

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.

On the left, a browser window shows the rendered HTML source of a page titled "Index". The HTML includes a form with a text input field labeled "Nom:" and a "Create" submit button. The form's action is set to `/RW/Index` and its method is `post`.

On the right, a code editor window shows the C# code for the `Index.cshtml` view. The code uses the `Html.BeginForm()` helper to generate the form. The code is as follows:

```
1 @model Basic.Models.Person
2
3 @{
4     ViewBag.Title = "Index";
5 }
6
7 <h2>Index</h2>
8
9
10 @using (Html.BeginForm())
11 {
12     @:Nom: <input type="text" id="Nom" name="Nom" />
13     @:<input type="submit" value="Create" />
14 }
15
```

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

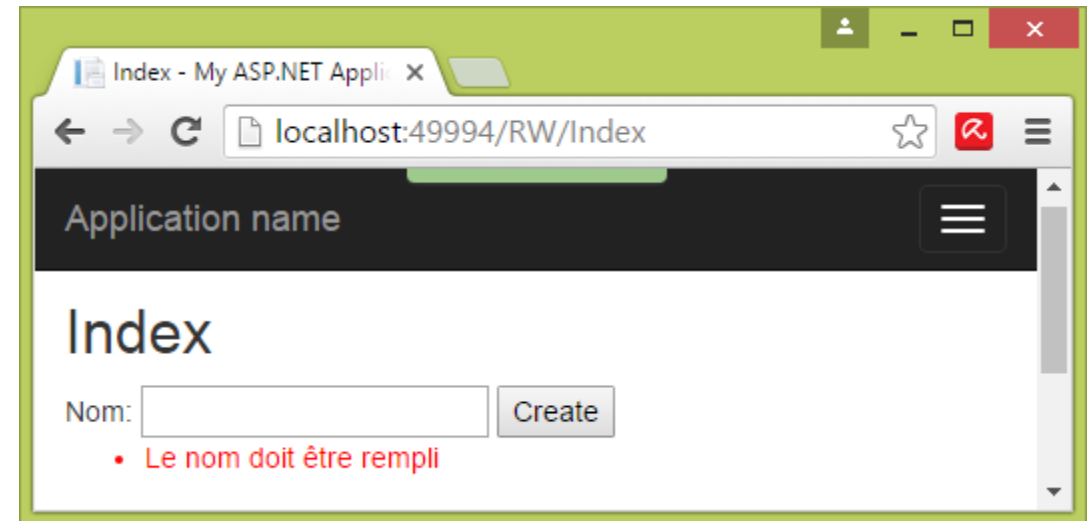
Ce helper permet d'afficher toutes les erreurs de validations du Model dans une liste.

```
<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
 world")

➔ <textarea cols="20" id="text" name="text" rows="2">hello
 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

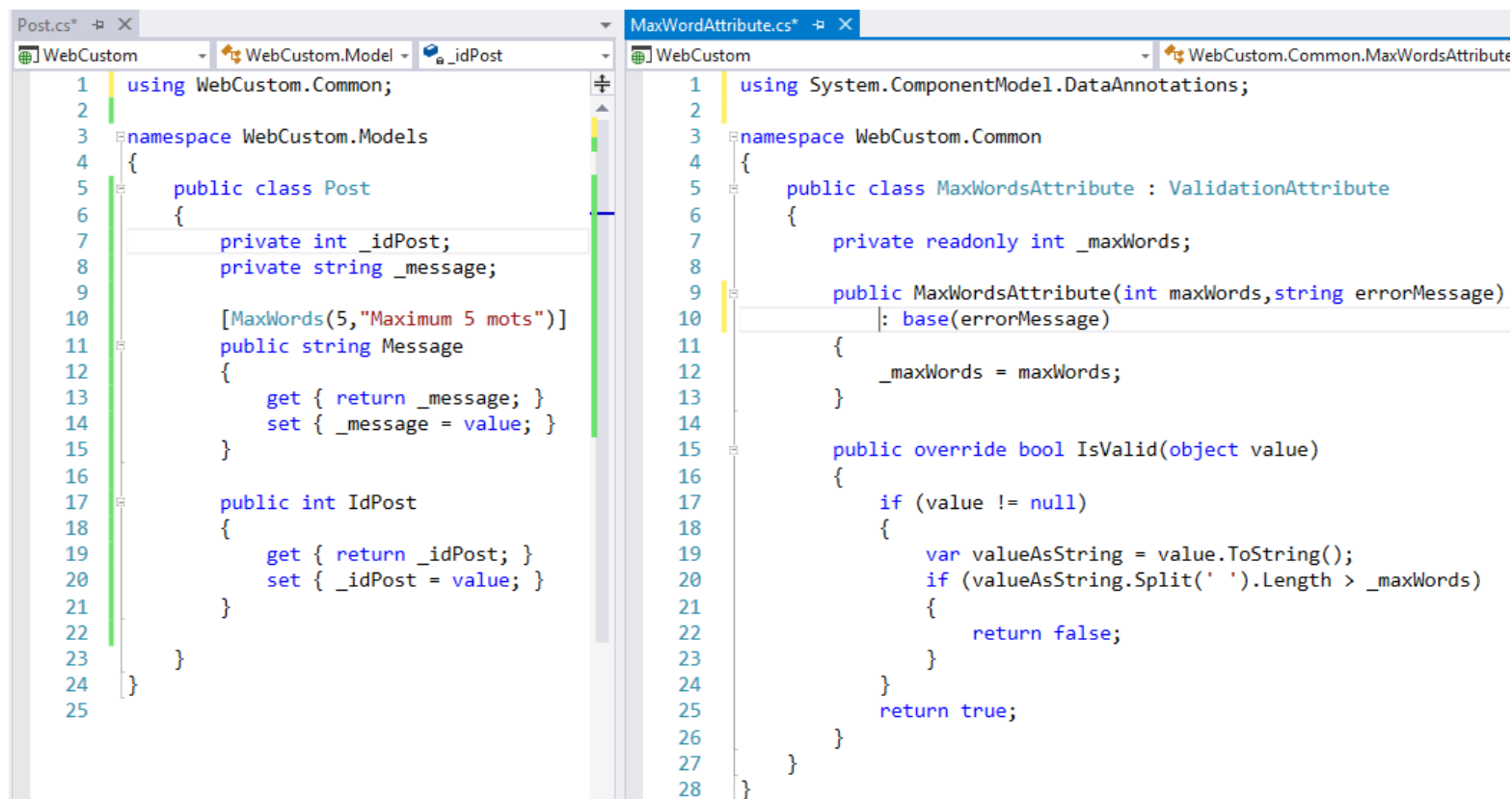
Si nous avons besoin d'ajouter des contraintes business, nous pouvons passer par les *Custom Attribute*

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; }
}
```