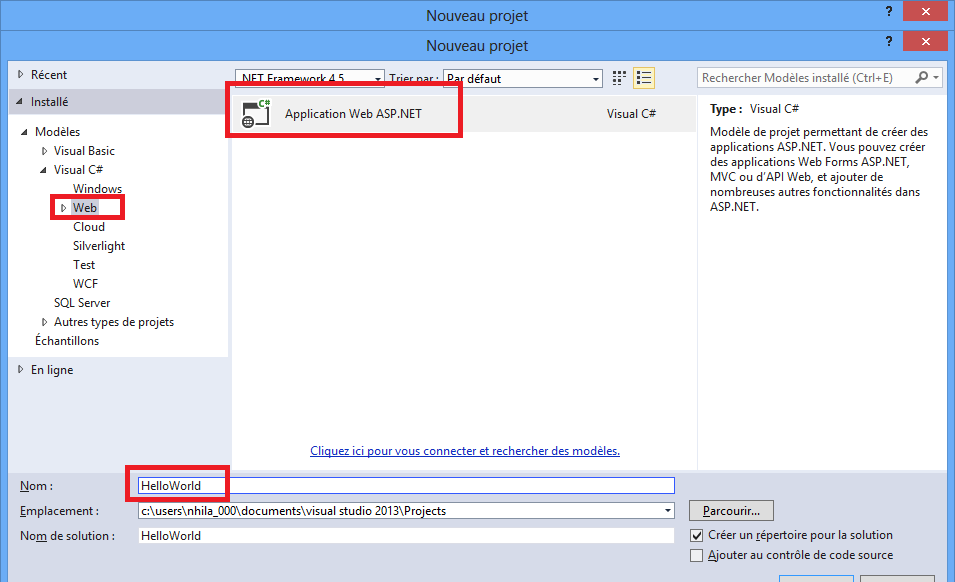
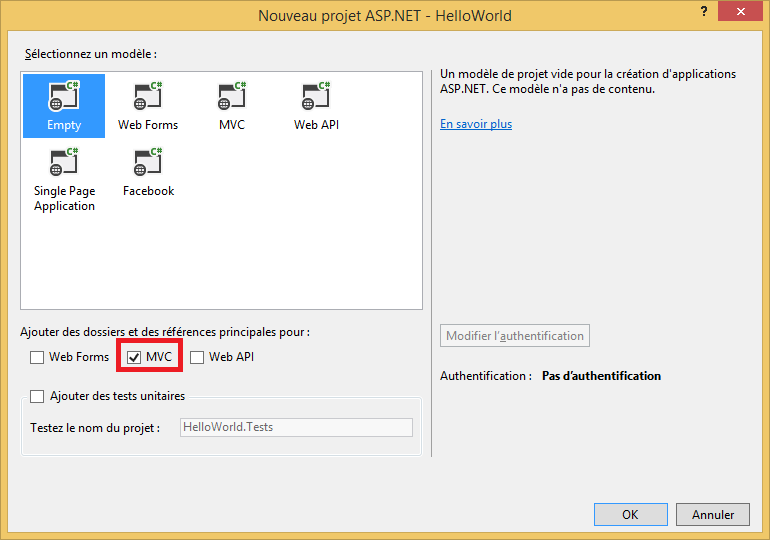
**Hello World MVC**

**Création du projet avec Visual Studio**

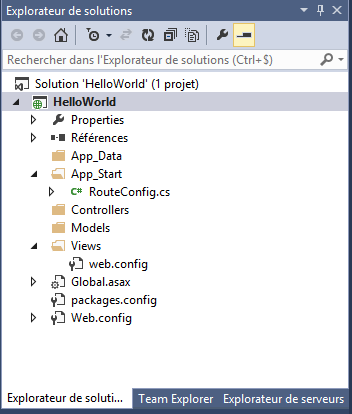
Alors on attaque avec Visual Studio, notre outil de développement dans lequel nous allons réaliser notre Hello World.. La première chose à faire est de créer un nouveau projet ASP.NET MVC. Rendez-vous dans le menu Fichier et choisissez Nouveau Projet, puis dans les modèles, choisissez un projet Visual C#/Web de type Application Web ASP.NET :

Création d'un projet ASP.NET

Visual Studio vous propose de sélectionner un modèle pour votre projet. Comme nous voulons tout faire par nous-mêmes, nous allons partir sur le modèle « empty », qui est donc vide, sans oublier de cocher MVC (ce qui nous permettra d'inclure tout ce qui est nécessaire pour utiliser le framework MVC) :

Choix du modèle vide

Pour l’instant laissez la case qui permet de créer un projet de test unitaire décochée. Nous y reviendrons ultérieurement dans un chapitre dédié, mais sachez dès à présent que c’est un point à ne pas manquer car il est très important de réaliser des tests automatiques de son application et que l’architecture de MVC le facilite.  
Un projet par défaut se crée, contenant déjà quelques dossiers et fichiers

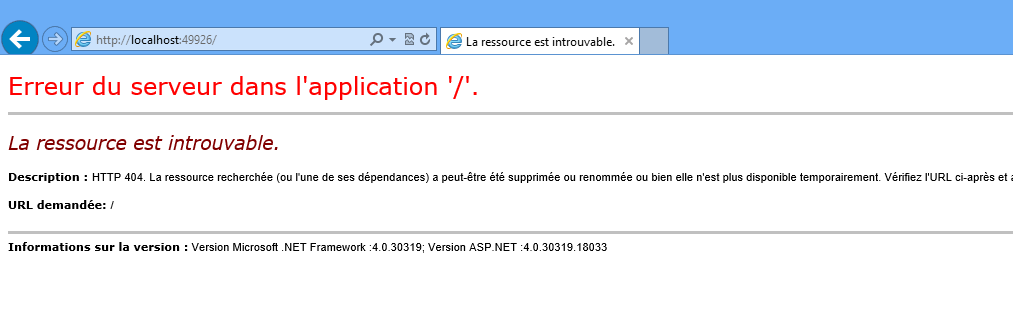


Arborescence de la solution après choix d'un modèle vide

Malgré le choix d'un modèle vide, la solution générée ne l'est pas tant que ça. Il y a quand même quelques fichiers et surtout quelques répertoires. Ici Visual Studio nous a créé le squelette par défaut d'une application MVC.   
Globalement il n’y a pas grand-chose, mais regardons un peu ce qu’il y a à l'intérieur :

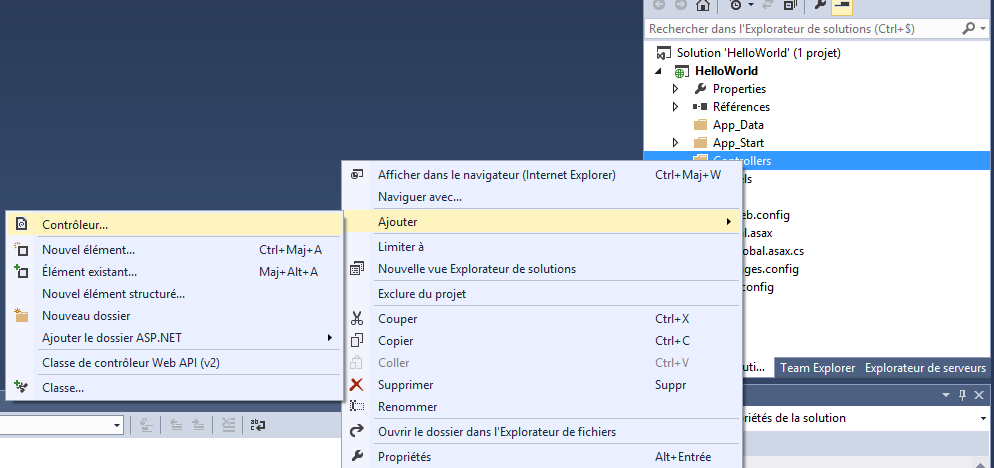
* Le répertoire App\_Data est un répertoire où nous pourrons stocker des données, fichiers binaires ou bases de données. Sachez qu’en général il est peu utilisé dans une vraie application mais qu’il peut bien dépanner dans notre phase d’apprentissage.
* Le répertoire App\_Start contient en général la logique de configuration, il y a dedans un fichier RouteConfig.cs, que nous pouvons ignorer pour l’instant et qui sert à configurer l’application.
* Le répertoire Controller contiendra en toute logique les contrôleurs qui servent à gérer les actions.
* Le répertoire Models, où nous mettrons tout ce qui a rapport avec notre modèle, c’est-à-dire les classes qui interagissent avec les données et qui gèrent les règles spécifiques de notre application.
* Le répertoire Views contiendra toutes les vues de notre application, grosso modo ce qui permettra le rendu de nos pages à base de HTML. Ceci fera bien sûr l’objet d’une bonne explication un peu plus tard.
* Un fichier Global.asax, qui est exécuté au tout début du lancement de l’application et qui nous permettra entre autre de configurer l’application. Si vous l’ouvrez, vous constaterez qu’il fait appel aux méthodes définies dans la classe RouteConfig présente dans le répertoire App\_Start
* Un fichier packages.config que nous pouvons ignorer (il sert à la configuration des packages nuget que nous utiliserons plus loin, mais il n'est pas intéressant ici de s'y plonger).
* Un autre fichier Web.config (différent de celui présent dans le répertoire Views), bien connu des développeurs ASP.NET WebForms, qui contient des éléments de configuration de l’application.

Place à l’action ! Démarrons l’application par défaut générée en appuyant sur F5 (ou passez par le menu Déboguer, Démarrer le débogage). Votre navigateur par défaut démarre et ouvre une page dont l’adresse ressemble à http://localhost:49926/, avec une belle page d’erreur :

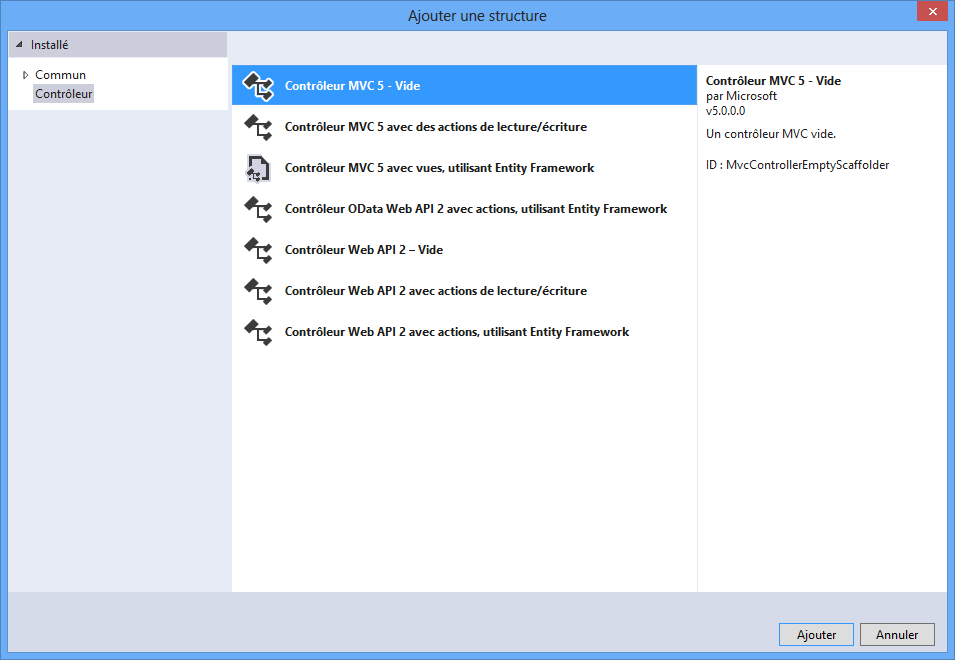


**Le contrôleur**

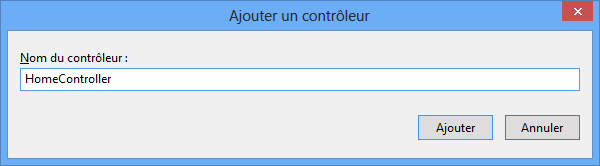
Alors cette page d’erreur ? Il nous a dit que la ressource « / » était introuvable. Et oui, il n’y a rien dans notre application web, elle ne sait pas traiter l’action associée à la route demandée. Nous allons ajouter un contrôleur afin de pouvoir traiter la demande d'affichage. Cliquez droit sur le répertoire Controllers et ajoutez un nouveau contrôleur :

Ajout d'un contrôleur

Sur la fenêtre, il y a plusieurs choix avec différents modèles. Nous y reviendrons plus tard et pour l’instant, gardez le modèle " Contrôleur MVC 5 - Vide " :

Choix du type de contrôleur

Puis nommez-le Puis nommez-le HomeController

Nommez le contrôleur HomeController

Le code suivant se génère pour nous :

public class HomeController : Controller

{

//

// GET: /Home/

public ActionResult Index()

{

return View();

}

}

Ce qu’il est important de constater c’est que la classe dérive de la classe [Controller](https://msdn.microsoft.com/en-us/library/system.web.mvc.controller.aspx) du framework MVC. Comme nous voulons faire apparaître le message « Hello World MVC » dans notre application, nous allons modifier la méthode Index  générée pour avoir :

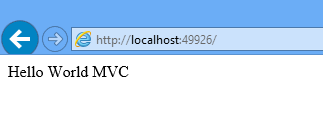
public string Index()

{

return "Hello World MVC";

}

Démarrons à nouveau notre application web. Nous n’avons cette fois-ci plus d’erreur et nous pouvons voir un magnifique message sur la page web :

Affichage de notre Hello World

**La route**

Que s’est-il passé ? Pour le comprendre, saisissez la nouvelle URL suivante : http://localhost:49926/Home/Index. Nous obtenons exactement la même chose. Regardez la fin de l'URL : /Home/Index.

Avec cette deuxième URL, nous avons en fait demandé au framework MVC de nous exécuter la méthode Index  du contrôleur HomeController  et c'est la classe et la méthode que nous avons créées.

Pourquoi la méthode de cette classe a-t-elle été appelée ? Pour le comprendre, ouvrez le fichier RouteConfig.cs qui se trouve dans le répertoire App\_Start. Nous pouvons notamment voir la ligne suivante :

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

C’est la définition de la route par défaut. Cela veut dire que lorsqu’on demande d’afficher l’URL http://localhost:49926/Home/Index, le framework MVC interprète cette URL et la transforme en action. En l’occurrence, l’URL ici est « Home/Index ». Suivant le schéma défini dans la route par défaut, à savoir {controller}/{action}/{id}, il transforme l’URL en :

* controlleur = Home
* action = Index
* id = null

Ceci a pour effet d’instancier le contrôleur HomeController  et d’exécuter la méthode Index . Celle-ci renvoie une chaîne de caractères et c’est ce qui s’affiche dans la page. Il ne s’agit pas d’HTML ici, mais bien d’une chaîne de caractères. Faites afficher le code source de la page dans votre navigateur et vous verrez que vous n’aurez que des caractères, pas de balises HTML ou autre…  
Si vous changez l’URL et que vous changez Index par Index2 par exemple, alors vous aurez à nouveau une page d’erreur… et oui, pas de méthode Index2  dans notre contrôleur ! Vous savez ce qu’il vous reste à faire pour pouvoir afficher quelque chose avec l’URL http://localhost:49926/Home/Index2... : ajouter une méthode Index2, par exemple :

public string Index2()

{

return "Hello les contrôleurs";

}

Bon, c’est bien beau, mais notre URL http://localhost:49926/, elle affichait bien quelque chose ! Alors que nous n’avions défini ni de contrôleur à instancier, ni d’action à exécuter… Et oui, c’est parce que dans la définition de la route par défaut :

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

il y a justement des valeurs par défaut qui remplacent les éléments qui ne seraient pas fournis dans l’URL. Ainsi, si l’action n’est pas fournie, il utilisera l’action (donc la méthode) Index  par défaut. De même, si le nom du contrôleur n’est pas fourni, alors il utilisera le contrôleur Home  par défaut. C’est pour cela que les URL suivantes produiront exactement la même chose :

* http://localhost:49926/
* http://localhost:49926/Home
* http://localhost:49926/Home/Index

Par contre, bien évidemment, l’URL suivante produira une erreur : http://localhost:49926/Index2  
Et oui, le format de route ne peut pas deviner qu’Index2  est l’action du contrôleur par défaut. Ici, il essaie de trouver le contrôleur Index2 qui n’existe bien sûr pas…  
Nous reviendrons plus en détail sur ces routes dans un chapitre ultérieur mais sachez donc qu’en MVC, lorsque vous naviguez sur une URL, ce n’est pas un fichier que vous tentez d’afficher, non. Vous tentez d’exécuter une action d’un contrôleur !

Bon, et cet id alors qu’il y a à la fin de la route alors ? C’est quoi donc ?

Et bien il s’agit de la possibilité de passer des paramètres à notre action. Remplacez donc la méthode Index  par la suivante :

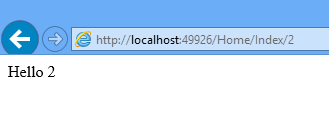
public string Index(int id)

{

return "Hello " + id;

}

Maintenant, vous allez pouvoir naviguer sur l’URL suivante http://localhost:49926/Home/Index?id=2 et obtenir :

Passage de paramètres à la méthode du contrôleur

Remarquez que cette URL http://localhost:49926/Home/Index/2 donne exactement la même chose.  
Automatiquement, le framework MVC traduit le dernier élément dans le paramètre id. Ici, depuis que nous avons ajouté le paramètre dans le contrôleur, l’id devient obligatoire. Si vous naviguez à nouveau sur l’URL http://localhost:49926/Home/Index vous aurez une exception car le paramètre est requis dans la méthode Index de notre contrôleur. Pour que le paramètre soit facultatif, il faudra utiliser un entier de type « nullable », c’est-à-dire qui peut valoir « null  » :

public string Index(int? id)

{

return "Hello " + id;

}

Remarquez que vous pouvez également passer une chaîne de caractères en modifiant la méthode :

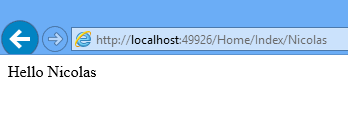
public string Index(string id)

{

return "Hello " + id;

}

Et vous pouvez naviguer sur http://localhost:49926/Home/Index/Nicolas, vous aurez alors :

  
Notez que vous êtes obligés d’utiliser le paramètre s’appelant id , il n’est pas possible de définir la méthode Index ainsi :

public string Index(string nom)

{

return "Hello " + nom;

}

Et non ! Rappelez-vous la définition de notre route : {controller}/{action}/{id}. On voit bien id , pas nom  ! Le paramètre n’est pas reconnu, il vaudra donc null . Pour résoudre cela il faudrait modifier la route par défaut ou ajouter une nouvelle route… mais bon, nous verrons ça plus tard.   
Je ne sais pas ce que vous en pensez, mais notre page web n’est pas très jolie… et puis ce n’est même pas de l’HTML, comment je vais bien pouvoir mettre de la couleur, des titres ? Bien sûr, on pourrait imaginer renvoyer de l’HTML depuis notre méthode, du genre :

public string Index(string id)

{

return @"

<html>

<head>

<title>Hello World</title>

</head>

<body>

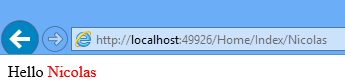
<p>Hello <span style=""color:red"">" + id + @"</span></p>

</body>

</html>";

}

Cela donnerait le résultat suivant :

Le contrôleur renvoie du HTML sous la forme d'une chaîne de caractères

Mais là, Tout est mélangé… niveau séparation des intentions, on repassera !

**La vue**

C’est là qu’intervient la vue. La vue, c’est ce qu’on voit. En l’occurrence, ça sera de l’HTML. Revenons sur la toute première version de notre contrôleur, générée par Visual Studio, qui était :

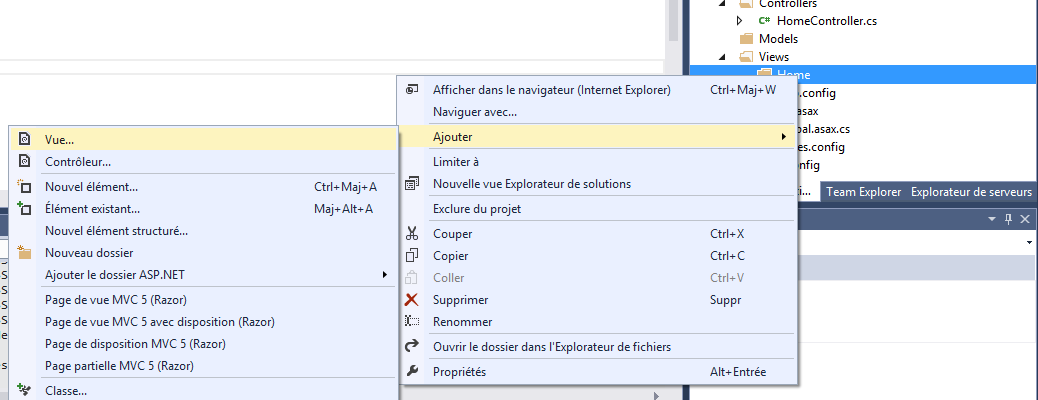
public ActionResult Index()

{

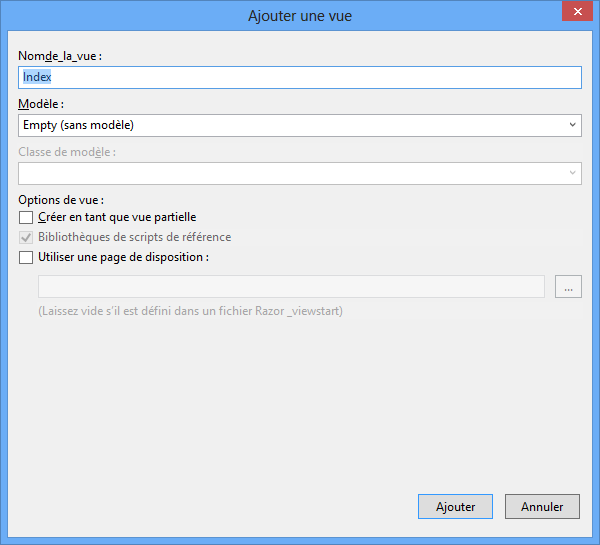
return View();

}

Ici, l’action renvoie un objet [ActionResult](https://msdn.microsoft.com/fr-fr/library/system.web.mvc.actionresult.aspx) en appelant la méthode View() , héritée de la classe Controller  du framework MVC. Créons ensuite un répertoire sous le répertoire Views qui s’appelle Home. Et dedans, nous allons y mettre une vue que nous appellerons Index.cshtml, de manière à avoir Views\Home\Index.cshtml. Pour ce faire, créez donc le sous-répertoire Home sous le répertoire Views (bouton droit sur le dossier Views, puis ajouter Nouveau dossier, s'il n'a pas encore été ajouté) et cliquez-droit sur le répertoire Home pour ajouter une vue :

Ajouter une vue

Nous l’appelons donc Index. Vous pouvez décocher tout le reste, nous ne nous en servirons pas pour l’instant, le modèle étant "empty" :



Ajout d'une vue Index

Et voilà, nous nous retrouvons avec un fichier Index.cshtml qui contient du HTML que nous pouvons modifier pour y ajouter un petit message :

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>Index</title>

</head>

<body>

<div>

Hello la vue

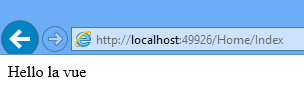
</div>

</body>

</html>

Il n’y a pas que du HTML là-dedans, nous verrons ça plus loin. D'ailleurs, l'extension du fichier .cshtml nous fait penser que nous pourrons à la fois mettre du C# et du HTML...

Et si nous redémarrons l’application nous aurons :

Affichage d'une vue

Et cette fois-ci, c’est du vrai HTML. Vous pouvez le vérifier en affichant le code source de la page.

public ActionResult Index()

{

return View();

}

Remarquez que dans l’action du contrôleur ci-dessus, appeler la méthode View()  équivaut à appeler la méthode View  en lui passant le nom de la vue en paramètre :

public ActionResult Index()

{

return View("Index");

}

En fait, automatiquement la méthode View()  va chercher la vue qui est placée dans le répertoire Views, dans le sous répertoire du même nom que le contrôleur et qui s’appelle comme la méthode de l’action appelée. Si vous renommez la méthode Index  en Index2  et que vous naviguez sur l’URL suivante : http://localhost:49926/Home/Index2, il vous dira que la vue Index2 est introuvable. Ce qui est vrai !  
Pour afficher la vue Index, il suffira de remplacer l’appel à View()  par View("Index") . Ou bien créer une vue s’appelant Index2.cshtml…  
Ce qui nous permettrait par exemple de gérer le cas où un id  est null  ou vide, en choisissant d'afficher une vue d'erreur dédiée. Remplacez donc la méthode Index par :

public ActionResult Index(string id)

{

if (string.IsNullOrWhiteSpace(id))

return View("Error");

else

return View();

}

Puis créez un répertoire Shared sous le répertoire Views. Maintenant, ajoutez une vue que vous appelez Error, comme nous l'avons fait précédemment et mettez-y le HTML suivant :

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>Error</title>

</head>

<body>

<h1>Une erreur s'est produite lors du traitement de votre demande.</h1>

</body>

</html>

Enfin, naviguez sur http://localhost:49926/Home/Index/Nicolas. Vous aurez alors le « Hello la vue » que nous avons dans la vue Index. Mais si vous naviguez sur http://localhost:49926/Home/Index/ par exemple, vous aurez alors l’affichage de la vue Error.cshtml (et oui ! il n’y a pas d’id  !), que nous vennons de rajouter :

Affichage de la vue Error

Error.cshtml est une vue qui en général est générée lors de la création d'un projet et qui contient un magnifique message d’erreur de ce genre. Étant donné que nous avons choisi de créer un modèle de projet vide, cette vue n'apparaît pas, c'est pourquoi nous l'avons rajoutée.

Il faut savoir que par défaut, ASP.NET MVC va chercher la vue dans le sous-répertoire du même nom que le contrôleur, mais que si elle n’existe pas, il ira voir ensuite dans le sous-répertoire Shared. Ce qu’il a fait ici.  
Bon, c’est bien beau ça, mais comment nous pouvons faire en sorte d’afficher le prénom de la personne dans notre vue ?  
En fait, il est très facile de faire passer des valeurs entre le contrôleur et la vue grâce à un dictionnaire de clé/valeur : le **ViewData**.  
Par exemple, pour pouvoir afficher le paramètre récupéré depuis l’URL, je vais pouvoir faire ceci dans le contrôleur :

public ActionResult Index(string id)

{

if (string.IsNullOrWhiteSpace(id))

return View("Error");

else

{

ViewData["Nom"] = id;

return View();

}

}

Avec ce code, je positionne la valeur de la chaîne id  dans le dictionnaire et que cette valeur sera accessible via la clé « Nom ».  
Puis dans la vue, je pourrais récupérer et afficher cette valeur en utilisant ce même dictionnaire :

<body>

<div>

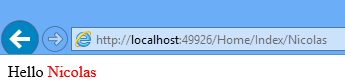
Hello <span style="color:red">@ViewData["Nom"]</span>

</div>

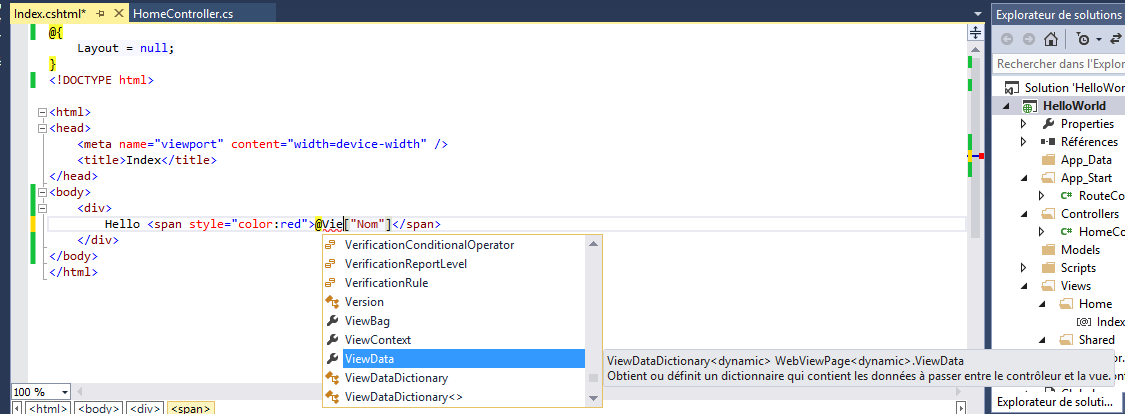
</body>

Remarquez que j’utilise le caractère @ pour utiliser du code C# directement dans ma vue.

Par exemple pour écrire dans la page HTML la valeur contenue dans le dictionnaire avec la clé Nom, je pourrais utiliser la syntaxe @ViewData["Nom"] . Et nous aurons :



Ce qui est intéressant, c’est que même dans la vue, nous disposons de la complétion automatique pour nous aider dans notre saisie :

Utilisation de la complétion automatique dans la vue

Remarquez que vous pouvez également utiliser l’objet ViewBag  qui est de type dynamic . Avec par exemple dans le contrôleur :

public ActionResult Index(string id)

{

if (string.IsNullOrWhiteSpace(id))

return View("Error");

else

{

ViewBag.Nom = id;

return View();

}

}

Et dans la vue :

Hello <span style="color:red">@ViewBag.Nom</span>

Vous pouvez utiliser la syntaxe que vous préférez.   
Et voilà pour le VC de MVC.

**Le modèle**

Reste le M, mais dans cette introduction à MVC je ne vais pas du tout m’étendre sur ce fameux modèle, nous y reviendrons un peu plus loin. Car modèle rime souvent avec base de données, même si ce n’est pas une obligation. Ici pour l’instant, notre modèle va rimer avec données en dur dans le code.   
Créons donc une classe qui va nous permettre de simuler le chargement d’une liste de clients. Faites un clic droit sur le répertoire Models et choisissez d’ajouter une classe toute bête, que nous allons appeler Client qui contiendra les propriétés d’un client :

public class Client

{

public string Nom { get; set; }

public int Age { get; set; }

}

Ajoutez également une classe Clients  qui simulera un chargement :

public class Clients

{

public List<Client> ObtenirListeClients()

{

return new List<Client>

{

new Client { Age = 33, Nom = "Nicolas"},

new Client { Age = 30, Nom = "Delphine"},

new Client { Age = 33, Nom = "Jérémie"},

new Client { Age = 1, Nom = "Timéo"}

};

}

}

Ce modèle est à instancier dans le contrôleur. Créons par exemple deux actions dans le contrôleur :

public ActionResult ListeClients()

{

Clients clients = new Clients();

ViewData["Clients"] = clients.ObtenirListeClients();

return View();

}

public ActionResult ChercheClient(string id)

{

ViewData["Nom"] = id;

Clients clients = new Clients();

Client client = clients.ObtenirListeClients().FirstOrDefault(c => c.Nom == id);

if (client != null)

{

ViewData["Age"] = client.Age;

return View("Trouve");

}

return View("NonTrouve");

}

Bon, rien de transcendant dans ce modèle, mais il va me servir à illustrer un peu mieux les vues.  
Puis vous l’aurez deviné, il va nous falloir trois vues :

* ListeClients.cshtml
* Trouve.cshtml
* NonTrouve.cshtml

Commençons par la vue Trouve.cshtml, qui doit se trouver dans le sous-répertoire du même nom que le contrôleur. Elle contiendra du HTML indiquant que le client a été trouvé, puis son âge :

<body>

<div>

Le client @ViewData["Nom"] a bien été trouvé et a @ViewData["Age"] ans.

</div>

</body>

Pour la vue NonTrouve.cshtml, le principe est le même :

<body>

<div style="color:red">

Le client @ViewData["Nom"] n'a pas été trouvé !

</div>

</body>

Enfin pour la vue ListeClients, vous avez compris le principe, nous allons pouvoir afficher la liste de tous les clients avec cette vue :

<body>

<table>

<tr>

<th>Nom</th>

<th>Age</th>

</tr>

@foreach (HelloWorld.Models.Client client in ViewData["Clients"] as List<HelloWorld.Models.Client>)

{

<tr>

<td>@client.Nom</td>

<td>@client.Age</td>

</tr>

}

</table>

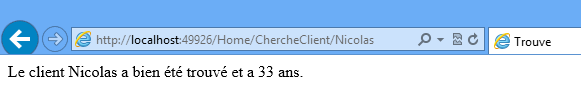
</body>

Voyez comme il est très facile de mixer du HTML et du C#…

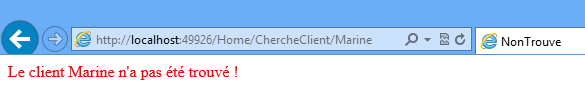
**Le résultat**

Alors, ça donne quoi ?  
Commençons par trouver un client avec l’URL suivante :  
http://localhost:49926/Home/ChercheClient/Nicolas

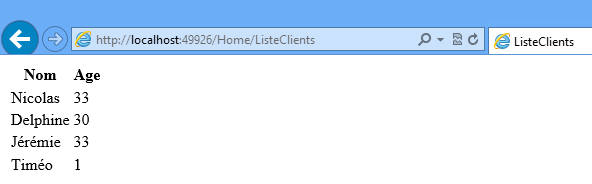
Nous aurons :

Affichage de la vue qui cherche un client

Puis, essayons avec un client inconnu, par exemple en naviguant sur http://localhost:49926/Home/ChercheClient/Marine, nous aurons :

Affichage de la vue où le client n'est pas trouvé

Enfin, nous pourrons afficher la liste de tous les clients, avec http://localhost:49926/Home/ListeClients.  
Ce qui donne :

Affichage de tous les clients  
Voilà pour cette première prise en main d’ASP.NET MVC. Nous avons pu faire un classique Hello World finalement sans avoir à faire grand-chose. Bon, nous sommes loin d’avoir tout vu et justement nous nous plongerons dans les détails des trois éléments de MVC dans la partie suivante.

**En résumé**

* Une URL est transformée en action via le mécanisme de routing d’ASP.NET.
* Le contrôleur est une classe qui dérive de la classe de base Controller  et dont les méthodes reflètent les actions faites par l’utilisateur.
* La vue peut combiner facilement des balises HTML et du code C#, pour par exemple afficher des informations remontées par le contrôleur grâce à l’objet ViewData .
* Le modèle est exploité par le contrôleur et transmis à la vue pour être affiché.#