**Guía de ASP.Net MVC 4**

Contents

[Iniciando con ASP.Net MVC 4 2](#_Toc488628189)

[El Modelo 3](#_Toc488628190)

[La vista 3](#_Toc488628191)

[El controlador 3](#_Toc488628192)

[ASP.NET MVC 1 Overview 3](#_Toc488628193)

[ASP.NET MVC 2 Overview 4](#_Toc488628194)

[ASP.NET MVC 3 Overview 4](#_Toc488628195)

[ASP.NET MVC 4 Overview 4](#_Toc488628196)

[**CONTROLLER** 5](#_Toc488628197)

[Action Method 6](#_Toc488628198)

[Action Results 6](#_Toc488628199)

[Actions 8](#_Toc488628200)

[Action Selectors 8](#_Toc488628201)

[**MODEL** 9](#_Toc488628202)

[**VIEW** 10](#_Toc488628203)

[Razor Syntax 11](#_Toc488628204)

[**Enviar información desde el controlador a la vista.** 13](#_Toc488628205)

[**Integrate Controller, View and Model** 20](#_Toc488628206)

[Modelo 20](#_Toc488628207)

[Controller 20](#_Toc488628208)

[View 20](#_Toc488628209)

[**Forms and HTML Helpers** 22](#_Toc488628210)

[GET o POST 22](#_Toc488628211)

[Tipos de HtmlHelper 22](#_Toc488628212)

[HtmlHelper 29](#_Toc488628213)

[**Data Annotations and Validation** 31](#_Toc488628214)

[DataAnnotations 31](#_Toc488628215)

[ASP.NET MVC Client Side validation requirements 32](#_Toc488628216)

[**Archivos de configuración.** 34](#_Toc488628217)

# 

# Iniciando con ASP.Net MVC 4

MVC es un **patrón arquitectónico** enfocado a la forma de presentar la información de una aplicación al usuario, así como a la forma de recibir e interpretar las acciones que este realiza sobre el programa. MVC tiene sus orígenes por ahí de 1970.

Surgió inicialmente como un patrón para la creación de aplicaciones de escritorio, sin embargo, con el tiempo (y el surgimiento de la Web) tuvo gran auge en el mundo de aplicaciones en internet y ahí fue se le reconoce más, solo para aparecer también en el desarrollo móvil

### El Modelo

El modelo es el que contiene y gestiona la información de la aplicación, también “**contiene**” la lógica relacionada con las reglas del negocio. En cierto sentido debe ser el cerebro de nuestra app.

El modelo debe saber absolutamente nada de la interfaz gráfica, y su única preocupación debe ser la de llevar a cabo todo lo que nuestra aplicación debe hacer en lo que concierne al problema que estamos buscando resolver con la app. El modelo es el encargado de **conectarse con la base de datos**, **conectarse con servicios web** y cualquier tipo de **manipulación de datos en general**. Idealmente el modelo debe ser algo que sea independiente de la plataforma en la que se esté ejecutando. Es decir, más allá de las limitantes del lenguaje, deberíamos poder llevar el modelo de una app para iOS a una aplicación web.

Es completamente normal que el modelo sea la mayor parte del código de tu app, sin embargo, a su vez, el modelo puede estar separado en otras capas, tal vez una de lógica de la aplicación, otra que únicamente esté diseñada para el acceso a datos y otra que se conecte con servicios web, que en MVC el modelo se presente como una sola sección de tu código, no significa que este deba ser un monolito de código espagueti imposible de separar en capas.

### La vista

La vista contiene todos los elementos que son visibles al usuario, ya sea una **página web en HTML** o un Storyboard en una app para iOS. Sin embargo, la vista no siempre está formada por archivos de interfaz gráfica, sino que también se pueden incluir cualquier otra herramienta que ayude a presentar los datos. Al contrario del modelo, la vista debe saber nada de la lógica de la aplicación. En teoría, su responsabilidad debe limitarse a definir la estructura, apariencia y comportamiento de los datos de la aplicación en la pantalla.

Sin embargo, te encontrarás con que muchas veces, algunas reglas del negocio aparecen en la vista en forma de validaciones, si bien esto no está definido dentro del modelo, puesto que no debería saber nada de la app, sin embargo, las validaciones (y los mensajes que estas generan, están relacionadas con los datos que el usuario metió) así que…

La vista nunca, nunca debe saber nada del modelo, y viceversa, para la comunicación entre ellos se dará a través de la última pieza de este patrón.

### El controlador

El Controlador es el intermediario, este se encarga de **gestionar la aplicación**, estos preparan la vista con los datos traídos del modelo y **se asegura que exista una comunicación entre ellos**: toma las cosas que le da la vista y las transforma a algo que el modelo entienda, así mismo, toma las cosas que el modelo le da y las convierte en algo que puede ser mostrado por la vista.

En el controlador también puede ocurrir el proceso de autenticación y autorización, para decidir si un usuario tiene permiso de realizar cualquier acción que desee. Este también decide si responde a una petición de la forma en que la vista la generó, y también es el responsable de la navegación entre las vistas de una app.

### ASP.NET MVC 1 Overview

In February 2007, Scott Guthrie (“ScottGu”) of Microsoft sketched out the core of ASP.NET MVC while flying on a plane to a conference on the East Coast of the United States. It was a simple application, containing a few hundred lines of code, but the promise and potential it offered for parts of the Microsoft web developer audience was huge.

All these highlighted a philosophy that placed a high value on community interaction throughout the development process. The end result was that the offi cial MVC 1.0 release — including code and unit tests — had already been used and reviewed by the developers who would be using it. ASP.NET MVC 1.0 was released on 13 March 2009.

### ASP.NET MVC 2 Overview

ASP.NET MVC 2 was released just one year later, in March 2010. Some of the main features in MVC 2 included:

* UI helpers with automatic scaffolding with customizable templates
* Attribute-based model validation on both client and server
* Strongly typed HTML helpers
* Improved Visual Studio tooling

There were also lots of API enhancements and “pro” features, based on feedback from developers building a variety of applications on ASP.NET MVC 1, such as:

* Support for partitioning large applications into areas
* Asynchronous controllers support
* Support for rendering subsections of a page/site using Html.RenderAction
* Lots of new helper functions, utilities, and API enhancements

One important precedent set by the MVC 2 release was that there were very few breaking changes.

I think this is a testament to the architectural design of ASP.NET MVC, which allows for a lot of extensibility without requiring core changes.

### ASP.NET MVC 3 Overview

ASP.NET MVC 3 shipped just 10 months after MVC 2, driven by the release date for Web Matrix.

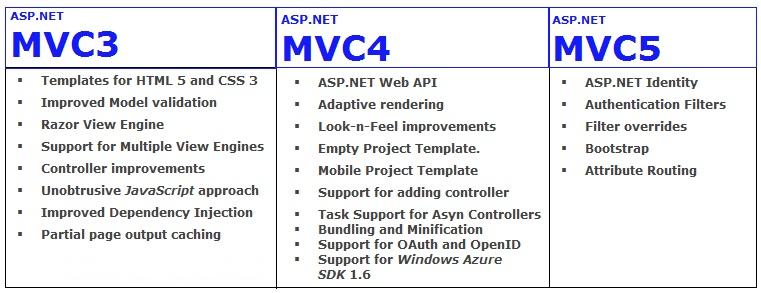
Some of the top features in MVC 3 included:

* The Razor view engine
* Support for .NET 4 Data Annotations
* Improved model validation
* Greater control and flexibility with support for dependency resolution and global action filters
* Better JavaScript support with unobtrusive JavaScript, jQuery Validation, and JSON binding
* Use of NuGet to deliver software and manage dependencies throughout the platform

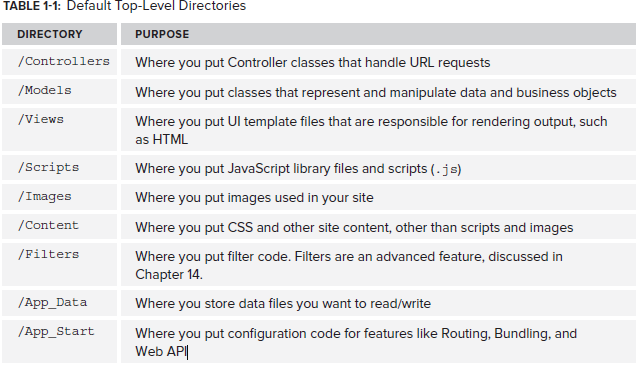
### ASP.NET MVC 4 Overview

The MVC 4 release is building on a pretty mature base and is able to focus on some more advanced scenarios. Some top features include:

* ASP.NET Web API
* Enhancements to default project templates
* Mobile project template using jQuery Mobile
* Display Modes
* Task support for Asynchronous Controllers
* Bundling and minification



***Ejemplo de la estructura de un proyecto MVC:***



## **CONTROLLER**

**Gestiona** cualquier **solicitud (request)** de una **URL**. El controlador es una clase, derivada de la clase base **System.Web.Mvc.Controller**. La clase Controller contiene métodos públicos denominados Métodos de acción (**Action methods**). El controlador y su método de acción gestionan solicitudes de navegador entrantes, recuperan los datos de modelo necesarios y devuelven respuestas apropiadas. En ASP.NET MVC, cada nombre de clase de controlador debe terminar con una palabra "**Controller**" y se puede crear un nuevo "**Controller**" utilizando diferentes **scaffolding templates**. También puede crear una plantilla personalizada.

***Ejemplo:***

public class HomeController : Controller

{

// GET: Metodo de Accion

public ActionResult Index()

{

return View();

}

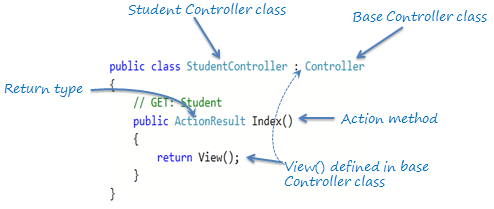
}

### Action Method

Todos los métodos públicos de una clase **Controller** se denominan **Action methods**. Son como cualquier otro método normal con las siguientes restricciones:

* El método de acción debe ser público. No puede ser privado o protegido.
* El método de acción no puede sobrecargarse.
* El método de acción no puede ser un método estático.

***Ejemplo:***



### Action Results

MVC framework includes various result classes, which can be return from an action methods

|  |  |
| --- | --- |
| **Result Class** | **Description** |
| ViewResult | Represents HTML and markup. |
| EmptyResult | Represents No response. |
| ContentResult | Represents string literal. |
| FileContentResult/ FilePathResult/ FileStreamResult | Represents the content of a file |
| JavaScriptResult | Represent a JavaScript script. |
| JsonResult | Represent JSON that can be used in AJAX |
| RedirectResult | Represents a redirection to a new URL |
| RedirectToRouteResult | Represent another action of same or other controller |
| PartialViewResult | Returns HTML from Partial view |
| HttpUnauthorizedResult | Returns HTTP 403 status |

La clase **ActionResult** es una clase base de todas las clases de resultados anteriores, por lo que puede ser tipo de retorno de métodos de acción que devuelve cualquier tipo de resultado enumerado anteriormente. Sin embargo, puede especificar la clase de resultado apropiada como un tipo de retorno del método de acción.

|  |  |  |
| --- | --- | --- |
| **Result Class** | **Description** | **Base Controller method** |
| ViewResult | Represents HTML and markup. | View() |
| EmptyResult | Represents No response. |  |
| ContentResult | Represents string literal. | Content() |
| FileContentResult, | Represents the content of a file | File() |
| FilePathResult, |  |  |
| FileStreamResult |  |  |
| JavaScriptResult | Represent a JavaScript script. | JavaScript() |
| JsonResult | Represent JSON that can be used in AJAX | Json() |
| RedirectResult | Represents a redirection to a new URL | Redirect() |
| RedirectToRouteResult | Represent another action of same or other controller | RedirectToRoute() |
| PartialViewResult | Returns HTML | PartialView() |
| HttpUnauthorizedResult | Returns HTTP 403 status |  |

***Ejemplos:***

//ViewResult

public ActionResult AR\_ViewResult()

{

return View();

}

//ContentResult

public ActionResult AR\_ContentResult()

{

return Content("This method returns plain text.");

}

//JsonResult

public ActionResult AR\_JsonResult()

{

var employee = new

{

Name = "Eduardo",

Age = "30",

Occupation = "Software Developer"

};

return Json(employee, JsonRequestBehavior.AllowGet);

}

//FileResult

public ActionResult AR\_FileResult()

{

return File(Server.MapPath("~/App\_Data/documentation.pdf"),

contentType: "application/pdf",

fileDownloadName: "documentation.pdf");

}

//RedirectToAction

public ActionResult AR\_RedirectToAction()

{

return RedirectToAction("Contact");

}

public ActionResult Contact()

{

return Content("This is the redirected Contact method.");

}

### Actions

Los métodos de acción son los métodos que tienen asignación uno a uno con las solicitudes de los usuarios. Estos métodos ejecutan peticiones y crean una respuesta. Cada Controlador puede tener un método de acción por defecto, sin embargo, que debe ser declarado en el método de **RegisteRoutes()** del archivo **RouteConfig**.

### Action Selectors

Los selectores de acción son los atributos que se pueden aplicar a los métodos de acción. Con la ayuda de selectores de acción, el motor de enrutamiento selecciona el método de acción correcto para manejar la solicitud específica. Los selectores de acción también deciden el comportamiento del método sobre la base del nombre dado delante del método de acción.

Hay tres tipos de atributos selectores de acción:

* ***ActionName***
* ***NonAction***
* ***ActionVerbs***
* **ActionName**: Permite usar un nombre diferente al nombre real del método.

[ActionName("PagInicio")]

public string Accion()

{

return "This is ASP.Net MVC ActionName";

}

In order to have two action methods with the same name inside a controller, we'll use the **ActionName** attribute

*//First Index Method*

public ActionResult Index()

{

return Content("This is the first Index action");

}

*//Second Index Method*

  [ActionName("IndexNew")]

public ActionResult Index(string something)

{

return Content("This is the second Index action");

}

* **NonAction**: Es un atributo incorporado que indica que un método de un controlador no es un método de acción. Se utiliza cuando se desea que un método no se trate como un método de acción.

[NonAction]

public string GetTimeAsString()

{

return "This is detail at " + DateTime.Now.ToString("T");

}

Al Ejecutar la acción muestra lo siguiente en el navegador:

**Server Error in '/' Application.**

*The resource cannot be found.*

**Description:** HTTP 404. The resource you are looking for (or one of its dependencies) could have been removed, had its name changed, or is temporarily unavailable.  Please review the following URL and make sure that it is spelled correctly.   
  
Requested URL**: /Home/GetTimeAsString**

* **ActionVerbs:** Se utiliza cuando se desea controlar la selección de un método de acción basado en un método de solicitud Http. Por ejemplo, puede definir **dos métodos** de acción diferentes **con el mismo nombre**, pero un método de acción responde a una solicitud **HTTP Get** y otro método de acción responde a una solicitud **HTTP Post**.

Si no aplica ningún atributo, entonces lo considera una **solicitud GET de forma predeterminada.**

Varios verbos de acción pueden aplicarse a un solo método de acción utilizando el atributo **AcceptVerbs**.

MVC soporta los siguientes **ActionVerbs**:

|  |  |
| --- | --- |
| **Http method** | **Usage** |
| GET | To retrieve the information from the server. Parameters will be appended in the query string. |
| POST | To create a new resource. |
| PUT | To update an existing resource. |
| HEAD | Identical to GET except that server do not return message body. |
| OPTIONS | OPTIONS method represents a request for information about the communication options supported by web server. |
| DELETE | To delete an existing resource. |
| PATCH | To full or partial update the resource. |

***Ejemplos:***

[HttpGet]

public ActionResult Editar(string nombre)

{

ViewBag.HttpGet = nombre;//Request.QueryString["nombre"];

return View();

}

//[HttpPost]

//public ActionResult Editar(int id)

//{

// ViewBag.HttpPost = id;

// return View();

//}

[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Get)]

public ActionResult GetAndPostAction()

{

return RedirectToAction("Index");

}

## **MODEL**

El modelo representa los datos específicos del dominio y la lógica empresarial en la arquitectura MVC. Mantiene los datos de la aplicación. Los objetos de modelo recuperan y almacenan el estado del modelo en el almacén de persistencia como una base de datos. La clase de modelo contiene datos en propiedades públicas. Todas las clases Model residen en la carpeta Model en la estructura de carpetas MVC.

***Ejemplo:***

namespace GuiaMVC4.Models

{

public class Usuario

{

public string Mensaje { get; set; }

public DateTime FechaAlta { get; set; }

public string Vista { get; set; }

public List<Ciudad> ListaCiudades { get; set; }

}

public class Ciudad

{

public Ciudad(string city)

{

ciudad = city;

}

public string ciudad { get; set; }

}

}

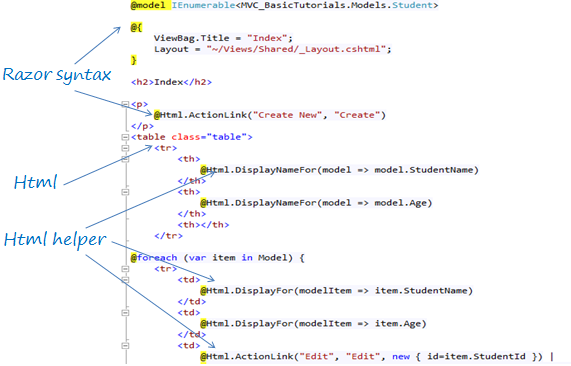
**VIEW**

La vista es la interfaz del usuario, muestra datos del modelo al usuario y también les permite modificar los datos.

ASP.NET MVC supports following types of view files:

|  |  |
| --- | --- |
| **View file extension** | **Description** |
| .cshtml | C# Razor view. Supports C# with html tags. |
| .vbhtml | Visual Basic Razor view. Supports Visual Basic with html tags. |
| .aspx | ASP.Net web form |
| .ascx | ASP.NET web control |

***Ejemplo:***



### Razor Syntax

Razor es uno de los motores de vista soportados en ASP.NET MVC, permite escribir una mezcla de HTML y el código del lado del servidor utilizando C # o Visual Basic. La sintaxis para **VB.Net** tiene extensión **.vbhtml** y la sintaxis **C#** tiene extensión de archivo **.cshtml.**

* **Inline expression**  
  Comience con el símbolo @ para escribir el lado del servidor con código Html. Por ejemplo, escriba @Variable\_Name para mostrar un valor de una variable del lado del servidor. Por ejemplo:

<h1>Razor syntax demo</h1>

<h2>@DateTime.Now.ToShortDateString()</h2>

**Result:**

08-09-2014

* **Multi-statement Code block**  
  Puede escribir varias líneas de código del lado del servidor entre llaves @{ ... }

Cada línea debe terminar con punto y coma igual que C #.

@{

var date = DateTime.Now.ToShortDateString();

var message = "Hello World";

}

<h2>Today's date is: @date </h2>

<h3>@message</h3>

**Result:**

Today's date is: 08-09-2014

Hello World!

* **Display text from code block**

Use @: or <text>/<text> to display texts within code block.

@{

var date = DateTime.Now.ToShortDateString();

string message = "Hello World!";

@:Today's date is: @date <br />

@message

}

**Result:**

Today's date is: 08-09-2014

Hello World!

Display text using <text> within a code block as shown below.

@{

var date = DateTime.Now.ToShortDateString();

string message = "Hello World!";

<text>Today's date is:</text> @date <br />

@message

}

**Result:**

Today's date is: 08-09-2014

Hello World!

* **if-else condition**

Write if-else condition starting with @ symbol. The if-else code block must be enclosed in braces { }, even for single statement.

@if(DateTime.IsLeapYear(DateTime.Now.Year) )

{

@DateTime.Now.Year @:is a leap year.

}

else {

@DateTime.Now.Year @:is not a leap year.

}

**Result:**

2014 is not a leap year.

* **for loop**

@for (int i = 0; i < 5; i++) {

@i.ToString() <br />

}

**Result:**

0

1

2

3

4

* **Model**

Use @model to use model object anywhere in the view.

@model Student

<h2>Student Detail:</h2>

<ul>

<li>Student Id: @Model.StudentId</li>

<li>Student Name: @Model.StudentName</li>

<li>Age: @Model.Age</li>

</ul>

**Result:**

**Student Detail:**

- Student Id: 1

- Student Name: John

- Age: 18

* **Declare Variables**

Declare a variable in a code block enclosed in brackets and then use those variables inside html with @ symbol.

@{

string str = "";

if(1 > 0)

{

str = "Hello World!";

}

}

<p>@str</p>

**Result:**

Hello World!

## **Enviar información desde el controlador a la vista.**

En ASP.NET MVC hay 3 formas de pasar datos del **controlador** a la **vista** y también se puede utilizar **Session** para persistir los datos durante una sesión de usuario.

* **ViewBag**: Es la versión mejorada de ViewData, sus puntos importantes son:
* ViewBag es una propiedad dinámica.
* **No es necesario usar casting.**
* Trabaja con framework arriba de 4.0
* ViewBag es más lento que ViewData
* Su ciclo de vida solo dura cuando ocurre la solicitud.
* Si la redirección ocurre entonces su valor se convierte en nulo.

***Ejemplo:***

public ActionResult Vista\_ViewBag()

{

DateTime aDay = DateTime.Now;

ViewBag.Mensaje = "Mensaje ViewBag desde HomeController";

ViewBag.FechaAlta = aDay;

ViewBag.Vista = "ViewBag";

ViewBag.Ciudades = new List<string>()

{

"Monterrey", "Guadalajara", "Saltillo","Monclova"

};

return View();

}

* **ViewData**: Es un diccionario key/value, unos de sus puntos importantes son:
* ViewData es un objeto diccionario.
* Almacenar cualquier tipo de dato, pero a la hora de recuperar el dato no tiene tipo, tenemos **que realizar conversiones (casting).**
* Trabaja con framework arriba de 3.5
* ViewData es más rápido que ViewBag
* Su ciclo de vida solo dura cuando ocurre la solicitud.
* Si la redirección ocurre entonces su valor se convierte en nulo.
* Se requiere el encasillado para obtener datos y comprobar valores nulos para evitar errores.

***Ejemplo:***

public ActionResult Vista\_ViewData()

{

DateTime aDay = DateTime.Now;

ViewData["Mensaje"] = "Mensaje ViewData desde HomeController";

ViewData["FechaAlta"] = aDay;

ViewData["Vista"] = "ViewData";

ViewData["Ciudades"] = new List<string>()

{

"Monterrey", "Guadalajara", "Saltillo","Monclova"

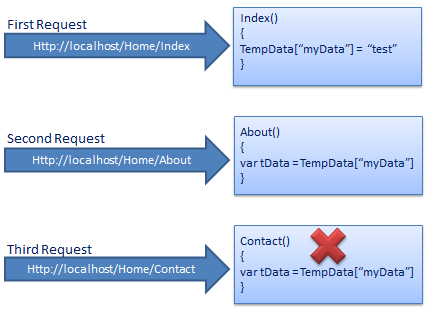
};

return View();

}

* **Tempdata**: Deriva de la clase TempDataDictionary, sus puntos importantes son:
* También es un diccionario pero con la característica de que se mantiene durante una solicitud HTTP.
* Es usada para pasar datos de controlador a controlador o de una acción a otra acción.
* Expirará automáticamente el valor de TempData, es decir que cualquier dato que ponga en TempData se desecha después de la siguiente solicitud completa.
* Similar a un almacén temporal en el que podemos guardar datos desde una acción y consultarlos desde otra.
* Necesita casting.
* Se utiliza para almacenar sólo mensajes de una vez como mensajes de error, mensajes de validación.
* Llame a **TempData.Keep()** para retener los valores TempData en una tercera solicitud consecutiva.

***Ejemplo:***

****

public ActionResult Vista\_TempData()

{

DateTime aDay = DateTime.Now;

TempData["Mensaje"] = "Mensaje TempData desde HomeController";

TempData["FechaAlta"] = aDay;

TempData["Vista"] = "TempData";

var Ciudades = new List<string>()

{

"Monterrey", "Guadalajara", "Saltillo","Monclova"

};

TempData["Ciudades"] = Ciudades;

return RedirectToAction("Vista\_TempData2");

}

public ActionResult Vista\_TempData2()

{

string data;

if (TempData["myData"] != null)

data = TempData["myData"] as string;

TempData.Keep();

return View();

}

* **Sesssion**:
* Es usada para pasar datos de controlador a controlador, vista a controlador y de controlador a vista
* Sesión es capaz de almacenar datos mucho más largo tiempo, hasta que la sesión del usuario no está caduca (por defecto el tiempo de expiración de la sesión es de 20 minutos, pero se puede aumentar).
* Necesita casting.
* Sesión es válido para distintos pedidos y no una simple redirección
* sintaxis: Session[“value”]=values;
* **Model:**

La propiedad Model no funciona como las anteriores, sino que lo que se **pasa es un objeto**, que se manda de la acción hacia la vista.

A diferencia de ViewData y ViewBag que existen tanto en el controlador como en la vista, el controlador no tiene una propiedad Model. En su lugar se usa una sobrecarga del método **View()** y se manda el objeto como parámetro.

Pasos:

* Generar una clase que contiene las entidades a utilizar.
* Instanciarla la clase del modelo en el controlador.
* Generar la vista y mandar llamar el @model.

***Ejemplo:***

**MODEL**

public class **Usuario**

{

public string Mensaje { get; set; }

public DateTime FechaAlta { get; set; }

public string Vista { get; set; }

public List<Ciudad> ListaCiudades { get; set; }

}

**CONTROLLER**

public ActionResult Vista\_ViewModel()

{

DateTime aDay = DateTime.Now;

List<Ciudad> lsciudad = new List<Ciudad>();

lsciudad.Add(new Ciudad("Monterrey"));

lsciudad.Add(new Ciudad("Guadalajara"));

lsciudad.Add(new Ciudad("Saltillo"));

lsciudad.Add(new Ciudad("Monclova"));

**Usuario data = new Usuario();**

data.Mensaje = "Mensaje ViewModel desde HomeController";

data.FechaAlta = aDay;

data.Vista = "ViewModel";

data.ListaCiudades = lsciudad;

return View(data);

}

**VIEW**

@{

Layout = null;

ViewBag.Title = "Vista Model";

@model GuiaMVC4.Models.Usuario

}

<head>

<meta name="viewport" content="width=device-width" />

<title>@ViewBag.Title</title>

</head>

<body>

<div>

<b> @Model.Mensaje </b><strong>@Model.FechaAlta.ToLongDateString()</strong>

<hr />

<p>Ejemplo de @Model.Vista</p>

<ul>

@foreach (var ciudad in @Model.ListaCiudades)

{

<li>@ciudad.ciudad</li>

}

</ul>

</div>

</body>

* **ViewModel:**

Permite configurar varias entidades de uno o más modelos de datos o fuentes en un único objeto.

Optimizado para el consumo y renderizado por la vista.

1. Si necesita pasar más de una cosa a una vista fuertemente tipificada (que es la mejor práctica),

     Generalmente querrá crear una clase separada para hacerlo.

2. Esto le permite validar su ViewModel de forma diferente a su modelo de dominio para

     Escenarios de validación basados en atributos

3. Se puede utilizar para ayudar a dar forma y formatear datos.

     Por ejemplo: ¿necesita un valor de fecha o dinero con formato de una manera particular?

           ViewModel es el mejor lugar para hacerlo.

4. El uso de un ViewModel puede hacer que la interacción entre el modelo y la vista sea más simple

**MODEL**

***Product.cs***

  public class Product

    {

        public Product() { Id = Guid.NewGuid(); }

        public Guid Id { get; set; }

        public string ProductName { get; set; }

        public virtual ProductCategory ProductCategory { get; set; }

    }

***ProductCategory.cs***

    public class ProductCategory

    {

        public int Id { get; set; }

        public string CategoryName { get; set; }

        public virtual ICollection<Product> Products { get; set; }

    }

**VIEWMODEL**

***ProductViewModel.cs***

 public class ProductViewModel

    {

        public Guid Id { get; set; }

        [Required(ErrorMessage = "required")]

        public string ProductName { get; set; }

        public int SelectedValue { get; set; }

        public virtual ProductCategory ProductCategory { get; set; }

        [DisplayName("Product Category")]

        public virtual ICollection<ProductCategory> ProductCategories { get; set; }

   }

**CONTROLLER ACTION METHODS**

 [HttpGet]

        public ActionResult AddProduct()

        {

            //instantiate the product repository

            var productRepository = new ProductRepository();

            //for get product categories from database

            var prodcutCategories = productRepository.GetAllProductCategories();

            //for initialize viewmodel

            var productViewModel = new ProductViewModel();

            //assign values for viewmodel

            productViewModel.ProductCategories = prodcutCategories;

            //send viewmodel into UI (View)

            return View("AddProduct", productViewModel);

        }

     [HttpPost]

     public ActionResult AddProduct(ProductViewModel productViewModel) //save entered data

     {

        //instantiate the product repository

        var productRepository = new ProductRepository();

//get product category for selected drop down list value

 var prodcutCategory=productRepository.GetProductCategory(productViewModel.SelectedValue);

        //for get all product categories

        var prodcutCategories = productRepository.GetAllProductCategories();

        //for fill the drop down list when validation fails

        productViewModel.ProductCategories = prodcutCategories;

        //for initialize Product domain model

        var productObj = new Product

          {

              ProductName = productViewModel.ProductName,

              ProductCategory = prodcutCategory,

          };

        if (ModelState.IsValid) //check for any validation errors

          {

             //code to save valid data into database

           return RedirectToAction("AddProduct");

          }

          else

          {

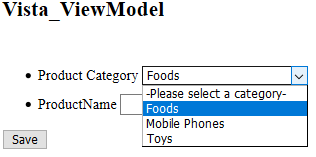
             //when validation failed return viewmodel back to UI (View)

             return View(productViewModel);

          }

    }

**VIEW**



@model ASPViewModel.ViewModels.ProductViewModel  **//set your ViewModel here**

<div class="boxedForm">

    @using (Html.BeginForm())

    {

        Html.EnableClientValidation(true);

        <ul style="margin-top:50px;">

            <li style="width: 370px">

                @Html.LabelFor(m => m.ProductCategories)

                @Html.DropDownListFor(m => m.SelectedValue,  
new SelectList(Model.ProductCategories, "Id","CategoryName"), "-Please select a category-")

                @Html.ValidationMessageFor(m => m.ProductCategory.Id)

            </li>

            <li style="width: 370px;margin-top:10px;">

                @Html.LabelFor(m => m.ProductName)

                @Html.EditorFor(m => m.ProductName)

                @Html.ValidationMessageFor(m => m.ProductName)

            </li>

        </ul>

        <div class="action">

            <button class="actionButton" type="submit">

                <span>Save</span>

            </button>

        </div>

    }

</div>

**Repository Methods**

***ProductRepository.cs***

  public class ProductRepository

    {

        /// <summary>

        /// to get product category

        /// </summary>

        public ProductCategory GetProductCategory(int categoryId)

        {

            var productCategoryList = GetAllProductCategoriesMockDatas();

            return (from p in productCategoryList

                    where (p.Id == categoryId)

                    select p).FirstOrDefault();

        }

        /// <summary>

        /// to get all product categories

        /// </summary>

        /// <returns></returns>

        public List<ProductCategory> GetAllProductCategories()

        {

            var productCategoryList = GetAllProductCategoriesMockDatas();

            return (from p in productCategoryList

                    select p)

                    .OrderBy(p => p.CategoryName)

                    .ToList();

        }

        /// <summary>

        /// to Get All Product Categories mock datas

        /// </summary>

        private List<ProductCategory> GetAllProductCategoriesMockDatas()

        {

            var productCategoryList = new List<ProductCategory>();

            productCategoryList.Add(new ProductCategory

            {

                Id = 1,

                CategoryName = "Foods",

            });

            productCategoryList.Add(new ProductCategory

            {

                Id = 2,

                CategoryName = "Toys",

            });

            productCategoryList.Add(new ProductCategory

            {

                Id = 3,

                CategoryName = "Mobile Phones",

            });

            return productCategoryList;

        }

    }

## **Integrate Controller, View and Model**

Modelo

***Ejemplo:***

namespace GuiaMVC4.Models

{

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

}

}

Controller

***Ejemplo:***

public ActionResult Estudiantes()

{

var studentList = new List<Student>{

new Student() { StudentId = 1, StudentName = "John", Age = 18 } ,

new Student() { StudentId = 2, StudentName = "Steve", Age = 21 } ,

new Student() { StudentId = 3, StudentName = "Bill", Age = 25 } ,

new Student() { StudentId = 4, StudentName = "Ram" , Age = 20 } ,

new Student() { StudentId = 5, StudentName = "Ron" , Age = 31 } ,

new Student() { StudentId = 4, StudentName = "Chris" , Age = 17 } ,

new Student() { StudentId = 4, StudentName = "Rob" , Age = 19 }

};

return View(studentList);

}

View

***Ejemplo:***

@\*@model GuiaMVC4.Models.Student\*@

@model IEnumerable<GuiaMVC4.Models.Student>

@{

Layout = null;

ViewBag.Title = "Estudiantes";

}

<!DOCTYPE html>

<html>

<head>

<meta name="viewport" content="width=device-width" />

<title>@ViewBag.Title</title>

<link href="~/Content/tabla.css" rel="stylesheet" />

</head>

<body>

<div id="wrapper">

<h2>Estudiantes</h2>

<hr />

<p>@Html.ActionLink("Create New", "Create")</p>

<br />

<table id="keywords" cellspacing="0" cellpadding="0">

<thead>

<tr>

<th><strong>@Html.DisplayNameFor(model => model.StudentName)</strong></th>

<th><strong>@Html.DisplayNameFor(model => model.Age)</strong></th>

<th><strong>Acciones</strong></th>

</tr>

</thead>

@foreach (var item in Model)

{

<tbody>

<tr>

<td class="lalign">@Html.DisplayFor(modelItem => item.StudentName)</td>

<td class="lalign">@Html.DisplayFor(modelItem => item.Age)</td>

<td class="lalign">

@Html.ActionLink("Edit", "Edit", new { id = item.StudentId }) |

@Html.ActionLink("Details", "Details", new { id = item.StudentId }) |

@Html.ActionLink("Delete", "Delete", new { id = item.StudentId })

</td>

</tr>

</tbody>

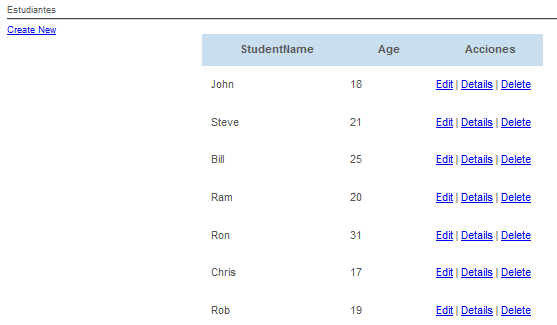
}

</table>

</div>

</body>

</html>



## **Forms and HTML Helpers**

GET o POST

Tanto GET como POST, justamente por ser métodos **ambos** de HTTP, **ejecutan** un **request** y **response**.

**GET** es **obtener información del servidor**. Traer datos que están en el servidor, ya sea en un archivo o base de datos, al cliente. Independientemente de que para eso tengamos que enviar (request) algún dato que será procesado para luego devolver la respuesta (response) que esperamos, como por ejemplo un identificador para obtener una noticia de la base de datos.

**POST** sin embargo **es enviar información desde el cliente para que sea procesada y actualice o agregue información en el servidor**, como sería la carga o actualización en sí de una noticia. Cuando enviamos (request) datos a través de un formulario, estos son procesados y luego a través de una redirección por ejemplo devolvemos (response) alguna página con información.

Las llamadas GET pueden ser cacheadas (historial del navegador), indexadas por buscadores, agregar los enlaces a nuestros favoritos o hasta pasar una url completa a otra persona para que directamente ingrese a esa página. Con el método POST sin embargo no se puede hacer esto.

Tipos de HtmlHelper

HtmlHelper Class: Es una clase que sirve para representar etiquetas HTML, podríamos decir que es un método que devuelve una cadena de texto y esta cadena de texto puede ser lo que nosotros queramos, desde etiquetas HTML, scripts

Los **HtmlHelper** se categorizan en tres tipos:

* Inline HTML helper
* Built-In HTML helper
* **Standard HTML helper**
* **Strongly typed HTML helper**
* **Template HTML helper**
* Custom HTML helper
* **Using static methods**
* **Using extension methods**
* **Using the @helper**
* **Inline HTML helper**

***Ejemplo:***

**Controller**

public ActionResult InlineHTMLhelper()

{

return View();

}

**View**

@{

Layout = null;

ViewBag.Title = "Inline HTML helper";

}

@helper OrderedList(string[] words)

{

<ol>

@foreach (string word in words)

{

<li>@word</li>

}

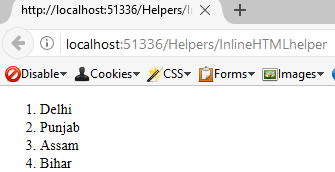
</ol>

}

<div>

@OrderedList(new string[] { "Delhi", "Punjab", "Assam", "Bihar" })

</div>



* **Custom HTML Helpers**Se pueden crear de 3 maneras:
* **Using static methods**
* **Using extension methods**
* **Using the @helper**

***Ejemplo, Using static methods***

Crear un folder ‘**CustomHTMLHelpers**’, agregar una clase static con un método static y mandarlo llamar en la vista.

**namespace GuiaMVC4.CustomHTMLHelpers**

public static class CustomHelper

{

public static MvcHtmlString Image(string source, string altTxt, string width, string height)

{

//TagBuilder creates a new tag with the tag name specified

var ImageTag = new TagBuilder("img");

//MergeAttribute Adds attribute to the tag

ImageTag.MergeAttribute("src", source);

ImageTag.MergeAttribute("alt", altTxt);

ImageTag.MergeAttribute("width", width);

ImageTag.MergeAttribute("height", height);

//Return an HTML encoded string with SelfClosing TagRenderMode

return MvcHtmlString.Create(ImageTag.ToString(TagRenderMode.SelfClosing));

}

}

**View**

@using GuiaMVC4.CustomHTMLHelpers

<h1>Custom HTML helper</h1>

<h2>Using static methods</h2>

@CustomHelper.Image("/Images/user.png", "UserPic", "100", "100")

***Ejemplo, Using extension methods***

public static MvcHtmlString Image\_HtmlHelper(this HtmlHelper htmlhelper, string source, string altTxt, string width, string height)

{

var ImageTag = new TagBuilder("img");

ImageTag.MergeAttribute("src", source);

ImageTag.MergeAttribute("alt", altTxt);

ImageTag.MergeAttribute("width", width);

ImageTag.MergeAttribute("height", height);

return MvcHtmlString.Create(ImageTag.ToString(TagRenderMode.SelfClosing));

}

**View**

@using GuiaMVC4.CustomHTMLHelpers

<h1>Custom HTML helper</h1>

<h2>Using static methods</h2>

@CustomHelper.Image("/Images/user.png", "UserPic", "100", "100")

<h2>Using Extension methods</h2>

@Html.Image\_HtmlHelper("/Images/user.png", "UserPic", "100", "100")

***Ejemplo, Using the @helper***

**View**

@helper Retrieve\_div(string content) {

<div style = "background-color:lightgrey;height:80px;width:500px" >

<mark>

<label style = "font-size:large;font-family:Arial" >

@content

</label>

</mark>

</div>

}

<h2>Using the @@helper</h2>

@Retrieve\_div(@"Este es un nuevo HTML Helper using the @helper");

* **Built-In HTML helper**Se pueden crear de 3 maneras:
* **Standard HTML helper**
* **Strongly typed HTML helper**
* **Template HTML helper**

**Standard HTML helper**

Se usan para representar el tipo **más común de controles** HTML como **TextBox, DropDown**, **Radiobutton**, etc.

**Formulario**: Para crear el elemento Form, podemos usar el método de extensión **BeginForm ()** y **EndForm ()**.

Están clasificados de la siguiente manera:

* URL Helpers
  + HTML Links
  + Image Links
* HTML Form Elements

**URL Helpers**

<h3>HTML Links</h3>

<strong>Link con texto: </strong>

@Html.ActionLink("Texto del link", "Name Controller")

<h3>Image Links</h3>

<strong>Link con imagen: </strong>

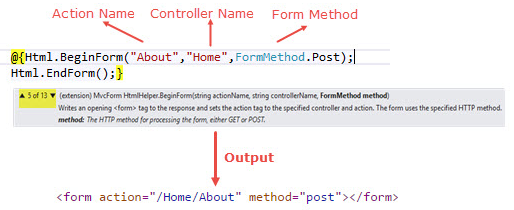
<a href="@Url.Action("Index", "Name Controller")">

<img src="/Images/img.jpg" alt="View Details" />

</a>

**HTML Form Elements**

***Ejemplo:***



@using (Html.BeginForm("InlineHTMLhelper", "HelpersController", FormMethod.Post))

{

<strong>TextBox: </strong>

@Html.TextBox("myTextBox", "", new { @class = "form-control", @placeholder = "Input text" })

<br />

<strong>TextArea: </strong>

@Html.TextArea("txtComments", "", htmlAttributes: new { @style = "width: 450px;", @placeholder = "Enter Comments here" })

<br />

<strong>CheckBox: </strong>

@Html.CheckBox("isNewlyEnrolled", true)

<br />

<strong>RadioButton: </strong>

Male: @Html.RadioButton("Gender", "Male");

Female: @Html.RadioButton("Gender", "Female");

<br />

<strong>DropDownList: </strong>

@Html.DropDownList("Country",

new List<SelectListItem>{

new SelectListItem{Text = "Monterrey", Value= "0"},

new SelectListItem{Text = "Guadalajara", Value= "1"},

new SelectListItem{Text = "Tampico", Value= "2"}

})

<br />

<strong>Password: </strong>

@Html.Password("OnlinePassword")

<br />

<strong>Hidden: </strong>

@Html.Hidden("idHidden")

<br />

<strong>Label: </strong>

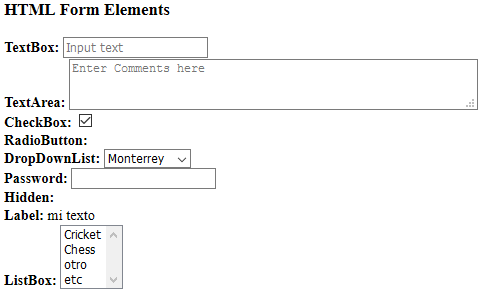
@Html.Label("Nombre", "mi texto")

<br />

<strong>ListBox: </strong>

@Html.ListBox("ListBox1", new MultiSelectList(new[] { "Cricket", "Chess", "otro", "etc" }))

}



**Strongly typed HTML helper**

Los strongly typed helper se utilizan para pasar datos del **modelo al helper** usando la sintaxis de la expresión. La clase **'HtmlHelper <TModel>'** es una subclase fuertemente tipada de la clase **'HtmlHelper'**.

Creando un **Strongly Custom Helper**

Para crear un ayudante personalizado de tipo fuerte escribimos el método de extensión para la clase **'HtmlHelper <TModel>**'.

* **Expresión:**

Mientras usamos un strongly typed helper en la vista usamos expresiones lambda como

**@ Html.TextBoxFor (m => m.Name)**

La clase Expression se utiliza para obtener la información del modelo de esta expresión lambda.

* **ModelMetaData:**

Esta clase se utiliza para obtener información del modelo con sus datos de la expresión lambda.

***Ejemplo:***

***MODELO***

namespace GuiaMVC4.Models

{

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

public Gender StudentGender { get; set; }

public bool isNewlyEnrolled { get; set; }

public string Password { get; set; }

public string Description { get; set; }

public string Gender { get; set; }

}

public enum Gender

{

Male,

Female,

Otro

}

}

***VISTA***

@using (Html.BeginForm()) {

@Html.AntiForgeryToken()

@Html.ValidationSummary(true)

<fieldset>

<legend>Student</legend>

<div class="editor-label">

@Html.LabelFor(model => model.StudentName)

</div>

<div class="editor-field">

@Html.EditorFor(model => model.StudentName)

@Html.ValidationMessageFor(model => model.StudentName)

</div>

<div class="editor-label">

@Html.LabelFor(model => model.Age)

</div>

<div class="editor-field">

@Html.EditorFor(model => model.Age)

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

</div>

<div class="editor-label">

@Html.LabelFor(model => model.isNewlyEnrolled)

</div>

<div class="editor-field">

@Html.CheckBoxFor(model => model.isNewlyEnrolled)

@Html.ValidationMessageFor(model => model.isNewlyEnrolled)

</div>

<div class="editor-label">

@Html.LabelFor(model => model.Password)

</div>

<div class="editor-field">

@Html.PasswordFor(model => model.Password)

@Html.ValidationMessageFor(model => model.Password)

</div>

<div class="editor-label">

@Html.LabelFor(model => model.Description)

</div>

<div class="editor-field">

@Html.TextAreaFor(model => model.Description)

@Html.ValidationMessageFor(model => model.Description)

</div>

<div class="editor-label">

@Html.LabelFor(model => model.Gender)

</div>

<div class="editor-field">

Hombre: @Html.RadioButtonFor(m => m.Gender, "Male")

Mujer: @Html.RadioButtonFor(m => m.Gender, "Female")

@Html.ValidationMessageFor(model => model.Gender)

</div>

<p>

<input type="submit" value="Create" />

</p>

</fieldset>

}

<div>

@Html.ActionLink("Back to List", "Index")

</div>

**Template HTML helper**

Fueron introducidos en MVC 2 y están clasificados en 2 categorías:

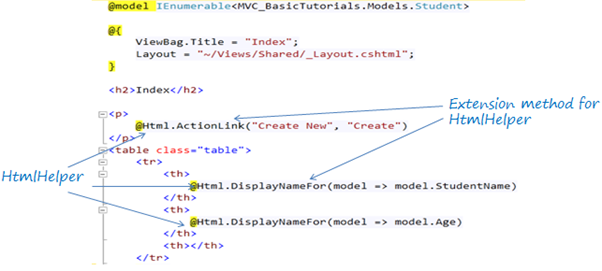
* Display Templates
* Editor Templates

**There are 3 DISPLAY templated helpers**  
**@Html.Display("EmployeeData")** - Used with a view that is not strongly typed. For example, if you have stored data in ViewData, then we can use this templated helper using the key that was used to store data in ViewData.  
**@Html.DisplayFor(model => model)** - Used with strongly typed views. If your model has properties that return complex objects, then this templated helper is very useful.  
**@Html.DisplayForModel()** - Used with strongly typed views. Walks thru each property, in the model to display the object.   
  
  
  
**Along the same lines, there are 3 EDIT templated helpers**  
@Html.Editor("EmployeeData")  
@Html.EditorFor(model => model)  
@Html.EditorForModel()

HtmlHelper

La clase **HtmlHelper** genera elementos html utilizando el objeto de clase de modelo en **razor view**. Vincula el objeto modelo a los elementos html para mostrar el valor de las propiedades del modelo en elementos html y también asigna el valor de los elementos html a las propiedades del modelo mientras se envía el formulario web.

***Ejemplo:***



@**Html** is an object of **HtmlHelper class**. (El símbolo @ es usado para acceder al objeto del lado del servidor).

Html is a property of type **HtmlHelper** included in base class of **razor view WebViewPage. ActionLink()** and **DisplayNameFor()** is extension methods included in **HtmlHelper class**.

**HtmlHelper** class generates html elements. For example:

@Html.ActionLink("Create New", "Create") genera una etiqueta

<a href="/Student/Create">Create New</a>

Hay muchos métodos de extensión para la clase **HtmlHelper**, que crea diferentes controles html, estos derivan del namespace  **System.Web.Mvc.Html**

|  |  |
| --- | --- |
| **Class** | **Description** |
| [ChildActionExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.childactionextensions(v=vs.118).aspx) | Represents support for calling child action methods and rendering the result inline in a parent view. |
| [DisplayExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.displayextensions(v=vs.118).aspx) | Represents support for rendering object values as HTML. |
| [DisplayNameExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.displaynameextensions(v=vs.118).aspx) | Provides a mechanism to get display names. |
| [DisplayTextExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.displaytextextensions(v=vs.118).aspx) | Provides a way to render object values as HTML. |
| [EditorExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.editorextensions(v=vs.118).aspx) | Represents support for the HTML **input** element in an application. |
| [EnumHelper](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.enumhelper(v=vs.118).aspx) | Provides methods for working with enumeration values and select lists. |
| [FormExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.formextensions(v=vs.118).aspx) | Represents support for HTML in an application. |
| [InputExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.inputextensions(v=vs.118).aspx) | Represents support for HTML input controls in an application. |
| [LabelExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.labelextensions(v=vs.118).aspx) | Represents support for the HTML **label** element in an ASP.NET MVC view. |
| [LinkExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.linkextensions(v=vs.118).aspx) | Represents support for HTML links in an application. |
| [MvcForm](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.mvcform(v=vs.118).aspx) | Represents an HTML **form** element in an MVC view. |
| [NameExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.nameextensions(v=vs.118).aspx) | [Gets the HTML ID and name attributes of the HtmlHelper string.](https://msdn.microsoft.com/en-us/library/system.web.mvc.htmlhelper(v=vs.118).aspx) |
| [PartialExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.partialextensions(v=vs.118).aspx) | Represents the functionality to render a partial view as an HTML-encoded string. |
| [RenderPartialExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.renderpartialextensions(v=vs.118).aspx) | Provides support for rendering a partial view. |
| [SelectExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.selectextensions(v=vs.118).aspx) | Represents support for making selections in a list. |
| [TextAreaExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.textareaextensions(v=vs.118).aspx) | Represents support for HTML **textarea** controls. |
| [ValidationExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.validationextensions(v=vs.118).aspx) | Provides support for validating the input from an HTML form. |
| [ValueExtensions](https://msdn.microsoft.com/en-us/library/system.web.mvc.html.valueextensions(v=vs.118).aspx) | Provides a mechanism to create custom HTML markup compatible with the ASP.NET MVC model binders and templates. |

En la tabla siguiente se enumeran los métodos **HtmlHelper** y el control html que cada método genera.

|  |  |  |
| --- | --- | --- |
| **HtmlHelper** | **Strogly Typed HtmlHelpers** | **Html Control** |
| Html.ActionLink |  | Anchor link |
| Html.TextBox | Html.TextBoxFor | Textbox |
| Html.TextArea | Html.TextAreaFor | TextArea |
| Html.CheckBox | Html.CheckBoxFor | Checkbox |
| Html.RadioButton | Html.RadioButtonFor | Radio button |
| Html.DropDownList | Html.DropDownListFor | Dropdown, combobox |
| Html.ListBox | Html.ListBoxFor | multi-select list box |
| Html.Hidden | Html.HiddenFor | Hidden field |
| Password | Html.PasswordFor | Password textbox |
| Html.Display | Html.DisplayFor | Html text |
| Html.Label | Html.LabelFor | Label |
| Html.Editor | Html.EditorFor | Generates Html controls based on data type of specified model property e.g. textbox for string property, numeric field for int, double or other numeric type. |

La diferencia entre llamar a los métodos **HtmlHelper** y utilizar etiquetas **html** es que el método HtmlHelper está diseñado para que **sea fácil enlazar para ver datos o datos de modelo.**

## **Data Annotations and Validation**

DataAnnotations

Consiste en agregar atributos en las propiedades con las reglas necesarias, existen un conjunto de atributos pre-definidos que son:

public class UserFilters

{

[Required(ErrorMessage = "El nombre es requerido")]

public string Name { get; set; }

[Required(ErrorMessage = "El email es requerido")]

[EmailAddress(ErrorMessage = "El email no tiene un formato correcto")]

public string Email { get; set; }

[Required(ErrorMessage = "La edad es requerida")]

[Range(14, 120, ErrorMessage = "La edad debe estar en un rango entre 14 y 120")]

public int Age { get; set; }

}

Luego de definir las reglas en nuestro modelo, en la acción del controlador ya no es necesario validar cada uno de los campos, en este caso es suficiente con llamar a ModelState.IsValid que retonar un boolean indicando si el modelo es válido o no, reescribiendo el código de la acción tenemos:

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

ASP.NET MVC Client Side validation requirements

First, here’s the list of things you need to do to enable client-side validation in an ASP.NET MVC project Make sure your client side code is loading both:

* jquery.validate.js
* jquery.validate.unobtrusive.js

Make sure your web.config has the following keys in appSettings with the follwoing values:

<appSettings>

...

<add key="ClientValidationEnabled" value="true" />

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

</appSettings>

These settings can be overridden in a controller, make sure that is not happening. For example this would turn off client side validation if executed inside a controller’s action:

public ActionResult Index()

{

HtmlHelper.ClientValidationEnabled = false;

return View();

}

The next requirement is that you use attributes from ***System.ComponentModel.DataAnnotations*** in the Model class that is used in view where you want client-side validation enabled.

For example, if we want the Email field to be a valid email, and make the password and email fields required we would create a model like this:

using System.ComponentModel.DataAnnotations;

public class LoginViewModel

{

[Required]

[EmailAddress]

public string Email { get; set; }

[Required]

public string Password { get; set; }

}

Finally, we have to use the HtmlHelpers that generate the correct markup for all of this to work, and they have to be inside a form, for example

@model LoginViewModel

<form>

@Html.LabelFor(m => m.Email)

@Html.TextBoxFor(m => m.Email)

@Html.ValidationMessageFor(m => m.Email)

@Html.LabelFor(m => m.Password)

@Html.PasswordFor(m => m.Password)

@Html.ValidationMessageFor(m => m.Password)

</form>

Getting away with using Client Side validation without a model

The last two requirements are actually optional. It is possible to take advantage of client side validation without having to create a model class and annotate it, which can be useful if you only use a couple of parameters (such as in the Login example).

If you inspect the markup that the helpers generate you’ll see that it’s actually pretty simple:

<input type="text" name="email" data-val="true" data-val-required="The email is required"/>

<span class="field-validation-valid" data-valmsg-for="email" data-valmsg-replace="true"></span>

It turns out that to enable client side validation without using the ***HtmlHelpers*** and a model you just have to add an input with ***data-val="true"***and then ***data-val-*** followed by validation method that you want to apply (e.g. ***data-val-required***), the value of which will be the error message presented to the user (e.g. ***data-val-required="This is the error message***"). This works because the MVC’s “unobtrusive validation” works by looking for inputs that are annotated with ***data-val*** attributes.

The ***data-valmsg-for‘s*** value is the ***name*** (not the ***id***) of the input it refers to, and ***data-valmsg-replace="true***" just means that the default message should be replaced, for example you could have a default message for the email field:

<span class="field-validation-valid" data-valmsg-for="email" data-valmsg-replace="true">The email must be unique</span>

This message would then be replaced by any validation error that occurs in the email field, for example “The email is required”. If ***data-valmsg-replace="false"*** then the original message will never be replaced. The only consequence of an error is that the span’s class is changed from ***field-validation-valid to field-validation-error*** (this happens irrespectively of the value of ***data-valmsg-replace="false"***).

Some validation methods have parameters, for example ***RegularExpression***. The way these work is very similar, they just need additional ***data-val-*** for their parameters. If you want to validate a text field using a regular expression for 5 to 8 digits, it would look like this:

<input type="text" name="number" id="number" data-val="true" data-val-regex="Valid input is 5 to 8 digits" data-val-regex-pattern="^\d{5,8}$"/>

If you create the markup yourself you can get away without having to create a model for your view. Using the login example from above, your controller action for handling the user logging in could simply be:

public ActionResult Login(string email, string password)

{

...

}

You’d have to make any server-side checks on the parameters yourself though.

Here is the list of the ***System.ComponentModel.DataAnnotation*** attributes you can use, and their ***data-val*** counterparts:

* Compare
  + data-val-equalto="Error message"
  + data-val-equalto-other="The name of the other field"
* CreditCard
  + data-val-creditcard="Error message"
* EmailAddress
  + data-val-email="Error message"
* MaxLength
  + data-val-maxlength="Error message"
  + data-val-maxlength-max="Maximum length (e.g. 5)"
* MinLength
  + data-val-minlength="Error message"
  + data-val-minlength-min="Minimum length (e.g. 2)"
* Range
  + data-val-range="Error message"
  + data-val-range-max="Max value"
  + data-val-range-min="Min value"
* RegularExpression
  + data-val-regex="Error message"
  + data-val-regex-pattern="The regular expression (e.g. ^[a-z]+$)"
* Required
  + data-val-required="Error message"
* StringLength
  + data-val-length="Error message"
  + data-val-length-max="Maximum number of characters"

**Archivos de configuración.**

Habilitar trace.axd en **web.config**

<system.web>

<trace enabled="true" pageOutput="false"/>

</system.web>

