllers nous limiterons les actions utilisateurs.

C'est plus précisément deux méthodes de cet attribut qu'il nous faudra redéfinir par un override de celle-ci:

- **AuthorizeCore** →

Nous permet d'établir une vérification et ainsi définir si nous sommes autorisé ou non à accéder au bloc de code soumis à l'attribut; Il nous retourne donc un booléen, mais demande un paramètre `HttpContextBase` (ce qui tombe bien, notre controller en contient justement un en propriété).

- **HandleUnauthorizedRequest** → Définit une action en cas de refus. Ne retourne rien, mais demande un paramètre `AuthorizationContext`.

# AuthorizeAttribute : Attributs personnalisés

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
// nous permet de définir que notre attribut pourra affecter des classes et des méthodes

public class AuthRequiredAttribute : AuthorizeAttribute
{
    protected override bool AuthorizeCore(HttpContextBase httpContext)
        //nous vérifions si le UserSession est différent de null
    {
        return UserSession.CurrentUser != null;
    }

    protected override void HandleUnauthorizedRequest(AuthorizationContext filterContext)
        //Redirige vers une autre action si non-autorisé
    {
        filterContext.Result = new RedirectToRouteResult(new RouteValueDictionary(new { Controller = "Auth", Action = "Login" }));
    }
}
```

# Areas

ASP.NET MVC

# Areas

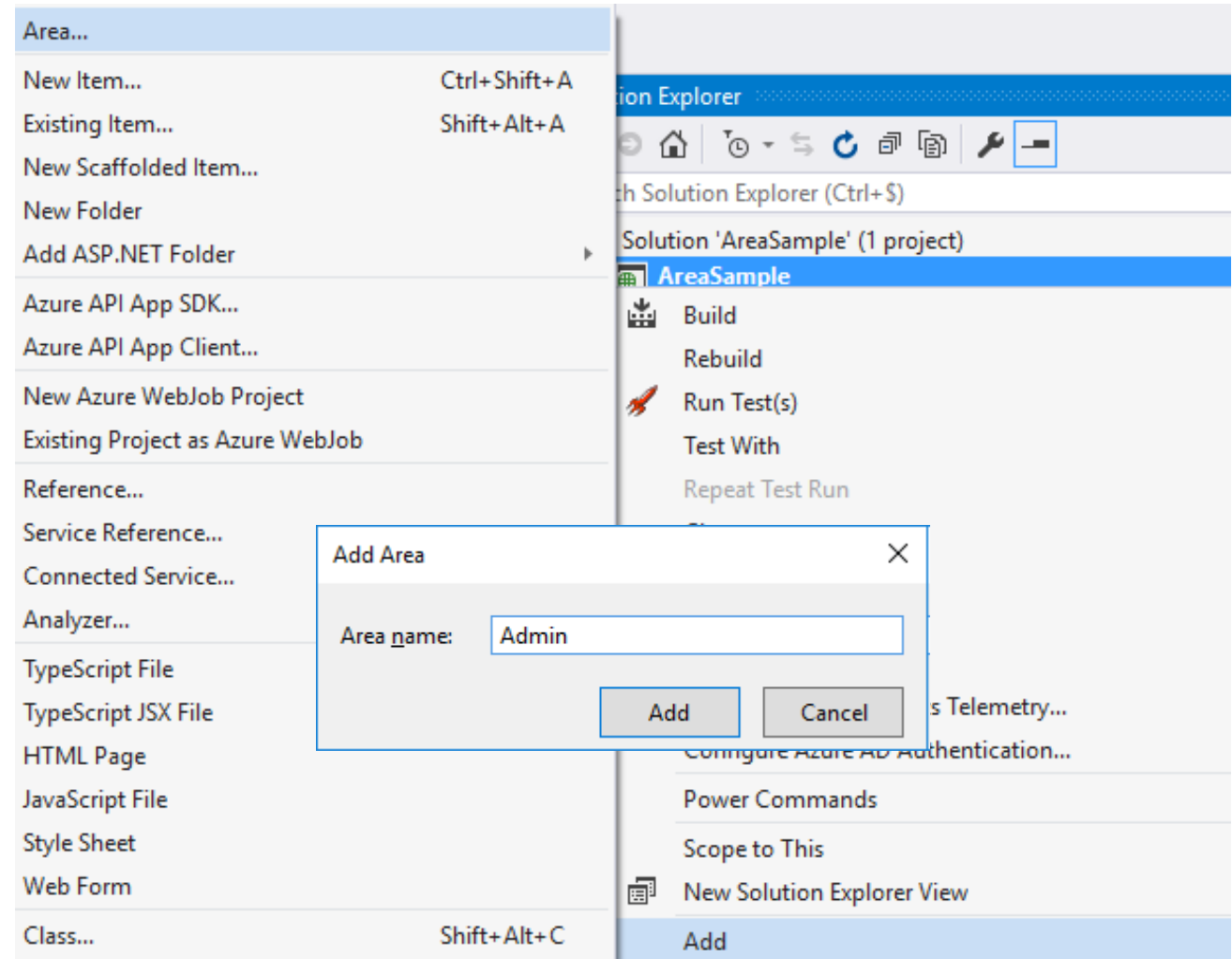
Les zones (Area) ont été introduites en ASP.NET MVC 2 et permettent de séparer nos Modèles, Contrôleurs et vues en « section » fonctionnelles.

Par exemple, pour une zone d'administration dans notre application Web.

La séparation « fonctionnelle » nous permet de faciliter la maintenance de nos applications.

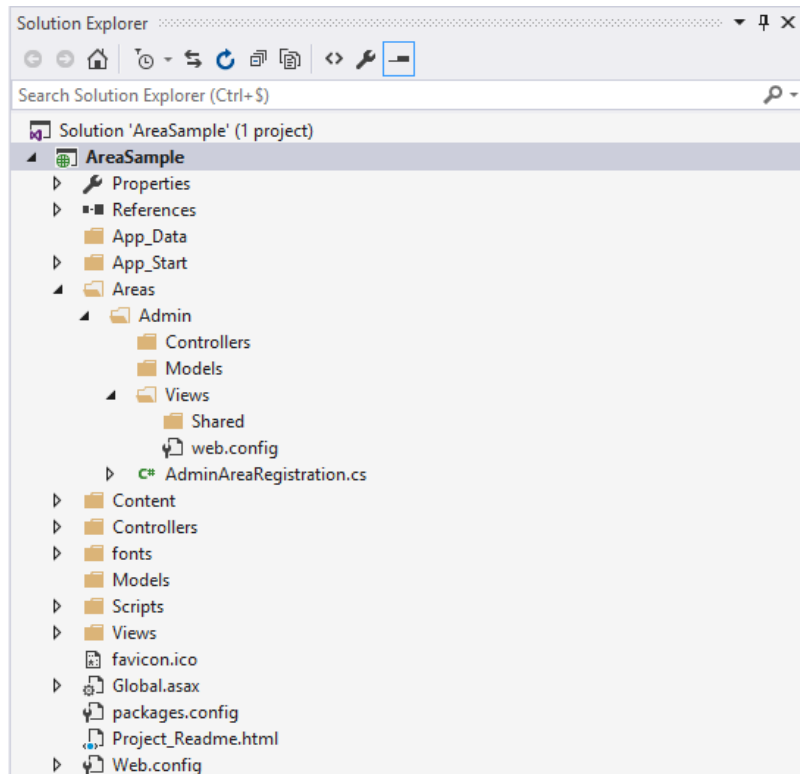
L'enregistrement d'une zone se fait en plusieurs étapes.

La première consistant à ajouter cette zone à notre projet via un Click droit → Add Item → Area sur notre projet.



# Areas

Nous obtenons une structure MVC dans un sous-dossier portant le nom de notre area.



Un fichier *nomAreaAreaRegistration.cs* est également créé et contient la définition des routes pour l'area.

```
using System.Web.Mvc;

namespace AreaSample.Areas.Admin
{
    public class AdminAreaRegistration : AreaRegistration
    {
        public override string AreaName
        {
            get
            {
                return "Admin";
            }
        }

        public override void RegisterArea(AreaRegistrationContext context)
        {
            context.MapRoute(
                "Admin_default",
                "Admin/{controller}/{action}/{id}",
                new { action = "Index", id = UrlParameter.Optional }
            );
        }
    }
}
```

# Areas

## Conflits avec les routes

Si nous avons deux contrôleurs portant le même nom dans l'area et le site principal, le moteur MVC ne pourra pas les distinguer et nous obtiendrons le message d'erreur ci-après.

Afin de remédier à ce problème, nous devons éditer le fichier *RouteConfig.cs* se trouvant dans le dossier *app\_start* et ajouter le namespace du contrôleur par défaut que nous désirons utiliser en cas de conflit.



# Area

The diagram illustrates the relationship between two namespaces in an ASP.NET MVC application. On the left, the `AreaSample.Controllers` namespace contains three controller classes: `HomeController`, `About()`, and `Contact()`. On the right, the `AreaSample` namespace contains a `RouteConfig` class with a `RegisterRoutes` method. This method configures routes, including a default route that maps to the `HomeController` in the `AreaSample.Controllers` namespace. Red boxes highlight the `AreaSample.Controllers` namespace and the `"AreaSample.Controllers"` string in the route configuration, with lines connecting them to show the reference.

```
using System.Web.Mvc;

namespace AreaSample.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult About()
        {
            ViewBag.Message = "Your application description page.";
            return View();
        }

        public ActionResult Contact()
        {
        }
    }
}

using System.Web.Routing;

namespace AreaSample
{
    public class RouteConfig
    {
        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 },
                namespaces: new string[] { "AreaSample.Controllers" }
            );
        }
    }
}
```

# Area

Si vous utilisez l'attribut route, vous devez ajouter l'attribut *RouteArea*.

Cet attribut contiendra le nom de l'area et donc pas besoin de le spécifier au sein de l'attribut route.

Il est également possible de changer le préfix. En effet par défaut si l'area s'appelle admin, la route commencera par *admin/....* Si vous désirez changer le préfixe, utilisez la propriété *AreaPrefix* de l'attribut.

```
namespace AreaSample.Areas.Admin.Controllers

[RouteArea("Admin", AreaPrefix = "Manage")]
public class HomeController : Controller
{
    // GET: Admin/Home

    [Route("")]
    [Route("home")]
    [Route("home/index")]
    public ActionResult Index()
    {
        return View();
    }
}
```



# Security

ASP.NET MVC

# MVC Input et Output

Security – Asp.Net MVC

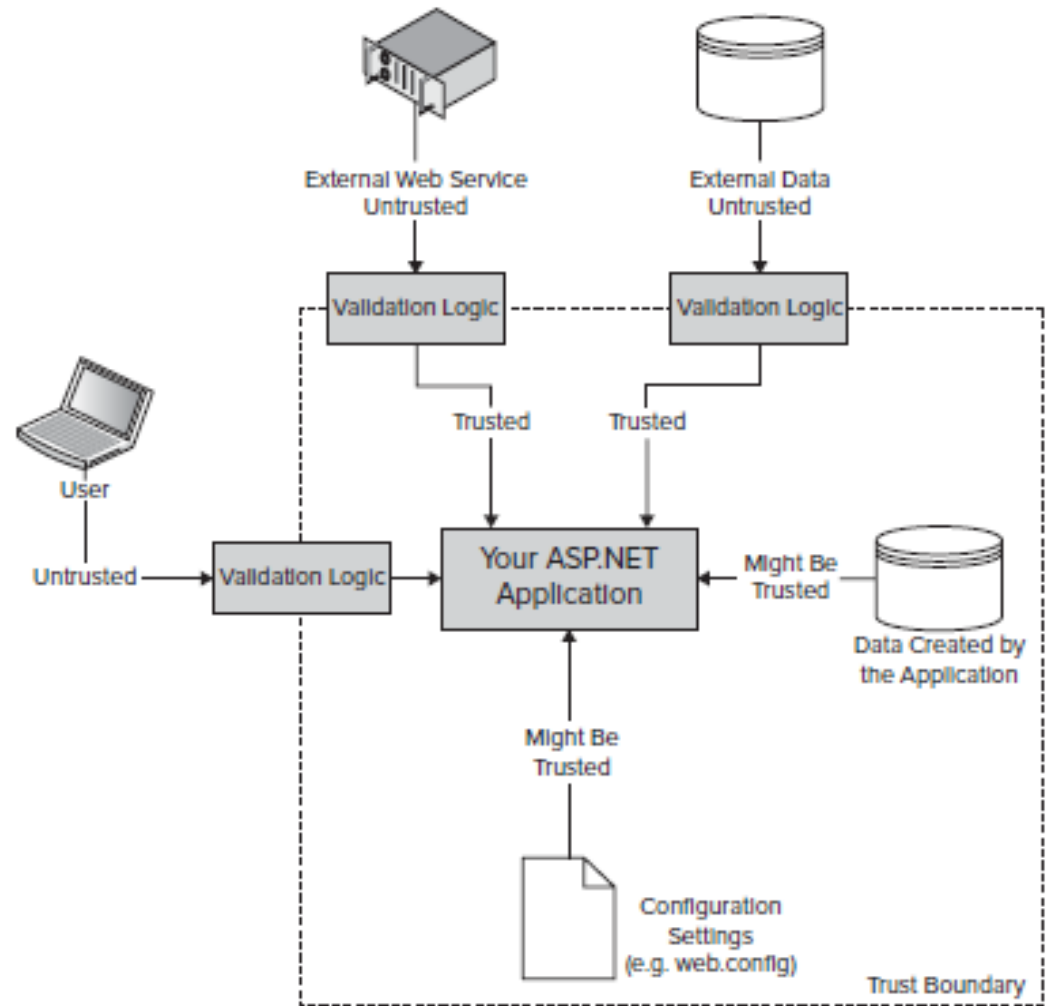
# MVC Input et Output

## Comment se protéger des attaques XSS (Cross Site Scripting)?

Tous les formulaires sont potentiellement vulnérables aux attaques XSS.

Ces attaques consistent à injecter du javascript via les inputs (formulaires, services , etc...) afin de rediriger les utilisateurs, changer le contenu de la page ou encore voler les infos stockées en session.

Le principe de base est de valider toutes les entrées dans notre zone de confiance



# MVC Input et Output

Asp.Net MVC protège déjà certaines injections.

## Not Vulnerable to XSS

Votre commentaire :

Valider



# MVC Input et Output

Cependant dans certains cas, nous avons besoin d'autoriser les tags HTML....

Nous pouvons dans ce cas, utiliser l'attribut [`ValidateInput(false)`] sur l'action du controller qui devra accepter les tags...

En faisant cela, nous acceptons donc directement le js ou l'html,... en input pour notre action...

Lorsque je veux ensuite afficher la donnée, ASP MVC encode le résultat en html grâce à l'@.

Si j'ai cependant besoin d'afficher avec formatage html, je devrais utiliser `@Html.Raw(...)`

The image shows a Visual Studio environment with a C# controller and a Razor view. The controller has two actions: `Index()` and `VulnerableXSS()`. The `VulnerableXSS()` action is annotated with `[HttpPost]` and `[ValidateInput(false)]`. The `VulnerableXSS(string Comment)` action sets `ViewBag.Comment = Comment;` and returns `View()`. The view contains the following HTML:

```
<h2>Vulnerable to XSS</h2>
<form method="post">
  <b><u>Votre commentaire :</u></b><br />
  @Html.EditorFor(M=>M.Comment)
  <input type="submit" class="btn btn-info"/>
  @Html.Raw(ViewBag.Comment)
</form>
```

Annotations:

- 1: Points from the `VulnerableXSS()` action to the `<h2>Vulnerable to XSS</h2>` tag in the view.
- 2: Points from the `[ValidateInput(false)]` attribute to the `<input type="submit" class="btn btn-info"/>` tag in the view.
- 3: Points from the `ViewBag.Comment = Comment;` line in the controller to the `@Html.Raw(ViewBag.Comment)` line in the view.
- 4: Points from the `@Html.Raw(ViewBag.Comment)` line in the view to the Chrome error message.

Chrome error message:

Cette page ne fonctionne pas

Chrome a détecté un code inhabituel sur cette page et a bloqué cette dernière pour protéger vos informations personnelles (mots de passe, numéros de téléphone et de cartes de paiement).

Essayez de [consulter la page d'accueil du site.](#)

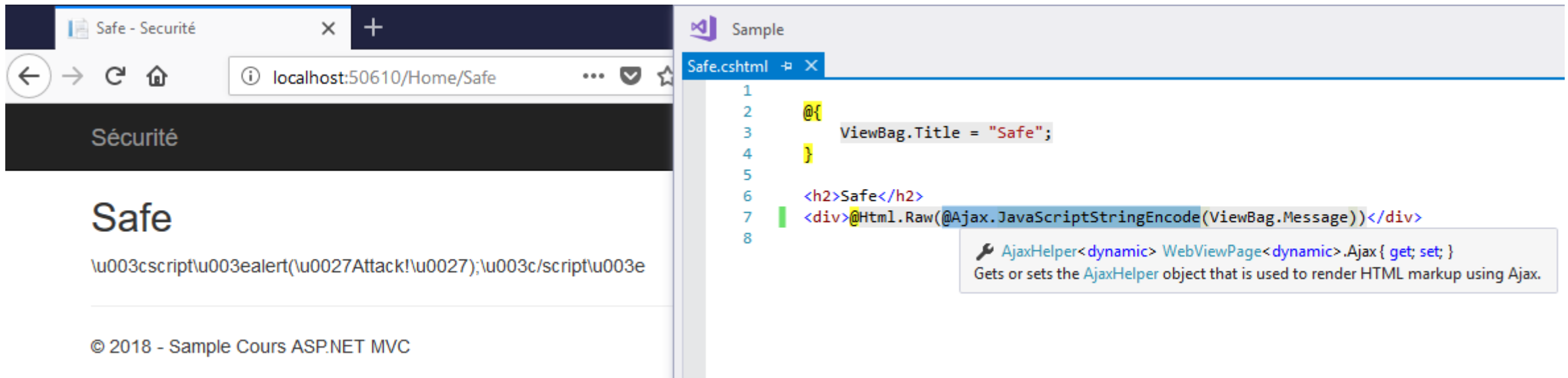
ERR\_BLOCKED\_BY\_XSS\_AUDITOR

# MVC Input et Output

Si nous voulons que seul une propriété permette l'encodage : [AllowHtml]

Nous devons également prêter attention aux paramètres de l'url, aux paramètres renvoyés par un services, une DB,...

Si l'helper `Html.Raw` peut nous éviter des surprise, nous pouvons également utiliser `@Ajax.JavaScriptStringEncode`



The screenshot shows a web browser window on the left and a Visual Studio code editor on the right. The browser window displays a security warning in French: "Sécurité" (Security) and "Safe". Below the warning, the text "Safe" is visible, followed by a JavaScript alert message: `\u003cscript\u003ealert(\u0027Attack!\u0027);\u003c/script\u003e`. The Visual Studio code editor shows a file named `Safe.cshtml` with the following code:

```
1
2  @{
3      ViewBag.Title = "Safe";
4  }
5
6  <h2>Safe</h2>
7  <div>@Html.Raw(@Ajax.JavaScriptStringEncode(ViewBag.Message))</div>
8
```

A tooltip for the `AjaxHelper` property is visible, stating: "Gets or sets the `AjaxHelper` object that is used to render HTML markup using Ajax."

# MVC Input et Output

Microsoft propose également une librairie : *AntiXSS library*.

Vous trouverez de la lecture supplémentaire sur : <http://go.microsoft.com/fwlink/?LinkID=293690&clcid=0x409>

Et pour utiliser la librairie, un simple *using* suffit :

```
@using Microsoft.Security.Application
```

```
<div class="messages">@Encoder.JavaScriptEncode(ViewBag.Msg)</div>
```

# MVC Input et Output

Une autre type d'attaque : Cross-Site Request Forgery (CSRF)

Le principe de cette attaque est de détourner une requête afin de se faire passer par le client ayant initié le dialogue avec le serveur.

## **Exemple :**

Vous recevez par mail un lien permettant de , par exemple, voir les résultat d'un concours que vous avez peut-être gagné.

`<a href=« http://www.lesite.be/Concours/?id=100&cmd=.....>View Result</a>`

Si vous cliquez sur le lien, l'action de la vue va donc s'exécuter et comme vous êtes un user authentifié, le pirate peut accéder au système via votre compte.



# MVC Input et Output

Pour prévenir ce genre d'attaque :

- Assurez vous qu'il n'est pas possible d'utiliser plusieurs fois le lien.
- La méthode GET ne doit permettre que la récupération de données et non la modification
- Assurez-vous que la requête ne peut pas être rejouée si par exemple un pirate utilise Javascript pour simuler un POST

L'html helper `@Html.AntiForgeryToken()` permet d'éviter le replay en fournissant un token unique pour vos requêtes.

Cette fonction utilise un Hidden input mais également un cookie afin de vérifier la provenance et la légitimité de la requête

## View

```
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken();
    @Html.EditorForModel();
    <input type="submit" value="Submit" />
}
```

## Action

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Crsf(Person P)
{
    return View();
}
```

# Testing

Asp.net MVC

# Testing

Le processus de création d'une application se focalise principalement sur les fonctionnalités qui doivent être développées afin de répondre à une demande, de résoudre un problème spécifique,...

Mais avant de mettre en production ladite application, il est impératif de s'assurer que ces fonctionnalités soient implémentées de manière fiable et robuste.

- **Les Tests Manuels**

C'est la manière la plus facile de tester une application et la plus littérale : On lance l'application et on tente d'utiliser l'application comme un utilisateur « normal » le ferait.

C'est pourtant la manière la moins robuste de tester et celle-ci a son lot de points négatifs.

- Basé sur le jugement humain, ces tests sont subjectifs. Comment faire la différence entre le string 1 et l'int 1 sur l'écran?
- L'ordinateur est plus efficace et plus précis. Le test manuel se fait au travers de l'UI et doit, pour pouvoir être intéressant, respecter des étapes dans un ordre précis et non pas comme le testeur pense que ça doit fonctionner
- Les tests manuels prennent du temps

# Testing

- **Les Test automatisés**

Et si nous demandions à un logiciel de tester une logiciel?

Il « suffit » d'écrire les tests de définir les résultats souhaités et laisser l'exécution des tests nous révéler les échecs et les succès.

# Testing

Il existe trois catégories de tests permettant d'identifier les bugs :

- Unit Tests:

Ils permettent de tester les fonctionnalités basiques. En mettant en place une série de tests unitaires, nous pouvons nous assurer que les aspects fonctionnels de notre application sont correctement codés

Exemple : Vérifier le type de retour d'une fonction

- Integration Tests :

Ils permettent de vérifier que 1 ou 2 composants fonctionnent correctement ensemble. Ils peuvent par exemple être utilisés pour vérifier que deux classes interagissent correctement ensemble.

Ils sont également utilisés afin de vérifier que toute l'application, incluant la base de données et les services externes, fonctionne et délivre un résultat attendu

- Acceptance Tests :

Ils testent les fonctionnalités et les composants en ciblant les besoins énoncés par le « client »

# Testing

## Unit Test

Un test unitaire est généralement composé de trois phases :

### 1. Arrange

Nous créons les instances, variables, ...nécessaires pour le test. Nous assignons également des valeurs aux propriétés nécessaires.

Les tests unitaires doivent être conçu afin de permettre l'automatisation de ceux-ci et donc de pouvoir les exécuter rapidement et facilement durant le cycle de vie de l'application.

C'est très important car des bugs peuvent apparaitre durant le développement et les tests unitaires doivent nous assurer de la robustesse de notre code.

### 2. Act

Nous appelons la fonctionnalité que nous désirons tester et nous stockons les résultats dans des variables afin de pouvoir vérifier celles-ci.

### 3. Assert

Nous vérifions les résultats attendus. Si la vérification passe : le test réussi. Sinon, celui-ci échoue.

# Testing

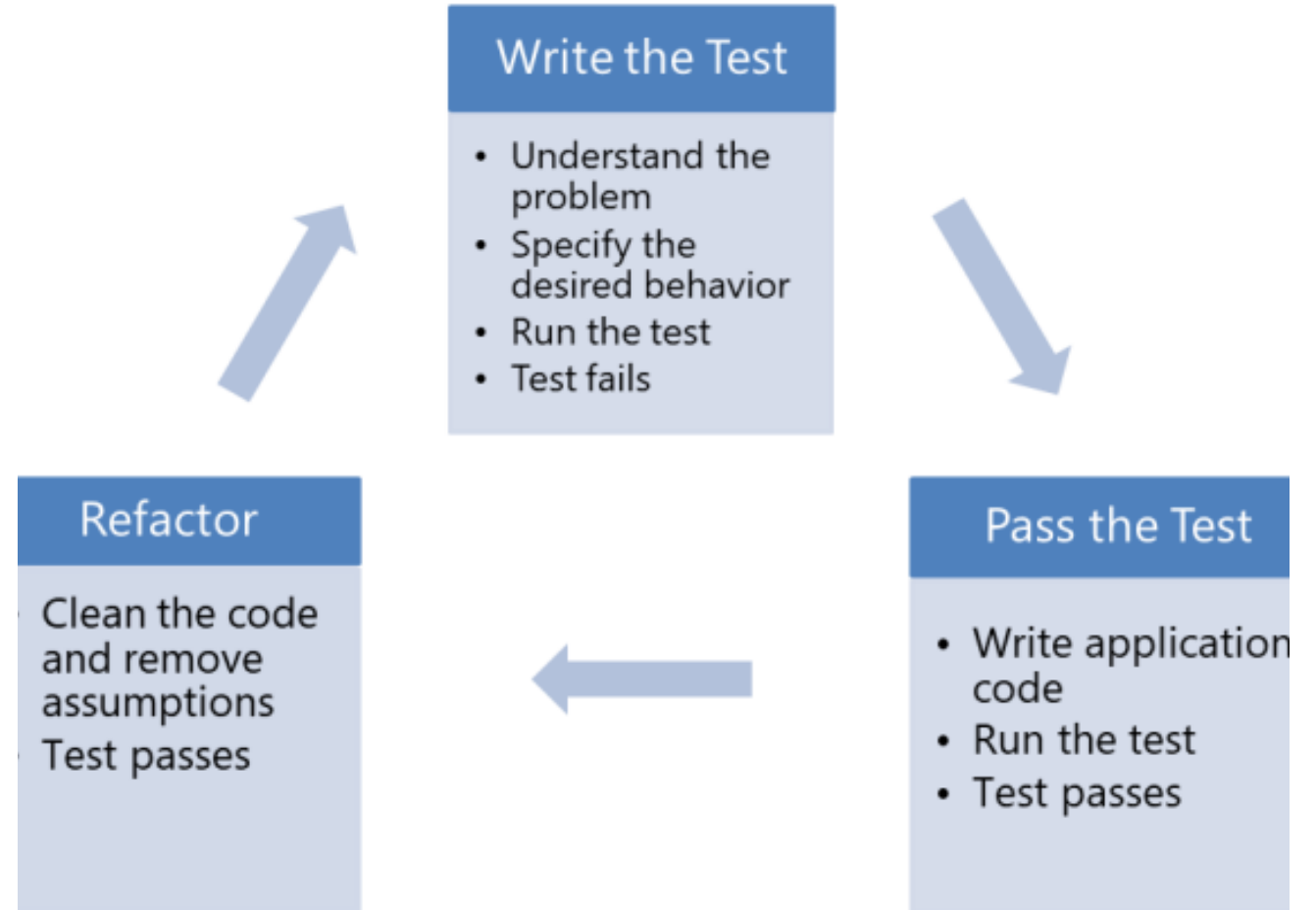
## Test Driven

Il est de plus en plus fréquent d'utiliser les tests unitaires dans la méthodologie de développement tel la méthodologie Agile, itérative ou Waterfall.

Mais une méthodologie spécifique est liée aux tests unitaires : Test Driven (TDD).

Les principes :

- Écrire le code avant les tests
- Avancer fonctionnalité par fonctionnalité
- Ecrire uniquement le code nécessaire pour passer les tests



# Composants MVC et Unit Testing

Testing



# Composants MVC et Unit Testing

Afin de permettre les test unitaires, il est nécessaire de tendre vers des composants évitant l'interdépendance forte : des « Loosely coupled Components ».

En Orienté objet, deux classes peuvent être « faiblement couplées » si une classe peut appeler une méthode de l'autre classe sans code spécifique à celle-ci.

Il est donc facile de remplacer une classe par un autre composant implémentant les même fonctionnalités.

Comment ?

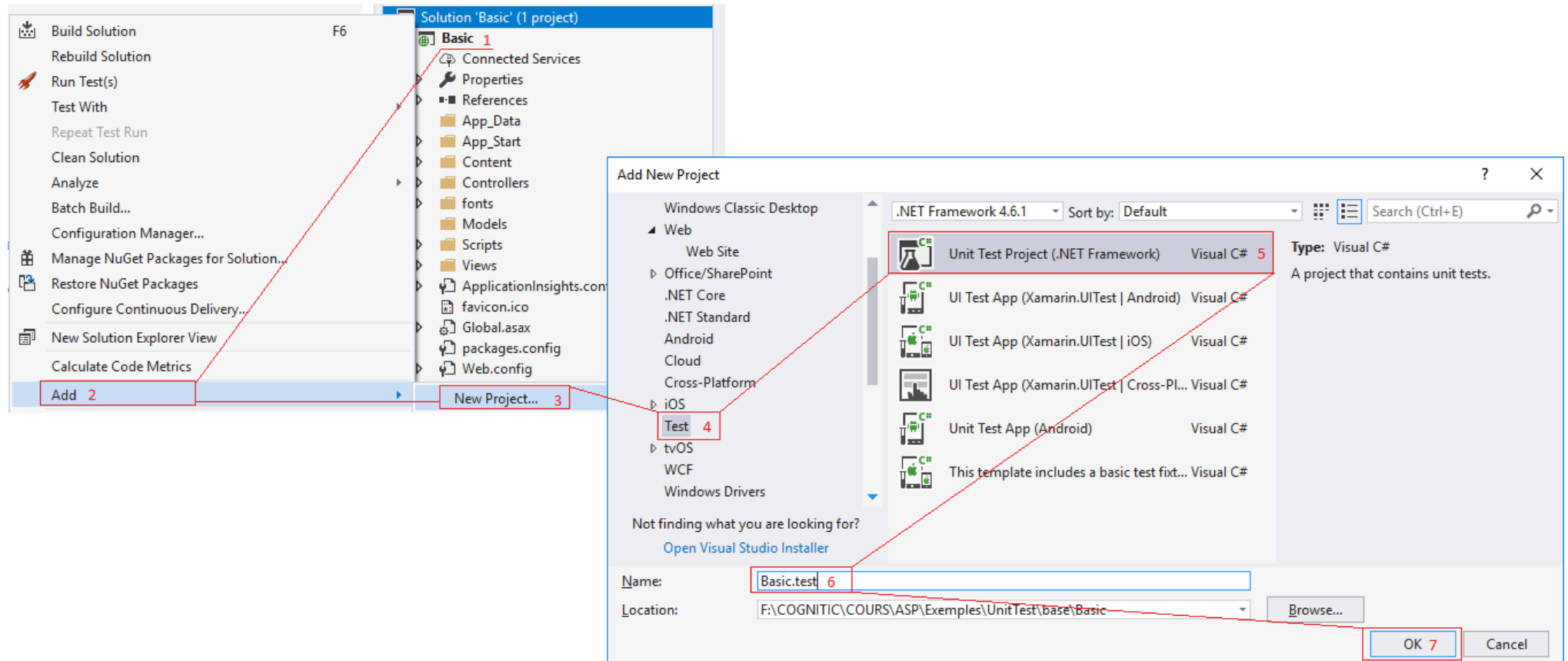
En utilisant les interfaces.

Chaque classe qui implémente une interface doit implémenter les propriétés et les méthodes.

Cela créé donc un faible couplage puisqu'on peut utiliser l'interface dans notre classe plutôt que la classe concrète.

# Composants MVC et Unit Testing

Pour tester une application MVC, nous devons ajouter un nouveau projet à notre solution.



# Composants MVC et Unit Testing

Pour commencer, le plus simple est de se focaliser sur les classes du Modèle puisqu'en MVC, ces classes ne dépendent pas des autres composants.

On peut donc facilement instancier ces classes en mémoire, attribuer les valeurs aux propriétés, appeler les méthodes et vérifier le résultat renvoyé.

```
PersonTest.cs
1 using System;
2 using System.Collections.Generic;
3 using Basic.Models;
4 using Microsoft.VisualStudio.TestTools.UnitTesting;
5
6 namespace Basic.test
7 {
8     [TestClass]
9     public class PersonTest
10     {
11         [TestMethod]
12         public void TestGetYoung()
13         {
14             //PREPARE
15             List<Person> ExceptedList = new List<Person>();
16             ExceptedList.Add(new Person() { Age = 12, City = "Charleroi", Email = "Alica@yopmail." });
17             ExceptedList.Add(new Person() { Age = 10, City = "Bruxelles", Email = "Eliot@yopmail." });
18             ExceptedList.Add(new Person() { Age = 11, City = "Philippeville", Email = "Mike@yopmail." });
19             Person p = new Person();
20             //ACT
21
22             List<Person> lp = p.getYoung();
23
24             //Assert
25             Assert.IsNotNull(lp);
26             Assert.AreEqual(ExceptedList.Count, lp.Count);
27             Assert.AreEqual(ExceptedList[0].Name, lp[0].Name);
28         }
29     }
30 }
```

```
Person.cs
25
26
27 [Range(7.0, 77.0, ErrorMessage = "Erreur, vous ne pouvez plus lire tintin")]
28 public int Age { get; set; }
29 public string Street { get; set; }
30 public string City { get; set; }
31 public string State { get; set; }
32 public int Zipcode { get; set; }
33
34 public bool savePerson()
35 {
36     throw new NotImplementedException();
37 }
38
39 public List<Person> getYoung()
40 {
41     List<Person> lp = new List<Person>();
42     lp.Add(new Person() { Age = 12, City = "Charleroi", Email = "Alica@yopmail." });
43     lp.Add(new Person() { Age = 10, City = "Bruxelles", Email = "Eliot@yopmail." });
44     lp.Add(new Person() { Age = 11, City = "Philippeville", Email = "Mike@yopmail." });
45
46     return lp;
47 }
48
49 }
```

# Composants MVC et Unit Testing

L'annotation *[TestMethod]* renseigne au moteur de test qu'il s'agit d'une méthode permettant le testing. Les *Assert* permettent de poser les vérifications de succès ou d'échec du test.

Les tests unitaires suivent le pattern A/A/A

**ARRANGE** : Déclaration des variables et objets nécessaires au testing

**ACT** : Exécution de la méthode à tester

**ASSERT** : Vérifications

# Composants MVC et Unit Testing

Méthode	Description
AreEqual<T>(T, T) AreEqual<T>(T, T, string)	Vérifie que les deux <T> ont la même valeur
AreNotEqual<T>(T, T) AreNotEqual<T>(T, T, string)	Vérifie que les deux <T> n'ont pas la même valeur
AreSame<T>(T, T) AreSame<T>(T, T, string)	Vérifie que les deux variables pointent sur le même objet
AreNotSame<T>(T, T) AreNotSame<T>(T, T, string)	Vérifie que les deux variables ne pointent pas sur le même objet
Fail() Fail(string)	Echec
Inconclusive() Inconclusive(string)	Indique que le résultat du test est incertain
IsTrue(bool) IsTrue(bool, string)	Évalue l'expression booléenne et renvoi le résultat
IsFalse(bool) IsFalse(bool, string)	Évalue l'expression booléenne et renvoi l'opposé du résultat
IsNull(object) IsNull(object, string)	Vérifie si l'objet est null

# Composants MVC et Unit Testing

Ces méthodes static permettent de vérifier certains Aspect des tests. Si une *Assertion* échoue, une exception est lancée et fait échouer le test.

Chaque test est exécuté séparément si un échoue, les autres continue à fonctionner.

Le string inclus dans les signatures permet d'envoyer un message vers l'exception afin de mieux tracer les erreurs.

Méthode	Description
IsNotNull(object) IsNotNull(object, string)	Vérifie que l'objet n'est pas null
IsInstanceOfType(object, Type) IsInstanceOfType(object, Type, string)	Vérifie si l'objet est une instance de <T>
IsNotInstanceOfType(object, Type) IsNotInstanceOfType(object, Type, string)	Vérifie si l'objet n'est pas une instance de <T>

# Composants MVC et Unit Testing

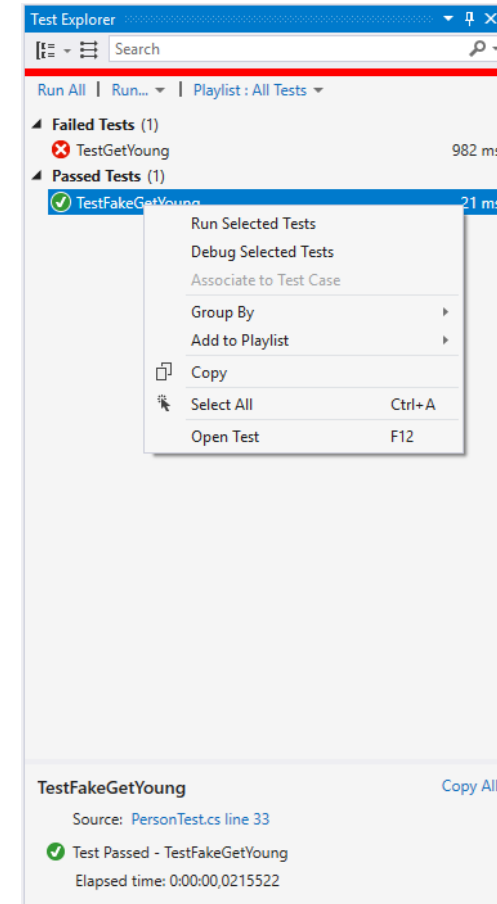
Pour lancer notre test :

- Build de la solution

- Affichage de l'explorateur de test

  - Menu → Test → Window → Test explorer

  - Ou ctrl+E,T



# Composants MVC et Unit Testing

## Facile?

Dans l'exemple précédent, nous simulons une liste dans la méthode même que l'on doit tester. Ce n'est évidemment pas le cas dans le monde réel.

## Un pattern à la rescousse : Le Repository

Un repository est basé sur une interface qui définit les propriétés et les méthodes nécessaires pour une application qui désire enregistrer des données.

Ce pattern nous permet par la suite de définir une source de donnée qui peut être un context EntityFramework, une couche Ado.net ou une FakeDb.



# Composants MVC et Unit Testing

## Interface

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Basic.Models.Interfaces
8 {
9     public interface IPersonContext
10     {
11         IQueryable<Person> GetAll();
12         Person GetById(int id);
13         Person Add(Person Entity);
14         bool Delete(Person Entity);
15         int SaveChanges();
16     }
17 }
```

## Implémentation

```
1 using Basic.Models.Global;
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Web;
6
7 namespace Basic.Models.Interfaces
8 {
9     public class PersonContext : IPersonContext
10     {
11         CampingEntities camping = new CampingEntities();
12         public Person Add(Person Entity)
13         {
14             return MapToPersonLocal(camping.Person.Add(MapToPerson(Entity)));
15         }
16
17         public bool Delete(Person Entity) {...}
18
19         public IQueryable<Person> GetAll() {...}
20
21         public Person GetById(int id) {...}
22
23         public int SaveChanges() {...}
24
25         private Person MapToPersonLocal(Global.Person person) {...}
26         private Global.Person MapToPerson(Person entity) {...}
27     }
28 }
```

# Composants MVC et Unit Testing

Afin de pouvoir effectuer les tests unitaires, nous ne pouvons pas utiliser l'implémentation précédente car elle se base directement sur notre couche Entity et donc notre test ferait partie des *Test d'intégration*.

Nous devons créer un FakeRepository afin de s'abstraire de la couche Db (EntityFramework, Ado.net,...)

Nous simulerons une collection (source de données) pour effectuer nos tests.

```
6 namespace Basic.Models.Interfaces
7 {
8     public class FakePersonContext : IPersonContext
9     {
10         //Permet de simuler une db
11         SetMap _map = new SetMap();
12
13         public IQueryable<Person> Persons
14         {
15             set { _map.Use<Person>(value); }
16         }
17
18         public bool ChangesSaved { get; set; }
19         public Person Add(Person Entity)
20         {
21             _map.Get<Person>().Add(Entity);
22             return Entity;
23         }
24         public IQueryable<Person> GetAll()
25         {
26             return _map.Get<Person>().AsQueryable();
27         }
28
29         public Person GetById(int id)
30         {
31             Person item = (from p in this.GetAll()
32                             where p.Id == id
33                             select p).First();
34
35             return item;
36         }
37         public int SaveChanges()...
42
43         public bool Delete(Person Entity)...
44     }
45 }
```

# Composants MVC et Unit Testing

Voici donc notre méthode de test

Il ne diffère pas beaucoup de notre test précédent MAIS celui-ci à l'avantage de permettre de tester le fonctionnement réel du repository sans devoir modifier le code de celui-ci afin d'ajouter des fakes datas

```
[TestMethod]
public void TestFakeGetYoung()
{
    //PREPARE
    var context = new FakePersonContext();
    context.Persons = new[]
    {
        new Person() { Age = 12, City = "Charleroi", Email = "Alica@yopmail.com", FirstName = "Alicia", Name = "Stein", Zipcode = 1337 },
        new Person() { Age = 10, City = "Bruxelles", Email = "Eliot@yopmail.com", FirstName = "Eliot", Name = "Ness", Zipcode = 1050 },
        new Person() { Age = 11, City = "Philippeville", Email = "Mike@yopmail.com", FirstName = "Mike", Name = "Kelkel", Zipcode = 54000 }
    }.AsQueryable();

    Person p = new Person();
    //ACT

    List<Person> lp = p.getYoung();

    //Assert
    Assert.IsNotNull(lp);
    Assert.AreEqual(3, lp.Count);
    Assert.AreEqual("Stein", lp[0].Name);
}
```

# Composants MVC et Unit Testing

Nous devons aller plus loin car nous devons nous assurer que l'appel de notre code se fait dans le bon « context »

Unit test : test sans prendre en compte la couche data.

Autres cas : le code doit interagir avec la couche Data (Entity Framework, Ado.net,...)

Le premier choix qui s'offre à nous est d'utiliser un constructeur afin de définir le repository lors de l'accès à notre classe et à ses méthodes

```
private IPersonContext _context;

public Person()
{
    _context = new PersonContext();
}

public Person(IPersonContext ip)
{
    _context = ip;
}

public List<Person> getYoung()
{
    //Avant le context
    //List<Person> lp = new List<Person>();
    //lp.Add(new Person() { Age = 12, City = "Charleroi", Ema:
    //lp.Add(new Person() { Age = 10, City = "Bruxelles", Ema:
    //lp.Add(new Person() { Age = 11, City = "Philippeville", I

    //return lp;

    //Après l'ajout des constructeurs
    return _context.GetAll().ToList();
}
```

# Composants MVC et Unit Testing

## Echec du test d'intégration : Erreur couche DB

```
[TestMethod]
public void TestGetYoung()
{
    //PREPARE
    List<Person> ExceptedList = new List<Person>();
    ExceptedList.Add(new Person() { Age = 12, City = "Charleroi", Email = "Alica@yopmail.com",
    ExceptedList.Add(new Person() { Age = 10, City = "Bruxelles", Email = "Eliot@yopmail.com",
    ExceptedList.Add(new Person() { Age = 11, City = "Philippeville", Email = "Mike@yopmail.com"
    Person p = new Person();
    //ACT

    List<Person> lp = p.getYoung();

    //Assert
    Assert.IsNotNull(lp);
    Assert.AreEqual(ExceptedList.Count, lp.Count);
    Assert.AreEqual(ExceptedList[0].Name, lp[0].Name);
}
```

Failed Tests (1)  
✖ TestGetYoung 982 ms

## Succès du test unitaire : Méthode de la classe valide

```
[TestMethod]
public void TestFakeGetYoung()
{
    //PREPARE
    var context = new FakePersonContext();
    context.Persons = new[]
    {
        new Person() { Age = 12, City = "Charleroi", Email = "Alica@yopmail.com", FirstName = "Alicia",
        new Person() { Age = 10, City = "Bruxelles", Email = "Eliot@yopmail.com", FirstName = "Eliot",
        new Person() { Age = 11, City = "Philippeville", Email = "Mike@yopmail.com", FirstName = "Mike",
    }.AsQueryable();

    Person p = new Person(context);
    //ACT

    List<Person> lp = p.getYoung();

    //Assert
    Assert.IsNotNull(lp);
    Assert.AreEqual(3, lp.Count);
    Assert.AreEqual("Stein", lp[0].Name);
}
```

Passed Tests (1)  
✔ TestFakeGetYoung 21 ms