

Capítulo 3

Aplicando Técnicas en una Aplicación ASP.NET MVC 6

Objetivo

Al finalizar el capítulo, el alumno:

- Diseña el URL Routing de una aplicación MVC.
- Aplica action filters y crea unos personalizados.
- Crea Html Helpers personalizados.
- Comprende el model binding y lo personaliza.
- Emplea y hace uso de View Components en MVC.
- Utiliza ViewModels.
- Realizar inyección de dependencia en vista.


Temas

1. Revisión de Razor.
2. Configuración de URL Routing de MVC
3. Uso y creación de Action Filters.
4. Uso y creación de HTML Helpers.
5. Revisión de Model Binders y Value Providers en MVC.
6. Creación y uso de ViewModels.
7. Introducción a los View components.

1. Revisión de Razor

Revisión de Razor

- Razor es sintaxis de marcado de código de lado del servidor que se agrega directamente a html. Fácil de aprender y usar. Es similar a C#.
- El uso de la @ funciona de dos maneras básicas:
 - **@expresión:** Renderiza la expresión en el navegador. Así @item.Nombre muestra el valor de item.Nombre. Es decir @expresión equivale a <%: expresión %>
 - **@{ código }:** Permite ejecutar un código que no genera salida HTML. Es decir @{código} equivale a <% Código %>

3 - 4Copyright © Todos los Derechos Reservados - Cibertec Perú SAC.

Razor es sintaxis de marcado de código de lado del servidor que se agrega directamente a html. Fácil de aprender y usar. Es similar a C#.

Lo que más choca de Razor es que, a diferencia del motor ASPX donde tenemos el tag que inicia el código de servidor y el que lo termina (<%%>), solo hay tag para iniciar código de servidor. El motor Razor es lo suficientemente inteligente para saber cuándo termina el código de servidor, sin necesidad de que lo explicitemos.

En Razor el símbolo de la arroba (@) marca el inicio de código de servidor. Y como comentaba antes, no hay símbolo para indicar que se termina el código de servidor: el motor Razor deduce cuando termina en base al contexto.

Por ejemplo, para mostrar una variable de servidor (item.Nombre p.ej.) simplemente la precedemos de una @item.Nombre.

El uso de la @ funciona de dos maneras básicas:

- **@expresión:** Renderiza la expresión en el navegador. Así @item.Nombre muestra el valor de item.Nombre. Es decir @expresión equivale a <%: expresión %>
- **@{ código }:** Permite ejecutar un código que no genera salida HTML. Es decir @{código} equivale a <% Código %>

1.1 Bloque de código en Razor se tiene que indicar de la siguiente manera:

```
<!-- Single statement block -->
@{ var myMessage = "Hello World"; }
```

1.2 Bloque de código en Razor multilínea:

```
@*
  Este es un bloque de comentario
*@

@{
  var greeting = "Welcome to our site!";
  var weekDay = DateTime.Now.DayOfWeek;
  var greetingMessage = greeting + " Today is: " + weekDay;
}
```

1.3 Definición de Variables:

Se debe usar la palabra clave **var** o el tipo de dato (int, float, decimal, bool, string, DateTime).

```
// Using the var keyword:
var greeting = "Welcome to W3Schools";
var counter = 103;
var today = DateTime.Today;

// Using data types:
string greeting = "Welcome to W3Schools";
int counter = 103;
DateTime today = DateTime.Today;
```

1.4 Operadores:

Tipos de operadores	Descripción
Operadores de asignación	=
Operadores matemáticos	+, -, * y /
Operadores de comparación	==, !=, <, >, <=, >=
Operadores lógicos	&&,
Operadores de concatenación	+

1.5 Estructuras repetitivas:

```
<ul>
  @for (int i = 0; i < 10; i++)
  {
    <li>@i</li>
  }
</ul>
```

```
<ul>
  @foreach (var x in Request.ServerVariables)
  {
    <li>@x</li>
  }
</ul>
```

```
@{
  var i = 0;
  while (i < 5)
  {
    i += 1;
    <p>Line @i</p>
  }
}
```

1.6 Estructuras condicionales:

```
@{var price = 50;}
<html>
<body>
  @if (price > 30)
  {
    <p>The price is too high.</p>
  }
</body>
</html>
```

```
@{var price = 20;}
<html>
<body>
    @if (price > 30)
    {
        <p>The price is too high.</p>
    }
    else
    {
        <p>The price is OK.</p>
    }
</body>
</html>
```

```
@{var price = 25;}
<html>
<body>
    @if (price >= 30)
    {
        <p>The price is high.</p>
    }
    else if (price > 20 && price < 30)
    {
        <p>The price is OK.</p>
    }
    else
    {
        <p>The price is low.</p>
    }
</body>
</html>
```

Consideraciones a la sintaxis

Expresiones complejas

Como hemos visto el motor Razor interpreta cuándo empieza y cuándo termina el código de servidor. Pero no siempre lo consigue adecuadamente. Por ejemplo, el siguiente código Razor:

```
@{ int a = 10; int b = 3; }

El valor de 10 - 3 es: @a-b
```

Genera el siguiente HTML: El valor de 10 - 3 es: 10-b

Es decir, Razor ha interpretado que el código de servidor terminaba al encontrar el símbolo de resta. En este caso, esa presunción es totalmente errónea, pero por suerte podemos usar los paréntesis para que haya solo una expresión detrás de la arroba:

```
El valor de 10 - 3 es: @(a-b)
```

Con ese código la vista mostrará el valor correcto (7). Recordemos la clave: el motor Razor espera una y solo una expresión detrás de la @. Por eso debemos usar los paréntesis.

"Romper" el código de servidor

A veces la problemática es justo la contraria de la que hemos visto con las expresiones complejas: a veces hay código que Razor interpreta que es de servidor pero realmente parte de ese código es HTML que debe enviarse al cliente. Veamos un ejemplo:

```
@for (int i = 0; i < 10; i++)  
{  
    El valor de i es: @i <br />  
}
```

A priori podríamos esperar que este código generara 10 líneas de código HTML. Pero el resultado es un error de compilación. La razón es que Razor interpreta que la línea "El valor de i es: @i" es código de servidor. Para "romper" este código de servidor y que Razor "sepa" que realmente esto es código que debe enviarse tal cual al cliente, tenemos dos opciones:

1. Intercalar una etiqueta HTML. Al intercalar una etiqueta HTML Razor "se da cuenta" que allí empieza un código de cliente:

```
@for (int i = 0; i < 10; i++)  
{  
    <span>El valor de i es:</span> @i <br />  
}
```

2. Usar la construcción @: que indica explícitamente a Razor que lo que sigue es código de cliente:

```
@for (int i = 0; i < 10; i++)  
{  
    @:El valor de i es: @i <br />  
}
```

La diferencia es que en el primer caso las etiquetas se envían al navegador, mientras que en el segundo caso no.

Consideraciones en correos electrónicos

La gente que ha desarrollado el motor Razor ha tenido en cuenta una excepción para los correos electrónicos. Es decir, Razor es de nuevo, lo suficientemente inteligente para saber que una expresión es un correo electrónico. Así pues, el siguiente código no da error y envía el HTML esperado al cliente:

Envíame un mail: a usuario@servidor.com

En este caso, Razor interpreta que se trata de un correo electrónico, por lo que no trata la @ como inicio de código de servidor. Este comportamiento a veces tampoco se desea. Por ejemplo, imaginemos lo siguiente:

```
<div id="div_@Model.Index">Div usado</div>
```

Estoy asumiendo que el ViewModel que recibe la vista tiene una propiedad llamada Index. Supongamos que dicha propiedad (Model.Index) vale 10. La verdad es que uno pensaría que eso generaría el código HTML:

```
<div id="div_10">Div usado</div>
```

Pero el resultado es bien distinto. El código HTML generado es:

```
<div id="div_@Model.Index">Div usado</div>
```

Es decir, Razor no ha interpretado la @ como inicio de código de servidor, y eso es porque ha aplicado la excepción de correo electrónico. La solución pasa por usar los paréntesis:

```
<div id="div_@(Model.Index)">Div usado</div>
```

Ahora Razor sabe que Model.Index es código de servidor y lo evaluará, generando el HTML que estábamos esperando.

A veces Razor falla incluso más espectacularmente. Dado el siguiente código:

```
@for (int i = 0; i <= 1; i++)  
{  
    <div id="div_@i">Div @i</div>  
}
```

En base a lo que hemos visto podríamos esperar que generase el siguiente HTML:

```
<div id="div_@i">Div 0</div>  
<div id="div_@i">Div 1</div>
```

Pues ¡no! Este código hace que el motor de Razor dé un error (*The for block is missing a closing "}" character. Make sure you have a matching "}" character for all the "{" characters within this block, and that none of the "}" characters are being interpreted as markup*).

Por supuesto, la solución para generar el HTML que queremos pasa por usar el paréntesis igual que antes:

```
@for (int i = 0; i <= 1; i++)  
{  
    <div id="div_@(i)">Div @i</div>  
}
```

Genera el HTML esperado:

```
<div id="div_0">Div 0</div>
<div id="div_1">Div 1</div>
```

Escapar la arroba

A veces es necesario indicarle a Razor que una arroba es eso... una simple arroba y que no haga nada específico. Que no asuma nada, que no piense nada, que simplemente envíe una @ al cliente. ¿Un ejemplo? El siguiente código:

```
<style>
@ @media screen
{
    body { font-size: 13px;}
}
</style>
```

Si no usáramos la doble arroba (@@), Razor nos generaría un error, ya que @media lo interpreta como "enviar el contenido de la variable media al cliente". En este caso al usar @@ Razor simplemente sabe que debe enviar una @ al cliente y lo que el navegador recibe es:

```
<style>
@media screen
{
    body { font-size: 13px;}
}
</style>
```

Hemos visto la sintaxis básica del motor de vistas Razor y las principales consideraciones que debemos tener siempre presentes.

2. Configuración de URL Routing de MVC

Configuración de URL Routing de MVC

Cuando se crea una aplicación ASP.NET MVC, se define una tabla de enrutamiento que se encarga de decidir qué controlador gestiona cada petición web, basándose en la URL de dicha petición

En cada petición URL no se asigna un archivo físico del disco, tal como una página .aspx, sino que se asigna la acción de un controlador

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

3 - 11

Copyright © Todos los Derechos Reservados - Cibertec Perú SAC.



Cuando se crea una aplicación ASP.NET MVC, se define una tabla de enrutamiento que se encarga de decidir qué controlador gestiona cada petición web basándose en el URL de dicha petición.

En cada petición URL no se asigna un archivo físico del disco, tal como una página .aspx, sino que se asigna la acción de un controlador. Las rutas son lógicas; es decir, siguen la estructura establecida en la tabla de enrutamiento, definida en la clase **Startup**, método **Configure** que por default contiene lo siguiente:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

La ruta Default responde a los requests de la siguiente forma:

- http://webserver - Controller: Home, Action: Index
- http://webserver/Productos - Controller: Productos, Action: Index
- http://webserver/Clientes/Details/5 - Controller: Clientes, Action: Details, Id: 5

Típicamente es suficiente con la ruta Default para que una aplicación pueda funcionar correctamente, pero a veces hay necesidades particulares que requiere agregar otras.

1. Implementación de URL Routing

Tener en cuenta que la coincidencia de un URL con alguna de las rutas configuradas se realiza en orden de arriba hacia abajo, por lo que se deben colocar de la más específica a la más general (el orden importa).

Se puede agregar más rutas en esta configuración de varias formas, tal como se muestra a continuación:

- Ruta Estática: No se usa { }, por lo que el URL debe coincidir exactamente.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}")
    .MapRoute(
        name: "Catalogo",
        template: "Catalogo",
        defaults: new { controller="Product", action="Index"});
});
```

<http://webserver/Catalogo> - Controller: Product, Action: Index

- Ruta Dinámica: Observar en este caso como el action está después del id. También, notar que hay una propiedad "constraints", la cual recibe un regular expression para verificar alguna parte del URL, en este ejemplo lo que hace es verificar que el id sea un número; en caso no se cumpla este constraint o restricción, la ruta no reconocerá al URL como válido.

```
.MapRoute(
    name: "CatalogoDinamica",
    template: "Catalogo/{id}/{action}",
    defaults: new { controller = "Product", action = "details" },
    constraints: new { id=@"\d+" });
```

<http://webserver/Catalogo/5/Details>

- Controller: Product, Action: Details, Id: 5

- Ruta SEO Friendly: Es una buena práctica colocar algo descriptivo a parte del id, así no se use en el procesamiento del action. Esto ayuda a que el url sea SEO Friendly, es decir, que se pueda indexar y luego, encontrar en una búsqueda.

```
.MapRoute(
    name: "CatalogoSEO",
    template: "CatalogoSEO/{nombre}/{id}",
    defaults: new { controller = "Product", action = "details"},
    constraints: new { id = @"\d+" });
```

<http://webserver/CatalogoSEO/5/libro-mvc-dot-net> -

Controller: Product, Action: Details, Id: 5

En este ejemplo, el nombre no se toma en cuenta para el routing, solo es algo descriptivo. Entonces, cuando se implementa la página de listado de productos y se programa la parte de ver detalles, el URL que se debe crear no solo debe incluir el id, sino también el nombre.

2. Implementación de Routing en los controladores

Se puede sobrescribir el patrón de rutas en los controladores y acciones, para ellos deber hacer uso de la clase **Route**.

Colocando la ruta base a nivel del controlador

```
[Route("CatalogoProductos")]
[Route("[Controller]")]
Public class ProductController : Controller
{
    [Route("[Action]/{id}")]
    [Route("Editar/{id}")]
    //GET: Product/Create
    public IActionResult Edit(int id)
    {
        var bus = new ProductBus();
        var model = bus.GetProducto(new Product() { ProductID = id });
        return View(model);
    }
}
```

En este caso se podría llamar al catálogo de producto haciendo uso de las siguientes direcciones:

<http://localhost:49349/CatalogoProductos/Edit/3>
<http://localhost:49349/CatalogoProductos/Editar/3>
<http://localhost:49349/Product/Edit/3>
<http://localhost:49349/Product/Editar/3>

Colocando la ruta en la acción: no se debe indicar el atributo **Route** en el controlador. Si las demás acciones no tienen el atributo **Route**, entonces toma el definido en el MapRoute de la clase Startup

```
[Route("Productos/Editar/{id}")]
//GET: Product/Create
public IActionResult Edit(int id)
{
    var bus = new ProductBus();
    var model = bus.GetProducto(new Product() { ProductID = id });
    return View(model);
}

}
```

3. Uso y creación de Action Filters

Uso y creación de Action Filters

Los Action Filters agregan lógica antes y después de la ejecución de los Action Methods. Se pueden definir para cada action method o a nivel de todo el controller, agregando el nombre del action filter entre [].

- **Action Filter por defecto:**

ResponseCache


```
[ResponseCache(Duration = 20)]
3 references
public IActionResult Index()
{
    return View();
}
```

Authorize

```
[Authorize(Roles="Administrator")]
public ActionResult Create()
{
}
```

3 - 18

Copyright © Todos los Derechos Reservados - Cibertec Perú S.A.C.



Los Action Filters agregan lógica antes y después de la ejecución de los Action Methods. Se pueden definir para cada action method o a nivel de todo el controller, agregando el nombre del action filter entre [].

Existen action filters que por default ofrece el Framework MVC, por ejemplo:

- ResponseCache

Indica que se debe guardar por una cierta cantidad de tiempo el resultado del action. Se verá en el capítulo de ASP.NET Core.

```
[ResponseCache(Duration = 20)]
3 references
public IActionResult Index()
{
    return View();
}
```

- Authorize

Indica restricciones de acceso para ciertos roles o usuarios. Se verá en el capítulo de Seguridad.

```
[Authorize(Roles="Administrator")]  
public ActionResult Create()  
{
```

También, se pueden crear action filters personalizados, para lo cual se debe heredar de la clase base **ActionFilterAttribute** (namespace: Microsoft.AspNet.Mvc.Filter), que implementa los siguientes métodos que pueden sobre escribirse:

- OnActionExecuting: es llamado antes de la ejecución de un action method.
- OnActionExecuted: es llamado después de la ejecución de un action method.
- OnResultExecuting: es llamado antes que se ejecute el resultado de un action method.
- OnResultExecuted: es llamado después que se ejecute el resultado de un action method.

```
public class LoggingFilterAttribute : ActionFilterAttribute  
{  
    protected static readonly log4net.ILog log =  
        log4net.LogManager.GetLogger(MethodBase.GetCurrentMethod().DeclaringType);  
  
    //Antes de que el action method se ejecute.  
    0 references  
    public override void OnActionExecuting(ActionExecutingContext filterContext)  
    {  
        var message = string.Format("Inicia ejecucion de: Controller {0}, Action {1}, Hora Inicio {2}",  
            filterContext.Controller.ToString(),  
            filterContext.ActionDescriptor.Name,  
            DateTime.Now.ToLongTimeString());  
        log.Debug(message);  
    }  
  
    //Despues de que el action method se ejecutó.  
    0 references  
    public override void OnActionExecuted(ActionExecutedContext filterContext)  
    {  
        var message = string.Format("Termina ejecucion de: Controller {0}, Action {1}, Hora Fin {2}",  
            filterContext.Controller.ToString(),  
            filterContext.ActionDescriptor.Name,  
            DateTime.Now.ToLongTimeString());  
        log.Debug(message);  
    }  
}
```

Para aplicarlo, se agrega [HandleLoggingDAT] antes de la declaración de un action method o de todo el controller.

ExceptionHandlerAttribute: Se utiliza como clase base para crear filtros personalizados que permitan atrapar errores

```
1 reference
public class HandleCustomError: ExceptionFilterAttribute
{
    protected static readonly ILog log =
        LogManager.GetLogger(MethodBase.GetCurrentMethod().DeclaringType);

    1 reference
    public override void OnException(ExceptionContext filterContext)
    {
        var message = string.Format("Controller: {0}, Action: {1}, Exception: {2}, Hora: {3}",
            filterContext.RouteData.Values["controller"].ToString(),
            filterContext.RouteData.Values["action"].ToString(),
            filterContext.Exception.ToString(),
            DateTime.Now.ToString());

        log.Error(message);

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

4. Uso y creación de TagHelpers

Uso y creación de TagHelpers

- Los TagHelpers son la nueva característica de ASP.NET MVC. Los TagHelpers permiten el pre procesamiento de atributos HTML en el lado del servidor.

```
<a asp-controller="Home" asp-action="About">About</a>
```

Va a permitir renderizar algo como lo siguiente:

```
<a href="/Home/About">About</a>
```

Otra manera de crear un enlace usando los Html Helper es usar lo siguiente:

```
<a href="@Url.Action("About",  
"Home")" class="someClass">About this site</a>
```

3 - 22

Copyright © Todos los Derechos Reservados - Cibertec Perú SAC.



Los TagHelpers son la nueva característica de ASP.NET MVC. Los TagHelpers permiten el pre procesamiento de atributos HTML en el lado del servidor.

Ejemplo: Definir un enlace con TagHelpers de la siguiente manera:

```
<a asp-controller="Home" asp-action="About">About</a>
```

Va a permitir renderizar algo como lo siguiente:

```
<a href="/Home/About">About</a>
```

Otra manera de crear un enlace usando los Html Helper es usar lo siguiente:

```
<a href="@Url.Action("About", "Home")" class="someClass">About this site</a>
```

Agregando Tag Helpers a su vista

Pueda habilitar Tag Helpers desde cualquier assembly/namespace agregando la directiva @addTagHelper en las vistas (archivos .cshtml)

```
@addTagHelper "*", Microsoft.AspNet.Mvc.TagHelpers"
```


En la sentencia se indica que incluya todos los helpers de assembly Microsoft.AspNet.Mvc.TagHelpers.

Para hacer que los Tag Helpers estén disponibles en todo el proyecto se debe agregar la directiva **@addTagHelper** en el archivo Views/_ViewImports.cshtml

Lista de TagHelpers

- **Anchor Tag Helper**

Este tag es aplicado al elemento html `<a>`. Soporta los siguientes atributos:

- asp-action
- asp-controller
- asp-fragment
- asp-host
- asp-protocol
- asp-route

Ejemplo:

```
<a asp-controller="Home" asp-action="About">About</a>
```

- **Input TagHelper**

Es aplicado a los elementos html de tipo `<input>`. Es similar al `TextBoxFor` al HTML Helper soportado en versiones anteriores y que permite enlazar a un campo del modelo. Soporta los siguientes atributos:

- asp-for: atributo referido a la propiedad en el modelo.
- asp-format: aplica un format después que el valor fue recibido.

Ejemplo:

```
<input asp-for="Birthday" asp-format="{0:yyyy-MM-dd}" />
```

- **Label TagHelper**

Tiene el mismo comportamiento de los métodos `HtmlExtension.LabelFor` pero ahora con este tag solo se usa el atributo 'asp-for' que se relaciona con el campo del modelo.

```
<label asp-for="Birthday" />
```

- **Select TagHelper**

Aplica al elemento html de tipo `<select>`, soporta dos atributos 'asp-for' y 'asp-items'.

```
<select asp-for="Country" asp-items="ViewBag.Countries">
```

Se puede agregar una opción por defecto.

```
<select asp-for="Country" asp-items="ViewBag.Countries">  
  <option selected="selected" value="">Choose Country</option>  
</select>
```

- **TextArea TagHelper**

Es aplicable al elemento 'textarea' de Html.

```
<textarea asp-for="Information"></textarea>
```

- **ValidationMessage TagHelper**

Permite mostrar mensajes de validación.

```
<input asp-for="Birthday" asp-format="{0:yyyy-MM-dd}" />
<span asp-validation-for="Birthday" />
```

- **FormTagHelper**

Usado para generar elemento html de tipo <form>. Soporta los siguientes atributos:

- asp-action: referida a nombre de la acción del controlador.
- asp-controller: referida al nombre controlador.
- asp-anti-forgery: permite evitar el Cross-Site Request Forgery (CSRF)

```
<form asp-action="FormSave" asp-controller="Home" asp-anti
forgery="true">
```

- **Cache TagHelper**

Permite que ciertas partes del html sea colocado en caché en memoria basado en ciertos parámetros que se indiquen como el tiempo que va a durar la caché.

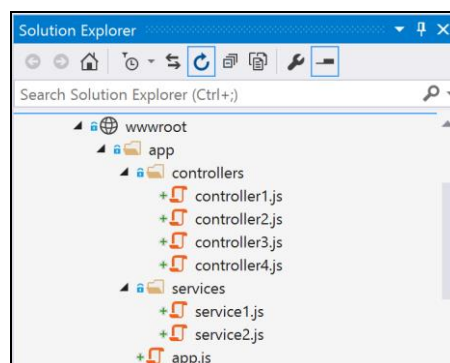
```
<cache expires-after="@TimeSpan.FromMinutes(10)">
    @Html.Partial("_WhatsNew")
    *last updated @DateTime.Now.ToLongTimeString()
</cache>
```

- **Link/Script TagHelpers**

Los Tag Helpers Link y Script están destinados para agregar de una manera más sencilla los tag de html de tipo <script> y <link>.

```
<link asp-href-include="~/app/**/*.css"></script>
<script asp-src-include="~/app/**/*.js"></script>
```

Ejemplo:



```
<script asp-src-include="~/app/**/*.js"></script>
```

En el ejemplo va a generar la etiqueta html de tipo `<script>` con todas la coincidencias en la carpeta app.

```
<script src="/app/app.js"></script>
<script src="/app/controllers/controller1.js"></script>
<script src="/app/controllers/controller2.js"></script>
<script src="/app/controllers/controller3.js"></script>
<script src="/app/controllers/controller4.js"></script>
<script src="/app/services/service1.js"></script>
<script src="/app/services/service2.js"></script>
```

- **asp-src-exclude:** también se puede usar el atributo para excluir archivos de la generación de html.
- **asp-href-exclude:** también se puede usar el atributo para excluir archivos de la generación de html.

• Environment TagHelper

El tag Environment es usado para generar html dependiendo del entorno (desarrollo, QA, producción, etc.) donde se esté ejecutando la aplicación web.

```
<environment names="Development">
  <link rel="stylesheet" href="~/css/site1.css" />
  <link rel="stylesheet" href="~/css/site2.css" />
</environment>
<environment names="Staging,Production">
  <link rel="stylesheet" href="~/css/site.min.css" asp-file-version="true"/>
</environment>In the Development environment
```

Creación de Tag Helpers Personalizados (Custom Tag Helper)

Para construir un Tag Helper personalizado se debe heredar de la clase abstracta **TagHelper** que a su vez implementa la interfaz **ITagHelper**.

Por ejemplo, si se desea implementar el Tag Helper **EmailTagHelper**, se deben seguir los siguientes pasos:

1. Crear una clase y nombrarla haciendo uso de CamelCase, luego heredar de la clase **TagHelper**:

```
public class EmailTagHelper : TagHelper
```
2. Agregar atributos a este Tag Helper mediante propiedades de la clase. Para nombrar las propiedades se debe usar CamelCase.

```
public string MailTo { get; set; }
```
3. Agregar la lógica para mostrar el html que va a generar el Tag Helper. Para ello, se debe hacer override al método **Process** de la clase base.
4. En el archivo `_ViewImports.cshtml`, agregar la referencia al tag creado, mediante `@addTagHelper`.
5. Y por último, para usar el Tag Helper en la vista se debe obviar la palabra "TagHelper" y para determinar los nombres del tag/atributos, a la clase o propiedades, añadir un guión entre cada palabra que la conforme (allí la razón de uso de CamelCase para nombres, clases y propiedades).

Ejemplo:

<email mail-to="visualstudio2017@cibertec.edu.pe">

```
0 references
public class EmailBasicTagHelper : TagHelper
{
    2 references
    public string MailTo { get; set; }

    2 references
    public string Name { get; set; }

    3 references
    public override Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a";
        output.Attributes["href"] = "mailto:" + MailTo;

        if(!String.IsNullOrEmpty(Name))
            output.Content.SetContent(Name);
        else
            output.Content.SetContent(MailTo);

        return base.ProcessAsync(context, output);
    }
}
```

Personalizar nombres de TagHelper y Atributos

- Para personalizar el nombre de un TagHelper se debe hacer uso de **HtmlTargetElement**, es un atributo que se coloca en la clase, en cuyo primer parámetro se le indica el nombre del TagHelper y en el segundo se le indica qué atributos del TagHelper son requeridos. [HtmlTargetElement("email", Attributes = "p-mail-to")], en este caso el nombre del TagHelper y cómo se llamaría en la vista sería "email" y se indica que "p-mail-to" va a ser el atributo requerido.

- Para personalizar los atributos de un TagHelper se debe hacer uso de **HtmlAttributeName**, es un atributo que se coloca en cada propiedad de la clase y como primer parámetro recibe el nombre del atributo del TagHelper. [HtmlAttributeName("p-mail-to")]

```
[HtmlTargetElement("email", Attributes = EmailToAttributeName)]  
3 references  
public class EmailTagHelper : TagHelper  
{  
    private const string EmailToAttributeName = "p-mail-to";  
    private const string NombreAttributeName = "p-nombre";  
  
    [HtmlAttributeName(EmailToAttributeName)]  
    2 references  
    public string MailTo { get; set; }  
  
    [HtmlAttributeName(NombreAttributeName)]  
    2 references  
    public string Name { get; set; }  
  
    1 reference  
    public override Task ProcessAsync(TagHelperContext context, TagHelperOutput output)  
    {  
        output.TagName = "a";  
        output.Attributes["href"] = "mailto:" + MailTo;  
  
        if(!String.IsNullOrEmpty(Name))  
            output.Content.SetContent(Name);  
        else  
            output.Content.SetContent(MailTo);  
  
        return base.ProcessAsync(context, output);  
    }  
}
```

5. Revisión de Model Binders y Value Providers en MVC

Revisión de Model Binders y Value Providers

- **Model Binders**

Supongamos que no sabemos nada acerca de las características que tiene ASP.NET MVC respecto al Model Binding, que como vamos a ver hace el trabajo más fácil.

```
public ActionResult Edit()
{
    var album = new Album();
    album.Titulo = Request.Form["Titulo"];
    album.Precio = Decimal.Parse(Request.Form["Precio"]);
    //Continúa con las demás propiedades...
    return View(album);
}
```

Con el Binding de MVC es mucho más fácil.

```
public ActionResult Edit(Album album)
{
    //Lógica del action method...
    return View(album);
}
```

3 - 29

Copyright © Todos los Derechos Reservados - Cibertec Perú SAC.



Model Binders

Supongamos que no sabemos nada acerca de las características que tiene ASP.NET MVC respecto al Model Binding, que como vamos a ver hace el trabajo más fácil.

Considerar que tenemos una vista de edición de álbum, sabemos que los campos que contiene la vista van a enviarse en un post al servidor, y si se desea recuperar los valores para actualizar un álbum, puede optar por tomar los valores directamente desde el request (Request.Form["Campo"]):

```
public ActionResult Edit()
{
    var album = new Album();
    album.Titulo = Request.Form["Titulo"];
    album.Precio = Decimal.Parse(Request.Form["Precio"]);
    //Continúa con las demás propiedades...
    return View(album);
}
```

Como se puede ver, un código como este se vuelve muy tedioso, y eso que el código solo muestra cómo asignar dos propiedades. Se tiene que tomar de cada valor de la propiedad de la colección Form (que contiene todos los valores del formulario publicado, por su nombre) y pasar esos valores en propiedades del álbum. Cualquier propiedad que no es de tipo cadena también requerirá una conversión de tipos.

Es por esto que se debe aprovechar el model binding. Para esto se requiere que, en la vista, el nombre de los elementos HTML tenga su tributo nombre igual al nombre de la propiedad de la clase del modelo, en este caso del álbum (Titulo y Precio). Se podría modificar la vista para utilizar diferentes nombres, pero hacerlo solo haría que el código del action method sea más difícil de escribir.

Con esto, en lugar de tener que explorar los valores del formulario de la solicitud, el action method Editar simplemente toma un álbum como parámetro:

```
public ActionResult Edit(Album album)
{
    //Lógica del action method...
    return View(album);
}
```

Se puede usar el atributo Bind, para indicar qué campos se desea enlazar al momento de invocar a la acción.

```
public IActionResult CreateProd([Bind("ProductID, Name")] Product product)
```

El Model binding va a trabajar cuando se tiene un parámetro en el action method. Pero también puede invocar explícitamente al model binding utilizando los métodos TryUpdateModelAsync o TryUpdateModel:

- TryUpdateModelAsync. También invoca de manera asíncrona al model binding, pero no produce una excepción. TryUpdateModel retona un booleano, true si el model binding fue exitoso y false en caso contrario.
- TryUpdateModel. También invoca al model binding, pero no produce una excepción. TryUpdateModel retona un booleano, true si el model binding fue exitoso y false en caso contrario.

Tener en cuenta que los usuarios pueden proporcionar valores que no pueden ser enlazados a la correspondiente propiedad del modelo, tal como fechas inválidas o valores de texto para campos numéricos. Cuando invocamos al model binding explícitamente, tenemos la responsabilidad de hacer frente a tales errores. El model binding expresa errores de enlace lanzando la excepción InvalidOperationException. Como un enfoque alternativo, podemos utilizar el método que devuelve true si el proceso de enlace tiene éxito y false si hay errores. La única razón para favorecer el uso de TryUpdateModelAsync sobre UpdateModel, es por la simplicidad del tratamiento de excepciones, pero no hay ninguna diferencia funcional en el proceso.

Por cada valor el model binder enlaza este al model, y registra una entrada en el estado modelo. Se puede comprobar el estado del modelo en cualquier momento después del enlace para ver si tuvo éxito.

```

[ActionName("Create")]
[HttpPost]
0 references
public IActionResult CreateProd()
{
    Product product = new Product();

    bool x = TryUpdateModelAsync(product).Result;

    var bus = new ProductBus();

    if (bus.Insert(product) > 0)
    {
        return RedirectToAction("Index");
    }

    return View(product);
}

```

Custom Binder

Se deben crear dos clases: que implementen la siguiente interfaz:

IModelBinder: Permite implementar binders para leer y enlazar los valores que los **ValueProvider** han leído de la petición **Http**.

```

namespace Microsoft.AspNetCore.Mvc.ModelBinding
{
    ...public interface IModelBinder
    {
        ...Task<ModelBindingResult> BindModelAsync(ModelBindingContext bindingContext);
    }
}

```

Dónde:

- **BindModelAsync**: Este método se invoca al momento de realizarse el enlace a los datos que llegan en la petición **Http**.

Para usar este Custom Binder, se debe usar el **ModelBinder** de la siguiente manera:

```

[ActionName("Create")]
[HttpPost]
0 references
public IActionResult CreateProd([ModelBinder(BinderType = typeof(ProductBinder))] Product product)
{

```

El mecanismo que permite entender de donde provienen los datos para poder acceder a estos y utilizarlos para crear o actualizar objetos del Modelo, es el **Value Provider**.

Value Provider

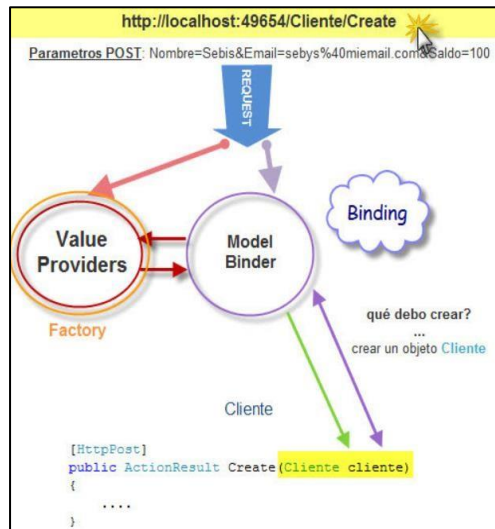
Es el que brinda acceso a la información que se puede utilizar para el proceso de **Model Binding**.

Se basan en la implementación de las siguientes interfaces:

IValueProvider: Permite implementar el comportamiento de lectura de los valores que se envían desde el browser.

Dónde:

- ContainsPrefix: se llama por el model binder para determinar si el value provider puede resolver los datos para un prefijo dado.
- GetValue: devuelve un valor para una clave dada de dato, o null si el value provider no tiene ningún dato adecuado.



Value Provider es el que brinda acceso a la información que se puede utilizar para el proceso de Model Binding.

MVC posee varios Value Providers que el Default Model Binder utiliza (todos ellos implementan la interface `IValueProvider`):

- `FormValueProvider`
- `RouteDataValueProvider`
- `QueryStringValueProvider`
- `HttpFileCollectionValueProvider`

Custom Provider

Se deben crear dos clases: que implementen las siguientes interfaces:

`IValueProvider`: Permite implementar el comportamiento de lectura de los valores que se envían desde el browser.

```

namespace Microsoft.AspNet.Mvc.ModelBinding
{
    ... public interface IValueProvider
    {
        ... bool ContainsPrefix(string prefix);
        ... ValueProviderResult GetValue(string key);
    }
}

```

Dónde:

- ContainsPrefix: se llama por el model binder para determinar si el value provider puede resolver los datos para un prefijo dado.
- GetValue: devuelve un valor para una clave dada de dato, o null si el value provider no tiene ningún dato adecuado.

IValueProviderFactory: Permite implementar la clase Factory que va a responder como alternativa de lectura a los valores que llegan del browser.

```
namespace Microsoft.AspNet.Mvc.ModelBinding
{
    public interface IValueProviderFactory
    {
        Task<IValueProvider> GetValueProviderAsync(ValueProviderFactoryContext context);
    }
}
```

Dónde:

- GetValueProviderAsync: Devuelve un ValueProvider.

Una vez creado el Provider, se debe configurar en el Startup.cs, en el método ConfigureServices y agregar la siguiente línea por ejemplo:


```
services.AddMvc(options => options.ValueProviderFactories.Insert(0, new ProductValueProviderFactory()));
```

Donde, ProductValueProviderFactory, es el proveedor personalizado que se está agregando.

6. Creación y uso de ViewModels

Creación y uso de ViewModels


- Con frecuencia, una Vista tiene que mostrar una variedad de datos que no corresponde directamente a un modelo de dominio.
- Uno de los enfoques que se puede tomar, y el que ofrece ventajas como intellisense y permite la creación de vistas strongly-typed, es el de escribir una clase personalizada para la Vista, es decir, un ViewModel.



```
graph TD; View[View] --> ViewModel[ViewModel]; ViewModel --> Model[Model];
```

3 - 35

Copyright © Todos los Derechos Reservados - Cibertec Perú SAC.



Con frecuencia, una Vista tiene que mostrar una variedad de datos que no corresponden directamente a un modelo de dominio.

Por ejemplo, es posible tener que diseñar una vista destinada a mostrar los detalles del producto y también información que es accesoria a la del producto, pero que no forma parte de su modelo.

Un método fácil de mostrar los datos extra, que no es una parte del modelo principal, es simplemente enviar los datos en el ViewBag. Sin duda, hace el trabajo y proporciona un enfoque flexible para mostrar datos dentro de una vista, pero no es la mejor solución.

Uno de los enfoques que se puede tomar, y el que ofrece ventajas como intellisense y permite la creación de vistas strongly-typed, es el de escribir una clase personalizada para la Vista, es decir, un ViewModel.

Se puede pensar en un ViewModel, como un Modelo que existe solo para proporcionar información a una Vista en particular, y que se basa en Modelos ya existentes.

Tener en cuenta que el término ViewModel en el contexto de MVC es diferente del concepto de ViewModel en el patrón MVVM (Model View ViewModel).

7. Introducción a los View Components

Introducción a los View Components

Los View Components son muy similares a lo que en versiones anteriores se conocían como Vistas parciales, las cuales buscan renderizar pequeños bloques de HTML, sirve en casos en los que se requiera manipular pequeñas porciones de una página.

1. Implementar una clase que herede de ViewComponent.

```
public class PriorityListViewComponent : ViewComponent
{
```

2. Crear la vista asociada a este componente

3 - 37

Copyright © Todos los Derechos Reservados - Cibertec Perú SAC.



Los View Components son muy similares a lo que en versiones anteriores se conocían como Vistas parciales, las cuales buscan renderizar pequeños bloques de HTML, sirve en casos en los que se requiera manipular pequeñas porciones de una página, por ejemplo, con manejo asíncronico de una página. La principal diferencia es que un ViewComponent es mucho más poderoso que una Vista Parcial debido a las formas de manipular la comunicación.

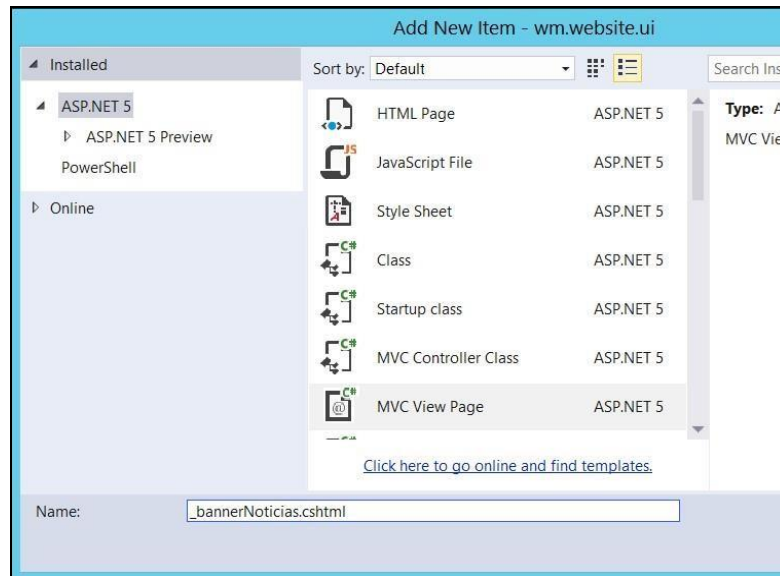
Los View components pueden ser usados para resolver problemas que pueden ser complejos si se implementan con vistas parciales, como, por ejemplo:

- Menús dinámicos
- Panel de Login.
- Shopping Cart
- Artículos recientemente publicados

Un View Component consiste en dos partes, **la clase** (deriva la clase ViewComponent) y la **vista que con Razor** llaman a métodos que se encuentran en la clase.

Ejemplo de Vista Parcial

1. Crear una vista en el folder Views/Shared que será invocada desde el _Layout.cshtml



2. Reemplace todo el contenido por el contenido a continuación (el css se agrega en el mismo archivo solamente por simplicidad, no es una buena práctica bajo ninguna razón).

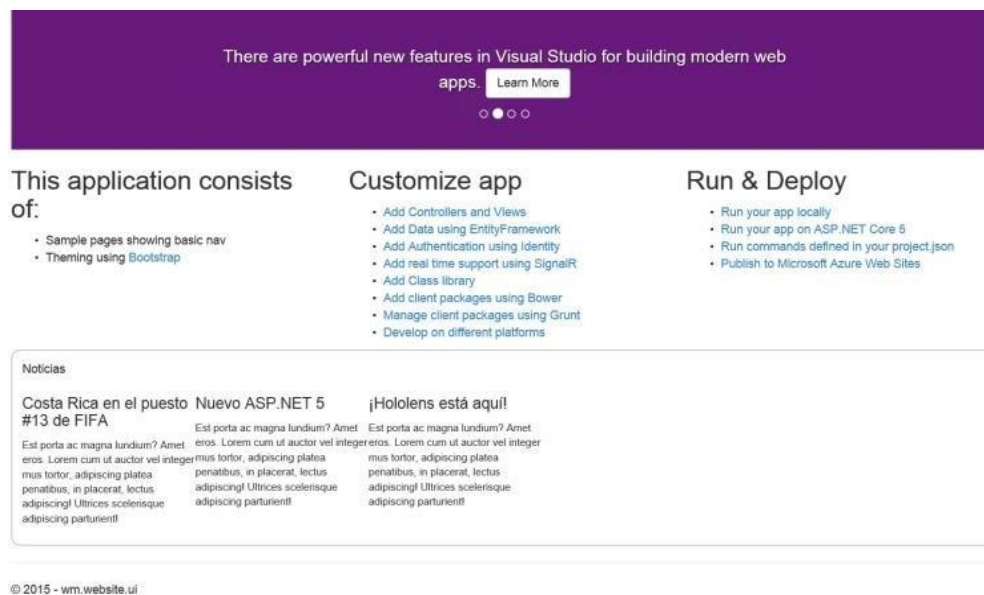
```
<div class="bannerNoticias">
  <p>Noticias</p>
  <div class="noticias">
    <h4>Costa Rica en el puesto #13 de FIFA</h4>
    <p>Est porta ac magna lundium? Amet eros. Lorem cum ut auctor vel
integer mus tortor, adipiscing platea penatibus, in placerat, lectus adipiscing! Ultrices
scelerisque adipiscing parturient!</p>
  </div>
  <div class="noticias">
    <h4>jHololens está aquí!</h4>
    <p>Est porta ac magna lundium? Amet eros. Lorem cum ut auctor vel
integer mus tortor, adipiscing platea penatibus, in placerat, lectus adipiscing! Ultrices
scelerisque adipiscing parturient!</p>
  </div>
  <div class="clearer"></div>
</div>
<style>
  .bannerNoticias { border:solid 1px #aaa; border-radius:10px; padding:12px;
}
  .bannerNoticias .noticias { display:inline-block; float:left;
    max-width:200px; font-size:12px;
  }
  .clearer{ clear:both;
}
</style>
```

3. En el archivo de Layout, inserte la siguiente línea de código debajo de la sentencia `@RenderBody`

```
@RenderBody()

@await Html.PartialAsync("_bannerNoticias")
```

4. Ejecute el proyecto (F5) y podrá apreciar una Vista parcial incrustada en cada página del sitio.



Ejemplo de ViewComponent

En este caso crearemos un viewcomponent que logre lo mismo que la vista parcial anterior, sin embargo, esta vez obtendremos las noticias desde el código, lo que más adelante podría ser reemplazado por datos desde una base de datos.

1. Crear un folder llamado ViewComponents en la raíz del proyecto.
2. Cree una clase llamada BannerInteractivoViewComponent.cs
3. Reemplace el contenido de la clase por el siguiente.

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;
namespace <nombredeespacio>.ViewComponents
{
    public class BannerInteractivoViewComponent : ViewComponent
    {
        public IViewComponentResult Invoke()
        {
            noticias = new Dictionary<string, string>() {
                { "Costa Rica en el puesto #13 de FIFA", "Est porta ac magna lundium? Amet eros. Lorem cum ut auctor vel integer mus tortor, adipiscing platea penatibus, in placerat, lectus adipiscing! Ultrices scelerisque adipiscing parturient!" }
            };
            return View(noticias);
        }
    }
}
```

```

adipiscing! Ultrices scelerisque adipiscing parturient!"),
{ "Nuevo ASP.NET 5", "Est porta ac magna lundium? Amet eros. Lorem cum ut auctor vel
integer mus tortor, adipiscing platea penatibus, in placerat, lectus adipiscing! Ultrices
scelerisque adipiscing parturient!"))}
return View(noticias);
}
}
}

```

4. Ahora, cree un folder en la dirección Views/Shared/Components y dentro de ese folder, cree otro llamado BannerInteractivo (Views/Shared/Components/BannerInteractivo) y luego cree una vista que se llame Default.cshtml, agregue el siguiente contenido a la vista (el css se agrega en el mismo archivo solamente por simplicidad, no es una buena práctica bajo ninguna razón).

```

@model Dictionary<string, string>
<h2>¡Esto podría venir de una base de datos!</h2>
<div class="bannerNoticias">
  <p>Noticias</p>
  @foreach (var item in Model)
  {
    <div class="noticias">
      <h4>@item.Key</h4>
      <p>@item.Value</p>
    </div>
  }
  <div class="clearer"></div>
</div>
<style>
  .bannerNoticias {
    border: solid 1px #aaa; border-radius: 10px; padding: 12px;
  }
  .bannerNoticias .noticias { display: inline-block; float: left;
    max-width: 200px; font-size: 12px;
  }
  .clearer { clear: both;
  }
</style>

```

5. **En el `_layout.cshtml` agregue la siguiente línea debajo del `@await Html.PartialAsync("_bannerNoticias")` agregado previamente, esto para invocar nuestro ViewComponent.**

```
@Component.Invoke("BannerInteractivo")
```

6. Si ejecuta el proyecto (F5) verá el mismo resultado 2 veces, sin embargo, con el ViewComponent podrá sacar los datos desde una base de datos más adelante e inclusive ejecutarlo de manera asincrónica.