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 défini 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 …

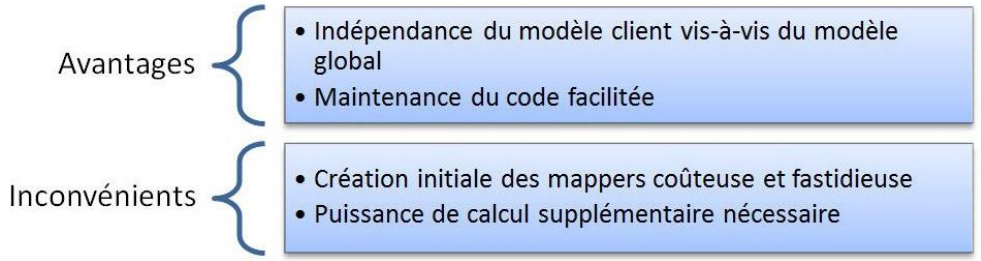
   Il n'y a donc AUCUNE LOGIQUE d'interraction (réaction aux événements UI, Gestion de formulaire,…) dans ce modèle.

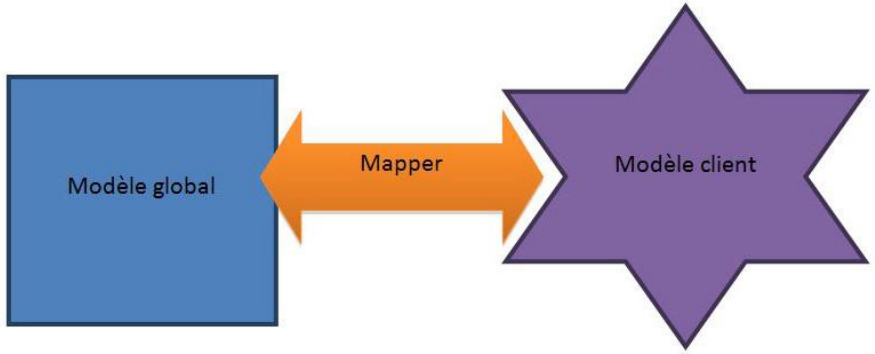
Les Modèles

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.

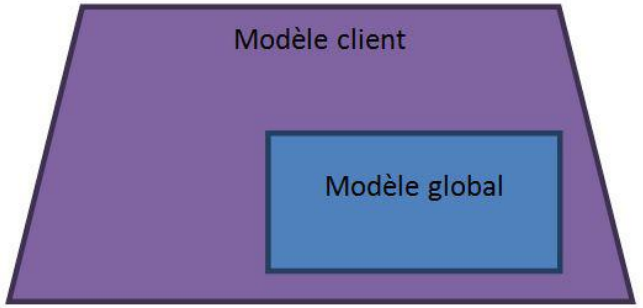




Les Modèles

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.

Avantages	• Peu de puissance de calcul nécessaire • Gestion des états et retours aux valeurs d'origine des objets facilitée
Inconvénients	• Dépendance forte du modèle client sur le modèle global • Nécessite un contrôle complet du code correspondant au modèle global



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

Basic.Controllers.PersonController | Save(Person P)

```
    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);
        }
    }
}
```

Person.cs* | Basic | Basic.Models.Person | Zipcode

```
 7  namespace Basic.Models
 8  {
 9      public class Person
10      {
11          public int Id { get; set; }
12          public string Name { get; set; }
13          public int Age { get; set; }
14          public string Street { get; set; }
15          public string City { get; set; }
16          public string State { get; set; }
17          public int Zipcode { get; set; }
18
19          internal bool savePerson()
20          {
21              throw new NotImplementedException();
22          }
23      }
24  }
25
```

Les Modèles

Scafolding

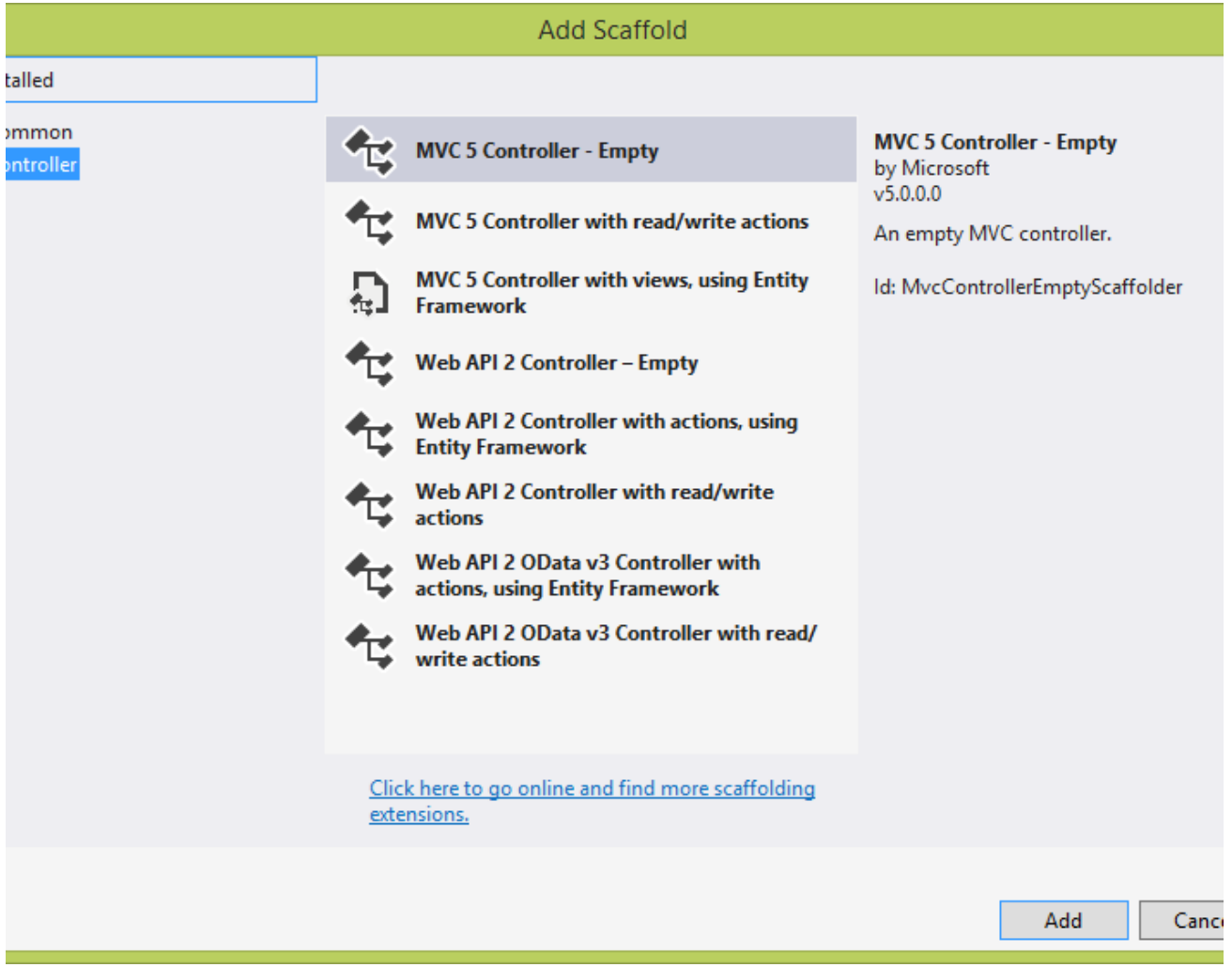
En ASP.NET, le scafolding permet de générer le squelette CRUD (**C**reate, **R**ead, **U**pdate et **D**elete)

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 scafolding afin de générer la totalité de votre application.

Utilisez le scafolding 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 vide mais ne contiennent

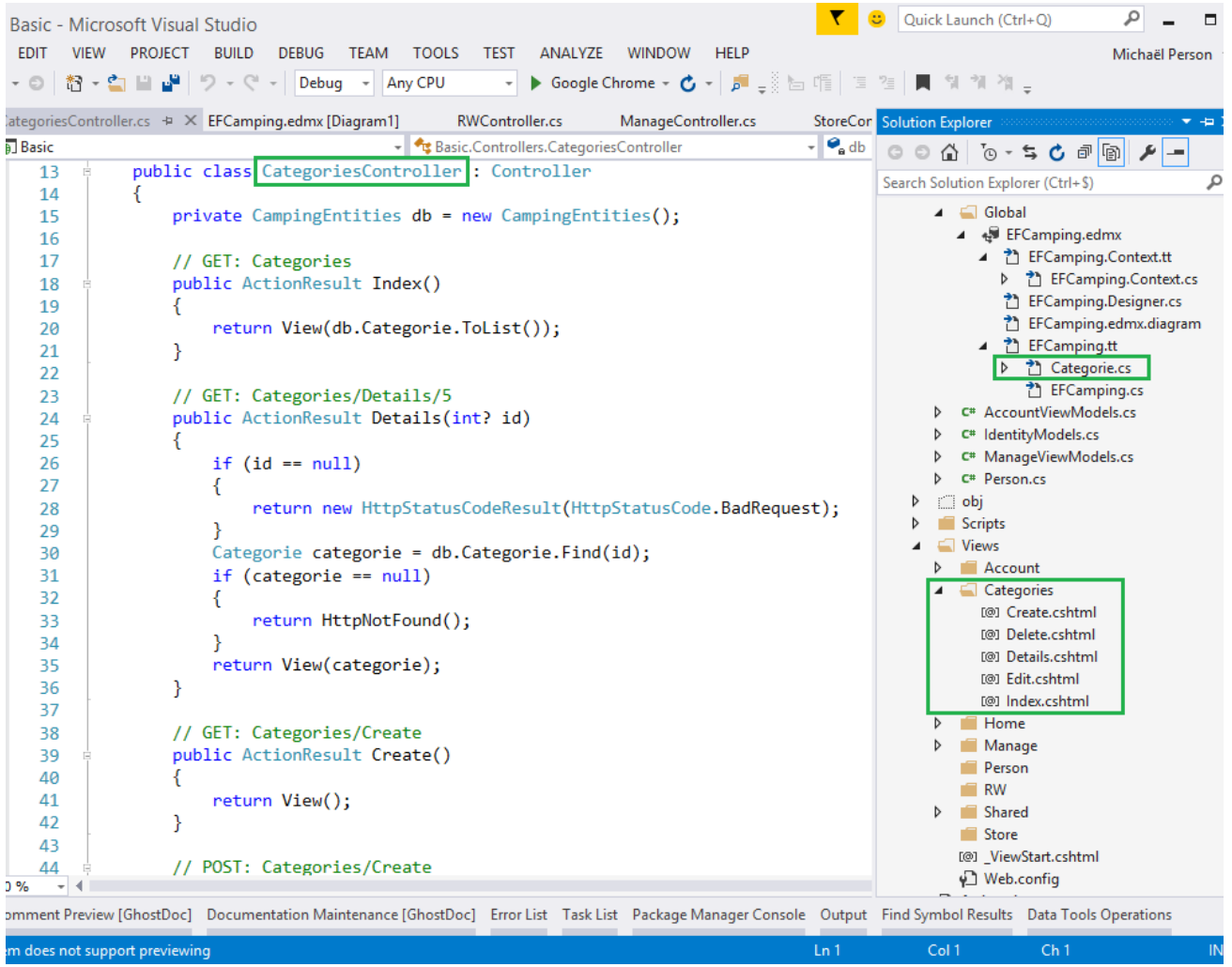
Aucun code fonctionel.

```
}

// 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]
ublic 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 connait 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.Lengts</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 coooooool";
string title = "Syntaxe de Razor";
int number = 1558;
var stringBuilder = new System.Text.StringBuilder();
```

- **Chaines de caractères**

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

Pour les textes avec les guillemets, vous devez utiliser l’opérateur de chaine 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

- **<u>Commentaires</u>**

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 *@
 }
```