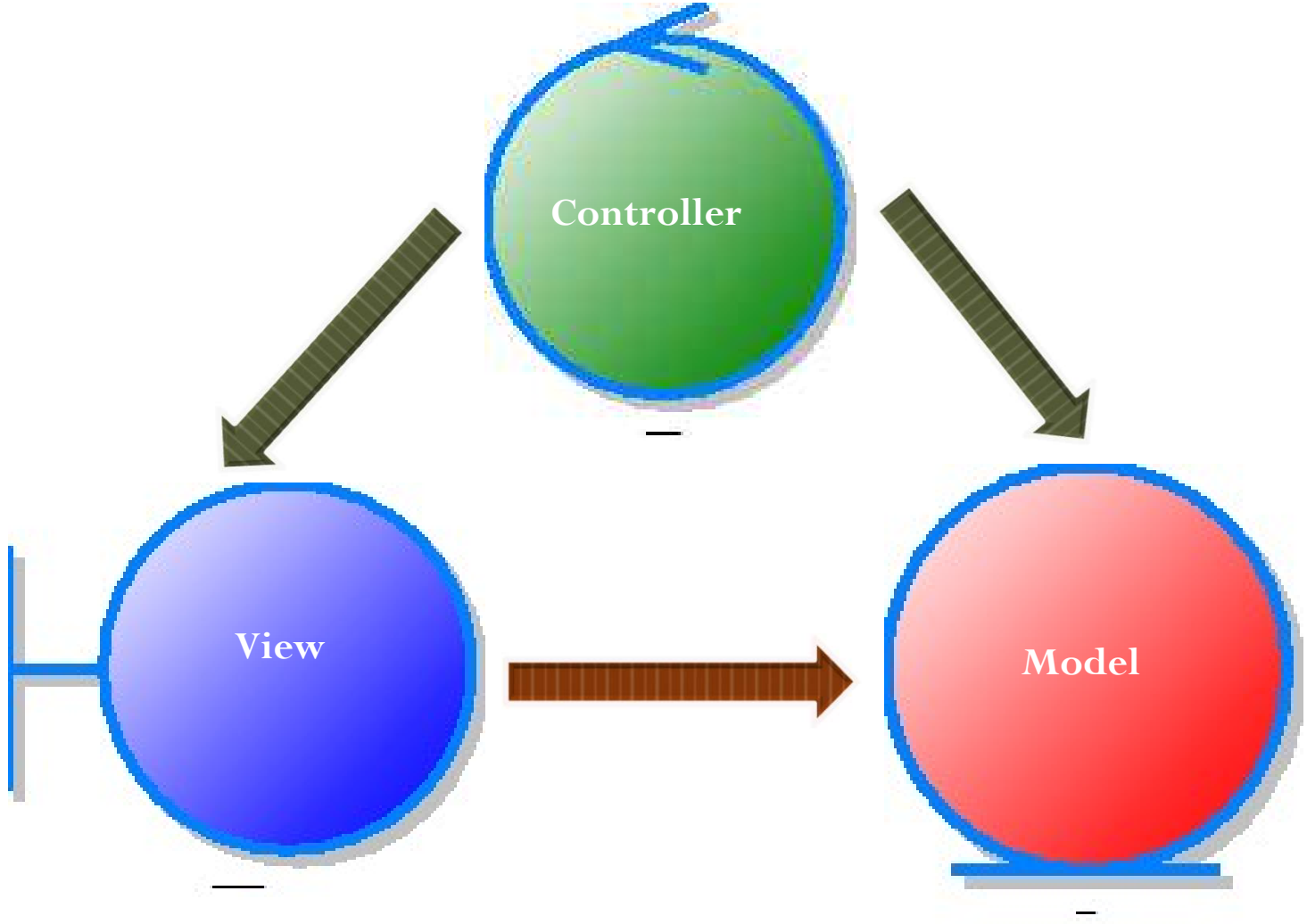
ASP.NET MVC

ASP.NET MVC



Créer une application ASP.Net

- Créer un projet Application Web ASP.NET MVC 5 (ou 4)

Les contrôleurs

- Rôle
  - Le contrôleur reçoit les requêtes du client
  - Prépare le modèle de données pour la vue.
  - Sélectionne et envoie la vue adéquate au client.
- Un contrôleur hérite de la classe Controller.
- Le nom de la classe du contrôleur doit se terminer par le suffixe « Controller ».

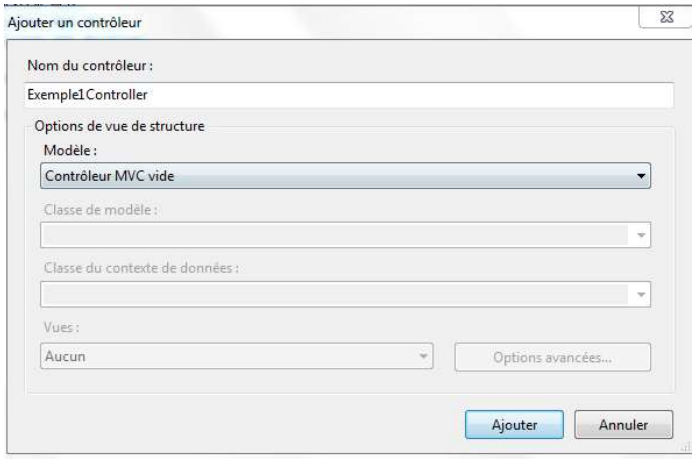
Les Vues

- Une vue est un fichier template utilisé par le moteur ASP.NET MVC pour générer dynamiquement la réponse HTML

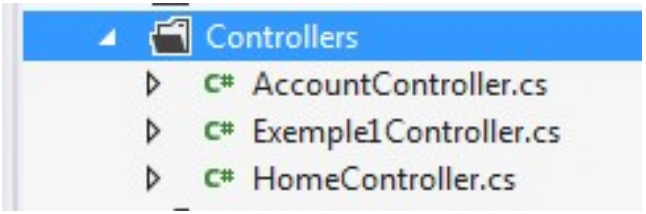
Atelier

1- Créer un projet Application web ASP.NET MVC

2- Créer un contrôleur
(Exemple1Controller, Modèle: Contrôleur MVC vide)



Un nouveau fichier nommé Exemple1Controller.cs est créé dans dossier « Controllers » qui contient la classe Exemple1Controller



Ajouter la méthode suivante dans le contrôleur:

```
public string Message()
    {
        return
"<h1>Bonjour</h1>";

    }
```

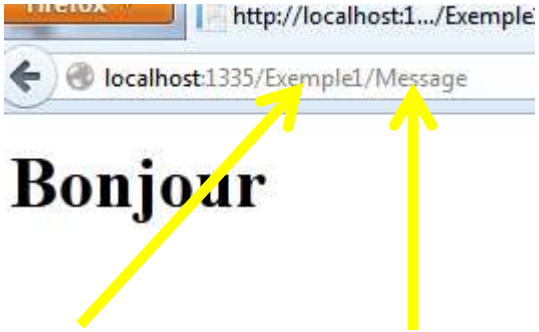
```
public class Exemple1Controller :
Controller
    {//

 // GET: /Exemple1/

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

Modifier la méthode (action) comme suit:

```
public string  Index()
        {
           return "<h1>Action par défaut</h1>";
        }
```

Tester Le contrôleur Exemple1 dans le navigateur.

Contrôleur, l'action par défaut Index est appelée



Contrôleur

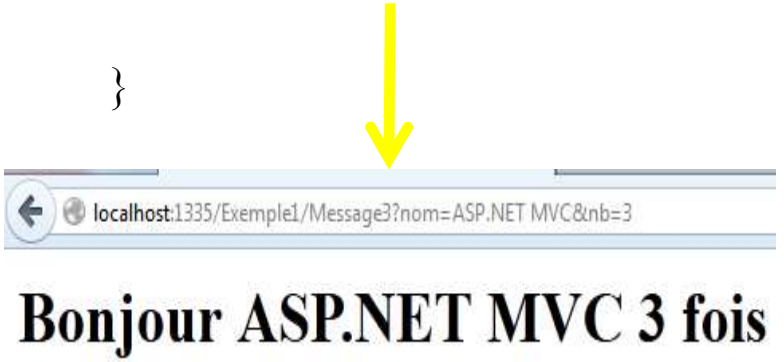
Action

Mapping par défaut d'une URL ASP.NET MVC: **/Contrôleur/Action/Paramètres**

Ajouter les deux actions suivantes :

```
public string Message2(string id)
        {
            return "<h2>Bonjour " + id +
"</h2>";

        }
```


```
 public string Message3(string nom , int
nb)
        {
            return "<h2>Bonjour " + nom +
" " + nb +  " fois</h2>";

        }
```

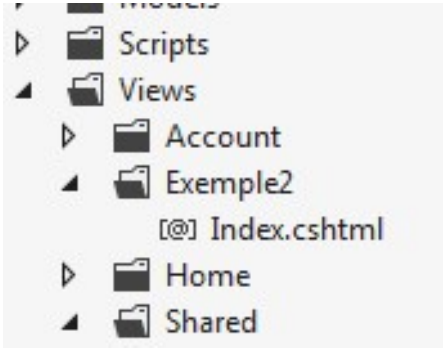


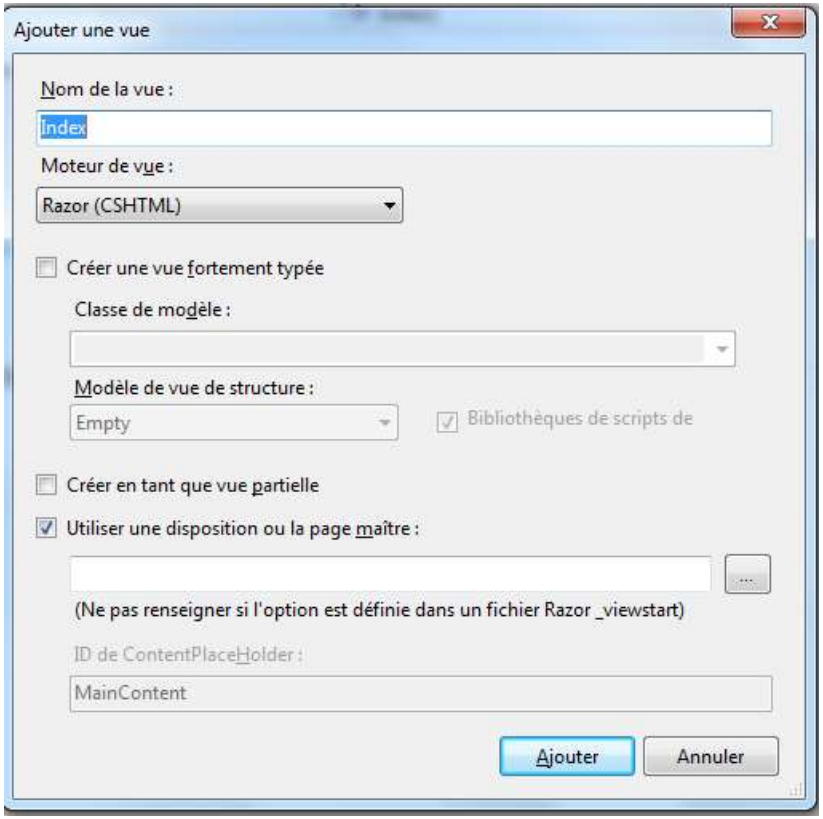
3- créer une vue

- Ajouter un contrôleur nommé exemple2
- Cliquer sur l'action Index avec le bouton droit et sélectionner la commande « ajouter une vue »

Dans le dossier Views, un nouveau sous dossier qui porte le nom du contrôleur (Exemple2) est créé.

Dans le dossier Exemple2, une vue qui porte le nom de l'action Index (.cshtml) est créée.



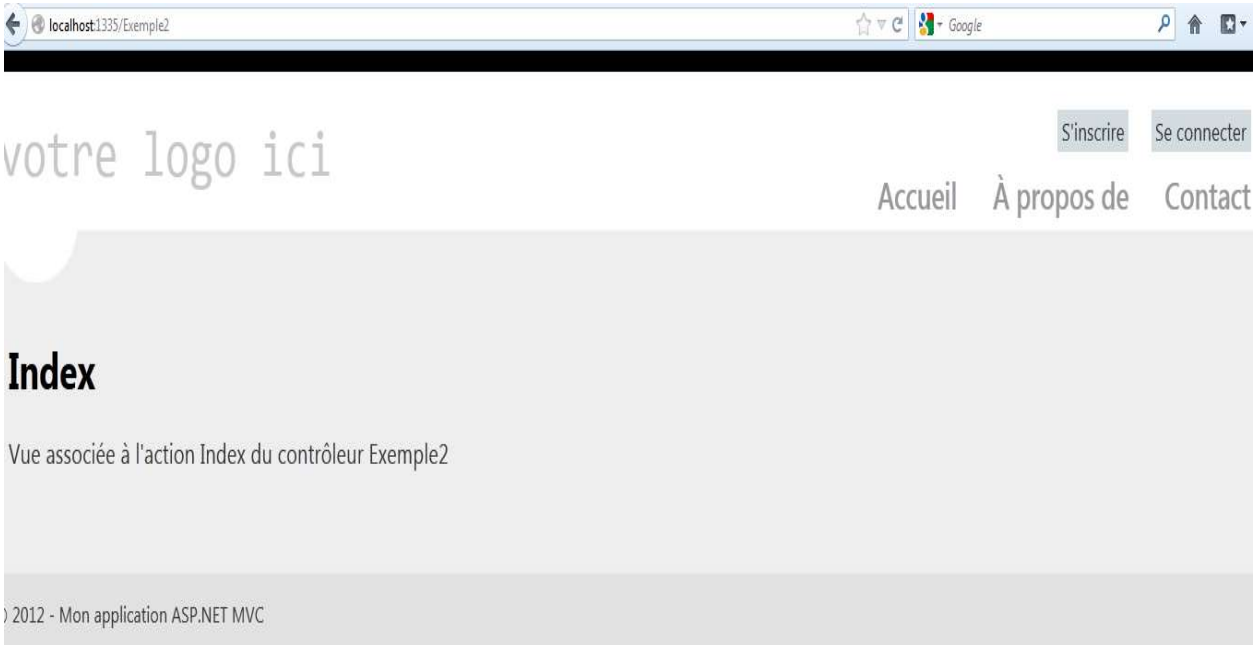


Index.cshtml | Exemple2Controller.cs

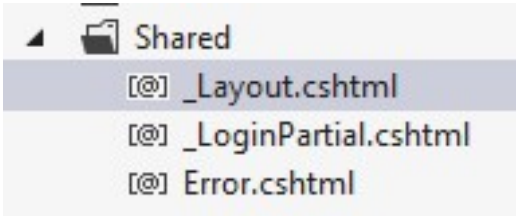
```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>
```

Ajouter le code suivant dans la page Index.cshtml

Vue associée à l'action Index du contrôleur Exemple2 .



Le contenu du dossier shared est partagé par toutes les vues



Le contenu des vues est affiché par @RenderBody()

```
<div id="body">
    @RenderSection("featured", required: false)
    <section class="content-wrapper main-content clear-fix">
        @RenderBody()
    </section>
</div>
```

La page _Layout.cshtml est le modèle utilisé par toutes les vues.

Supprimer le contenu « - Mon application ASP.NET MVC » de la balise tiltle

```
<meta charset="utf-8" />
<title>@ViewBag.Title </title>
<link href="~/favicon.ico" rel="shortcut i
<meta name="viewport" content="width=devic
```

4- Passer des données du contrôleur vers la vue

Ajouter dans le contrôleur Exemple2 l'action suivante:

```
public ActionResult Message3(string
nom, int nb)
        {

            // ViewBag est un objet
dynamique permet de passer des données
à une vue
            ViewBag.message = "Bonjour
" +  nom;
            ViewBag.nombre = nb;
            return View();

        }
```

- Ajouter une vue associé à l'action Message3
- Ajouter le code suivant, dans la vue nouvellement créée

```
<ul>
  @for (int i=0; i < ViewBag.nombre; i++) {
    <li>@ViewBag.message</li>
  }
</ul>
```

Le modèle

Le modèle est constitué de classes POCO (Plain Old CLR Objects

Exemple:

```
public class Photo
    {
        //Le nom de l'identifiant doit être soit
        //ID soit NomClasseID
        public int PhotoID { get; set; }
public string Titre { get; set; }
public string Description { get; set; }
public DateTime DateCreation { get; set; }
        public string Proprietaire { get; set; }
public byte[] Fichier { get; set; }
        //Propriété de navigation
        //virtual pour activer le chargement différé (Lazy loading)
        public virtual ICollection<Commentaire> Commentaires { get; set; }



    }
```

Les annotations

- Annotations d'affichage et d'édition

```
[DisplayName("Date de création")]
        [DataType(DataType.Date)]

[DisplayFormat(DataFormatString="{0:dd/MM/yy}",ApplyFormatInEditMode=true)]
        public DateTime DateCreation { get; set; }
```

- Annotations de validation

```
public class Personne
{
    public int PersonnID { get; set; }

    [Required(ErrorMessage=“Entrez un nom")]
    public string Nom { get; set; }

    [Range(15, 40)]
    public int Age{ get; set; }

    [Required]
    [DataType(DataType.EmailAddress)]
    public string AdresseEmail{ get; set; }
}
```

Ajouter le contexte Entity Framework

- Créer une classe qui hérite de DbContexte (System.Data.Entity)

```
public class PartagePhotoDB:DbContext
    {
        public DbSet<Photo> Photos { get; set; }
        public DbSet<Commentaire> Commentaires { get; set; }
    }
```

- Lors de la première exécution de l'application EF cherchera une chaine de connexion nommée **PartagePhotoDB** pour créer la base de données, s'il n'existe pas il génère une connexion par défaut.

Contrôleurs

Les actions

- Une action est une méthode publique qui retourne un objet de type ActionResult ou un type dérivé.
- Types dérivés de ActionResult
  - ViewResult: pour retourner une vue.
  - PartialViewResult: représente une vue partielle (une partie d'une page html) qui peut être utilisée dans plusieurs vues de l'application.
  - RedirectToRouteResult: redirige le navigateur vers une autre action RedirectResult:redirection vers une URL dans l'application ou une URL externe.
  - ContentResult: retourne des données au navigateur au format texte , XML, JSON

Le contrôleur Categories avec l'action Index

```
public class CategoriesController : Controller
    {
        //Création du contexte de données
        restoEntities db = new restoEntities();
        //Action Index
        public ActionResult Index()
        {  /*Le contrôleur passe le modèle (categories) à
            * la vue nommée Index.cshtml (nom de l'action),
stockée dans le dossier views/Categories.
View() retourne un ActionResult de type ViewResult
            */
           return View(db.categories.ToList());
            //On peut aussi fournir explicitement le nom de la
vue
            return View("Index", db.categories.ToList());
        }
}
```

L’action Details

```
//int? : type nullable (int + null)
        public ActionResult Details(int? ID)
        {

            if (ID==null)
                return HttpNotFound();
            // Utiliser une requête LINQ
           /* var cat = (from c in db.Categories
                       where c.ID == ID
                       select c).FirstOrDefault();*/
            //Utilise la méthode Find qui en paramètre
            //la valeur d'une clé primaire
            var cat = db.Categories.Find(ID);

            if (cat == null)
                return HttpNotFound();

            return View(cat);
        }
```

L’action Creer

```
/* Par défaut [HttpGet] appelée lors de l’affichage du formulaire*/
public ActionResult Creer()
        {
            return View(new Categorie());
        }

[HttpPost] /* appelée lors de l’envoi du formulaire (envoyé par méthode POST)*/
[ValidateAntiForgeryToken]
public ActionResult Creer(Categorie cat)
{// On doit tester la validité des données envoyées
    if (ModelState.IsValid)
    {
        db.Categories.Add(cat);
        db.SaveChanges();
        /* RedirectToAction retourne un ActionResult
         * det Type RedirectToRouteResult
         */
        return RedirectToAction("Index");
    }
    return View(cat);

}
```

L'action Modifier

```
public ActionResult Modifier(int? ID)
{    if (ID==null)
        /* HttpNotFound() retourne un
ActionResult de type HttpNotFoundResult
         */
        return HttpNotFound();
    var cat = db.Categories.Find(ID);
    if (cat == null)
        return HttpNotFound();
    return View(cat);}
[HttpPost]
public ActionResult Modifier(Categorie
c)
{
    if (ModelState.IsValid)
      {/* Cette linge indique que c est un
   objet (une entée)
         * existante qui est dans l'état
   modifié, l'état est
         * défini par la constante
   EntityState.Modified
         * L'appel à SaveChanges
   déclenchera dans ce cas un Update
         */
   /* La méthode Entry (de la classe
   DbContext) retourne, une entité
         * de type DbEntityEntry, attachée
   au contexte de données db et on
         * modifie l'état de l'entité à
   l'aide de sa propriété State.
         */
        db.Entry(c).State =
EntityState.Modified;
        db.SaveChanges();
        return
RedirectToAction("Index");
    }
    return View(c);
}
```

L'action Supprimer

```
//Suppression
public ActionResult Supprimer(int? ID)
{
    // Il faut ajouter les tests sur
null


    return View(db.Categories.Find(ID));
}
/* L'attribut ActionName définit le nom
de l'action, ainsi l'action
 * Supprimer est associée à la méthode
ConfirmerSuppression
 */
[HttpPost , ActionName("Supprimer")]
public ActionResult
ConfirmerSuppresion(Categorie cat)
{// Il faut ajouter les tests sur null
    // Etape 1: recherche de l'entité
cat dans le contexte de données
    Categorie categorie = (from cc in
db.Categories
    where cc.ID == cat.ID
    select cc).FirstOrDefault();
     // Etape 2: suppression de l'entité
     db.Categories.Remove(categorie);
    //Ou bien nous pouvons aussi
remplacer les étapes 1 et 2 par:
     //db.Entry(cat).State =
EntityState.Deleted;
     //
     db.SaveChanges();
     return RedirectToAction("Index");
}
```

Actions filles

- L'annotation [ChildActionOnly] permet de définir une action fille qui retourne une partie du contenu d'une vue, une action fille peut être appelée dans une vue par le helper Html.Action()

Les filtres

Actions et filtres

Reqête

MVC crée une instance du contrôleur

Pre-execution
Code du filtre

Action is Executed

Post-execution
Code du filtre

Le modèle est combine avec la vue

Réponse HTML

Filtres de sécurité

- Authorize
- ValidateAntiForgeryToken
- RequireHttps

Les vues

- Les moteurs de vues
- Syntaxe Razor
- Les Html Helpers
  - Action
  - Affichage
  - Edition
  - Validation

Les moteurs de vue Asp.Net MVC

- Razor:depuis la version Asp.Net MVC 3
- ASPX: moteur de vue par défaut pour les versions Asp.net MVC 1 et 2
- NHaml: version .Net du moteur de vues Haml utilisé par Ruby On Rails.
- Spark:moteur de vue utilisé aussi par le framework MonoRail (un framework basé sur Asp.Net et inspiré par Ruby On Rails).

Syntaxe Razor

- Le symbole @ différencie le code C# du code HTML dans une vue.
- @@: affiche le symbole @
- @: définit une ligne de texte dans le code C#, pour plusieurs lignes il faut utiliser <text> </text>
- @* commentaire razor *@
- Par défaut Razor encode les chaînes de caractères avant de les envoyer au navigateur, par exemple si @Model.Categorie contient " Salade ", alors il remplace les caractères < et > par '<' et '>'.
- Pour générer les chaînes sans encodage HTML, il faut utiliser @Html.Raw(Model.Categorie)

Exemples

```
@* Commentaire Razor *@


<span>
    Prix TTC : @(Model.Prix * 1.2)
</span>

@if (Model.Count > 15)
{
    <ol>
        @foreach(var item in Model)
   {
            <li>@item.Nom</li>
   }
    </ol>
}
```

Lier une vue au modèle

- Une collection d'entités

```
@model IEnumerable<app3.Models.Categorie>
@foreach (var item in Model) {
    <div>
     Nom:    @item.libelle
    </div>
    }
```

- Une entité
  - `@model app3.Models.Categorie`
  - `Libellé: @Model.libelle`

Les Html Helpers
Actions

- @Html.ActionLink: génère un élément <a>

```
@Html.ActionLink("Afficher Détails", "Details", new { id=1 })
```



```
<a href="/Plats/Details/1">Afficher Détails</a>
```

- @Html.Action: génère un lien (uniquement la valeur de href dans le cas de ActionLink)

```
@Html.Action("Details", new { id=1 })
```

```
Plats/Details/1
```

Affichage

- **Html.DisplayNameFor():** affiche la légende définie par l'annotation [DisplayName] dans le modèle si elle est définie, sinon ce helper affiche le nom de l'attribut.

```
@Html.DisplayNameFor(model => model.DateCreation)
```


Date de création

- **Html.DisplayFor():** affiche la valeur de la propriété en prenant en compte éventuellement les annotations définies dans le modèle.

```
@Html.DisplayFor(model => model.DateCreation)
```


25/11/2013

Le helper Html.BeginForm()

Html.BeginForm(): génère l'élément Html form, les attributs du formulaire peuvent être passés en arguments du helper.

```
@using (Html.BeginForm()) {
    /* Champs du formulaire */
}
```

```
<form action="/Categories/Creer" method="post"></form>
```

```
@using (Html.BeginForm("Creer", "Plat", FormMethod.Post,
new { enctype = "multipart/form-data" }))
{
    // Champs du formulaire
}
```

```
<form action="/Plat/Creer" enctype="multipart/form-
data" method="post"></form>
```

Les helpers d'édition

```
@Html.LabelFor(model => model.DateCreation)
```



```
<label for="DateCreation">  Date de création</label>
```

- LabelFor génère un élément <label> et prend en compte l'annotation DisplayNameFor
- EditorFor génère l'élément adéquat selon le type de la propriété et prend en compte l'annotation DataType,
- Si la vue est liée à un modèle EditorFor remplit le champ généré avec la valeur adéquate

```
@Html.EditorFor(model => model.DateCreation)
```



```
<input type="date" name="DateCreation">
```

Les helpers de validation

- **Html.ValidationSummary()**

```
@Html.ValidationSummary()
```

```
<ul>
  <li>Entrez votre nom</li>
  <li>Entrez une adresse email valide</li>
</ul>
```

- **Html.ValidationMessageFor ()**

```
@Html.ValidationMessageFor(model => model.Email)
```

Entrez une adresse email valide

- Les helpers de validation utilisent les annotations de validation définies dans le modèle
- Dans un contrôleur la propriété ModelState.IsValid permet de vérifier si l'utilisateur a saisi des données valides.

Les vues partielles

- Une vue partielle est un bloc de code que nous pouvons insérer dans d'autres vues.
- Par convention le nom d'une vue commence par un souligné « _ »
- Vues partielles fortement typées et vues partielles dynamiques:
  - Créez une vue partielle fortement typée si la vue utilise toujours le même modèle, et une vue partielle dynamique dans le cas contraire.
- Utiliser une vue partielle
  - Html.Partial(_NomVuePartielle): passe le modèle de la vue parente à la vue partielle.
  - Html.Action(): permet de passer un modèle différent de celui de la page parente à la vue partielle.

Les annotations avec EF Model First et Database First

Nous allons ajouter des annotations à la classe Intervenant générée automatiquement par l'Entity Framework:

```
namespace BdConferenceApp.Models
{
    using System;
    using System.Collections.Generic;

    public partial class Intervenant
    {
public int IntervenantID { get; set; }
        public string Nom { get; set; }
        public string AdresseEmail { get; set; }


    }
}
```

- Pour ajouter des annotations à la classe Intervenant, il faut créer une classe partielle Intervenant et une classe qui doit contenir les annotations souhaitées (IntervenantAnnotations) dans le même espace de noms que la classe partielle générée par l'EF.
- Ensuite, il faut associer la classe IntervenantAnnotations à la classe Intervenant à l'aide de l'annotation MetaDataType

```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
namespace BdConferenceApp.Models
{/* Classe de définition des métadonnées (annotations )
     * pour Intervenant
*/
    public class IntervenantAnnotations
    {
        [Required]
        [DisplayName("Nom de l'intervenant")]
        public Object Nom { get; set; }

        [DataType(DataType.EmailAddress)]
        public Object AdresseEmail { get; set; }
 }

/* Définir les métadonnées de la classe Intervenant
 */[MetadataType(typeof(IntervenantAnnotations))]
    public partial class Intervenant { }


   public class SessionMetaDonnees
   {[Required]
       [DataType(DataType.MultilineText)]
       public Object Description { get; set; }
   }}
```