eBook Gratuit

APPRENEZ asp.net-mvc

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#asp.net-mvc

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: asp-net-mvc

It is an unofficial and free asp.net-mvc ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official asp.net-mvc.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec asp.net-mvc

Remarques

Le modèle d'architecture MVC (Model-View-Controller) sépare une application en trois composants principaux: le modèle, la vue et le contrôleur. Le framework ASP.NET MVC offre une alternative au modèle ASP.NET Web Forms pour la création d'applications Web. Le framework ASP.NET MVC est un framework de présentation léger et hautement testable qui (comme avec les applications Web Forms) est intégré aux fonctionnalités ASP.NET existantes, telles que les pages maîtres et l'authentification basée sur l'appartenance. Le framework MVC est défini dans l'assembly System.Web.Mvc.

Le framework MVC comprend les composants suivants:

- **Modèles** . Les objets de modèle sont les parties de l'application qui implémentent la logique du domaine de données de l'application. Souvent, les objets de modèle extraient et stockent l'état du modèle dans une base de données. Par exemple, un objet Product peut extraire des informations d'une base de données, les utiliser, puis réécrire les informations mises à jour dans une table Products d'une base de données SQL Server. Dans les petites applications, le modèle est souvent une séparation conceptuelle plutôt que physique. Par exemple, si l'application lit uniquement un jeu de données et l'envoie à la vue, l'application ne comporte pas de couche de modèle physique et de classes associées. Dans ce cas, l'ensemble de données joue le rôle d'objet de modèle.
- **Vues** Les vues sont les composants qui affichent l'interface utilisateur de l'application. En règle générale, cette interface utilisateur est créée à partir des données du modèle. Un exemple serait une vue d'édition d'une table Products qui affiche les zones de texte, les listes déroulantes et les cases à cocher en fonction de l'état actuel d'un objet Product.
- **Contrôleurs** . Les contrôleurs sont les composants qui gèrent l'interaction de l'utilisateur, travaillent avec le modèle et sélectionnent finalement une vue à afficher qui affiche l'interface utilisateur. Dans une application MVC, la vue affiche uniquement les informations. le contrôleur gère et répond aux entrées et interactions de l'utilisateur. Par exemple, le contrôleur gère les valeurs de chaîne de requête et transmet ces valeurs au modèle, qui à son tour peut utiliser ces valeurs pour interroger la base de données.

Versions

Version	Version .NET	Date de sortie
MVC 1.0	.NET 3.5	2009-03-13
MVC 2.0	.NET 3.5 / 4.0	2010-03-10
MVC 3.0	.NET 4.0	2011-01-13
MVC 4.0	.NET 4.0 / 4.5	2012-08-15

Version	Version .NET	Date de sortie
MVC 5.0	.NET 4.5	2013-10-17
MVC 5.1	.NET 4.5	2014-01-17
MVC 5.2	.NET 4.5	2014-08-28
MVC 6.0	.NET 4.5	2015-11-18
Core MVC 1.0	.NET 4.5	2016-07-12
Core MVC 1.1	.NET 4.5	2016-11-18

Examples

Bonjour MVC!

ASP.NET MVC est un framework d'applications Web open source. MVC lui-même est un modèle de conception qui est construit autour de trois composants principaux: *model-view-controller* .

Modèle - Les modèles reflètent vos objets métier et permettent de transmettre des données entre les contrôleurs et les vues.

Affichage - Les vues sont les pages qui rendent et affichent les données du modèle à l'utilisateur. Les vues ASP.NET MVC sont généralement écrites à l'aide de la syntaxe Razor.

Controller - Les contrôleurs gèrent les requêtes HTTP entrantes provenant d'un client et retournent généralement un ou plusieurs modèles à une vue appropriée.

Les fonctionnalités ASP.NET MVC:

1. Idéal pour développer des applications complexes mais légères
2. Il fournit un cadre extensible et enfichable qui peut être facilement remplacé et personnalisé. Par exemple, si vous ne souhaitez pas utiliser le moteur Razor ou ASPX View Engine intégré, vous pouvez utiliser tout autre moteur d'affichage tiers ou même personnaliser les moteurs existants.
3. Utilise la conception basée sur les composants de l'application en la divisant logiquement en composants Model, View et Controller. Cela permet aux développeurs de gérer la complexité des projets à grande échelle et de travailler sur des composants individuels.
4. La structure MVC améliore le développement piloté par les tests et la testabilité de l'application, car tous les composants peuvent être conçus sur la base d'interfaces et testés à l'aide d'objets simulés. Par conséquent, ASP.NET MVC Framework est idéal pour les projets avec une grande équipe de développeurs Web.
5. Prend en charge toutes les fonctionnalités ASP.NET existantes telles que l'autorisation et l'authentification, les pages maîtres, la liaison de données, les contrôles utilisateur, les adhésions, le routage ASP.NET, etc.
6. Il n'utilise pas le concept de View State (qui est présent dans ASP.NET). Cela aide à créer

des applications légères et à donner le contrôle total aux développeurs.

Application MVC simple

Nous allons créer une application MVC simple qui affiche les détails de la personne. Créez un nouveau projet MVC à l'aide de Visual Studio. Ajoutez le nouveau modèle nommé dossier *Personne* au modèles comme suit:

```
public class Person
{
    public string Surname { get; set; }
    public string FirstName { get; set; }
    public string Patronymic { get; set; }
    public DateTime BirthDate { get; set; }
}
```

Ajouter un nouveau contrôleur au dossier Contrôleurs:

```
public class HomeController : Controller
{
    //Action Method
    public ActionResult Index()
    {
        // Initialize model
        Person person = new Person
        {
            Surname = "Person_SURNAME",
            FirstName = "Person_FIRSTNAME",
            Patronymic = "Person_PATRONYMIC",
            BirthDate = new DateTime(1990, 1, 1)
        };

        // Send model to View for displaying to user
        return View(person);
    }
}
```

Ajoutez enfin View à */ Views / Home /* folder nommé *Index.cshtml* :

```
@* Model for this view is Person *@
@model Hello_MVC.Models.Person

<h2>Hello @Model.FirstName !</h2>

<div>
    <h5>Details:</h5>
    <div>
        @Html.LabelFor(m => m.Surname)
        @Html.DisplayFor(m => m.Surname)
    </div>
    <div>
        @Html.LabelFor(m => m.FirstName)
        @Html.DisplayFor(m => m.FirstName)
    </div>
    <div>
        @Html.LabelFor(m => m.Patronymic)
        @Html.DisplayFor(m => m.Patronymic)
```

```
    </div>
    <div>
        @Html.LabelFor(m => m.BirthDate)
        @Html.DisplayFor(m => m.BirthDate)
    </div>
</div>
```

Lire Démarrer avec asp.net-mvc en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/769/demarrer-avec-asp-net-mvc

Chapitre 2: ActionResult

Remarques

Un `ActionResult` est le meilleur en tant que point de terminaison Web dans MVC. Vous pouvez accéder à la méthode ActionResult en entrant l'adresse Web appropriée configurée par votre moteur de routage.

Examples

Renvoyer une page de vue

Cette ActionResult renvoie une page de vue Razor. Sous le modèle de routage standard, cette méthode ActionResult est accessible à l' adresse http: // localhost / about / me

La vue sera automatiquement recherchée sur votre site à `~/Views/About/Me.cshtml`

```
public class AboutController : Controller
{
    public ActionResult Me()
    {
        return View();
    }
}
```

Renvoyer un fichier

Un `ActionResult` peut renvoyer `FileContentResult` en spécifiant le chemin d'accès et le type de fichier à partir de la définition d'extension, appelée type MIME.

Le type MIME peut être défini automatiquement en fonction du type de fichier à l'aide de la méthode `GetMimeMapping` ou défini manuellement au format approprié, par exemple "text / plain".

Comme `FileContentResult` nécessite le `FileContentResult` un tableau d'octets en tant que flux de fichiers, `System.IO.File.ReadAllBytes` peut être utilisé pour lire le contenu du fichier en tant que tableau d'octets avant d'envoyer le fichier demandé.

```
public class FileController : Controller
{
     public ActionResult DownloadFile(String fileName)
     {
          String file = Server.MapPath("~/ParentDir/ChildDir" + fileName);
          String mimeType = MimeMapping.GetMimeMapping(path);

          byte[] stream = System.IO.File.ReadAllBytes(file);
          return File(stream, mimeType);
     }
}
```

Retourne un Json

Le résultat de l'action peut renvoyer Json.

1.Retourner Json pour transmettre json dans ActionResult

```
public class HomeController : Controller
{
    public ActionResult HelloJson()
    {
        return Json(new {message1="Hello", message2 ="World"});
    }
}
```

2.Retourner du contenu pour transmettre json dans ActionResult

```
public class HomeController : Controller
{
    public ActionResult HelloJson()
    {
        return Content("Hello World", "application/json");
    }
}
```

Lire ActionResult en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6246/actionresult

Chapitre 3: ActionResult

Examples

Voir résultat

```
public ActionResult Index()
{
    // Renders a view as a Web page.
    return View();
}
```

Les méthodes d'action renvoient généralement un résultat appelé résultat d'action. La classe ActionResult est la classe de base pour tous les résultats d'action. ActionInvoker décide du type de résultat d'action à renvoyer en fonction de la tâche exécutée par la méthode d'action.

Il est possible d'être explicite sur le type à retourner, mais généralement pas nécessaire.

```
public ViewResult Index()
{
    // Renders a view as a Web page.
    return View();
}
```

PartialViewResult

```
public ActionResult PopulateFoods()
{
     IEnumerable<Food> foodList = GetAll();

    // Renders a partial view, which defines a section of a view that can be rendered inside
another view.
    return PartialView("_foodTable", foodVms);;
}
```

Les méthodes d'action renvoient généralement un résultat appelé résultat d'action. La classe ActionResult est la classe de base pour tous les résultats d'action. ActionInvoker décide du type de résultat d'action à renvoyer en fonction de la tâche exécutée par la méthode d'action.

Il est possible d'être explicite sur le type à retourner, mais généralement pas nécessaire.

```
public PartialViewResult PopulateFoods()
{
    IEnumerable<Food> foodList = GetAll();

    // Renders a partial view, which defines a section of a view that can be rendered inside
another view.
     return PartialView("_foodTable", foodVms);
}
```

RedirectResult

```
public ActionResult Index()
{
    //Redirects to another action method by using its URL.
    return new RedirectResult("http://www.google.com");
}
```

Les méthodes d'action renvoient généralement un résultat appelé résultat d'action. La classe ActionResult est la classe de base pour tous les résultats d'action. ActionInvoker décide du type de résultat d'action à renvoyer en fonction de la tâche exécutée par la méthode d'action.

Il est possible d'être explicite sur le type à retourner, mais généralement pas nécessaire.

```
public RedirectResult Index()
{
    //Redirects to another action method by using its URL.
    return new RedirectResult("http://www.google.com");
}
```

RedirectToRouteResult

```
public ActionResult PopulateFoods()
{
    // Redirects to another action method. In this case the index method
    return RedirectToAction("Index");
}
```

Les méthodes d'action renvoient généralement un résultat appelé résultat d'action. La classe ActionResult est la classe de base pour tous les résultats d'action. ActionInvoker décide quel type d'action doit être renvoyé en fonction de la tâche exécutée par la méthode d'action.

Il est possible d'être explicite sur le type à retourner, mais généralement pas nécessaire.

```
public RedirectToRouteResult PopulateFoods()
{
   // Redirects to another action method. In this case the index method
   return RedirectToAction("Index");
}
```

Si vous souhaitez rediriger vers une autre action avec un paramètre, vous pouvez utiliser la surcharge RedirectToAction :

```
public ActionResult SomeActionWithParameterFromThisController(string parameterName)
{
   // Some logic
}
.....................
.....................
.....................
return RedirectToAction("SomeActionWithParameterFromThisController", new { parameterName =
parameter });
```

ContentResult

```
public ActionResult Hello()
{
    // Returns a user-defined content type, in this case a string.
    return Content("hello world!");
}
```

Les méthodes d'action renvoient généralement un résultat appelé résultat d'action. La classe ActionResult est la classe de base pour tous les résultats d'action. ActionInvoker décide du type de résultat d'action à renvoyer en fonction de la tâche exécutée par la méthode d'action.

Il est possible d'être explicite sur le type à retourner, mais généralement pas nécessaire.

```
public ContentResult Hello()
{
    // Returns a user-defined content type, in this case a string.
    return Content("hello world!");
}
```

Vous pouvez en savoir plus à ce sujet ici: Asp.Net Mvc: ContentResult vs. string

JsonResult

```
public ActionResult LoadPage()
{
    Student result = getFirst();

    //Returns a serialized JSON object.
    return Json(result, JsonRequestBehavior.AllowGet);
}
```

Les méthodes d'action renvoient généralement un résultat appelé résultat d'action. La classe ActionResult est la classe de base pour tous les résultats d'action. ActionInvoker décide du type de résultat d'action à renvoyer en fonction de la tâche exécutée par la méthode d'action.

Il est possible d'être explicite sur le type à retourner, mais généralement pas nécessaire.

```
public JsonResult LoadPage()
{
    Student result = getFirst();

    //Returns a serialized JSON object.
    return Json(result, JsonRequestBehavior.AllowGet);
}
```

Lire ActionResult en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6487/actionresult

Chapitre 4: ActionResult

Syntaxe

- // La méthode ActionResult renvoie une instance dérivée de ActionResult. Vous êtes en mesure de créer une méthode d'action pouvant renvoyer toute instance encapsulée dans le type ActionResult approprié.
- // Les types de retour ActionResult intégrés sont:
- Vue(); // ViewResult rend une vue en tant que page Web
- Vue partielle(); // PartialViewResult rend une vue partielle, qui peut être utilisée dans le cadre d'une autre vue.
- Réorienter(); // RedirectResult redirige vers une autre méthode d'action en utilisant son URL.
- RediectToAction (); RedirectToRoute (); // RedirectToRouteResult redirige vers une autre méthode d'action.
- Contenu(); // ContentResult renvoie un type de contenu défini par l'utilisateur.
- Json (); // JsonResult renvoie un objet JSON sérialisé.
- JavaScript (); // JavaScriptResult renvoie un script pouvant être exécuté côté client.
- Fichier(); // FileResult renvoie une sortie binaire à écrire dans la réponse.
- // EmptResult représente une valeur de retour utilisée si la méthode d'action doit renvoyer un résultat nul.

Examples

Méthodes d'action

Lorsque l'utilisateur entre une URL, par exemple: http://example-website.com/Example/HelloWorld , l'application MVC utilisera les règles de routage pour analyser cette URL et extraire le sous-chemin qui déterminera le contrôleur, l'action et les paramètres possibles. Pour l'URL ci-dessus, le résultat sera / Example / HelloWorld, qui, par défaut, donne les résultats des règles de routage, fournit le nom du contrôleur: Exmaple et le nom de l'action: HelloWorld.

```
public class ExampleController: Controller
{
    public ActionResult HelloWorld()
    {
        ViewData["ExampleData"] = "Hello world!";
        return View();
    }
```

```
}
```

La méthode ActionResult ci-dessus "HelloWorld" rendra la vue appelée HelloWorld, où nous pourrons alors utiliser les données de ViewData.

Mappage des paramètres de la méthode d'action

S'il y avait une autre valeur dans l'URL comme: / Example / ProcessInput / 2, les règles de routage menaceraient le dernier nombre en tant que paramètre passé dans l'action ProcessInput de l'exemple de contrôleur.

```
public ActionResult ProcessInput(int number)
{
    ViewData["OutputMessage"] = string.format("The number you entered is: {0}", number);
    return View();
}
```

Appel d'une actionResult dans une autre actionResult

Nous pouvons appeler un résultat d'action dans un autre résultat d'action.

```
public ActionResult Action1()
{
    ViewData["OutputMessage"] = "Hello World";
    return RedirectToAction("Action2","ControllerName");
    //this will go to second action;
}


public ActionResult Action2()
{
    return View();
    //this will go to Action2.cshtml as default;
}
```

Lire ActionResult en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6635/actionresult

Chapitre 5: Aide HTML

Introduction

Les aides HTML sont des méthodes utilisées pour rendre les éléments HTML dans une vue. Ils font partie de l'espace de noms `System.Web.Mvc.HtmlHelper` .

Il existe différents types d'aides HTML:

Standard HTML Helpers : ils permettent de rendre des éléments HTML normaux, par exemple `Html.TextBox()` .

Helpers HTML fortement typés : ces helpers génèrent des éléments HTML basés sur les propriétés du modèle, par exemple `Html.TextBoxFor()` .

Helpers HTML personnalisés : l'utilisateur peut créer une méthode d'assistance personnalisée qui renvoie `MvcHtmlString` .

Examples

Assistant HTML personnalisé - Nom d'affichage

```
/// <summary>
/// Gets displayName from DataAnnotations attribute
/// </summary>
/// <typeparam name="TModel"></typeparam>
/// <typeparam name="TProperty"></typeparam>
/// <param name="htmlHelper"></param>
/// <param name="expression"></param>
/// <returns></returns>
public static MvcHtmlString GetDisplayName<TModel, TProperty>(this HtmlHelper<TModel>
htmlHelper, Expression<Func<TModel, TProperty>> expression)
{
    var metaData = ModelMetadata.FromLambdaExpression(expression, htmlHelper.ViewData);
    var value = metaData.DisplayName ?? (metaData.PropertyName ??
ExpressionHelper.GetExpressionText(expression));
    return MvcHtmlString.Create(value);
}
```

Custom Helper - Bouton de soumission de rendu

```
/// <summary>
/// Creates simple button
/// </summary>
/// <param name="poHelper"></param>
/// <param name="psValue"></param>
/// <returns></returns>
public static MvcHtmlString SubmitButton(this HtmlHelper poHelper, string psValue)
{
    return new MvcHtmlString(string.Format("<input type=\"submit\" value=\"{0}\">", psValue));
```

```
}
```

Liste exhaustive des échantillons HtmlHelper, y compris la sortie HTML

`HtmlHelper.Action()`

- `@Html.Action(actionName: "Index")`
  output: **Le HTML rendu par une méthode d'action appelée `Index()`**
- `@Html.Action(actionName: "Index", routeValues: new {id = 1})`
  output: **Le HTML rendu par une méthode d'action appelée `Index(int id)`**
- `@(Html.Action("Index", routeValues: new RouteValueDictionary(new Dictionary<string, object>{ {"id", 1} })))`
  output: **Le HTML rendu par une méthode d'action appelée `Index(int id)`**
- `@Html.Action(actionName: "Index", controllerName: "Home")`
  output: **le HTML rendu par une méthode d'action appelée `Index()` dans le `HomeController`**
- `@Html.Action(actionName: "Index", controllerName: "Home", routeValues: new {id = 1})`
  output: **Le HTML rendu par une méthode d'action appelée `Index(int id)` dans le `HomeController`**
- `@Html.Action(actionName: "Index", controllerName: "Home", routeValues: new RouteValueDictionary(new Dictionary<string, object>{ {"id", 1} }))`
  output: **Le HTML rendu par une méthode d'action appelée `Index(int id)` dans le `HomeController`**

`HtmlHelper.ActionLink()`

- `@Html.ActionLink(linkText: "Click me", actionName: "Index")`
  sortie: **`<a href="Home/Index">Click me</a>`**
- `@Html.ActionLink(linkText: "Click me", actionName: "Index", routeValues: new {id = 1})`
  sortie: **`<a href="Home/Index/1">Click me</a>`**
- `@Html.ActionLink(linkText: "Click me", actionName: "Index", routeValues: new {id = 1}, htmlAttributes: new {@class = "btn btn-default", data_foo = "bar")`
  sortie: **`<a href="Home/Index/1" class="btn btn-default" data-foo="bar">Click me</a>`**
- `@Html.ActionLink()`
  sortie: **`<a href=""></a>`**

`@HtmlHelper.BeginForm()`

- `@using (Html.BeginForm("MyAction", "MyController", FormMethod.Post, new {id="form1",@class = "form-horizontal"}))`
  output: **`<form action="/MyController/MyAction" class="form-horizontal" id="form1" method="post">`**

HTML Helpers standard avec leurs sorties HTML

Html.TextBox ()

- `@Html.TextBox("Name", null, new { @class = "form-control" })`

output: **<input class="form-control" id="Name"name="Name"type="text"value=""/>**

- `@Html.TextBox("Name", "Stack Overflow", new { @class = "form-control" })`
  output: **<input class="form-control" id="Name"name="Name"type="text" value="Stack Overflow"/>**

Html.TextArea ()

- `@Html.TextArea("Notes", null, new { @class = "form-control" })`
  sortie: **<textarea class="form-control" id="Notes" name="Notes" rows="2" cols="20"></textarea>**
- `@Html.TextArea("Notes", "Please enter Notes", new { @class = "form-control" })`
  output: **<textarea class="form-control" id="Notes" name="Notes" rows="2" cols="20" >Please enter Notes</textarea>**

Html.Label ()

- `@Html.Label("Name","FirstName")`
  output: **<label for="Name"> FirstName </label>**
- `@Html.Label("Name", "FirstName", new { @class = "NameClass" })`
  output: **<label for="Name" class="NameClass">FirstName</label>**

Html.Hidden ()

- `@Html.Hidden("Name", "Value")`
  output: **<input id="Name" name="Name" type="hidden" value="Value" />**

Html.CheckBox ()

- `@Html.CheckBox("isStudent", true)`
  output: **<input checked="checked" id="isStudent" name="isStudent" type="checkbox" value="true" />**

Html.Password ()

- `@Html.Password("StudentPassword")`
  output: **<input id="StudentPassword" name="StudentPassword" type="password" value="" />**

Assistant personnalisé - Bouton de rendu de la radio avec étiquette

```
       public static MvcHtmlString RadioButtonLabelFor<TModel, TProperty>(this
HtmlHelper<TModel> self, Expression<Func<TModel, TProperty>> expression, bool value, string
labelText)
    {
        // Retrieve the qualified model identifier
        string name = ExpressionHelper.GetExpressionText(expression);
        string fullName = self.ViewContext.ViewData.TemplateInfo.GetFullHtmlFieldName(name);

        // Generate the base ID
        TagBuilder tagBuilder = new TagBuilder("input");
        tagBuilder.GenerateId(fullName);
        string idAttr = tagBuilder.Attributes["id"];

        // Create an ID specific to the boolean direction
        idAttr = string.Format("{0}_{1}", idAttr, value);

        // Create the individual HTML elements, using the generated ID
```

```
        MvcHtmlString radioButton = self.RadioButtonFor(expression, value, new { id = idAttr
});
        MvcHtmlString label = self.Label(idAttr, labelText);

        return new MvcHtmlString(radioButton.ToHtmlString() + label.ToHtmlString());
    }
```

Exemple: `@Html.RadioButtonLabelFor(m => m.IsActive, true, "Yes")`

Assistant personnalisé - Sélecteur de date et d'heure

```
public static MvcHtmlString DatePickerFor<TModel, TProperty>(this HtmlHelper<TModel>
htmlHelper, Expression<Func<TModel, TProperty>> expression, object htmlAttributes)
{
    var sb = new StringBuilder();
    var metaData = ModelMetadata.FromLambdaExpression(expression, htmlHelper.ViewData);
    var dtpId = "dtp" + metaData.PropertyName;
    var dtp = htmlHelper.TextBoxFor(expression, htmlAttributes).ToHtmlString();
    sb.AppendFormat("<div class='input-group date' id='{0}'> {1} <span class='input-group-
addon'><span class='glyphicon glyphicon-calendar'></span></span></div>", dtpId, dtp);
    return MvcHtmlString.Create(sb.ToString());
}
```

Exemple:

```
@Html.DatePickerFor(model => model.PublishedDate,  new { @class = "form-control" })
```

Si vous utilisez **Bootstrap.v3.Datetimepicker** Votre JavaScript est comme ci-dessous -

```
$('#dtpPublishedDate').datetimepicker({ format: 'MMM DD, YYYY' });
```

Lire Aide HTML en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/2290/aide-html

Chapitre 6: Annotations de données

Introduction

Nous pouvons ajouter des validations à notre application en ajoutant des annotations de données à nos classes de modèles. Les annotations de données nous permettent de décrire les règles que nous souhaitons appliquer à nos propriétés de modèle, et ASP.NET MVC se chargera de les appliquer et d'afficher les messages appropriés aux utilisateurs.

Examples

Attributs de validation de base utilisés dans ViewModel

Modèle

```
using System.ComponentModel.DataAnnotations;

public class ViewModel
{
    [Required(ErrorMessage="Name is required")]
    public string Name { get; set; }

    [StringLength(14, MinimumLength = 14, ErrorMessage = "Invalid Phone Number")]
    [Required(ErrorMessage="Phone Number is required")]
    public string PhoneNo { get; set; }

    [Range(typeof(decimal), "0", "150")]
    public decimal? Age { get; set; }

    [RegularExpression(@"^\d{5}(-\d{4})?$", ErrorMessage = "Invalid Zip Code.")]
    public string ZipCode {get;set;}

    [EmailAddress(ErrorMessage = "Invalid Email Address")]
    public string Email { get; set; }

    [Editable(false)]
    public string Address{ get; set; }
}
```

Vue

```
// Include Jquery and Unobstructive Js here for client side validation

@using (Html.BeginForm("Index","Home") {

    @Html.TextBoxFor(model => model.Name)
    @Html.ValidationMessageFor(model => model.Name)

    @Html.TextBoxFor(model => model.PhoneNo)
    @Html.ValidationMessageFor(model => model.PhoneNo)
```

```
    @Html.TextBoxFor(model => model.Age)
    @Html.ValidationMessageFor(model => model.Age)

    @Html.TextBoxFor(model => model.ZipCode)
    @Html.ValidationMessageFor(model => model.ZipCode)

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

    @Html.TextBoxFor(model => model.Address)
    @Html.ValidationMessageFor(model => model.Address)

    <input type="submit" value="submit" />
}
```

Manette

```
public ActionResult Index(ViewModel _Model)
{
    // Checking whether the Form posted is valid one.
    if(ModelState.IsValid)
    {
        // your model is valid here.
        // perform any actions you need to, like database actions,
        // and/or redirecting to other controllers and actions.
    }
    else
    {
        // redirect to same action
        return View(_Model);
    }
}
```

Validation à distance

La validation à distance utilisée pour vérifier si le contenu entre dans le contrôle d'entrée est valide ou non en envoyant une requête ajax au serveur pour le vérifier.

Travail

`RemoteAttribute` fonctionne en effectuant un appel AJAX du client vers une action de contrôleur avec la valeur du champ en cours de validation. L'action du contrôleur renvoie ensuite une réponse `JsonResult` indiquant le succès ou l'échec de la validation. Retourner `true` de votre action indique que la validation a réussi. Toute autre valeur indique un échec. Si vous retournez `false` , le message d'erreur spécifié dans l'attribut est utilisé. Si vous retournez autre chose, par exemple une chaîne ou même un entier, le message d'erreur apparaîtra. Sauf si vous avez besoin que votre message d'erreur soit dynamique, il est logique de renvoyer true ou false et de laisser le validateur utiliser le message d'erreur spécifié sur l'attribut.

ViewModel

```
public class ViewModel
{
    [Remote("IsEmailAvailable", "Group", HttpMethod = "POST", ErrorMessage = "Email already
exists. Please enter a different email address.")]
    public string Email{ get; set; }
}
```

Manette

```
[HttpPost]
public JsonResult IsEmailAvailable(string Email)
{
    // Logic to check whether email is already registered or Not.
    var emailExists = IsEmailRegistered();
    return Json(!emailExists);
}
```

Live Demo Fiddle

Vous pouvez transmettre des propriétés supplémentaires du modèle à la méthode du contrôleur à l'aide de la propriété `AdditionalFields` de `RemoteAttribute` . Un scénario typique consisterait à transmettre la propriété ID du modèle dans un formulaire «Modifier», afin que la logique du contrôleur puisse ignorer les valeurs de l'enregistrement existant.

Modèle

```
  public int? ID { get; set; }
  [Display(Name = "Email address")]
  [DataType(DataType.EmailAddress)]
  [Required(ErrorMessage = "Please enter you email address")]
  [Remote("IsEmailAvailable", HttpMethod="Post", AdditionalFields="ID", ErrorMessage = "Email
already exists. Please enter a different email address.")]
  public string Email { get; set; }
```

Manette

```
[HttpPost]
public ActionResult Validate(string email, int? id)
{
    if (id.HasValue)
    {
        return Json(!db.Users.Any(x => x.Email == email && x.ID != id);
    }
    else
    {
        return Json(!db.Users.Any(x => x.Email == email);
    }
}
```

Démo de travail - Champs supplémentaires

Note supplémentaire

Le message d'erreur par défaut est naturellement vague, n'oubliez donc pas de remplacer le message d'erreur par défaut lorsque vous utilisez `RemoteAttribute` .

RequiredAttribute

L'attribut `Required` spécifie qu'une propriété est requise. Un message d'erreur peut être spécifié en utilisant la propriété `ErrorMessage` sur l'attribut.

Ajoutez d'abord l'espace de nom:

```
using System.ComponentModel.DataAnnotations;
```

Et appliquer l'attribut sur une propriété.

```
public class Product
{
   [Required(ErrorMessage = "The product name is required.")]
   public string Name { get; set; }

   [Required(ErrorMessage = "The product description is required.")]
   public string Description { get; set; }
}
```

Il est également possible d'utiliser des ressources dans le message d'erreur pour les applications globalisées. Dans ce cas, le `ErrorMessageResourceName` doit être spécifiée avec la clé des ressources de la classe de ressources (`resx` de fichier) qui doit être paramétré sur le `ErrorMessageResourceType` :

```
public class Product
{
   [Required(ErrorMessageResourceName = "ProductNameRequired",
             ErrorMessageResourceType = typeof(ResourceClass))]
   public string Name { get; set; }

   [Required(ErrorMessageResourceName = "ProductDescriptionRequired",
             ErrorMessageResourceType = typeof(ResourceClass))]
   public string Description { get; set; }
}
```

StringLengthAttribute

L'attribut `StringLength` spécifie la longueur minimale et maximale des caractères autorisés dans un champ de données. Cet attribut peut être appliqué aux propriétés, aux champs publics et aux paramètres. Le message d'erreur doit être spécifié sur la propriété `ErrorMessage` de l'attribut. Les propriétés `MinimumLength` et `MaximumLength` spécifient respectivement le minimum et le maximum.

Ajoutez d'abord l'espace de nom:

```
using System.ComponentModel.DataAnnotations;
```

Et appliquer l'attribut sur une propriété.

```
public class User
{
   // set the maximum
   [StringLength(20, ErrorMessage = "The username cannot exceed 20 characters. ")]
   public string Username { get; set; }

   [StringLength(MinimumLength = 3, MaximumLength = 16, ErrorMessage = "The password must have
between 3 and 16 characters.")]
   public string Password { get; set; }
}
```

Il est également possible d'utiliser des ressources dans le message d'erreur pour les applications globalisées. Dans ce cas, le `ErrorMessageResourceName` doit être spécifiée avec la clé des ressources de la classe de ressources (`resx` de fichier) qui doit être paramétré sur le `ErrorMessageResourceType` :

```
public class User
{
   [StringLength(20, ErrorMessageResourceName = "StringLength",
                             ErrorMessageResourceType = typeof(ResoucesKeys))]
   public string Username { get; set; }

   [StringLength(MinimumLength = 3,
                        MaximumLength = 16,
                        ErrorMessageResourceName = "StringLength",
                        ErrorMessageResourceType = typeof(ResoucesKeys))]
   public string Password { get; set; }
}
```

Attribut de plage

L'attribut `Range` peut décorer des propriétés ou des champs publics et spécifie une plage entre laquelle un champ numérique doit figurer pour être considéré comme valide.

```
[Range(minimumValue, maximumValue)]
public int Property { get; set; }
```

En outre, il accepte une propriété `ErrorMessage` facultative qui peut être utilisée pour définir le message reçu par l'utilisateur lorsque des données non valides sont entrées:

```
[Range(minimumValue, maximumValue, ErrorMessage = "{your-error-message}")]
public int Property { get; set; }
```

Exemple

```
[Range(1,100, ErrorMessage = "Ranking must be between 1 and 100.")]
public int Ranking { get; set; }
```

Attribut RegularExpression

L'attribut `[RegularExpression]` peut décorer des propriétés ou des champs publics et spécifie une expression régulière qui doit correspondre pour que la propriété soit considérée comme valide.

```
[RegularExpression(validationExpression)]
public string Property { get; set; }
```

En outre, il accepte une propriété `ErrorMessage` facultative qui peut être utilisée pour définir le message reçu par l'utilisateur lorsque des données non valides sont entrées:

```
[RegularExpression(validationExpression, ErrorMessage = "{your-error-message}")]
public string Property { get; set; }
```

Exemples)

```
[RegularExpression(@"^[a-z]{8,16}?$", ErrorMessage = "A User Name must consist of 8-16
lowercase letters")]
public string UserName{ get; set; }
[RegularExpression(@"^\d{5}(-\d{4})?$", ErrorMessage = "Please enter a valid ZIP Code (e.g.
12345, 12345-1234)")]
public string ZipCode { get; set; }
```

Comparer l'attribut

L'attribut `Compare` compare deux propriétés d'un modèle.

Le message d'erreur peut être spécifié en utilisant la propriété `ErrorMessage` ou en utilisant des fichiers de ressources.

Pour utiliser l'attribut `Compare` devez `using` pour l'espace de noms suivant:

```
using System.ComponentModel.DataAnnotations;
```

Ensuite, vous pouvez utiliser l'attribut dans votre modèle:

```
public class RegisterModel
{
    public string Email { get; set; }

    [Compare("Email",  ErrorMessage = "The Email and Confirm Email fields do not match.")]
    public string ConfirmEmail { get; set; }
}
```

Lorsque ce modèle est validé, si les valeurs `Email` et `ConfirmEmail` sont différentes, la validation échouera.

Messages d'erreur localisés

Tout comme avec tous les attributs de validation, il est possible d'utiliser des messages d'erreur provenant de fichiers de ressources. Dans cet exemple, le message d'erreur sera chargé à partir du fichier de ressources `Resources` , le nom de la ressource est `CompareValidationMessage` :

```
public class RegisterModel
{
    public string Email { get; set; }
```

```
    ["Email", ErrorMessageResourceType = typeof(Resources),  ErrorMessageResourceName =
"CompareValidationMessage")]
    public string ConfirmEmail { get; set; }
}
```

Eviter les chaînes dans les noms de propriétés

Pour éviter d'utiliser la chaîne pour la valeur de la propriété, dans C # 6+, vous pouvez utiliser `nameof` mot-clé:

```
public class RegisterModel
{
    public string Email { get; set; }

    [Compare(nameof(Email),  ErrorMessage = "The Email and Confirm Email fields do not
match.")]
    public string ConfirmEmail { get; set; }
}
```

Placeholders dans les messages d'erreur

Vous pouvez utiliser des espaces réservés dans vos messages d'erreur. L'espace réservé `{0}` est remplacé par le nom complet de la propriété en cours et `{1}` est remplacé par le nom d'affichage de la propriété associée:

```
public class RegisterModel
{
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Display(Name = "Confirm Email")]
    [Compare("Email",  ErrorMessage = "The '{1}' and '{0}' fields do not match.")]
    public string ConfirmEmail { get; set; }
}
```

Si la validation du modèle échoue, le message d'erreur sera

> Les champs 'Email' et 'Confirm Email' ne correspondent pas.

Attribut de validation personnalisé

Lorsqu'il s'agit de valider certaines règles qui ne sont pas des validations de données génériques, par exemple en vous assurant qu'un champ est requis ou une plage de valeurs, mais qu'elles sont spécifiques à votre logique métier, vous pouvez créer votre propre ***validateur personnalisé*** . Pour créer un attribut de validation personnalisé, il vous suffit d' `inherit` classe **`ValidationAttribute`** et de `override` sa méthode **`IsValid`** . La méthode **`IsValid`** prend deux paramètres, le premier est un `object` nommé en tant que `value` et le second est un `ValidationContext object` nommé en tant que `validationContext` . **`Value`** fait référence à la valeur réelle du champ que votre validateur personnalisé va valider.

Supposons que vous souhaitiez valider un `Email` via un `Custom Validator`

```
public class MyCustomValidator : ValidationAttribute
{
    private static string myEmail= "admin@dotnetfiddle.net";

    protected override ValidationResult IsValid(object value, ValidationContext
validationContext)
    {
        string Email = value.ToString();
        if(myEmail.Equals(Email))
            return new ValidationResult("Email Already Exist");
        return ValidationResult.Success;
    }
}

public class SampleViewModel
{
    [MyCustomValidator]
    [Required]
    public string Email { get; set; }

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

Voici sa démo DotNetFiddle

Modèle EDMx - Annotation des données

Edmx modèle internel

```
public partial class ItemRequest
{
    public int RequestId { get; set; }
    //...
}
```

Ajouter des annotations de données à cela - si nous modifions directement ce modèle, lorsqu'une mise à jour du modèle est effectuée, les modifications sont perdues. alors

Pour ajouter un attribut dans ce cas, 'Obligatoire'

Créez une nouvelle classe - n'importe quel nom

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

//make sure the namespace is equal to the other partial class ItemRequest
namespace MvcApplication1.Models
{
    [MetadataType(typeof(ItemRequestMetaData))]
    public partial class ItemRequest
    {
    }

    public class ItemRequestMetaData
    {
        [Required]
```

```
        public int RequestId {get;set;}

        //...
    }
}
```

ou

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace YourApplication.Models
{
    public interface IEntityMetadata
    {
        [Required]
        Int32 Id { get; set; }
    }

    [MetadataType(typeof(IEntityMetadata))]
    public partial class Entity : IEntityMetadata
    {
        /* Id property has already existed in the mapped class */
    }
}
```

Annotations de données pour la première implémentation de la base de données (code de modèle généré automatiquement)

```
[MetadataType(typeof(RoleMetaData))]
public partial class ROLE
{
}

public class RoleMetaData
{
    [Display(Name = "Role")]
    public string ROLE_DESCRIPTION { get; set; }

    [Display(Name = "Username")]
    public string ROLE_USERNAME { get; set; }
}
```

Si vous avez utilisé database-first et que votre code de modèle a été généré automatiquement, ce message apparaîtra au-dessus de votre code de modèle:

> Ce code a été généré à partir d'un modèle. Les modifications manuelles apportées à ce fichier peuvent entraîner un comportement inattendu dans votre application. Les modifications manuelles apportées à ce fichier seront remplacées si le code est régénéré

Si vous souhaitez utiliser des annotations de données et que vous ne voulez pas qu'elles soient écrasées si vous actualisez edmx, ajoutez simplement une autre classe partielle à votre dossier de modèle qui ressemble à l'exemple ci-dessus.

Lire Annotations de données en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/1961/annotations-de-donnees

Chapitre 7: Appel jQuery Ajax avec Asp MVC

Examples

Publication d'objets JavaScript avec l'appel jQuery Ajax

Appels Ajax, demande et récupération de données pour donner à l'utilisateur une meilleure expérience d'interface utilisateur interactive. Cet article va vous montrer comment utiliser jQuery et envoyer des données via des appels Ajax. Pour cet exemple, nous allons poster l'objet JavaScript suivant sur notre serveur.

```
var post = {
    title: " Posting JavaScript objects with jQuery Ajax Call",
    content: " Posting JavaScript objects with jQuery Ajax Call",
    tags: ["asp mvc", "jquery"]
};
```

Le côté serveur

Le modèle côté serveur correspondant à l'objet javascript.

```
public class Post
{
    public string Title { get; set; }
    public string Content { get; set; }
    public string[] Tags { get; set; }
}
```

Tout ce que nous avons à faire est de créer une méthode de contrôleur ASP.NET MVC standard qui prend un seul paramètre du type Person, comme cela.

```
public class PostController : BaseController
{
    public bool Create(Post model)
    {
        //Do somthing
    }
}
```

Le côté client

Pour envoyer des objets JavaScript, nous devons utiliser la méthode JSON.stringify () pour envoyer l'objet à l'option data.

```
$.ajax({
    url: '@Url.Action("create", "Post")',
    type: "POST",
    contentType: "application/json",
    data: JSON.stringify({ model: post })
}).done(function(result){
```

```
    //do something
});
```

Lire Appel jQuery Ajax avec Asp MVC en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/9734/appel-jquery-ajax-avec-asp-mvc

Chapitre 8: Asp.net mvc envoyer un mail

Examples

Formulaire de contact dans Asp MVC

1. modèle:

```
 public class ContactModel
 {
    [Required, Display(Name="Sender Name")]
    public string SenderName { get; set; }
    [Required, Display(Name = "Sender Email"), EmailAddress]
    public string SenderEmail { get; set; }
    [Required]
    public string Message { get; set; }
 }
```

2. contrôleur:

```
public class HomeController
{
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<ActionResult> Contact(ContectModel model)
    {
        if (ModelState.IsValid)
        {
            var mail = new MailMessage();
            mail.To.Add(new MailAddress(model.SenderEmail));
            mail.Subject = "Your Email Subject";
            mail.Body = string.Format("<p>Email From: {0} ({1})</p><p>Message:</p><p>{2}</p>",
model.SenderName, mail.SenderEmail, model.Message);
            mail.IsBodyHtml = true;
            using (var smtp = new SmtpClient())
            {
                await smtp.SendMailAsync(mail);
                return RedirectToAction("SuccessMessage");
            }
        }
        return View(model);
    }

    public ActionResult SuccessMessage()
    {
        return View();
    }

}
```

3. Web.Config:

```
<system.net>
  <mailSettings>
```

```
    <smtp from="you@outlook.com">
      <network host="smtp-mail.outlook.com"
               port="587"
               userName="you@outlook.com"
               password="password"
               enableSsl="true" />
    </smtp>
  </mailSettings>
</system.net>
```

4. Voir:

Contact.cshtml

```
@model ContectModel
   @using (Html.BeginForm())
   {
       @Html.AntiForgeryToken()
       <h4>Send your comments.</h4>
       <hr />
       <div class="form-group">
           @Html.LabelFor(m => m.SenderName, new { @class = "col-md-2 control-label" })
           <div class="col-md-10">
               @Html.TextBoxFor(m => m.SenderName, new { @class = "form-control" })
               @Html.ValidationMessageFor(m => m.SenderName)
           </div>
       </div>
       <div class="form-group">
           @Html.LabelFor(m => m.SenderEmail, new { @class = "col-md-2 control-label" })
           <div class="col-md-10">
               @Html.TextBoxFor(m => m.SenderEmail, new { @class = "form-control" })
               @Html.ValidationMessageFor(m => m.SenderEmail)
           </div>
       </div>
       <div class="form-group">
           @Html.LabelFor(m => m.Message, new { @class = "col-md-2 control-label" })
           <div class="col-md-10">
               @Html.TextAreaFor(m => m.Message, new { @class = "form-control" })
               @Html.ValidationMessageFor(m => m.Message)
           </div>
       </div>
       <div class="form-group">
           <div class="col-md-offset-2 col-md-10">
               <input type="submit" class="btn btn-default" value="Send" />
           </div>
       </div>
   }
```

SuccessMessage.cshtml

```
<h2>Your message has been sent</h2>
```

Envoi de courrier électronique de la classe

Cette façon de faire peut être tellement utile, mais certaines personnes (comme moi) ont des problèmes de répétition de code et, comme vous nous le faites savoir, cela signifie que je dois

créer un contrôleur de contact avec le même code sur chaque projet. , Je pense que cela peut être utile aussi

Ceci est ma classe, qui peut être sur une DLL ou autre

```
 public class Emails
    {
        public static void SendHtmlEmail(string receiverEmail, string subject, string body,
bool Ssl = false)
        {
            //Those are read it from webconfig or appconfig
            var client = new SmtpClient(ConfigurationManager.AppSettings["MailServer"],
Convert.ToInt16

                (ConfigurationManager.AppSettings["MailPort"]))
            {
                Credentials = new
NetworkCredential(ConfigurationManager.AppSettings["MailSender"],
ConfigurationManager.AppSettings["MailSenderPassword"]),
                EnableSsl = Ssl
            };

            MailMessage message = new MailMessage();
            message.From = new MailAddress(ConfigurationManager.AppSettings["MailSender"]);
            message.To.Add(receiverEmail);
            // message.To.Add("sgermosen@praysoft.net");
            message.Subject = subject;
            message.IsBodyHtml = true;
            message.Body = body;
            client.Send(message);
        }

    }
```

comme vous le voyez, il va lire depuis le webconfig, donc, nous devons le configurer, cette configuration est pour Gmail, mais chaque hôte a sa propre configuration

```
 <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
    <add key="AdminUser" value="sgrysoft@gmail.com" />
    <add key="AdminPassWord" value="123456789" />
    <add key="SMTPName" value="smtp.gmail.com" />
    <add key="SMTPPort" value="587" />

  </appSettings>
```

Lire Asp.net mvc envoyer un mail en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/9736/asp-net-mvc-envoyer-un-mail

Chapitre 9: Cryptage Web.config

Examples

Comment protéger votre fichier Web.config

Il est recommandé de chiffrer votre fichier Web.config si vous y avez des informations sensibles, par exemple une chaîne de connexion avec un mot de passe.

Avec l' outil d'enregistrement ASP.NET IIS (Aspnet_regiis.exe), vous pouvez facilement chiffrer des sections spécifiques du fichier Web.config. Une commande avec des privilèges élevés est requise.

Exemple utilisant DataProtectionConfigurationProvider . Ce fournisseur utilise DPAPI pour chiffrer et déchiffrer les données:

```
aspnet_regiis.exe -pef "connectionStrings" c:\inetpub\YourWebApp -prov
"DataProtectionConfigurationProvider"
```

Exemple utilisant RSAProtectedConfigurationProvider :

```
aspnet_regiis.exe -pef "connectionStrings" c:\inetpub\YourWebApp -prov
"RSAProtectedConfigurationProvider"
```

Si vous ne spécifiez pas le paramètre -prov, il utilise par défaut **RSAProtectedConfigurationProvider** . Ce fournisseur est recommandé pour les scénarios Web Farm.

Pour rétablir la section **connectionStrings** en texte clair:

```
aspnet_regiis.exe -pdf "connectionStrings" c:\inetpub\YourWebApp
```

Plus d'informations sur aspnet_regiis.exe sont disponibles sur MSDN .

Lire Cryptage Web.config en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6373/cryptage-web-config

Chapitre 10: Dockerization de l'application ASP.NET

Examples

Dockerfile et Nuget

La dockerisation de l'application ASP.NET nécessite un fichier Dockerfile pour la configuration et son exécution en tant que conteneur de station d'accueil.

```
FROM microsoft/dotnet:latest

RUN apt-get update && apt-get install sqlite3 libsqlite3-dev

COPY . /app

WORKDIR /app

RUN ["dotnet", "restore"]

RUN ["dotnet", "build"]

RUN npm install && npm run postscript

RUN bower install

RUN ["dotnet", "ef", "database", "update"]

EXPOSE 5000/tcp

ENTRYPOINT ["dotnet", "run", "--server.urls", "http://0.0.0.0:5000"]
```

Un fichier de configuration de flux nuget permet de récupérer la source correcte. L'utilisation de ce fichier dépend de la configuration actuelle du projet et peut être adaptée aux exigences du projet de la suite.

```
<?xml version="1.0" encoding="utf-8"?>
  <configuration>
   <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json"  protocolVersion="3" />
   <packageSources>
   <packageRestore>
    <add key="enabled" value="True" />
    <add key="automatic" value="True" />
   <packageRestore>
  <bindingRedirects>
   <add key="skip" value="False" />
  </bindingRedirects>
</configuration>
```

Support POSTGRESQL.

```
"Data": {
    "DefaultConnection": {
        "ConnectionString":
"Host=localhost;Username=postgres;Password=******;Database=postgres;Port=5432;Pooling=true;"
    }
  },
```

Dockerisation

Il est inutile d'avoir un package .NET ou mono-aspnet.

Il est important de comprendre l'importance de la dockerisation. Installez dotnet sur Ubuntu ou le système d'exploitation sur lequel vous travaillez.

Installation de DOTNET

```
$ sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/dotnet-release/
trusty main" > /etc/apt/sources.list.d/dotnetdev.list'
$ sudo apt-key adv --keyserver apt-mo.trafficmanager.net --recv-keys 417A0893
$ sudo apt-get update

Ubuntu 16.04



$ sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/dotnet-release/
xenial main" > /etc/apt/sources.list.d/dotnetdev.list'
$ sudo apt-key adv --keyserver apt-mo.trafficmanager.net --recv-keys 417A0893
$ sudo apt-get update
```

Installer le SDK Core .NET

```
$ sudo apt-get install dotnet-dev-1.0.0-preview2-003121
```

COURTOISIE: https://www.microsoft.com/net/core#ubuntu

Pour l'installation de Docker, suivez https://docs.docker.com/engine/installation/linux/ubuntulinux/

POUR LE PORT:

```
Kestrel server port : 5000
Docker Deamon will listen to port :

 EXPOSE 5000/tcp
```

Pour construire docker:

```
 $ sudo docker build -t myapp .
```

Pour exécuter le conteneur Docker:

```
 $ sudo docker run -t -d -p 8195:5000 myapp
```

Pour visiter le site:

```
$ ifconfig

eth0 : ***.***.**
 server-ip-address
```

Le site sera disponible sur (compte tenu de cette configuration):

```
 http://server-ip-address:8195
```

Processus Docker. Il listera les processus en cours d'exécution.

```
  $ sudo docker ps
```

Pour supprimer le processus ou le conteneur.

```
 $ sudo docker rm -rf <process_id>
```

Lire Dockerization de l'application ASP.NET en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6740/dockerization-de-l-application-asp-net

Chapitre 11: Domaines

Introduction

Quelle est la zone?

Une zone est une unité plus petite de l'application MVC qui permet de séparer un grand nombre de modules d'application en groupes fonctionnels. Une application peut contenir plusieurs zones stockées dans le dossier Areas.

Chaque zone peut contenir différents modèles, contrôleurs et vues selon les besoins. Pour utiliser une zone, il est nécessaire d'enregistrer le nom de la zone dans `RouteConfig` et de définir le préfixe de la route.

Remarques

si vous voulez aller dans cette zone via votre contrôleur par défaut

```
      return RedirectToAction("Index","Home",new{area="areaname"});
```

Examples

Créer une nouvelle zone

Faites un clic droit sur le dossier / nom de votre projet et créez une nouvelle zone et nommez-la.

Dans mvc internet / application vide / basic, un dossier avec le nom de la zone sera créé, qui contiendra trois dossiers différents nommés controller, model et views et un fichier de classe appelé

" *areaname* AreaRegistration.cs"

Configurez RouteConfig.cs

Dans votre dossier App_start, ouvrez routeconfig.cs et faites ceci

```
 routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional
},
            namespaces:new []{"nameofyourproject.Controllers"}// add this line ;
        );
```

Créer un nouveau contrôleur et configurer areanameAreaRegistration.cs

maproute

Créer un nouveau contrôleur

ControllerName: "Home", ActionresultName: "Index"

ouvrez AreaRegistraion.cs et ajoutez le nom du contrôleur et le nom de l'action à réacheminer vers

```
    context.MapRoute(
            "nameofarea_default",
            "nameofarea/{controller}/{action}/{id}",  // url shown will be like this in
browser
            new {controller="Home", action = "Index", id = UrlParameter.Optional }
        );
```

Lire Domaines en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6310/domaines

Chapitre 12: Enregistrement des erreurs

Examples

Attribut simple

```
using System;
using System.Web;
using System.Web.Mvc;

namespace Example.SDK.Filters
{
    [AttributeUsage(AttributeTargets.Class, Inherited = false, AllowMultiple = false)]
    public sealed class CustomErrorHandlerFilter : HandleErrorAttribute
    {
        public override void OnException(ExceptionContext filterContext)
        {
            // RouteDate is useful for retrieving info like controller, action or other route
values
            string controllerName = filterContext.RouteData.Values["controller"].ToString();
            string actionName = filterContext.RouteData.Values["action"].ToString();

            string exception = filterContext.Exception.ToString(); // Full exception stack
            string message = filterContext.Exception.Message; // Message given by the
exception

            // Log the exception within database
            LogExtensions.Insert(exception.ToString(), message, controllerName + "." +
actionName);

            base.OnException(filterContext);
        }
    }
}
```

Ensuite, définissez-le dans `FilterConfig.cs`

```
filters.Add(new CustomErrorHandlerFilter());
```

retour de la page d'erreur personnalisée

```
public ActionResult Details( string product)
{
   ....
    if (productNotFound) {
        // http://www.eidias.com/blog/2014/7/2/mvc-custom-error-pages
        Response.Clear();
        Response.TrySkipIisCustomErrors = true;
        Response.Write(product + " product not exists");
        Response.StatusCode = (int)HttpStatusCode.NotFound;
        Response.End();
        return null;
    }
```

}

Créer un ErrorLogger personnalisé dans ASP.Net MVC

Étape 1: Création d'un filtre de journalisation des erreurs personnalisé qui écrira les erreurs dans les fichiers texte selon DateWise.

```
public class ErrorLogger : HandleErrorAttribute
{
    public override void OnException(ExceptionContext filterContext)
    {

        string strLogText = "";
        Exception ex = filterContext.Exception;
        filterContext.ExceptionHandled = true;
        var objClass = filterContext;
        strLogText += "Message ---\n{0}" + ex.Message;

        if (ex.Source == ".Net SqlClient Data Provider")
        {
            strLogText += Environment.NewLine + "SqlClient Error ---\n{0}" + "Check Sql
Error";
        }
        else if (ex.Source == "System.Web.Mvc")
        {
            strLogText += Environment.NewLine + ".Net Error ---\n{0}" + "Check MVC Code For
Error";
        }
        else if (filterContext.HttpContext.Request.IsAjaxRequest() == true)
        {
            strLogText += Environment.NewLine + ".Net Error ---\n{0}" + "Check MVC Ajax Code
For Error";
        }
        strLogText += Environment.NewLine + "Source ---\n{0}" + ex.Source;
        strLogText += Environment.NewLine + "StackTrace ---\n{0}" + ex.StackTrace;
        strLogText += Environment.NewLine + "TargetSite ---\n{0}" + ex.TargetSite;
        if (ex.InnerException != null)
        {
            strLogText += Environment.NewLine + "Inner Exception is {0}" +
ex.InnerException;//error prone
        }
        if (ex.HelpLink != null)
        {
            strLogText += Environment.NewLine + "HelpLink ---\n{0}" + ex.HelpLink;//error
prone
        }

        StreamWriter log;

        string timestamp = DateTime.Now.ToString("d-MMMM-yyyy", new CultureInfo("en-GB"));

        string error_folder = ConfigurationManager.AppSettings["ErrorLogPath"].ToString();

        if (!System.IO.Directory.Exists(error_folder))
        {
            System.IO.Directory.CreateDirectory(error_folder);
        }

        if (!File.Exists(String.Format(@"{0}\Log_{1}.txt", error_folder, timestamp)))
```

```
        {
            log = new StreamWriter(String.Format(@"{0}\Log_{1}.txt", error_folder,
timestamp));
        }
        else
        {
            log = File.AppendText(String.Format(@"{0}\Log_{1}.txt", error_folder, timestamp));
        }

        var controllerName = (string)filterContext.RouteData.Values["controller"];
        var actionName = (string)filterContext.RouteData.Values["action"];

        // Write to the file:
        log.WriteLine(Environment.NewLine + DateTime.Now);
        log.WriteLine("------------------------------------------------------------------------
-------------------------");
        log.WriteLine("Controller Name :- " + controllerName);
        log.WriteLine("Action Method Name :- " + actionName);
        log.WriteLine("------------------------------------------------------------------------
-------------------------");
        log.WriteLine(objClass);
        log.WriteLine(strLogText);
        log.WriteLine();

        // Close the stream:
        log.Close();
        filterContext.HttpContext.Session.Abandon();
        filterContext.Result = new RedirectToRouteResult
         (new RouteValueDictionary
         {
                 {"controller", "Errorview"}, {"action", "Error"}
         });

    }

}
```

Étape 2: Ajout d'un chemin physique sur le serveur ou le lecteur local où le fichier texte sera stocké

```
<add key="ErrorLogPath" value="C:\ErrorLog\DemoMVC\" />
```

Étape 3: ajout de **Errorview** Controller avec **Error** ActionMethod

Étape 4: Ajout de **Error.cshtml Affichage et affichage d'un message d'erreur personnalisé dans la vue**

Étape 5: Enregistrez le filtre ErrorLogger dans la classe FilterConfig

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new ErrorLogger());
    }
}
```

Étape 6: Enregistrez FilterConfig dans Global.asax

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        WebApiConfig.Register(GlobalConfiguration.Configuration);
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
```

Lire Enregistrement des erreurs en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/2268/enregistrement-des-erreurs

Chapitre 13: Extensions Ajax MVC

Introduction

Cela documente l'utilisation de la bibliothèque `System.Web.Mvc.Ajax` .

Citing MSDN docs "Chaque méthode d'extension rend un élément HTML. La méthode ActionLink rend un élément anchor (a) lié à une méthode d'action. La méthode RouteLink rend un élément anchor (a) lié à une URL, qui peut être transformé en un élément méthode d'action, un fichier, un dossier ou une autre ressource Cette classe contient également les méthodes BeginForm et BeginRouteForm qui vous aident à créer des formulaires HTML pris en charge par les fonctions AJAX.

Paramètres

Options AJAX	La description
Confirmer	Obtient ou définit le message à afficher dans une fenêtre de confirmation avant l'envoi d'une demande.
HttpMethod	Obtient ou définit la méthode de requête HTTP ("Get" ou "Post").
Mode d'insertion	Obtient ou définit le mode qui spécifie comment insérer la réponse dans l'élément DOM cible.
ChargementElementDuration	Obtient ou définit une valeur, en millisecondes, qui contrôle la durée de l'animation lors de l'affichage ou du masquage de l'élément de chargement.
LoadingElementId	Obtient ou définit l'attribut id d'un élément HTML affiché pendant le chargement de la fonction Ajax.
OnBegin	Obtient ou définit le nom de la fonction JavaScript à appeler immédiatement avant la mise à jour de la page.
OnComplete	Obtient ou définit la fonction JavaScript à appeler lorsque les données de réponse ont été instanciées, mais avant la mise à jour de la page.
OnFailure	Obtient ou définit la fonction JavaScript à appeler si la mise à jour de la page échoue.
OnSuccess	Obtient ou définit la fonction JavaScript à appeler après la mise à jour de la page.

Options AJAX	La description
UpdateTargetId	Obtient ou définit l'ID de l'élément DOM à mettre à jour à l'aide de la réponse du serveur.
URL	Obtient ou définit l'URL pour effectuer la demande.

Remarques

Le package `Jquery.Unobtrusive-Ajax` est requis dans le projet. Les fichiers javascript correspondants doivent être inclus dans un bundle (`jquery.unobtrusive-ajax.js` ou `jquery.unobtrusive-ajax.min.js`). Enfin, il doit également être activé dans le fichier `web.config` :

```
<appSettings>
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

Les actions appelées (`SomeAction` dans les exemples) doivent renvoyer un `Json` ou un `PartialView` .

Examples

Ajax Action Link

```
@* Renders an anchor (a) element that links to an action method.
 * The innerHTML of "target-element" is replaced by the result of SomeAction.
 *@
@Ajax.ActionLink("Update", "SomeAction", new AjaxOptions{UpdateTargetId="target-element" })
```

Ajax Forms

```
@* Adds AJAX functions support to a form.
 * The innerHTML of "target-element" is replaced by the result of SomeAction.
 *@
@using ( Ajax.BeginForm("SomeAction", "SomeController",
                        new AjaxOptions {
                            UpdateTargetId="target-element",
                            OnSuccess = "some_js_fun(context)"
                        })
)
{
    <!-- my form contents -->
}
```

Lire Extensions Ajax MVC en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/9007/extensions-ajax-mvc

Chapitre 14: Filtres d'action

Examples

Un filtre d'action de journalisation

```
 public class LogActionFilter : ActionFilterAttribute
 {
      public override void OnActionExecuting(ActionExecutingContext filterContext)
      {
           Log("OnActionExecuting", filterContext.RouteData);
      }

      public override void OnActionExecuted(ActionExecutedContext filterContext)
      {
           Log("OnActionExecuted", filterContext.RouteData);
      }

      public override void OnResultExecuting(ResultExecutingContext filterContext)
      {
           Log("OnResultExecuting", filterContext.RouteData);
      }

      public override void OnResultExecuted(ResultExecutedContext filterContext)
      {
           Log("OnResultExecuted", filterContext.RouteData);
      }


      private void Log(string methodName, RouteData routeData)
      {
           var controllerName = routeData.Values["controller"];
           var actionName = routeData.Values["action"];
           var message = String.Format("{0} controller:{1} action:{2}", methodName,
controllerName, actionName);
           Debug.WriteLine(message, "Action Filter Log");
      }
 }
```

Filtre d'action de contrôle de session - page & ajax request

Généralement, les processus d'authentification et d'autorisation sont exécutés par des supports de cookies et de jetons intégrés dans .net MVC. Mais si vous décidez de le faire vous-même avec `Session` vous pouvez utiliser la logique ci-dessous pour les requêtes de page et les requêtes ajax.

```
public class SessionControl : ActionFilterAttribute
{
    public override void OnActionExecuting ( ActionExecutingContext filterContext )
    {
        var session = filterContext.HttpContext.Session;

        /// user is logged in (the "loggedIn" should be set in Login action upon a successful
login request)
        if ( session["loggedIn"] != null && (bool)session["loggedIn"] )
```

```
            return;

        /// if the request is ajax then we return a json object
        if ( filterContext.HttpContext.Request.IsAjaxRequest() )
        {
            filterContext.Result = new JsonResult
            {
                Data = "UnauthorizedAccess",
                JsonRequestBehavior = JsonRequestBehavior.AllowGet
            };
        }
        /// otherwise we redirect the user to the login page
        else
        {
            var redirectTarget = new RouteValueDictionary { { "Controller", "Login" }, {
"Action", "Index" } };
            filterContext.Result = new RedirectToRouteResult(redirectTarget);
        }
    }

    public override void OnResultExecuting ( ResultExecutingContext filterContext )
    {
        base.OnResultExecuting(filterContext);

        /// we set a field 'IsAjaxRequest' in ViewBag according to the actual request type
        filterContext.Controller.ViewBag.IsAjaxRequest =
filterContext.HttpContext.Request.IsAjaxRequest();
    }
}
```

Emplacements d'utilisation du filtre d'action (global, contrôleur, action)

Vous pouvez placer des filtres d'action à trois niveaux possibles:

1. Global
2. Manette
3. action

Placer un filtre **globalement** signifie qu'il s'exécutera sur les requêtes à n'importe quelle route. En en plaçant un sur un **contrôleur,** il s'exécute sur les requêtes de toute action dans ce contrôleur. En en plaçant un sur une **action** , il s'exécute avec l'action.

Si nous avons ce filtre d'action simple:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = true)]
public class CustomActionFilterAttribute : FilterAttribute, IActionFilter
{
    private readonly string _location;

    public CustomActionFilterAttribute(string location)
    {
        _location = location;
    }

    public void OnActionExecuting(ActionExecutingContext filterContext)
    {
        Trace.TraceInformation("OnActionExecuting: " + _location);
```

```
    }

    public void OnActionExecuted(ActionExecutedContext filterContext)
    {
        Trace.TraceInformation("OnActionExecuted: " + _location);
    }
}
```

Nous pouvons l'ajouter au niveau global en l'ajoutant à la collection de filtres globale. Avec la configuration typique du projet ASP.NET MVC, cela se fait dans App_Start / FilterConfig.cs.

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new CustomActionFilterAttribute("Global"));
    }
}
```

Nous pouvons également l'ajouter sur le contrôleur et le niveau d'action comme dans un contrôleur:

```
[CustomActionFilter("HomeController")]
public class HomeController : Controller
{
    [CustomActionFilter("Index")]
    public ActionResult Index()
    {
        return View();
    }
}
```

Si nous exécutons l'application et regardons la fenêtre de sortie, nous verrons les messages suivants:

```
iisexpress.exe Information: 0 : OnActionExecuting: Global
iisexpress.exe Information: 0 : OnActionExecuting: HomeController
iisexpress.exe Information: 0 : OnActionExecuting: Index
iisexpress.exe Information: 0 : OnActionExecuted: Index
iisexpress.exe Information: 0 : OnActionExecuted: HomeController
iisexpress.exe Information: 0 : OnActionExecuted: Global
```

Comme vous pouvez le voir, lorsque la requête arrive, les filtres sont exécutés:

1. Global
2. Manette
3. action

Les excellents exemples de filtres placés au niveau mondial sont les suivants:

1. Filtres d'authentification
2. Filtres d'autorisation
3. Filtres de journalisation

Attribut du gestionnaire d'exception

Cet attribut gère toutes les exceptions non gérées dans le code (ceci concerne principalement les requêtes Ajax - qui traitent de JSON - mais peuvent être étendues)

```
public class ExceptionHandlerAttribute : HandleErrorAttribute
{
    /// <summary>
    ///   Overriden method to handle exception
    /// </summary>
    /// <param name="filterContext"> </param>
    public override void OnException(ExceptionContext filterContext)
    {
        // If exeption is handled - return ( don't do anything)
        if (filterContext.ExceptionHandled)
            return;

        // Set the ExceptionHandled to true ( as you are handling it here)
        filterContext.ExceptionHandled = true;

        //TODO: You can Log exception to database or Log File

        //Set your result structure
        filterContext.Result = new JsonResult
        {
            Data = new { Success = false, Message = filterContext .Exception.Message, data =
new {} },
            JsonRequestBehavior = JsonRequestBehavior.AllowGet
        };

    }
}
```

Alors disons que vous devez toujours envoyer une réponse JSON similaire à celle-ci:

```
{

    Success: true,  // False when Error

    data: {},

    Message:"Success" // Error Message when Error

}
```

Donc, au lieu de gérer les exceptions dans les actions du contrôleur, comme ceci:

```
public ActionResult PerformMyAction()
{
    try
    {
        var myData = new { myValue = 1};

        throw new Exception("Handled", new Exception("This is an Handled Exception"));

        return Json(new {Success = true, data = myData, Message = ""});
```

```
    }
    catch(Exception ex)
    {
        return Json(new {Success = false, data = null, Message = ex.Message});
    }
}
```

Tu peux le faire:

```
[ExceptionHandler]
public ActionResult PerformMyAction()
{
        var myData = new { myValue = 1};

        throw new Exception("Unhandled", new Exception("This is an unhandled Exception"));

        return Json(new {Success = true, data = myData, Message = ""});
}
```

OU vous pouvez ajouter au niveau du contrôleur

```
[ExceptionHandler]
public class MyTestController : Controller
{

    public ActionResult PerformMyAction()
    {
            var myData = new { myValue = 1};

            throw new Exception("Unhandled", new Exception("This is an unhandled Exception"));

            return Json(new {Success = true, data = myData, Message = ""});
    }
}
```

Lire Filtres d'action en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/1450/filtres-d-action

Chapitre 15: Html.AntiForgeryToken

Introduction

Le jeton anti-contrefaçon peut être utilisé pour protéger votre application contre la falsification de requêtes intersites. Pour utiliser cette fonctionnalité, appelez la méthode AntiForgeryToken à partir d'un formulaire et ajoutez l'attribut ValidateAntiForgeryTokenAttribute à la méthode d'action que vous souhaitez protéger.

Génère un champ de formulaire masqué (jeton anti-contrefaçon) validé lors de la soumission du formulaire.

Syntaxe

- @ Html.AntiForgeryToken ()

Remarques

Lorsque vous soumettez une requête ajax avec un jeton CSRF (`__RequestVerificationToken`), assurez-vous que le type de contenu n'est pas défini sur `application/json` . Si vous utilisez jQuery, il définit automatiquement le type de contenu sur `application/x-www-form-urlencoded` qui est alors reconnu par ASP.NET MVC.

Mise en garde

Soyez prudent lorsque vous définissez cette valeur. Son utilisation incorrecte peut ouvrir des vulnérabilités de sécurité dans l'application.

Examples

Utilisation de base

La méthode d'assistance `@Html.AntiForgeryToken()` protège contre les attaques par falsification de requête intersite (ou CSRF).

Vous pouvez l'utiliser simplement en utilisant l' `Html.AntiForgeryToken()` dans l'un de vos formulaires existants et en décorant son action de contrôleur correspondante avec l'attribut `[ValidateAntiForgeryToken]` .

Razor (YourView.cshtml)

```
@using (Html.BeginForm("Manage", "Account")) {
    @Html.AntiForgeryToken()
```

```
    <!-- ... -->
}
```

OU

```
<form>
    @Html.AntiForgeryToken()
    <!-- ... -->
</form>
```

Contrôleur (YourController.cs)

La méthode d'action cible:

```
[ValidateAntiForgeryToken]
[HttpPost]
public ActionResult ActionMethod(ModelObject model)
{
    // ...
}
```

Désactiver le contrôle heuristique d'identité

Souvent, vous verrez une exception

```
Anti forgery token is meant for user "" but the current user is "username"
```

Cela est dû au fait que le jeton Anti-Forgery est également lié à l'utilisateur actuellement connecté. Cette erreur apparaît lorsqu'un utilisateur se connecte mais que son jeton est toujours lié à un utilisateur anonyme du site.

Il existe plusieurs manières de résoudre ce problème, mais si vous préférez ne pas avoir de jetons CSRF liés à l'état de connexion d'un utilisateur, vous pouvez désactiver cette fonctionnalité.

Placez cette ligne dans votre `Global.asax` ou dans la logique de démarrage d'une application similaire.

```
AntiForgeryConfig.SuppressIdentityHeuristicChecks = true;
```

Valider tous les messages

En raison de la vulnérabilité causée par CSRF, il est généralement considéré comme une bonne pratique de rechercher un AntiForgeryToken sur tous les HttpPosts à moins qu’il n’y ait une bonne raison de le faire (problème technique avec la publication, autre mécanisme d’authentification et / ou post ne modifie pas l'état comme l'enregistrement dans une base de données ou un fichier). Pour vous assurer de ne pas oublier, vous pouvez ajouter un GlobalActionFilter spécial qui vérifie automatiquement tous les HttpPosts à moins que l'action ne soit décorée avec un attribut spécial "ignore".

```
[AttributeUsage(AttributeTargets.Class)]
public class ValidateAntiForgeryTokenOnAllPosts : AuthorizeAttribute
{
    public override void OnAuthorization(AuthorizationContext filterContext)
    {
        var request = filterContext.HttpContext.Request;

        //  Only validate POSTs
        if (request.HttpMethod == WebRequestMethods.Http.Post)
        {
            bool skipCheck =
filterContext.ActionDescriptor.IsDefined(typeof(DontCheckForAntiForgeryTokenAttribute), true)
                ||
filterContext.ActionDescriptor.ControllerDescriptor.IsDefined(typeof(DontCheckForAntiForgeryTokenAttrib
true);

            if (skipCheck)
                return;


            //  Ajax POSTs and normal form posts have to be treated differently when it comes
            //  to validating the AntiForgeryToken
            if (request.IsAjaxRequest())
            {
                var antiForgeryCookie = request.Cookies[AntiForgeryConfig.CookieName];

                var cookieValue = antiForgeryCookie != null
                    ? antiForgeryCookie.Value
                    : null;

                AntiForgery.Validate(cookieValue,
request.Headers["__RequestVerificationToken"]);
            }
            else
            {
                new ValidateAntiForgeryTokenAttribute()
                    .OnAuthorization(filterContext);
            }
        }
    }
}

/// <summary>
/// this should ONLY be used on POSTS that DO NOT MUTATE STATE
/// </summary>
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false,
Inherited = true)]
public sealed class DontCheckForAntiForgeryTokenAttribute : Attribute { }
```

Pour vous assurer qu'il est vérifié sur toutes les demandes, ajoutez-le simplement à vos filtres d'action globale.

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        //...
        filters.Add(new ValidateAntiForgeryTokenOnAllPosts());
        //...
    }
```

```
}
```

Utilisation avancée: appliquez le filtre Antiforgery par défaut pour chaque POST

Nous pourrions oublier d'appliquer l' `Antiforgery attribute` pour chaque requête `POST` afin que nous puissions le faire par défaut. Cet exemple s'assurera que le `Antiforgery filter` sera toujours appliqué à chaque requête `POST` .

`AntiForgeryTokenFilter` créer un nouveau filtre `AntiForgeryTokenFilter` :

```
//This will add ValidateAntiForgeryToken Attribute to all HttpPost action methods
public class AntiForgeryTokenFilter : IFilterProvider
{
    public IEnumerable<Filter> GetFilters(ControllerContext controllerContext,
ActionDescriptor actionDescriptor)
    {
        List<Filter> result = new List<Filter>();

        string incomingVerb = controllerContext.HttpContext.Request.HttpMethod;

        if (String.Equals(incomingVerb, "POST", StringComparison.OrdinalIgnoreCase))
        {
            result.Add(new Filter(new ValidateAntiForgeryTokenAttribute(), FilterScope.Global,
null));
        }

        return result;
    }
}
```

Enregistrez ensuite ce filtre personnalisé dans MVC, Application_Start:

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        //Cactch generic error
        filters.Add(new HandleErrorAttribute());

        //Anti forgery token hack for every post request
        FilterProviders.Providers.Add(new AntiForgeryTokenFilter());
    }
}



public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
```

```
}
```

Ainsi, toutes vos requêtes `POST` sont désormais protégées par défaut à l'aide d'attributs Antiforgery. Il n'est donc plus nécessaire d'avoir l'attribut `[ValidateAntiForgeryToken]` sur chaque méthode POST.

Utiliser AntiForgeryToken avec Jquery Ajax Request

Tout d'abord, vous créez le formulaire

```
@using (Html.BeginForm())
{
  @Html.AntiForgeryToken()
}
```

Méthode d'action

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Test(FormViewModel formData)
{
    // ...
}
```

Scénario

```
<script src="https://code.jquery.com/jquery-1.12.4.min.js"></script>
<script>
var formData = new FormData($('form')[0]);
$.ajax({
    method: "POST",
    url: "/demo/test",
    data: formData ,
    success: function (data) {
  console.log(data);
    },
    error: function (jqXHR, textStatus, errorThrown) {
        console.log(errorThrown);
    }
})
</script>
```

Assurez-vous que contentType n'est pas défini sur autre chose que `application/x-www-form-urlencoded` et si son Jquery non spécifié est défini par défaut sur `application/x-www-form-urlencoded`

Lire Html.AntiForgeryToken en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/1997/html-antiforgerytoken

Chapitre 16: Html.RouteLink

Paramètres

Paramètre	Détails
linkText	Le texte qui sera affiché pour le lien.
routeName	Le nom de la route pour laquelle renvoyer un chemin virtuel.

Examples

Exemple de base en utilisant le texte du lien et le nom de la route

Au lieu d'utiliser `Html.ActionLink` pour générer des liens dans une vue, vous pouvez utiliser

`Html.RouteLink`

Pour utiliser cette fonctionnalité, vous devez configurer un itinéraire, par exemple:

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapRoute(
    "SearchResults",
    "{controller}/{action}",
    new { controller = "Search", action = "Results" });
}
```

Ensuite, dans une vue, vous pouvez créer un lien vers cette route comme suit:

```
@Html.RouteLink("Search Results", "SearchResults");
```

Utiliser `RouteLink()` est pratique si vous changez de nom de contrôleur ou de nom de méthode d'action, car l'utilisation de `Html.ActionLink()` signifie que vous `Html.ActionLink()` modifier les paramètres du nom du contrôleur et de la méthode d'action dans l'appel afin qu'ils correspondent aux nouveaux noms. été changé.

Avec `RouteLink()` vous pouvez modifier les détails de l'itinéraire dans l'appel `MapRoute()` , c'est-à-dire dans un emplacement, et aucun code faisant référence à cet itinéraire via `RouteLink()` ne sera requis.

Lire Html.RouteLink en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6209/html-routelink

Chapitre 17: Injection de dépendance

Remarques

Le point entier de l'injection de dépendance (DI) est de réduire le couplage de code. Imaginez tout type d'interaction qui implique de changer quelque chose comme dans "l'exemple de la dépendance codée en dur".

Une grande partie de l'écriture du code est la possibilité de le tester. Chaque fois que nous découvrons une nouvelle dépendance, nous rendons notre code difficile à tester car nous n'avons aucun contrôle sur cette dépendance.

Comment testeriez-vous le code qui dépend de DataTime.Now par exemple? Cela change toujours, donc vous n'avez aucune référence. C'est à ce moment que vous injectez un paramètre stable comme point de départ. Vous pouvez le contrôler, vous pouvez écrire des tests en fonction de différentes valeurs et vous assurer que vous obtenez toujours le bon résultat.

Une bonne option est donc de passer une interface ou une classe abstraite en tant que paramètre dans le constructeur DI.

Une interface représente un contrat bien défini, vous pouvez toujours compter sur les méthodes pour y être et vous pouvez toujours compter sur les signatures de méthode.

Une fois que vous commencez à utiliser DI, d'autres aspects vont s'ouvrir. Par exemple, même si vous passez une interface à un moment donné, vous aurez besoin d'une véritable implémentation pour faire tout travail. C'est là que d'autres concepts apparaissent. Nous pouvons utiliser IOC (Inversion of Control) pour résoudre nos dépendances. Cela signifie que nous demandons à notre code de toujours utiliser une implémentation spécifique pour tout contrat. Bien sûr, il existe d'autres moyens de le faire. Nous pourrions toujours instancier chaque contrat avec une implémentation spécifique et à partir de ce moment, notre code peut utiliser cette partie:

```
public ILogging Logging { get; set }
```

à un moment donné, nous l'initialisons.

```
Logging = new FileLogging();
```

cela fonctionnera toujours tant que notre classe remplit le contrat prévu:

```
public class FileLogging : ILogging
```

à partir du moment d'initialisation, nous utilisons toujours l'objet Logging. Cela facilite la vie car si nous décidons de changer et d'utiliser un objet DatabaseLogging par exemple, il suffit de changer le code à un seul endroit et c'est exactement là que nous initialisons la classe Logging.

Est-ce que le DI n'est bon que pour les tests? Non, DI est également important lors de la rédaction

du code maintenable. Cela permet de séparer les préoccupations.

Lorsque vous écrivez un code, pensez-vous que ... est-il testable? Puis-je écrire un test, c'est-à-dire injecter une valeur DateTime au lieu d'utiliser DateTime.

Examples

Configurations Ninject

Après l'installation d'un conteneur IoC (Inversion of Control), des ajustements sont nécessaires pour que cela fonctionne. Dans ce cas, je vais utiliser Ninject. Dans le fichier NinjectWebCommon, situé dans le dossier App_Start, remplacez la méthode CreateKernel par:

```
private static IKernel CreateKernel()
    {
        // Create the kernel with the interface to concrete bindings
        var kernel = RegisterServices();
        try
        {
            kernel.Bind<Func<IKernel>>().ToMethod(ctx => () => new Bootstrapper().Kernel);
            kernel.Bind<IHttpModule>().To<HttpApplicationInitializationHttpModule>();

            return kernel;
        }
        catch
        {
            kernel.Dispose();
            throw;
        }
    }
```

Et la méthode RegisterServices avec:

```
 private static StandardKernel RegisterServices()
        {
            Container container = new Container();
            // encapsulate the interface to concrete bindings in another class or even in
another layer
            StandardKernel kernel = container.GetServices();
            return kernel;
        }
```

Créez une nouvelle classe pour la liaison appelée ici Container:

```
public class Container
{
    public StandardKernel GetServices()
    {
        // It is good practice to create a derived class of NinjectModule to organize the
binding by concerns. In this case one for the repository, one for service and one for app
service bindings
        return new StandardKernel(new NinjectRepositoryModule(),
            new NinjectServiceModule(),
            new NinjectAppServiceModule());
```

```
    }
}
```

Enfin, dans chaque classe NinjectModule dérivée, modifiez les liaisons surchargeant la méthode Load comme:

```
public class NinjectRepositoryModule: NinjectModule
{
    public override void Load()
    {
        // When we need a generic IRepositoryBase<> to bind to a generic RepositoryBase<>
        // The typeof keyword is used because the target method is generic
        Bind(typeof (IRepositoryBase<>)).To(typeof (RepositoryBase<>));

        // When we need a IUnitOfWorkbind to UnitOfWork concrete class that is a singleton
        Bind<IUnitOfWork>().To<UnitOfWork>().InSingletonScope();
    }
}
```

Un autre exemple de NinjectModule dérivé:

```
public class NinjectServiceModule :NinjectModule
{
    public override void Load()
    {
        // When we need a IBenefitService to BenefitService concrete class
        Bind<IBenefitService>().To<BenefitService>();

        // When we need a ICategoryService to CategoryService concrete class
        Bind<ICategoryService>().To<CategoryService>();

        // When we need a IConditionService to ConditionService concrete class
        Bind<IConditionService>().To<ConditionService>();
    }
}
```

Utilisation des interfaces

Dans la classe concrète qui a besoin du service, utilisez l'interface pour accéder au service au lieu de son implémentation comme:

```
 public class BenefitAppService
{
    private readonly IBenefitService _service;
    public BenefitAppService(IBenefitService service)
    {
        _service = service;
    }

    public void Update(Benefit benefit)
    {
        if (benefit == null) return
        _service.Update(benefit);
        _service.Complete();
    }
}
```

Maintenant, si vous avez besoin de quelque chose dans la classe concrète, n'interférez pas dans le code ci-dessus. Vous pouvez changer l'implémentation du service pour une autre différence, et tant que cela satisfait l'interface, vous êtes prêt à partir. En outre, il est très facile de le tester.

Injection de dépendance du constructeur

L'injection de dépendance de constructeur nécessite des paramètres dans le constructeur pour injecter des dépendances. Vous devez donc passer les valeurs lorsque vous créez un nouvel objet.

```
public class Example
{
    private readonly ILogging _logging;

    public Example(ILogging logging)
    {
        this._logging = logging;
    }
}
```

Dépendance codée en dur

```
public class Example
{
    private FileLogging _logging;

    public Example()
    {
        this._logging = new FileLogging();
    }
}
```

paramètre DI

```
public DateTime SomeCalculation()
{
     return DateTime.Now.AddDays(3);
}
```

contre

```
public DateTime SomeCalculation(DateTime inputDate)
{
    return inputDate.AddDays(3);
}
```

Injection de dépendances à injecter

Le résolveur de dépendance est utilisé pour éviter les classes étroitement couplées, améliorer la flexibilité et faciliter les tests. Vous pouvez créer votre propre injecteur de dépendance (non recommandé) ou utiliser un injecteur de dépendance bien écrit et testé. Dans cet exemple, je vais

utiliser Ninject .

Première étape: créez un résolveur de dépendance.

Tout d'abord, téléchargez *Ninject* depuis NuGet. Créez le dossier nommé *Infrastructure* et ajoutez la classe nommée *NinjectDependencyResolver* :

```
using Ninject;
using System;
using System.Collections.Generic;
using System.Web.Mvc;

public class NinjectDependencyResolver
    : IDependencyResolver
{
    private IKernel kernel;

    public NinjectDependencyResolver()
    {
        // Initialize kernel and add bindings
        kernel = new StandardKernel();
        AddBindings();
    }

    public object GetService(Type serviceType)
    {
        return kernel.TryGet(serviceType);
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return kernel.GetAll(serviceType);
    }

    private void AddBindings()
    {
        // Bindings added here
    }
}
```

MVC Framework appellera les méthodes *GetService* et *GetServices* lorsqu'elles ont besoin d'une classe pour traiter une demande entrante.

Etape 2: Enregistrez le résolveur de dépendance.

Nous avons maintenant notre résolveur de dépendances personnalisé et nous devons l'enregistrer pour que le framework MVC utilise notre résolveur de dépendances. Enregistrez le résolveur de dépendance dans *le* fichier *Global.asax.cs* :

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    DependencyResolver.SetResolver(new NinjectDependencyResolver());

    // .....
}
```

Troisième étape: Ajouter des liaisons.

Imaginez que nous avons l'interface et l'implémentation suivantes:

```
public interface ICustomCache
{
    string Info { get; }
}

public class CustomCache : ICustomCache
{
    public string Info
    {
        get
        {
            return "Hello from CustomCache.";
        }
    }
}
```

Si nous voulons utiliser CustomCache dans notre contrôleur sans coupler étroitement notre contrôleur avec CustomCache, nous devons lier *ICustomCache à CustomCache* et l'injecter à l'aide de Ninject. Tout d'abord, liez *ICustomCache à CustomCache* en ajoutant le code suivant à la méthode *AddBindings ()* de *NinjectDependencyResolver* :

```
private void AddBindings()
{
    // Bindings added here
    kernel.Bind<ICustomCache>().To<CustomCache>();
}
```

Ensuite, préparez votre contrôleur pour l'injection comme ci-dessous:

```
public class HomeController : Controller
{
    private ICustomCache CustomCache { get; set; }

    public HomeController(ICustomCache customCacheParam)
    {
        if (customCacheParam == null)
            throw new ArgumentNullException(nameof(customCacheParam));

        CustomCache = customCacheParam;
    }

    public ActionResult Index()
    {
        // cacheInfo: "Hello from CustomCache."
        string cacheInfo = CustomCache.Info;

        return View();
    }
}
```

Ceci est un exemple d' *injection* de *costructor* et c'est *une forme d'injection de dépendance* . Comme vous le voyez, notre contrôleur domestique ne dépend pas de la classe CustomCache. Si

nous voulons utiliser une autre implémentation de ICustomCache dans notre application, la seule chose que nous devons changer est de lier *ICustomCache* à une autre implémentation et c'est la seule étape que nous devons prendre. Ce qui est arrivé ici est-framework MVC a demandé à notre *résolveur enregistré de dépendance* de créer une instance de la classe *HomeController* via la méthode *GetService.* La méthode GetService demande au noyau Ninject de créer l'objet demandé et le noyau Ninject examine le type dans son terme et découvre que le constructeur de HomeController exige un *ICustomCache* et que la liaison a déjà été ajoutée pour *ICustomCache* . Ninject crée une instance de *classe* liée, l'utilise pour créer *HomeController* et la renvoie à MVC Framework.

Chaînes de dépendance.

Lorsque Ninject essaie de créer un type, il examine d'autres dépendances entre les types et les autres types et s'il existe un Ninject, il essaie également de les créer. Par exemple, si notre classe CustomCache nécessite ICacheKeyProvider et si la combinaison ajoutée pour ICacheKeyProvider Ninject peut la fournir pour notre classe.

Interface *ICacheKeyProvider* et implémentation *SimpleCacheKeyProvider* :

```
public interface ICacheKeyProvider
{
    string GenerateKey(Type type);
}

public class SimpleCacheKeyProvider
    : ICacheKeyProvider
{
    public string GenerateKey(Type type)
    {
        if (type == null)
            throw new ArgumentNullException(nameof(type));

        return string.Format("{0}CacheKey", type.Name);
    }
}
```

Classe *CustomCache* modifiée

```
public class CustomCache : ICustomCache
{
    private ICacheKeyProvider CacheKeyProvider { get; set; }

    public CustomCache(ICacheKeyProvider keyProviderParam)
    {
        if (keyProviderParam == null)
            throw new ArgumentNullException(nameof(keyProviderParam));

        CacheKeyProvider = keyProviderParam;
    }

    ...........
}
```

Ajouter une liaison pour *ICacheKeyProvider* :

```
private void AddBindings()
{
    // Bindings added here
    kernel.Bind<ICustomCache>().To<CustomCache>();
    kernel.Bind<ICacheKeyProvider>().To<SimpleCacheKeyProvider>();
}
```

Maintenant , quand nous naviguons à *HomeController* Ninject crée instance de *SimpleCacheKeyProvider* utilise pour créer *CustomCache* et utilise par exemple pour créer CustomCache *HomeController.*

Ninject possède un grand nombre de fonctionnalités comme l'injection de dépendances en chaîne et vous devriez les examiner si vous souhaitez utiliser Ninject.

Lire Injection de dépendance en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6392/injection-de-dependance

Chapitre 18: Le rasoir

Introduction

Qu'est-ce que le rasoir?

Razor est une syntaxe de balisage qui vous permet d'intégrer du code basé sur un serveur (Visual Basic et C #) dans des pages Web.

Le code basé sur un serveur peut créer du contenu Web dynamique à la volée, tandis qu'une page Web est écrite dans le navigateur. Lorsqu'une page Web est appelée, le serveur exécute le code basé sur le serveur dans la page avant de renvoyer la page au navigateur. En exécutant sur le serveur, le code peut effectuer des tâches complexes, comme accéder à des bases de données.

Syntaxe

- @ {...}
- @Nom de variable
- @(Nom de variable)
- @pour(...){ }
- @ (Expression explicite)
- @* commentaires *@

Remarques

ASP.NET Razor inclut des moteurs de visualisation pour C # et VB.

Le moteur de vue C # traite les fichiers avec une extension `.cshtml` , tandis que le moteur de vue VB fonctionne avec des fichiers `.vbhtml` .

Examples

Ajoutez des commentaires

Razor a sa propre syntaxe de commentaire qui commence par `@*` et se termine par `*@` .

Commentaire en ligne:

```
<h1>Comments can be @*hi!*@ inline</h1>
```

Commentaire multiligne:

```
@* Comments can spread
   over multiple
```

```
   lines *@
```

Commentaire HTML

Vous pouvez également utiliser la syntaxe de commentaire HTML normale commençant par `<!--` et se terminant par `-->` dans les vues de rasoir. Mais contrairement aux autres commentaires, le code Razor dans un commentaire HTML est toujours exécuté normalement.

```
@{
    var hello = "Hello World!";
}
<!-- @hello -->
```

L'exemple ci-dessus produit la sortie HTML suivante:

```
<!-- Hello World! -->
```

Commentaires dans un bloc de code:

```
@{
    // This is a comment
    var Input = "test";
}
```

Afficher le code HTML dans le bloc de code Razor

Dans un bloc de code Razor, le navigateur reconnaît uniquement le code HTML si le code est échappé.

Utilisez `@:` pour une ligne simple:

```
@foreach(int number in Model.Numbers)
{
    @:<h1>Hello, I am a header!</h1>
}
```

Utilisez `<text> ... </text>` pour Multi-line:

```
@{
    var number = 1;

    <text>
        Hello, I am text
        <br / >
        Hello, I am more text!
    </text>
}
```

Notez que Razor, à l'intérieur d'un bloc de code, comprendra les balises HTML. Par conséquent, l'ajout de la balise de `text` autour des balises HTML est inutile (mais toujours correct), par exemple:

```
@{
    var number = 1;
    <text>
        <div>
            Hello, I am text
            <br / >
            Hello, I am more text!
        </div>
    </text>
}
```

Syntaxe de base

Le code rasoir peut être inséré n'importe où dans le code HTML. Les blocs de code de rasoir sont placés dans `@{ ... }` . La variable en ligne et les fonctions commencent par `@` . Code à l'intérieur des parenthèses de rasoir suivent les règles normales de C # ou de VB.

Déclaration de ligne unique:

```
@{ var firstNumber = 1; }
```

Bloc de code multiligne:

```
@{
    var secondNumber = 2;
    var total = firstNumber + secondNumber;
}
```

En utilisant une variable inline:

```
<h1>The total count is @total</h1>
```

Utiliser une variable en ligne explicitement :

```
<h2>Item@(item.Id)</h2>
```

Pour cet exemple particulier, nous ne pourrons pas utiliser la syntaxe implicite car `Item@item.Id` ressemble à un email et sera rendu comme tel par Razor.

Placez le code à l'intérieur des instructions du flux de contrôle:

```
<h1>Start with some HTML code</h1>

@for (int i = 0; i < total; i++){
    Console.Write(i);
}

<p>Mix in some HTML code for fun!</p>
<p>Add a second paragraph.</p>

@if (total > 3)
{
```

```
    Console.Write("The total is greater than 3");
}
else
{
    Console.Write("The total is less than 3");
}
```

Cette même syntaxe serait utilisée pour toutes les instructions telles que `for` , `foreach` , `while` , `if` , `switch` , etc.

Ajouter du code à l'intérieur du code:

```
@if (total > 3)
{
    if(total == 10)
    {
        Console.Write("The total is 10")
    }
}
```

Notez que vous n'avez pas besoin de taper le `@` à la seconde `if` . Après le code, vous pouvez simplement taper un autre code derrière le code existant.

Si vous voulez ajouter du code après un élément HTML que vous **avez** besoin de taper un `@` .

Évasion @ caractère

Dans de nombreux cas, l'analyseur Razor est suffisamment intelligent pour déterminer si le signe `@` est destiné à être utilisé dans le cadre du code, au lieu de faire partie d'une adresse électronique. Dans l'exemple ci-dessous, échapper au signe `@` n'est pas nécessaire:

```
<p>Reach out to us at contact@mail.com</p>
```

Cependant, dans certains cas, l'utilisation du signe `@` est plus ambiguë et doit être explicitement échappée avec `@@` , comme dans l'exemple ci-dessous:

```
<p>Join us @@ Stack Overflow!</p>
```

Alternativement, nous pouvons utiliser un caractère HTML encodé `@`

```
<p>Join us &#64; Stack Overflow!</p>
```

Créer des classes et des méthodes en ligne à l'aide des fonctions @

L' `@functions` mot clé Razor `@functions` permet d'introduire des classes et des méthodes pour une utilisation en ligne dans un fichier Razor:

```
@functions
{
    string GetCssClass(Status status)
```

```
    {
        switch (status)
        {
            case Status.Success:
                return "alert-success";
            case Status.Info:
                return "alert-info";
            case Status.Warning:
                return "alert-warning";
            case Status.Danger:
            default:
                return "alert-danger";
         }
     }
}


<label class="alert @GetCssClass(status)"></label>
```

La même chose peut être faite pour les classes:

```
@functions
{
    class Helpers
    {
        //implementation
    }
}
```

Ajout d'un attribut personnalisé avec - (trait d'union) dans le nom

Si vous avez besoin d'ajouter un attribut par le biais d'un rasoir comportant un - (trait d'union) dans le nom que vous ne pouvez pas faire simplement

```
@Html.DropDownListFor(m => m.Id, Model.Values, new { @data-placeholder = "whatever" })
```

il ne compilera pas. Les attributs data- * sont valides et courants dans html5 pour ajouter des valeurs supplémentaires aux éléments.

Cependant, les éléments suivants fonctionneront

```
@Html.DropDownListFor(m => m.Id, Model.Values, new { @data_placeholder = "whatever" })
```

puisque "_" est remplacé par "-" lors du rendu.

Cela fonctionne bien car les traits de soulignement ne sont pas acceptables dans les noms d'attribut en HTML.

Modèles de l'éditeur

Les modèles d'éditeur sont un bon moyen de réutiliser le code Razor. Vous pouvez définir des modèles d'éditeur en tant que vues partielles Razor, puis les utiliser dans d'autres vues.

Les modèles d'éditeur existent généralement dans le dossier `Views/Shared/EditorTemplates/` , bien qu'ils puissent également être enregistrés dans le dossier `Views/ControllerName/EditorTemplates/` . Le nom de la vue est généralement le nom de l'objet pour `<type>.cshtml` vous souhaitez utiliser le modèle, comme `<type>.cshtml` .

Voici un template d'éditeur simple pour DateTime:

```
@model DateTime
<div>
   <span>
      @Html.TextBox("", Model.ToShortDateString(), new { data_date_picker="true" })
   </span>
</div>
```

Enregistrez le fichier sous le nom **Views / Shared / EditorTemplate / DateTime.cshtml** .

Ensuite, utilisez `EditorFor` pour appeler ce code de modèle dans une autre vue:

```
@Html.EditorFor(m => m.CreatedDate)
```

Il existe également un attribut UIHint pour spécifier le nom du fichier:

```
public class UiHintExampleClass
{
    [UIHint("PhoneNumber")]
    public string Phone { get; set; }
}
```

Définissez ce modèle de numéro de téléphone dans **Views / Shared / EditorTemplates / PhoneNumber.cshtml** .

Les modèles d'éditeur peuvent également être définis pour les types personnalisés.

Voici un type personnalisé appelé `SubModel` :

```
public class SubModel
{
    public Guid Id { get; set;}
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Model
{
    public Guid Id { get; set; }
    public DateTime Created {get; set; }

    public SubModel SubModel{get; set; }
}
```

Ceci est le EditorTemplate pour SubModel:

```
@model SubModel
<div class="form-group">
    @Html.LabelFor(m => m.FirstName)
    @Html.TextBoxFor(m => m.FirstName)
</div>
<div class="form-group">
    @Html.LabelFor(m => m.LastName)
    @Html.TextBoxFor(m => m.LastName)
</div>
```

Maintenant, la vue pour le modèle devient simplement:

```
@model Model
@Html.EditorFor(m => m.CreatedDate)
@Html.EditorFor(m => m.SubModel, new { @Prefix = "New"})
@* the second argument is how you can pass viewdata to your editor template*@
```

Transmettre le contenu du rasoir à un @helper

Envoyer une partie Razor à un @helper, par exemple un div HTML.

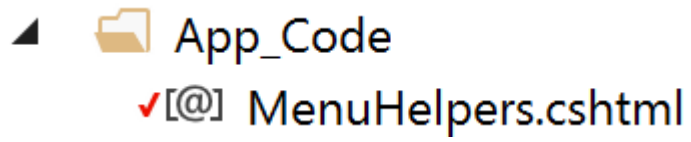
```
@helper WrapInBox(Func<Object, HelperResult> content)
{
    <div class="box">@content(null) </div>
}

//call
@WrapInBox(@<div>
                I'm a inner div
            </div>)
```

Partager @helpers entre les vues

@Helpers pourrait être partagé entre les vues.

Ils doivent être créés dans le dossier App_Code




```
@helper CreatePrimaryBootstrapButton(string label)
{
    <button type="button" class="btn btn-primary">@label</button>
}

//call

@MenuHelpers.CreatePrimaryBootstrapButton("my button")
```

Les globals `@Url` et `@Html` ne sont pas disponibles par défaut dans le @Helper défini dans App_code. Vous pouvez les ajouter comme suit (pour chaque fichier .cshtml dans votre dossier App_code)

```
@*  Make @Html and @Url available *@
@functions
{
    private new static HtmlHelper<object> Html
    {
        get { return ((WebViewPage)CurrentPage).Html; }
    }

    private static UrlHelper Url
    {
        get { return ((WebViewPage)CurrentPage).Url; }
    }
 }
```

Lire Le rasoir en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/5266/le-rasoir

Chapitre 19: Le routage

Introduction

Le routage est la manière dont ASP.NET MVC associe un URI à une action. Le module de routage est chargé de mapper les requêtes du navigateur entrant vers des actions de contrôleur MVC particulières.

MVC 5 prend en charge un nouveau type de routage, appelé routage d'attribut. Comme son nom l'indique, le routage d'attribut utilise des attributs pour définir des itinéraires. Le routage d'attributs vous permet de mieux contrôler les URI de votre application Web.

Examples

Routage personnalisé

Le routage personnalisé fournit un besoin spécialisé de routage pour gérer des demandes entrantes spécifiques.

Pour définir des itinéraires personnalisés, gardez à l'esprit que l'ordre des itinéraires que vous ajoutez à la table de routage est important.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    // this is an advanced custom route
    // you can define custom URL with custom parameter(s) point to certain action method
    routes.MapRoute(
    "CustomEntry", // Route name
    "Custom/{entryId}", // Route pattern
    new { controller = "Custom", action = "Entry" } // Default values for defined parameters
above
    );

    // this is a basic custom route
    // any custom routes take place on top before default route
    routes.MapRoute(
    "CustomRoute", // Route name
    "Custom/{controller}/{action}/{id}", // Route pattern
    new { controller = "Custom", action = "Index", id = UrlParameter.Optional } // Default
values for defined parameters above
    );

    routes.MapRoute(
    "Default", // Route name
    "{controller}/{action}/{id}", // Route pattern
    new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Default
values for defined parameters above
    );
}
```

`controller` noms de `controller` et d' `action` sont réservés. Par défaut, MVC mappe `{controller}` partie de l'URL sur la classe `<controller>Controller` , puis recherche une méthode portant le nom `<action>` sans ajouter de suffixe.

Bien que cela puisse être tentant de créer une famille de routes en utilisant `{controller}/{action}/{parameter}` template, sachez que cela vous permet de révéler la structure de votre application et de rendre les URL quelque peu fragiles. achemine et rompt les liens enregistrés par l'utilisateur.

Préférer un paramétrage de route explicite:

```
routes.MapRoute(
    "CustomRoute", // Route name
    "Custom/Index/{id}", // Route pattern
    new { controller = "Custom", action = nameof(CustomController.Index), id =
UrlParameter.Optional }
);
```

(vous ne pouvez pas utiliser le `nameof` opérateur pour le nom du contrôleur car il aura un suffixe supplémentaire `Controller`) qui doit être omis lors de la définition du nom du contrôleur dans l'itinéraire.

Ajout d'un itinéraire personnalisé dans Mvc

L'utilisateur peut ajouter un itinéraire personnalisé, mappant une URL à une action spécifique dans un contrôleur. Ceci est utilisé pour l'optimisation des moteurs de recherche et rend les URL lisibles.

```
routes.MapRoute(
  name: "AboutUsAspx", // Route name
  url: "AboutUs.aspx",  // URL with parameters
  defaults: new { controller = "Home", action = "AboutUs", id = UrlParameter.Optional }  //
Parameter defaults
);
```

Routage des attributs dans MVC

Avec les méthodes classiques de définition de route, les frameworks MVC WEB API 2 puis MVC 5 ont introduit le `Attribute routing` :

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        // This enables attribute routing and must go  before other routes are added to the
routing table.
        // This makes attribute routes have higher priority
        routes.MapMvcAttributeRoutes();
    }
}
```

Pour les routes avec le même préfixe dans un contrôleur, vous pouvez définir un préfixe commun pour des méthodes d'action complètes à l'intérieur du contrôleur à l'aide de l'attribut `RoutePrefix` .

```
[RoutePrefix("Custom")]
public class CustomController : Controller
{
    [Route("Index")]
    public ActionResult Index()
    {
        ...
    }
}
```

`RoutePrefix` est facultatif et définit la partie de l'URL préfixée par toutes les actions du contrôleur.

Si vous disposez de plusieurs routes, vous pouvez définir une route par défaut en capturant l'action en tant que paramètre, puis l'appliquer à la totalité du contrôleur, sauf si un attribut `Route` spécifique est défini sur certaines méthodes d'action qui remplacent la route par défaut.

```
[RoutePrefix("Custom")]
[Route("{action=index}")]
public class CustomController : Controller
{
    public ActionResult Index()
    {
        ...
    }

    public ActionResult Detail()
    {
        ...
    }
}
```

Bases du routage

Lorsque vous demandez l'url `yourSite/Home/Index` via un navigateur, le module de routage dirigera la requête vers la méthode d'action `Index` de la classe `HomeController` . Comment savoir comment envoyer la demande à la méthode spécifique de cette classe spécifique? il vient le RouteTable.

Chaque application possède une table de routage dans laquelle elle stocke le modèle de route et des informations sur la destination de la demande. Ainsi, lorsque vous créez votre application mvc, une route par défaut est déjà enregistrée dans la table de routage. Vous pouvez voir cela dans la classe `RouteConfig.cs` .

```
public static void RegisterRoutes(RouteCollection routes)
{
   routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
   routes.MapRoute("Default", "{controller}/{action}/{id}",
             new { controller = "Home", action = "Index", id = UrlParameter.Optional });
}
```

Vous pouvez voir que l'entrée a un nom et un modèle. Le modèle est le modèle de route à vérifier

lorsqu'une demande entre en jeu. Le modèle par défaut a `Home` comme valeur du segment d'URL du contrôleur et `Index` comme valeur du segment d'action. Cela signifie que si vous ne transmettez pas explicitement un nom de contrôleur et une action dans votre requête, il utilisera ces valeurs par défaut. C'est la raison pour laquelle vous obtenez le même résultat lorsque vous accédez à votre `yourSite/Home/Index` et à votre `yourSite`

Vous avez peut-être remarqué que nous avons un paramètre appelé id comme dernier segment de notre modèle de route. Mais dans les valeurs par défaut, nous spécifions que c'est facultatif. C'est la raison pour laquelle nous n'avons pas eu besoin de spécifier la valeur d'ID dans l'url que nous avons essayée.

Maintenant, retournez à la méthode d'action Index dans HomeController et ajoutez un paramètre à celle-ci.

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

Mettez maintenant un point de repère visuel en studio dans cette méthode. Exécutez votre projet et accédez à votre `yourSite/Home/Index/999` dans votre navigateur. Le point d'arrêt sera touché et vous devriez pouvoir voir que la valeur 999 est maintenant disponible dans le paramètre `id` .

Création d'un second motif de route

Disons que nous voudrions qu'il soit configuré de telle sorte que la même méthode d'action soit appelée pour un modèle de route différent. Nous pouvons le faire en ajoutant une nouvelle définition de route à la table de routage.

```
public static void RegisterRoutes(RouteCollection routes)
{
   routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

   // New custom route definition added
   routes.MapRoute("MySpecificRoute",
    "Important/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional });

   //Default catch all normal route definition
   routes.MapRoute("Default", "{controller}/{action}/{id}",
             new { controller = "Home", action = "Index", id = UrlParameter.Optional });
}
```

La nouvelle définition que j'ai ajoutée a un motif `Important/{id}` où id est à nouveau facultatif. Cela signifie que lorsque vous demandez votre `yourSiteName\Important` ou votre `yourSiteName\Important\888` , il sera envoyé à l'action Index de HomeController.

Enregistrement de l'ordre de définition de l'itinéraire

L'ordre d'enregistrement des itinéraires est important. Vous devez toujours enregistrer les modèles de route spécifiques avant la route par défaut générique.

Catch-all route

Supposons que nous voulons avoir un itinéraire qui autorise un nombre non lié de segments comme celui-ci:

- http://example.com/Products/ (voir tous les produits)
- http://example.com/Products/IT
- http://example.com/Products/IT/Laptops
- http://example.com/Products/IT/Laptops/Ultrabook
- http://example.com/Products/IT/Laptops/Ultrabook/Asus
- etc.

Nous devrions ajouter une route, normalement à la fin de la table de routage, car cela intercepterait probablement toutes les demandes, comme ceci:

```
routes.MapRoute("Final", "Route/{*segments}",
      new { controller = "Product", action = "View" });
```

Dans le contrôleur, une action qui pourrait gérer cela pourrait être:

```
public void ActionResult View(string[] segments /* <- the name of the parameter must match the
name of the route parameter */)
{
    // use the segments to obtain information about the product or category and produce data
to the user
    // ...
}
```

Route Catch-All pour activer le routage côté client

Il est recommandé de coder l'état de l'application Single Page (SPA) dans url:

```
my-app.com/admin-spa/users/edit/id123
```

Cela permet de sauvegarder et de partager l'état de l'application.
Lorsque l'utilisateur place une URL dans la barre d'adresse du navigateur et que les hits entrent, le serveur doit ignorer la partie côté client de l'URL demandée. Si vous diffusez votre SPA en tant que vue Razor rendue (résultat de l'action d'appel du contrôleur) plutôt qu'un fichier HTML statique, vous pouvez utiliser un itinéraire fourre-tout:

```
public class AdminSpaController
{
    [Route("~/admin-spa/{clienSidePart*}")]
    ActionResult AdminSpa()
    {
        ...
    }
}
```

Dans ce cas, le serveur ne renvoie que SPA et s'initialise ensuite en fonction de la route. Cette

approche est plus flexible car elle ne dépend pas du module de réécriture d'url .

Routage des attributs dans les zones

Pour utiliser le routage d'attribut dans les zones, les zones d'enregistrement et les `[RouteArea(...)]` sont requises.

Dans `RouteConfig.cs` :

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapMvcAttributeRoutes();
        AreaRegistration.RegisterAllAreas();
    }
}
```

Dans une définition de routage d'attribut de contrôleur de zone d'échantillon:

```
[RouteArea("AreaName", AreaPrefix = "AreaName")]
[RoutePrefix("SampleAreaController")]
public class SampleAreaController : Controller
{
    [Route("Index")]
    public ActionResult Index()
    {
        return View();
    }
}
```

Pour utiliser les liens `Url.Action` dans les zones:

```
@Url.Action("Index", "SampleAreaController", new { area = "AreaName" })
```

Lire Le routage en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/1534/le-routage

Chapitre 20: Modèles d'affichage et d'édition

Introduction

Lorsque vous traitez des objets dans une application MVC, si un objet doit être affiché à plusieurs endroits du même format, nous aurons besoin d'une présentation standardisée. ASP.NET MVC a simplifié ce type de standardisation en intégrant des modèles d'affichage et d'éditeur. En bref, les modèles d'affichage et d'éditeur sont utilisés pour standardiser la présentation présentée à l'utilisateur lors de la modification ou de l'affichage de certains types ou classes.

Examples

Modèle d'affichage

Modèle:

```
public class User
{
   public int ID { get; set; }
   public string FirstName { get; set; }
   public DateTime DateOfBirth { get; set; }
}
```

Si nous voulons afficher les utilisateurs dans différentes vues, il serait préférable de créer une présentation standardisée pour ces utilisateurs, quel que soit l'endroit où ils doivent être affichés. Nous pouvons accomplir cela en utilisant des modèles d'affichage.

Un modèle d'affichage est simplement une vue partielle liée à l'objet à afficher et se `Views/Shared/DisplayTemplates` dossier `Views/Shared/DisplayTemplates` (même si vous pouvez également la placer dans `Views/ControllerName/DisplayTemplates`). De plus, **le nom de la vue (par défaut) doit être le nom de l'objet que vous souhaitez utiliser comme modèle** .

> Views / Shared / DisplayTemplates / User.cshtml

```
 @model TemplatesDemo.Models.User

<div style="padding-bottom: 10px">
    <p><strong>ID:</strong> @Html.DisplayFor(m => m.ID)</p>
    <p><strong>Name:</strong> @Html.DisplayFor(m => m.FirstName)</p>
    <p><strong>Date of Birth:</strong> @Html.DisplayFor(m => m.DateOfBirth)</p>
</div>
<hr/>
```

Maintenant, si nous voulons afficher tous les utilisateurs de la base de données et les afficher dans différentes vues, nous pouvons simplement envoyer la liste des utilisateurs à la vue et utiliser le modèle d'affichage pour les afficher. Nous pouvons utiliser l'une des deux méthodes suivantes:

```
Html.DisplayFor()
Html.DisplayForModel()
```

`DisplayFor` appelle le modèle d'affichage pour le type de la propriété sélectionnée (par exemple `Html.DisplayFor(x => x.PropertyName)` . `DisplayForModel` appelle le modèle d'affichage pour le modèle `@model` de la vue

Vue

```
@model IEnumerable<TemplatesDemo.Models.User>
@{
    ViewBag.Title = "Users";
}

<h2>Users</h2>

@Html.DisplayForModel()
```

Modèle de l'éditeur

Les modèles d'affichage peuvent être utilisés pour standardiser la disposition d'un objet, voyons maintenant comment nous pouvons faire la même chose pour ces objets lors de leur édition. Tout comme les modèles d'affichage, il existe deux manières d'appeler des modèles d'éditeur pour un type donné:

```
Html.EditorFor()
Html.EditorForModel()
```

Les modèles d'éditeur, tout comme les modèles d'affichage, doivent exister dans **Views / Shared / EditorTemplates** ou **Views / ControllerName / EditorTemplates** . Pour cette démonstration, nous les créerons dans le dossier Shared. Encore une fois, **le nom de la vue (par défaut) doit être le nom de l'objet que vous souhaitez utiliser comme modèle.**

Modèle

```
public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public DateTime DateOfBirth { get; set; }
        public Roles Roles { get; set; }
        public int RoleId { get; set; }
    }

 public class Roles
    {
        public int Id { get; set; }
        public string Role { get; set; }
    }
```

Disons que nous voulons pouvoir éditer n'importe quel utilisateur de la base de données dans plusieurs vues. Nous utiliserons un **ViewModel** à cette fin.

ViewModel

```
public class UserEditorViewModel
    {
        public User User { get; set; }
        public IEnumerable<Roles> Roles { get; set; }
    }
```

En utilisant ce **ViewModel** , nous allons créer un **modèle d'éditeur**

> Vues / Shared / EditorTemplates / UserEditorViewModel.cshtml

```
@model TemplatesDemo.Models.UserEditorViewModel

<div class="form-group">
    @Html.DisplayNameFor(m => m.User.Id)
    @Html.EditorFor(m => m.User.Id)
</div>
<div class="form-group">
    @Html.DisplayNameFor(m => m.User.Name)
    @Html.EditorFor(m => m.User.Name)
</div>
<div class="form-group">
    @Html.DisplayNameFor(m => m.User.DateOfBirth)
    @Html.EditorFor(m => m.User.DateOfBirth)
</div>
<div class="form-group">
    @Html.DisplayNameFor(m => m.User.Roles.Role)
    @Html.DropDownListFor(m => m.User.RoleId, new SelectList(Model.Roles,"Id","Role"))
</div>
```

Nous allons obtenir l'utilisateur souhaité et la liste des rôles disponibles et les lier dans viewModel **UserEditorViewModel** dans l'action du contrôleur et envoyer le viewModel à la vue. Pour plus de simplicité, je lance le viewModel à partir de l'action

action

```
 public ActionResult Editor()
        {
            var viewModel = new UserEditorViewModel
            {
                User = new User
                {
                    Id = 1,
                    Name = "Robert",
                    DateOfBirth = DateTime.Now,
                    RoleId = 1
                },
                Roles = new List<Roles>()
                {
                    new Roles
                    {
                        Id = 1,
                        Role = "Admin"
                    },
                    new Roles
                    {
```

```
                        Id = 2,
                        Role = "Manager"
                    },
                    new Roles
                    {
                        Id = 3,
                        Role = "User"
                    }
                }
            };

            return View(viewModel);
        }
```

Nous pouvons utiliser le **modèle de l'éditeur** créé dans n'importe quelle vue que nous souhaitons

Vue

```
@model TemplatesDemo.Models.UserEditorViewModel

@using (Html.BeginForm("--Your Action--", "--Your Controller--"))
{
    @Html.EditorForModel()
    <input type="submit" value="Save" />
}
```

Lire Modèles d'affichage et d'édition en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/9784/modeles-d-affichage-et-d-edition

Chapitre 21: MVC vs Web Forms

Introduction

Avant de sauter dans ASP. NET MVC pour développer votre application Web, vous devez considérer les avantages et les inconvénients du framework et vous devez savoir qu'il existe un autre framework Web créé et géré par Microsoft, à savoir ASP .NET Web Forms.

Lequel choisiriez-vous est une question de connaissance des deux techonologies.

Syntaxe

- ASPX View Engine utilise "<% =%>" ou "<%:%>" pour afficher le contenu côté serveur.
- Le moteur de vue de rasoir utilise @ pour rendre le contenu côté serveur.

Remarques

https://www.asp.net/web-forms

https://www.asp.net/mvc

Examples

Avantages des formulaires Web ASP .NET

- Contrôles de pré-construction pour gérer les grilles, les entrées, les graphiques, les arbres, etc.
- Il prend en charge un modèle d'événement qui préserve l'état sur HTTP, ce qui profite au développement d'applications Web métier. L'application basée sur Web Forms fournit des dizaines d'événements pris en charge dans des centaines de contrôles serveur.
- Il utilise un modèle de contrôleur de page qui ajoute des fonctionnalités à des pages individuelles. Pour plus d'informations, voir Contrôleur de page sur le site Web MSDN.
- Il utilise des formulaires d'état d'affichage ou de serveur, ce qui facilite la gestion des informations d'état.
- Cela fonctionne bien pour les petites équipes de développeurs Web et de concepteurs qui veulent tirer parti du grand nombre de composants disponibles pour le développement rapide d'applications.
- En général, il est moins complexe pour le développement d'applications, car les composants (la classe Page, les contrôles, etc.) sont étroitement intégrés et nécessitent généralement

moins de code que le modèle MVC.

- Modèle de développement facile pour les développeurs issus du développement WindowsForm.

Que sont les formulaires Web

Avantages d'une application Web basée sur MVC

- Cela facilite la gestion de la complexité en divisant une application dans le modèle, la vue et le contrôleur (séparation des problèmes).
- Il n'utilise pas les formulaires d'affichage d'état ou de serveur. Cela rend le framework MVC idéal pour les développeurs qui veulent un contrôle total sur le comportement d'une application.
- Il utilise un modèle de contrôleur frontal qui traite les demandes d'application Web via un seul contrôleur. Cela vous permet de concevoir une application prenant en charge une infrastructure de routage riche. Pour plus d'informations, voir Front Controller sur le site Web MSDN.
- Il fournit un meilleur support pour le développement piloté par les tests (TDD).
- Cela fonctionne bien pour les applications Web qui sont prises en charge par de grandes équipes de développeurs et de concepteurs Web qui ont besoin d'un haut degré de contrôle sur le comportement des applications.

Que sont les formulaires Web

Désavantages

Formulaires Web:

- Cycle de vie complexe, chaque fois qu'une demande est faite sur le serveur, il existe au moins 5 méthodes à exécuter avant le gestionnaire d'événements.
- Dificult pour travailler avec les frameworks côté client comme JQuery ou Angular.
- Difficile de travailler avec le Javascript et le XML asyncronieux (AJAX)
- Manipulation de Viewstate
- Le côté client de la page et le code derrière sont étroitement liés.

MVC:

- Il faut plus de temps pour se développer en comparaison avec les formulaires Web.
- Les données sont envoyées au serveur sous forme de texte en clair, tandis que dans les formulaires Web, les données d'état sont chiffrées par défaut.

Razor View Engine VS ASPX View Engine

Rasoir (MVC)	ASPX (Web Forms)
L'espace de noms utilisé par Razor View Engine est System.Web.Razor	L'espace de noms utilisé par ASPX View Engine est System.Web.Mvc.WebFormViewEngine
Les extensions de fichier utilisées par Razor View Engine sont différentes d'un moteur de vue Web. Il utilise cshtml avec C # et vbhtml avec vb pour les vues, les vues partielles, les modèles et les pages de mise en page.	Les extensions de fichiers utilisées par les Web Form View Engines sont similaires aux formulaires Web ASP.Net. Il utilise l'extension ASPX pour afficher l'extension aspc des vues partielles ou des contrôles utilisateur ou des modèles et des extensions principales pour les pages de disposition / maquette.
Le moteur de rasage prend en charge le développement piloté par les tests (TDD).	Le moteur de vue Web Form ne prend pas en charge TDD (Test Driven Development) car il dépend de la classe System.Web.UI.Page pour rendre le test complexe.

Moteur de visualisation ASPX View Engine VS Razor

Lire MVC vs Web Forms en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/8584/mvc-vs-web-forms

Chapitre 22: Opération CRUD

Introduction

L'opération CRUD fait référence aux opérations classiques (créer, lire, mettre à jour, supprimer) en ce qui concerne les données.

Dans le contexte ASP MVC, il existe plusieurs façons de mettre vos données CRUD en œuvre à l'aide de modèles et, par la suite, des vues, des contrôleurs.

Un moyen simple consiste à utiliser la fonction d'échafaudage fournie par les modèles Visual Studio et à l'adapter à vos besoins.

N'oubliez pas que CRUD est défini de manière très large et qu'il présente de nombreuses variantes en fonction de vos besoins. Par exemple, la base de données en premier, l'entité en premier, etc.

Remarques

Pour simplifier, cette opération CRUD utilise un contexte de structure d'entité dans le contrôleur. Ce n'est pas une bonne pratique, mais cela dépasse le cadre de ce sujet. Cliquez dans le cadre des entités si vous souhaitez en savoir plus.

Examples

Créer - Partie Contrôleur

Pour implémenter la fonctionnalité de création, nous avons besoin de deux actions: *GET* et *POST* .

1. L'action *GET* utilisée pour retourner la vue affichera un formulaire permettant à l'utilisateur d'entrer des données à l'aide d'éléments HTML. Si des valeurs par défaut doivent être insérées avant que l'utilisateur ajoute des données, elles doivent être affectées aux propriétés du modèle de vue pour cette action.

2. Lorsque l'utilisateur remplit le formulaire et clique sur le bouton "Enregistrer", nous traiterons les données du formulaire. À cause de cela, nous avons maintenant besoin de l'action *POST* . Cette méthode sera responsable de la gestion des données et de leur enregistrement dans la base de données. En cas d'erreurs, la même vue renvoyée avec les données de formulaire stockées et le message d'erreur explique quel problème survient après la soumission de l'action.

Nous allons implémenter ces deux étapes dans deux méthodes Create () de notre classe de contrôleur.

```
    // GET: Student/Create
    // When the user access this the link ~/Student/Create a get request is made to controller
Student and action Create, as the page just need to build a blank form, any information is
needed to be passed to view builder
    public ActionResult Create()
    {
        // Creates a ViewResult object that renders a view to the response.
        // no parameters means: view = default in this case Create and model = null
        return View();
    }

    // POST: Student/Create
    [HttpPost]
    // Used to protect from overposting attacks, see
http://stackoverflow.com/documentation/asp.net-mvc/1997/html-antiforgerytoke for details
    [ValidateAntiForgeryToken]
    // This is the post request with forms data that will be bind the action, if in the data
post request have enough information to build a Student instance that will be bind
    public ActionResult Create(Student student)
    {
        try
        {
        //Gets a value that indicates whether this instance received from the view is valid.
            if (ModelState.IsValid)
            {
                // Adds to the context
                db.Students.Add(student);
                // Persist the data
                db.SaveChanges();
                // Returns an HTTP 302 response to the browser, which causes the browser to
make a GET request to the specified action, in this case the index action.
                return RedirectToAction("Index");
            }
        }
        catch
        {
            // Log the error (uncomment dex variable name and add a line here to write a log).
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the
problem persists see your system administrator.");
        }
        // view = default in this case Create and model = student
        return View(student);
    }
```

Créer - Afficher la pièce

```
@model ContosoUniversity.Models.Student

//The Html.BeginForm helper Writes an opening <form> tag to the response. When the user
submits the form, the request will be processed by an action method.
@using (Html.BeginForm())
{
    //Generates a hidden form field (anti-forgery token) that is validated when the form is
submitted.
    @Html.AntiForgeryToken()

<div class="form-horizontal">
    <h4>Student</h4>
    <hr />
```

```
    //Returns an unordered list (ul element) of validation messages that are in the
ModelStateDictionary object.
    @Html.ValidationSummary(true, "", new { @class = "text-danger" })

    <div class="form-group">
        //Returns an HTML label element and the property name of the property that is
represented by the specified expression.
        @Html.LabelFor(model => model.LastName, htmlAttributes: new { @class = "control-label
col-md-2" })

        <div class="col-md-10">
            //Returns an HTML input element for each property in the object that is
represented by the Expression expression.
            @Html.EditorFor(model => model.LastName, new { htmlAttributes = new { @class =
"form-control" } })

            //Returns the HTML markup for a validation-error message for each data field that
is represented by the specified expression.
            @Html.ValidationMessageFor(model => model.LastName, "", new { @class = "text-
danger" })
        </div>
    </div>

    <div class="form-group">
        @Html.LabelFor(model => model.FirstMidName, htmlAttributes: new { @class = "control-
label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.FirstMidName, new { htmlAttributes = new { @class =
"form-control" } })
            @Html.ValidationMessageFor(model => model.FirstMidName, "", new { @class = "text-
danger" })
        </div>
    </div>

    <div class="form-group">
        @Html.LabelFor(model => model.EnrollmentDate, htmlAttributes: new { @class = "control-
label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.EnrollmentDate, new { htmlAttributes = new { @class
= "form-control" } })
            @Html.ValidationMessageFor(model => model.EnrollmentDate, "", new { @class =
"text-danger" })
        </div>
    </div>

    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Create" class="btn btn-default" />
        </div>
    </div>
</div>
}

<div>
    //Returns an anchor element (a element) the text is Back to List and action is Index
    @Html.ActionLink("Back to List", "Index")
</div>
```

Détails - Partie contrôleur

Par l'URL `~/Student/Details/5` étant: (~: racine du site, Etudiant: Contrôleur, Détails: Action, 5: identifiant étudiant), il est possible de récupérer l'étudiant par son identifiant.

```
// GET: Student/Details/5
    public ActionResult Details(int? id)
    {
        // it good practice to consider that things could go wrong so,it is wise to have a
validation in the controller
        if (id == null)
        {
            // return a bad request
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        Student student = db.Students.Find(id);
        if (student == null)
        {
            // if doesn't found return 404
            return HttpNotFound();
        }
        return View(student);
    }
```

Détails - Voir la partie

```
// Model is the class that contains the student data send by the controller and will be
rendered in the view
@model ContosoUniversity.Models.Student

<h2>Details</h2>

<div>
   <h4>Student</h4>
<hr />
<dl class="dl-horizontal">
    <dt>
        //Gets the display name for the model.
        @Html.DisplayNameFor(model => model.LastName)
    </dt>

    <dd>
        //Returns HTML markup for each property in the object that is represented by the
Expression expression.
        @Html.DisplayFor(model => model.LastName)
    </dd>

    <dt>
        @Html.DisplayNameFor(model => model.FirstMidName)
    </dt>

    <dd>
        @Html.DisplayFor(model => model.FirstMidName)
    </dd>

    <dt>
        @Html.DisplayNameFor(model => model.EnrollmentDate)
    </dt>

    <dd>
```

```
        @Html.DisplayFor(model => model.EnrollmentDate)
    </dd>
    <dt>
        @Html.DisplayNameFor(model => model.Enrollments)
    </dt>
    <dd>
        <table class="table">
            <tr>
                <th>Course Title</th>
                <th>Grade</th>
            </tr>
            @foreach (var item in Model.Enrollments)
            {
                <tr>
                    <td>
                        @Html.DisplayFor(modelItem => item.Course.Title)
                    </td>
                    <td>
                        @Html.DisplayFor(modelItem => item.Grade)
                    </td>
                </tr>
            }
        </table>
    </dd>
</dl>
</div>
<p>
    //Returns an anchor element (a element) the text is Edit, action is Edit and the route
value is the model ID property.
    @Html.ActionLink("Edit", "Edit", new { id = Model.ID }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

Edit - Partie contrôleur

```
 // GET: Student/Edit/5
 // It is receives a get http request for the controller Student and Action Edit with the id
of 5
    public ActionResult Edit(int? id)
    {
         // it good practice to consider that things could go wrong so,it is wise to have a
validation in the controller
        if (id == null)
        {
            // returns a bad request
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }

        // It finds the Student to be edited.
        Student student = db.Students.Find(id);
        if (student == null)
        {
            // if doesn't found returns 404
            return HttpNotFound();
        }
        // Returns the Student data to fill out the edit form values.
        return View(student);
    }
```

Cette méthode est très similaire à la méthode d'action des détails, ce qui est un bon candidat pour un refactoring, mais elle est hors de portée de ce sujet.

```
    // POST: Student/Edit/5
    [HttpPost]

    //used to To protect from overposting attacks more details see
http://stackoverflow.com/documentation/asp.net-mvc/1997/html-antiforgerytoke
    [ValidateAntiForgeryToken]

    //Represents an attribute that is used for the name of an action.
    [ActionName("Edit")]
    public ActionResult Edit(Student student)
    {
        try
        {
            //Gets a value that indicates whether this instance received from the view is
valid.
            if (ModelState.IsValid)
            {
                // Two thing happens here:
                // 1) db.Entry(student) -> Gets a DbEntityEntry object for the student entity
providing access to information about it and the ability to perform actions on the entity.
                // 2) Set the student state to modified, that means that the student entity is
being tracked by the context and exists in the database, and some or all of its property
values have been modified.
                db.Entry(student).State = EntityState.Modified;

                // Now just save the changes that all the changes made in the form will be
persisted.
                db.SaveChanges();

                // Returns an HTTP 302 response to the browser, which causes the browser to
make a GET request to the specified action, in this case the index action.
                return RedirectToAction("Index");
            }
        }
        catch
        {
            //Log the error add a line here to write a log.
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the
problem persists, see your system administrator.");
        }

        // return the invalid student instance to be corrected.
        return View(student);
    }
```

Supprimer - Partie contrôleur

Est-ce une bonne pratique pour résister à la tentation de faire l'action de suppression dans la demande d'obtention. Ce serait une énorme erreur de sécurité, il faut toujours le faire dans la méthode post.

```
    // GET: Student/Delete/5
    public ActionResult Delete(int? id)
    {
```

```
        // it good practice to consider that things could go wrong so,it is wise to have a
validation in the controller
        if (id == null)
        {
            // returns a bad request
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }

        // It finds the Student to be deleted.
        Student student = db.Students.Find(id);
        if (student == null)
        {
            // if doesn't found returns 404
            return HttpNotFound();
        }
        // Returns the Student data to show the details of what will be deleted.
        return View(student);
    }

    // POST: Student/Delete/5
    [HttpPost]

    //Represents an attribute that is used for the name of an action.
    [ActionName("Delete")]

    //used to To protect from overposting attacks more details see
http://stackoverflow.com/documentation/asp.net-mvc/1997/html-antiforgerytoke
    [ValidateAntiForgeryToken]
    public ActionResult Delete(int id)
    {
        try
        {
            // Finds the student
            Student student = db.Students.Find(id);

            // Try to remove it
            db.Students.Remove(student);

            // Save the changes
            db.SaveChanges();
        }
        catch
        {
            //Log the error add a line here to write a log.
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the
problem persists, see your system administrator.");
        }

        // Returns an HTTP 302 response to the browser, which causes the browser to make a GET
request to the specified action, in this case the index action.
        return RedirectToAction("Index");
    }
```

Lire Opération CRUD en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6380/operation-crud

Chapitre 23: Règles de réécriture IIS

Examples

Forcer HTTPS à l'aide de la règle de réécriture

Cet exemple montre comment vous pouvez utiliser les règles de réécriture IIS pour forcer HTTPS en faisant en sorte que toutes les requêtes HTTP renvoient une redirection 301 (permanente) vers la page HTTPS.

C'est généralement mieux que d'utiliser l'attribut `[RequireHttps]` car l'attribut utilise une redirection 302, et le fait d'être dans le pipeline MVC est beaucoup plus lent que le faire au niveau IIS.

```
   <rewrite xdt:Transform="Insert">
      <rules>
        <rule name="Enforce HTTPS WWW" stopProcessing="true">
          <match url=".*" />
          <conditions logicalGrouping="MatchAll" trackAllCaptures="true">
            <add input="{HTTP_HOST}" pattern="^(?!www)(.*)"/>
            <add input="{URL}" pattern="^(.*)"/>
            <!-- {URL} Gives the base portion of the URL, without any querystring or extra
path information, for example, "/vdir/default.asp". -->
          </conditions>
          <action type="Redirect" url="https://www.{C:1}{C:2}" appendQueryString="true"
redirectType="Permanent" />
        </rule>
      </rules>
    </rewrite>
```

Lire Règles de réécriture IIS en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6358/regles-de-reecriture-iis

Chapitre 24: Regroupement et Minification

Examples

Minification

La minification permet de réduire la taille des fichiers CSS et Javascript pour accélérer les temps de téléchargement. Ce processus est effectué en supprimant tous les espaces vides, les commentaires et tout autre contenu non essentiel des fichiers.

Ce processus est effectué automatiquement lors de l'utilisation d'un objet `ScriptBundle` ou d'un objet `StyleBundle` . Si vous devez le désactiver, vous devez utiliser un objet `Bundle` base.

Exemple utilisant Minification

Le code suivant utilise les directives de préprocesseur pour appliquer le regroupement uniquement pendant les éditions afin de faciliter le débogage lors des non-éditions (car les fichiers non fournis sont généralement plus faciles à parcourir):

```
public static void RegisterBundles(BundleCollection bundles)
{
    #if DEBUG
        bundles.Add(new Bundle("~/bundles/jquery").Include("~/Scripts/jquery-{version}.js"));
        bundles.Add(new Bundle("~/Content/css").Include("~/Content/site.css"));
    #else
        bundles.Add(new ScriptBundle("~/bundles/jquery").Include("~/Scripts/jquery-
{version}.js"));
        bundles.Add(new StyleBundle("~/Content/css").Include("~/Content/site.css"));
    #endif
}
```

Bundles Script et Style

Voici l'extrait de code par défaut pour le fichier BundleConfig.cs.

```
using System.Web.Optimization;

public class BundleConfig
{
    // For more information on Bundling, visit http://go.microsoft.com/fwlink/?LinkId=254725
    public static void RegisterBundles(BundleCollection bundles)
    {
        bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                    "~/Scripts/jquery-{version}.js"));

    // 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 StyleBundle("~/Content/css").Include("~/Content/site.css"));

    bundles.Add(new StyleBundle("~/Content/themes/base/css").Include(
                "~/Content/themes/base/jquery.ui.core.css",
                "~/Content/themes/base/jquery.ui.resizable.css",
    }
 }
```

Les bundles sont enregistrés dans le fichier Global.asax dans la méthode Application_Start ():

```
using System.Web.Optimization;

protected void Application_Start()
{
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}
```

Les bundles doivent être rendus dans vos vues comme suit:

```
@using System.Web.Optimization

@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/modernizr")
@Styles.Render("~/Content/css")
@Styles.Render("~/Content/themes/base/css")
```

Notez que le regroupement ne se produit pas lorsque vous êtes en mode développement (où l'élément de compilation du fichier Web.config est défini sur debug = "true"). Au lieu de cela, les instructions de rendu dans vos vues incluront chaque fichier individuel dans un format non minifié et non intégré, pour faciliter le débogage.

Une fois l'application en mode de production (où l'élément de compilation dans le fichier Web.config est défini sur debug = "false"), le regroupement aura lieu.

Cela peut entraîner des complications pour les scripts qui référencent les chemins d'accès relatifs d'autres fichiers, tels que les références aux fichiers d'icône de Twitter Bootstrap. Cela peut être résolu en utilisant la classe CssRewriteUrlTransform de System.Web.Optimization:

```
 bundles.Add(new StyleBundle("~/bundles/css").Include(
                "~/Content/css/*.css", new CssRewriteUrlTransform()));
```

La classe CssRewriteUrlTransform réécrira les URL relatives dans les fichiers groupés en chemins absolus, de sorte que les références resteront intactes une fois que la référence appelante aura été déplacée vers l'emplacement du bundle (en utilisant le code ci-dessus, en partant de "~ / Content / css / bootstrap.css "à" ~ / bundles / css / bootstrap.css ").

Lire Regroupement et Minification en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/1959/regroupement-et-minification

Chapitre 25: Reliure modèle

Introduction

La liaison de modèle est le processus consistant à prendre des paramètres HTTP, généralement dans la chaîne de requête d'une requête GET ou dans le corps POST, et à les appliquer à un objet pouvant être validé et orienté objet sans actions du contrôleur. avoir une connaissance intime de la façon de récupérer les paramètres HTTP.

En d'autres termes, la liaison de modèle est ce qui permet aux actions, dans MVC, d'avoir un ou plusieurs paramètres, qu'il s'agisse d'un type de valeur ou d'un objet.

Remarques

Pour essayer de créer une instance dans l'action, le processus de modèle de liaison recherche les données à différents endroits:

- Données de formulaire
- Données d'itinéraire
- Chaîne de requête
- Fichiers personnalisés (cookies par exemple)

Examples

Liaison de valeur d'itinéraire

Étant donné un routage par défaut tel que `{controller=Home}/{action=Index}/{id?}` Si vous avez l'URL `https://stackoverflow.com/questions/1558902`

Cela irait au QuestionsController et la valeur 1558902 serait mappée à un paramètre id d'une action d'index, c.-à-d.

```
public ActionResult Index(int? id){
     //id would be bound to id of the route
}
```

Liaison de chaîne de requête

Pour prolonger la liaison, indiquez une URL telle que `https://stackoverflow.com/questions/1558902?sort=desc`

et routage comme `{controller=Home}/{action=Index}/{id?}`

```
public ActionResult Index(int? id, string sort){
     //sort would bind to the value in the query string, i.e. "desc"
}
```

Liaison aux objets

Vous travailliez souvent avec des classes viewmodel dans asp.net-mvc et souhaitiez les associer à des propriétés. Cela fonctionne de la même manière que la correspondance avec des paramètres individuels.

Disons que vous aviez un simple modèle de vue appelez PostViewModel comme ceci

```
public class PostViewModel{
  public int Id {get;set;}
  public int SnappyTitle {get;set;}
}
```

Ensuite, vous aviez publié des valeurs de Id et SnappyTitle à partir d'un formulaire dans la requête http, puis ils mappaient directement sur ce modèle si le modèle lui-même était le paramètre d'action, par exemple

```
public ActionResult UpdatePost(PostViewModel viewModel){
  //viewModel.Id would have our posted value
}
```

Il convient de noter que la liaison est insensible à la casse pour les noms de paramètre et de propriété. Il jettera également des valeurs si possible. Je laisse plus de cas pour des exemples spécifiques

Liaison Ajax

Ce sont des valeurs de formulaire qui vont dans la requête HTTP en utilisant la méthode POST. (y compris les requêtes POST jQuery).

Dis que tu as fait un post ajax comme

```
$.ajax({
    type: 'POST',
    url: window.updatePost,
    data:  { id: 21, title: 'snappy title' },
    //kept short for clarity
});
```

Ici, les deux valeurs de json, id et title, seraient liées à l'action correspondante, par exemple

```
public JsonResult UpdatePost(int id, string title) {
    ...
}
```

Générique, liaison de modèle basée sur la session

Parfois, nous avons besoin de préserver *tout le modèle* et de le transférer à travers des actions ou même des contrôleurs. Stockage du modèle à la session bonne solution pour ce type d'exigences.

Si nous combinons cela avec les puissantes fonctionnalités de *liaison de modèles* de MVC, nous obtenons une manière élégante de le faire. Nous pouvons créer une liaison de modèle générique basée sur la session en trois étapes simples:

Première étape: Créer un modèle de classeur

Créez un modèle de classeur lui-même. Personnellement, j'ai créé la classe *SessionDataModelBinder* dans le dossier / *Infrastructure* / *ModelBinders* .

```
using System;
using System.Web.Mvc;

public class SessionDataModelBinder<TModel>
    : IModelBinder
    where TModel : class
{
    private string SessionKey { get; set; }

    public SessionDataModelBinder(string sessionKey)
    {
        if (string.IsNullOrEmpty(sessionKey))
            throw new ArgumentNullException(nameof(sessionKey));
        SessionKey = sessionKey;
    }

    public object BindModel(
        ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        // Get model from session
        TModel model = controllerContext
            .HttpContext
            .Session[SessionKey] as TModel;
        // Create model if it wasn't found from session and store it
        if (model == null)
        {
            model = Activator.CreateInstance<TModel>();
            controllerContext.HttpContext.Session[SessionKey] = model;
        }
        // Return the model
        return model;
    }
}
```

Deuxième étape: enregistrer le classeur

Si nous avons un modèle comme ci-dessous:

```
public class ReportInfo
{
    public int ReportId { get; set; }
    public ReportTypes TypeId { get; set; }
}

public enum ReportTypes
{
    NotSpecified,
```

```
    Monthly, Yearly
}
```

Nous pouvons enregistrer un modèle de classeur basé sur la session pour ce modèle dans **Global.asax** dans la méthode *Application_Start* :

```
protected void Application_Start()
{
    .........

    // Model binders.
    // Remember to specy unique SessionKey
    ModelBinders.Binders.Add(typeof(ReportInfo),
        new SessionDataModelBinder<ReportInfo>("ReportInfo"));
}
```

Troisième étape: utilisez-le!

Nous pouvons maintenant bénéficier de ce modèle en *ajoutant* simplement des *paramètres à nos actions* :

```
public class HomeController : Controller
{
    public ActionResult Index(ReportInfo reportInfo)
    {
        // Simply set properties
        reportInfo.TypeId = ReportTypes.Monthly;

        return View();
    }

    public ActionResult About(ReportInfo reportInfo)
    {
        // reportInfo.TypeId is Monthly now because we set
        // it previously in Index action.
        ReportTypes currentReportType = reportInfo.TypeId;

        return View();
    }
}
```

Empêcher la liaison sur PostModel

Considérant un modèle (post):

```
public class User
{
    public string FirstName { get; set; }
    public bool IsAdmin { get; set; }
}
```

Avec une vue comme ça:

```
@using (Html.BeginForm()) {
```

```
    @Html.EditorFor(model => model.FirstName)
    <input type="submit" value="Save" />
}
```

Afin d'empêcher un utilisateur malveillant d'attribuer IsAdmin, vous pouvez utiliser l'attribut `Bind` dans l'action:

```
[HttpPost]
public ViewResult Edit([Bind(Exclude = "IsAdmin")] User user)
{
    // ...
}
```

Téléchargement de fichiers

Modèle:

```
public class SampleViewModel
{
    public HttpPostedFileBase file {get;set;}
}
```

Vue:

```
@model HelloWorldMvcApp.SampleViewModel

@using (Html.BeginForm("Index","Home",FormMethod.Post, new { enctype = "multipart/form-data"
}))
{
    <div class="form-group">
        @Html.TextBoxFor(model => model.file, new {@class="form-control", type="file"})
        @Html.ValidationMessageFor(model => model.file)
    </div>

    <button type="submit" class="btn btn-success submit">Upload</button>
}
```

Action:

```
[HttpPost]
public ActionResult Index(SampleViewModel model)
{

    if (model.file.ContentLength > 0)
    {
        string fileName = Path.GetFileName(model.file.FileName);
        string fileLocation = "~/App_Data/uploads/"+ fileName;
        model.file.SaveAs(Server.MapPath(fileLocation));
    }
    return View(model);
}
```

Validation manuelle des champs de date avec des formats dynamiques à

l'aide du modèle de classeur

Si différents utilisateurs ont besoin d'un format datetime différent, vous devrez peut-être analyser votre chaîne de date entrante à la date réelle en fonction du format. Dans ce cas, cet extrait peut vous aider.

```
public class DateTimeBinder : DefaultModelBinder
{
    public override object BindModel(ControllerContext controllerContext, ModelBindingContext
bindingContext)
    {

                var value = bindingContext.ValueProvider.GetValue(bindingContext.ModelName);
                DateTime date;
                var displayFormat = Session["DateTimeFormat"];
                if (value.AttemptedValue != "")
                {
                    if (DateTime.TryParseExact(value.AttemptedValue, displayFormat,
CultureInfo.InvariantCulture, DateTimeStyles.None, out date))
                    {
                        return date;
                    }
                    else
                    {
                        bindingContext.ModelState.AddModelError(bindingContext.ModelName,
"Invalid date format");
                    }
                }
            }

        return base.BindModel(controllerContext, bindingContext);
    }
```

Lire Reliure modèle en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/1258/reliure-modele

Chapitre 26: T4MVC

Introduction

T4MVC est un modèle T4 qui génère des helpers fortement typés à utiliser dans les mécanismes de routage MVC, par opposition aux chaînes magiques. T4MVC détectera les différents contrôleurs, actions et vues, et créera des références à ces vues, générant des erreurs de compilation au cas où une tentative de routage ou d'accès à une vue n'est pas valide.

Examples

Appel d'une action

Dans MVC, vous devez spécifier une action à des fins de routage, que ce soit pour un lien, une action de formulaire ou une redirection vers une action. Vous pouvez spécifier une action via l'espace de noms MVC.

Lorsqu'il est donné un contrôleur, tel que `HomeController` :

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ...
    }
    public ActionResult MyAction()
    {
        ...
    }
    public ActionResult MyActionWithParameter(int parameter)
    {
        ...
    }
}
```

T4MVC générera un contrôleur hérité qui remplace l'action. Cette substitution définira correctement les données de route afin que `UrlHelper` de MVC génère l'URL appropriée. Vous pouvez appeler cette méthode et la transmettre aux différentes méthodes pour `UrlHelper` . Les exemples ci-dessous supposent que l'itinéraire MVC par défaut est utilisé:

Lien

Pour générer une `a` étiquette avec le texte spécifié:

```
@Html.ActionLink("Link Text", MVC.Home.Index() )
//result: <a href="/">Link Text</a>
@Html.ActionLink("Link Text", MVC.Home.MyAction() )
//result: <a href="/Home/MyAction">Link Text</a>
//T4MVC also allows you to specify the parameter without creating an anonymous object:
```

```
@Html.ActionLink("Link Text", MVC.Home.MyActionWithParameter(1) )
//result: <a href="/Home/MyActionWithParameter/1">Link Text</a>
```

Pour générer une URL:

```
@Url.Action( MVC.Home.Index() )
//result: /
@Url.Action("Link Text", MVC.Home.MyAction() )
//result: /Home/MyAction
@Url.Action("Link Text", MVC.Home.MyActionWithParameter(1) )
//result: /Home/MyActionWithParameter/1
```

Notez que T4MVC suit les mêmes règles que MVC Routing - il ne spécifie pas les variables de route par défaut, de sorte que l'action `Index` sur le `HomeController` ne génère pas `/Home/Index` mais plutôt la forme parfaitement valide et abrégée de `/` .

Méthode de formulaire

Pour générer une balise de `form` avec l' `action` correcte spécifiée:

```
@Html.BeginForm( MVC.Home.Index(), FormMethod.Get /* or FormMethod.Post */ )
{
    //my form
}
//result:
<form action="/" method="GET">
    //my form
</form>
@Html.BeginForm( MVC.Home.MyActionWithParameter(1), FormMethod.Get /* or FormMethod.Post */ )
{
    //my form
}
//result:
<form action="/Home/MyActionWithParameter/1" method="GET">
    //my form
</form>
```

Redirect to Action

Dans un contrôleur, vous pouvez vouloir rediriger vers une action de l'action en cours. Cela peut être fait, comme par exemple:

```
public class RedirectingController : Controller
{
    public ActionResult MyRedirectAction()
    {
        ...
        return RedirectToAction( MVC.Redirecting.ActionToRedirectTo() );
        //redirects the user to the action below.
    }
    public ActionResult ActionToRedirectTo()
    {
        ...
    }
}
```

Lire T4MVC en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/9147/t4mvc

Chapitre 27: Traitement des erreurs HTTP

Introduction

Chaque site Web doit gérer les erreurs. Vous pouvez laisser vos utilisateurs voir les pages d'erreur 404 ou 500 stockées par IIS ou, à l'aide de Web.Config et d'un simple contrôleur, capturer ces erreurs et fournir vos propres pages d'erreur personnalisées.

Examples

Configuration de base

Cet exemple couvrira la création d'une page d'erreur personnalisée pour 404 Page Not Found et 500 Server Error. Vous pouvez étendre ce code pour capturer tout code d'erreur dont vous avez besoin.

Web.Config

Si vous utilisez IIS7 et supérieur, ignorez le noeud `<CustomError..` et utilisez `<httpErrors...`

Ajoutez ce qui suit dans le nœud `system.webServer` :

```
<httpErrors errorMode="Custom" existingResponse="Replace">
    <remove statusCode="404" />
    <remove statusCode="500" />
    <error statusCode="404" path="/error/notfound" responseMode="ExecuteURL" />
    <error statusCode="500" path="/error/servererror" responseMode="ExecuteURL" />
 </httpErrors>
```

Cela indique au site de diriger les erreurs 404 vers `~/error/notfound` et toute erreur 500 vers `~/error/servererror` . Il préservera également l'URL demandée (pensez à *transférer* plutôt qu'à *rediriger*) pour que l'utilisateur ne voie jamais l'URL de la page `~/error/...`

Ensuite, vous avez besoin d'un nouveau contrôleur d' `Error` afin ...

```
public class ErrorController : Controller
{
    public ActionResult servererror()
    {
        Response.TrySkipIisCustomErrors = true;
        Response.StatusCode = (int)HttpStatusCode.InternalServerError;
        return View();
    }

    public ActionResult notfound()
    {
        Response.TrySkipIisCustomErrors = true;
        Response.StatusCode = (int)HttpStatusCode.NotFound;
        return View();
    }
```

```
}
```

L'essentiel à noter ici est le `Response.TrySkipIisCustomErrors = true;` . Cela contournera IIS et forcera votre page d'erreur à travers.

Enfin, créez les `NotFound` et `ServerError` correspondantes et `ServerError` -les en forme de manière à ce qu'elles soient parfaitement `ServerError` la conception de votre site.

Hey presto - pages d'erreur personnalisées.

Lire Traitement des erreurs HTTP en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/9137/traitement-des-erreurs-http

Chapitre 28: Utilisation de plusieurs modèles en une seule vue

Introduction

Le sujet principal de cette rubrique utilisant plusieurs classes de modèles dans la couche d'affichage de MVC

Examples

Utilisation de plusieurs modèles dans une vue avec ExpandoObject dynamique

ExpandoObject (l'espace de noms `System.Dynamic`) est une classe qui a été ajoutée à `.Net Framework 4.0` . Cette classe nous permet d'ajouter et de supprimer dynamiquement des propriétés sur un objet lors de l'exécution. En utilisant l'objet Expando, nous pouvons ajouter nos classes de modèle dans un objet Expando créé dynamiquement. L'exemple suivant explique comment utiliser cet objet dynamique.

Modèle d'enseignant et d'étudiant:

```
public class Teacher
{
    public int TeacherId { get; set; }
    public string Name { get; set; }
}

public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
}
```

Méthodes de liste des enseignants et des étudiants:

```
public List<Teacher> GetTeachers()
{
    List<Teacher> teachers = new List<Teacher>();
    teachers.Add(new Teacher { TeacherId = 1, Name = "Teacher1" });
    teachers.Add(new Teacher { TeacherId = 2, Name = "Teacher2" });
    teachers.Add(new Teacher { TeacherId = 3, Name = "Teacher3" });
    return teachers;
}

public List<Student> GetStudents()
{
    List<Student> students = new List<Student>();
    students.Add(new Student { StudentId = 1, Name = "Student1"});
    students.Add(new Student { StudentId = 2, Name = "Student2"});
```

```
    students.Add(new Student { StudentId = 3, Name = "Student3"});
    return students;
}
```

Contrôleur (utilisant un modèle dynamique):

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Message = "Hello World";
        dynamic mymodel = new ExpandoObject();
        mymodel.Teachers = GetTeachers();
        mymodel.Students = GetStudents();
        return View(mymodel);
    }
}
```

Vue:

```
@using ProjectName ; // Project Name
@model dynamic
@{
    ViewBag.Title = "Home Page";
}
<h2>@ViewBag.Message</h2>

<h2>Teacher List</h2>

<table>
    <tr>
        <th>Id</th>
        <th>Name</th>
    </tr>
    @foreach (Teacher teacher in Model.Teachers)
    {
        <tr>
            <td>@teacher.TeacherId</td>
            <td>@teacher.Name</td>
        </tr>
    }
</table>

<h2>Student List</h2>

<table>
    <tr>
        <th>Id</th>
        <th>Name</th>
    </tr>
    @foreach (Student student in Model.Students)
    {
        <tr>
            <td>@student.StudentId</td>
            <td>@student.Name</td>
        </tr>
    }
</table>
```

Lire Utilisation de plusieurs modèles en une seule vue en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/10144/utilisation-de-plusieurs-modeles-en-une-seule-vue

Chapitre 29: Validation automatique côté client à partir des attributs

Remarques

Par défaut, Safari n'applique pas la validation d'éléments HTML5. Vous devez remplacer cela manuellement en utilisant d'autres moyens.

Examples

Modèle

```
public class UserModel
{

        [Required]
        [StringLength(6, MinimumLength = 3)]
        [RegularExpression(@"(\S)+", ErrorMessage = "White space is not allowed")]
        public string UserName { get; set; }

        [Required]
        [StringLength(8, MinimumLength = 3)]
        public string FirstName { get; set; }

        [Required]
        [StringLength(9, MinimumLength = 2)]
        public string LastName { get; set; }

        [Required]
        public string City { get; set; }

}
```

Paramètres web.config

```
<appSettings>
  <add key="ClientValidationEnabled" value="true"/>
  <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
 </appSettings>
```

Forfaits Nuget requis

```
<package id="jQuery" version="1.10.2" targetFramework="net452" />
<package id="jQuery.Validation" version="1.11.1" targetFramework="net452" />
<package id="Microsoft.jQuery.Unobtrusive.Validation" version="3.2.3" targetFramework="net452"
/>
```

Vue du formulaire

```
@model WebApplication4.Models.UserModel
@{
    ViewBag.Title = "Register";
}

<h2>@ViewBag.Title.</h2>

@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-
horizontal", role = "form" }))
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.FirstName, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.FirstName, new { @class = "form-control" })
            @Html.ValidationMessageFor(m=>m.FirstName)
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.LastName, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.LastName, new { @class = "form-control" })
            @Html.ValidationMessageFor(m => m.LastName)
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.UserName, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.UserName, new { @class = "form-control" })
            @Html.ValidationMessageFor(m => m.UserName)
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Configuration du bundle

```
  public class BundleConfig
    {

        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*"));
```

```
        }
    }
```

Global.asax.cs

```
public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            // Need to include your bundles
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
```

Lire Validation automatique côté client à partir des attributs en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/6044/validation-automatique-cote-client-a-partir-des-attributs

Chapitre 30: Validation du modèle

Examples

Valider le modèle dans ActionResult

```
[HttpPost]
public ActionResult ContactUs(ContactUsModel contactObject)
{
    // This line checks to see if the Model is Valid by verifying each Property in the Model
meets the data validation rules
    if(ModelState.IsValid)
    {
    }
    return View(contactObject);
}
```

La classe de modèle

```
public class ContactUsModel
{
    [Required]
    public string Name { get; set; }
    [Required]
    [EmailAddress] // The value must be a valid email address
    public string Email { get; set; }
    [Required]
    [StringLength(500)] // Maximum length of message is 500 characters
    public string Message { get; set; }
}
```

Supprimer un objet de la validation

Disons que vous avez le modèle suivant:

```
public class foo
{
    [Required]
    public string Email { get; set; }

    [Required]
    public string Password { get; set; }

    [Required]
    public string FullName { get; set; }
}
```

Mais vous souhaitez exclure FullName de la validation de modèle car vous utilisez également le modèle à un endroit où FullName n'est pas renseigné, vous pouvez le faire de la manière suivante:

```
ModelState.Remove("FullName");
```

Messages d'erreur personnalisés

Si vous souhaitez fournir des messages d'erreur personnalisés, procédez comme suit:

```
public class LoginViewModel
{
    [Required(ErrorMessage = "Please specify an Email Address")]
    [EmailAddress(ErrorMessage = "Please specify a valid Email Address")]
    public string Email { get; set; }

    [Required(ErrorMessage = "Type in your password")]
    public string Password { get; set; }
}
```

Lorsque vos messages d'erreur se trouvent dans un ResourceFile (.resx), vous devez spécifier le ResourceType et le ResourceName:

```
public class LoginViewModel
{
    [Required(ErrorMessageResourceType = typeof(ErrorResources), ErrorMessageResourceName =
"LoginViewModel_RequiredEmail")]
    [EmailAddress(ErrorMessageResourceType = typeof(ErrorResources), ErrorMessageResourceName
= "LoginViewModel_ValidEmail")]
    public string Email { get; set; }

    [Required(ErrorMessageResourceType = typeof(ErrorResources), ErrorMessageResourceName =
"LoginViewModel_RequiredPassword")]
    public string Password { get; set; }
}
```

Création de messages d'erreur personnalisés dans le modèle et dans le contrôleur

Disons que vous avez la classe suivante:

```
public class PersonInfo
{
    public int ID { get; set; }

    [Display(Name = "First Name")]
    [Required(ErrorMessage = "Please enter your first name!")]
    public string FirstName{ get; set; }

    [Display(Name = "Last Name")]
    [Required(ErrorMessage = "Please enter your last name!")]
    public string LastName{ get; set; }

    [Display(Name = "Age")]
    [Required(ErrorMessage = "Please enter your Email Address!")]
    [EmailAddress(ErrorMessage = "Invalid Email Address")]
    public string EmailAddress { get; set; }
}
```

Ces messages d'erreur personnalisés apparaîtront si votre `ModelState.IsValid` renvoie false.

Mais vous et moi savons qu'il ne peut y avoir qu'une seule adresse e-mail par personne, sinon vous enverrez des e-mails à des personnes potentiellement dangereuses et / ou à plusieurs personnes. C'est là que le contrôle entre en jeu. Supposons donc que les gens créent des comptes que vous pouvez enregistrer via l'action Créer.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "ID, FirstName, LastName, EmailAddress")]
PersonInfo newPerson)
{
    if(ModelState.IsValid) // this is where the custom error messages on your model will
display if return false
    {
        if(database.People.Any(x => x.EmailAddress == newPerson.EmailAddress))  // checking if
the email address that the new person is entering already exists.. if so show this error
message
        {
            ModelState.AddModelError("EmailAddress", "This email address already exists!
Please enter a new email address!");
            return View(newPerson);
        }

        db.Person.Add(newPerson);
        db.SaveChanges():
        return RedirectToAction("Index");
    }

    return View(newPerson);
}
```

J'espère que cela pourra aider quelqu'un!

Validation du modèle dans JQuery.

Dans les cas où vous devez vous assurer de la validation du modèle à l'aide de Jquery, la fonction .valid () peut être utilisée.

Les champs de classe de modèle

```
[Required]
[Display(Name = "Number of Hospitals")]
public int Hospitals{ get; set; }
[Required]
[Display(Name = "Number of Beds")]
public int Beds { get; set; }
```

Le code de vue

```
@using (Html.BeginForm(new {id = "form1", @class = "form-horizontal" }))
{

<div class="divPanel">
  <div class="row">
```

```
    <div class="col-md-3">
            @Html.LabelFor(m => m.Hospitals)
            @Html.TextBoxFor(m => m.Hospitals, new { @class = "form-control", @type =
"number"})
            @Html.ValidationMessageFor(m => m.Hospitals)

    </div>
    <div class="col-md-3">

            @Html.LabelFor(m => m.Beds)
            @Html.TextBoxFor(m => m.Beds, new { @class = "form-control", @type = "number"})
            @Html.ValidationMessageFor(m => m.Beds)
    </div>
<div class="col-md-3">
        <button type=button  class="btn btn-primary" id="btnCalculateBeds"> Calculate
Score</button>
    </div>
 </div>

  </div>
 }
```

Le script pour le contrôle de validation.

```
$('#btnCalculateBeds').on('click', function (evt) {
evt.preventDefault();

if ($('#form1').valid()) {
//Do Something.
}
}
```

Assurez-vous que les fichiers `jquery.validate` et `jquery.validate.unobtrusive` sont présents dans la solution.

Lire Validation du modèle en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/2683/validation-du-modele

Chapitre 31: ViewData, ViewBag, TempData

Introduction

`ViewData` et `ViewBag` sont utilisés pour transférer des données du contrôleur à la vue.

ViewData n'est rien d'autre qu'un dictionnaire d'objets et il est accessible par chaîne en tant que clé.

ViewBag est très similaire à ViewData. ViewBag est une propriété dynamique. ViewBag est juste un wrapper autour de ViewData.

TempData conserve les données pour le temps de la requête HTTP, ce qui signifie qu'il contient des données entre deux requêtes consécutives. TempData nous aide à transférer des données entre des contrôleurs ou entre des actions. Utilise la session en interne.

Syntaxe

1. ViewData [clé] = valeur;
2. ViewBag.Key = valeur;
3. TempData [clé] = valeur;

Examples

Que sont ViewData, ViewBag et TempData?

ViewData est le mécanisme permettant à un contrôleur de fournir des données à la vue qu'il présente, sans utiliser ViewModel. Plus précisément, ViewData est un dictionnaire disponible à la fois dans les méthodes d'action et les vues MVC. Vous pouvez utiliser ViewData pour transférer certaines données de votre méthode d'action vers la vue renvoyée par la méthode d'action.

Comme il s'agit d'un dictionnaire, vous pouvez utiliser la syntaxe du dictionnaire pour définir et obtenir des données.

```
ViewData[key] = value; // In the action method in the controller
```

Par exemple, si vous souhaitez transmettre un message de chaîne de votre méthode d'action Index à votre vue Index `Index.cshtml` , vous pouvez le faire.

```
public ActionResult Index()
{
   ViewData["Message"] = "Welcome to ASP.NET MVC";
   return View(); // notice the absence of a view model
}
```

Pour y accéder dans votre vue `Index.cshtml` , vous pouvez simplement accéder au dictionnaire ViewData avec la clé utilisée dans la méthode d'action.

```
<h2>@ViewData["Message"]</h2>
```

ViewBag est l'équivalent dynamique du dictionnaire ViewData non typé. Il tire parti du type `dynamic` C # pour l'expérience du sucre syntaxique.

La syntaxe pour définir certaines données sur ViewBag est

```
ViewBag.Key = Value;
```

Donc, si nous voulons passer la chaîne de message dans notre exemple précédent en utilisant ViewBag, ce sera

```
public ActionResult Index()
{
   ViewBag.Message = "Welcome to ASP.NET MVC";
   return View(); // not the absence of a view model
}
```

et dans votre vue d'index,

```
<h2>@ViewBag.Message</h2>
```

Les données ne sont pas partagées entre ViewBag et ViewData. ViewData nécessite une conversion de type typé pour obtenir des données à partir de types de données complexes et vérifier les valeurs NULL pour éviter les erreurs lorsque View Bag ne nécessite pas de transtypage.

TempData peut être utilisé lorsque vous souhaitez conserver les données entre une requête HTTP et la requête HTTP suivante uniquement. La durée de vie des données stockées dans TempDataDictionary se termine après la deuxième requête. Ainsi, TempData est utile dans les scénarios où vous suivez le modèle PRG.

```
[HttpPost]
public ActionResult Create(string name)
{
   // Create a user
   // Let's redirect (P-R-G Pattern)
   TempData["Message"] = "User created successfully";
   return RedirectToAction("Index");
}
public ActionResult Index()
{
  var messageFromPreviousCall = TempData["Message"] as String;
  // do something with this message
  // to do : Return something
}
```

Lorsque nous `return RedirectToAction("SomeActionMethod")` , le serveur enverra une réponse 302

au client (navigateur) avec la valeur d'en-tête d'emplacement définie sur l'URL "SomeActionMethod" et le navigateur effectuera une requête totalement nouvelle. ViewBag / ViewData ne fonctionnera pas dans ce cas pour partager certaines données entre ces 2 appels. Vous devez utiliser TempData dans de tels scénarios.

Cycle de vie TempData

Les données enregistrées dans TempData sont stockées dans la session et seront automatiquement supprimées à la fin de la première requête à laquelle les données sont accédées. S'il n'est jamais lu, il sera conservé jusqu'à ce que la lecture soit terminée ou que la session expire.

L'utilisation typique ressemble à la séquence suivante (où chaque ligne est appelée à partir d'une autre requête):

```
//first request, save value to TempData
TempData["value"] = "someValueForNextRequest";

//second request, read value, which is marked for deletion
object value = TempData["value"];

//third request, value is not there as it was deleted at the end of the second request
TempData["value"] == null
```

Ce comportement peut être un autre contrôleur avec les méthodes `Peek` et `Keep` .

- Avec `Peek` vous pouvez récupérer les données stockées dans TempData sans les marquer pour suppression, afin que les données soient toujours disponibles lors d'une prochaine demande.

  ```
  //first request, save value to TempData
  TempData["value"] = "someValueForNextRequest";

  //second request, PEEK value so it is not deleted at the end of the request
  object value = TempData.Peek("value");

  //third request, read value and mark it for deletion
  object value = TempData["value"];
  ```

- Avec `Keep` vous pouvez spécifier qu'une clé marquée pour suppression doit être conservée. Dans ce cas, la récupération des données et leur enregistrement depuis la suppression nécessite 2 appels de méthode:

  ```
  //first request, save value to TempData
  TempData["value"] = "someValueForNextRequest";

  //second request, get value marking it from deletion
  object value = TempData["value"];
  //later on decide to keep it
  TempData.Keep("value");

  //third request, read value and mark it for deletion
  ```

```
object value = TempData["value"];
```

Dans cet esprit, utilisez `Peek` lorsque vous souhaitez toujours conserver la valeur pour une autre demande et utilisez l' `Keep` lorsque la conservation de la valeur dépend d'une logique supplémentaire.

Lire ViewData, ViewBag, TempData en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/1286/viewdata--viewbag--tempdata

Chapitre 32: Vues partielles

Introduction

Une vue partielle est une vue rendue dans une autre vue. Les vues partielles peuvent être réutilisées et empêcher ainsi la duplication du code. Ils peuvent être rendus par Html.Partial ou Html.RenderPartial

Syntaxe

- @ Html.Partial ("ViewName")

  @ Html.Partial ("ViewName", ViewModel)

  @ {Html.RenderPartial ("ViewName");}

  Si votre vue partielle se trouve dans un autre dossier que le dossier partagé, vous devrez mentionner le chemin complet de la vue comme ci-dessous:

  -@Html.RenderPartial ("~ / Areas / Admin / Views / Shared / partial / _subcat.cshtml")

Examples

Vue partielle avec modèle

Un modèle peut également être ajouté à la vue partielle:

```
@model Solution.Project.Namespace.MyModelClass
<p>@Model.Property</p>
```

Dans la vue, vous pouvez maintenant utiliser:

```
<div>
    @Html.Partial("PartialViewExample", new MyModelClass(){Property="my property value"})
</div>
<div>
    @{ Html.RenderPartial("PartialViewExample", new MyModelClass(){Property="my property
value"}); }
</div>
```

Vue partielle sur une chaîne - pour le contenu du courrier électronique, etc.

Appeler la fonction

```
string InvoiceHtml = myFunction.RenderPartialViewToString("PartialInvoiceCustomer",
ToInvoice); // ToInvoice is a model, you can pass parameters if needed
```

Fonction pour générer du HTML

```
public static string RenderPartialViewToString(string viewName, object model)
{
    using (var sw = new StringWriter())
    {
        BuyOnlineController controller = new BuyOnlineController(); // instance of the
required controller (you can pass this as a argument if needed)

        // Create an MVC Controller Context
        var wrapper = new HttpContextWrapper(System.Web.HttpContext.Current);

        RouteData routeData = new RouteData();

        routeData.Values.Add("controller",
controller.GetType().Name.ToLower().Replace("controller", ""));

        controller.ControllerContext = new ControllerContext(wrapper, routeData, controller);

        controller.ViewData.Model = model;

        var viewResult = ViewEngines.Engines.FindPartialView(controller.ControllerContext,
viewName);

        var viewContext = new ViewContext(controller.ControllerContext, viewResult.View,
controller.ViewData, controller.TempData, sw);
        viewResult.View.Render(viewContext, sw);

        return sw.ToString();
    }
}
```

Vue partielle - PartialInvoiceCustomer

```
@model eDurar.Models.BuyOnlineCartMaster
 <h2>hello customer - @Model.CartID </h2>
```

Html.Partial Vs Html.RenderPartial

Html.Partial retourne une chaîne par contre **Html.RenderPartial** renvoie un void.

Html.RenderPartial

Cette méthode renvoie void et le résultat est directement écrit dans le flux de réponse HTTP. Cela signifie qu'il utilise le même objet TextWriter que celui utilisé dans la page Web / le modèle en cours. Pour cette raison, cette méthode est plus rapide que la méthode partielle. Cette méthode est utile lorsque l'affichage des données dans la vue partielle se trouve déjà dans le modèle de vue correspondant.

Exemple: Dans un blog pour afficher les commentaires d'un article, nous aimerions utiliser la méthode RenderPartial car une information d'article avec des commentaires est déjà remplie dans le modèle de vue.

```
@{Html.RenderPartial("_Comments");}
```

Html.Partial

Cette méthode retourne une chaîne codée en HTML. Cela peut être stocké dans une variable. Comme la méthode RenderPartial, la méthode partielle est également utile lorsque les données affichées dans la vue partielle sont déjà dans le modèle de vue correspondant.

Exemple: Dans un blog pour afficher les commentaires d'un article, vous pouvez utiliser la méthode Partial car une information d'article avec des commentaires est déjà remplie dans le modèle de vue.

```
@Html.Partial("_Comments")
```

Lire Vues partielles en ligne: https://riptutorial.com/fr/asp-net-mvc/topic/2171/vues-partielles

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec asp.net-mvc	Aaron Hudon, Adil Mammadov, Aditya Korti, Ameya Deshpande, Ashley Medway, Community, Hywel Rees, Rifaj, Shog9, Shyju, Supraj v, Syed Farjad Zia Zaidi, SztupY
2	ActionResult	hasan, SlaterCodes, Tetsuya Yamamoto
3	Aide HTML	Ashiquzzaman, Laurel, Lokesh_Ram, Pavel Pája Halbich, Peter Mortensen, QuantumHive, Tassadaque, Testing123, Tetsuya Yamamoto, The_Outsider, TheFallenOne
4	Annotations de données	abiNerd, dotnetom, dove, Edathadan Chief aka Arun, Ehsan Sajjad, Felipe Oriani, gunr2171, Karthikeyan, LaCartouche, mmushtaq, Ollie P, Rion Williams, SailajaPalakodeti, Stephen Muecke, Tetsuya Yamamoto, The_Outsider, tmg, Tsahi Asher
5	Appel jQuery Ajax avec Asp MVC	Ashiquzzaman, hasan
6	Asp.net mvc envoyer un mail	Ashiquzzaman, hasan, sGermosen
7	Cryptage Web.config	glaubergft, Jack Spektor
8	Dockerization de l'application ASP.NET	SUMIT LAHIRI
9	Domaines	Himaan Singh, Tetsuya Yamamoto
10	Enregistrement des erreurs	Andrus, Leandro Soares, Saineshwar
11	Extensions Ajax MVC	rll
12	Filtres d'action	Andrei Rînea, Dawood Awan, juunas, Laurel, Lokesh_Ram, Tolga Evcimen
13	Html.AntiForgeryToken	Aaron Hudon, Andrei Rînea, Art, felickz, Hanno, Jakotheshadows, Joshua Leigh, Martin Costello, Minh Nguyen , Rion Williams, SailajaPalakodeti, SlaterCodes, viggity
14	Html.RouteLink	Jason Evans
15	Injection de	Adil Mammadov, Andrei Dragotoniu, Cà phê đen, Dave,

	dépendance	PedroSouki
16	Le rasoir	Aditya Korti, aeliusd, Anik Saha, Arendax, Ashley Medway, Big Fan, Brandon Wood, Braydie, Denis Elkhov, dove, James Haug, Julian, Kuldeep Prajapati, Lee Chengkai, lloyd, RamenChef, SadiRubaiyet, Sain Pradeep, Sandro, Thennarasan, Tim Coker, TKharaishvili, Tot Zam, usr
17	Le routage	Alex Art., Andrei Rînea, chsword, Ciaran Bruen, Jarrod Dixon, Jason Evans, Karthikeyan, kkakkurt, Laurel, Lokesh_Ram, mstaessen, Pavel Voronin, SailajaPalakodeti, Sandro, Shyju, SlaterCodes, Stephen Muecke, Tetsuya Yamamoto, tmg, Tot Zam, user270576
18	Modèles d'affichage et d'édition	Adnan Niloy, SailajaPalakodeti
19	MVC vs Web Forms	DiegoS, Houssam Hamdan, The_Outsider
20	Opération CRUD	EvenPrime, Krzyserious, PedroSouki, Tetsuya Yamamoto
21	Règles de réécriture IIS	SlaterCodes
22	Regroupement et Minification	Ashley Medway, Beofett, hasan, Laurel, Lokesh_Ram, Paul DS, rageit, Rion Williams, Robban, Tetsuya Yamamoto, tmg
23	Reliure modèle	Adil Mammadov, Andrei Rînea, dove, Ehsan Sajjad, James Haug, Md Dinar, PedroSouki, rdans, Tolga Evcimen
24	T4MVC	James Haug
25	Traitement des erreurs HTTP	scgough
26	Utilisation de plusieurs modèles en une seule vue	hasan, Travis Tubbs
27	Validation automatique côté client à partir des attributs	Slick86
28	Validation du modèle	Aaron Hudon, Ankit Kumar Singh, GTown-Coder, hasan, Marimba, Nikunj Patel, Pavel Voronin, SlaterCodes, Stephen Muecke, The_Outsider, TheFallenOne, Vincentw
29	ViewData, ViewBag, TempData	bzlm, Daniel J.G., IanB, Rion Williams, SailajaPalakodeti, Shyju, TheFallenOne, tmg

30	Vues partielles	Adnan Niloy, Ashiquzzaman, Edathadan Chief aka Arun, glacasa, Jason Evans, Laurel, Lokesh_Ram, Marimba, SailajaPalakodeti, The_Outsider