

**CODES**

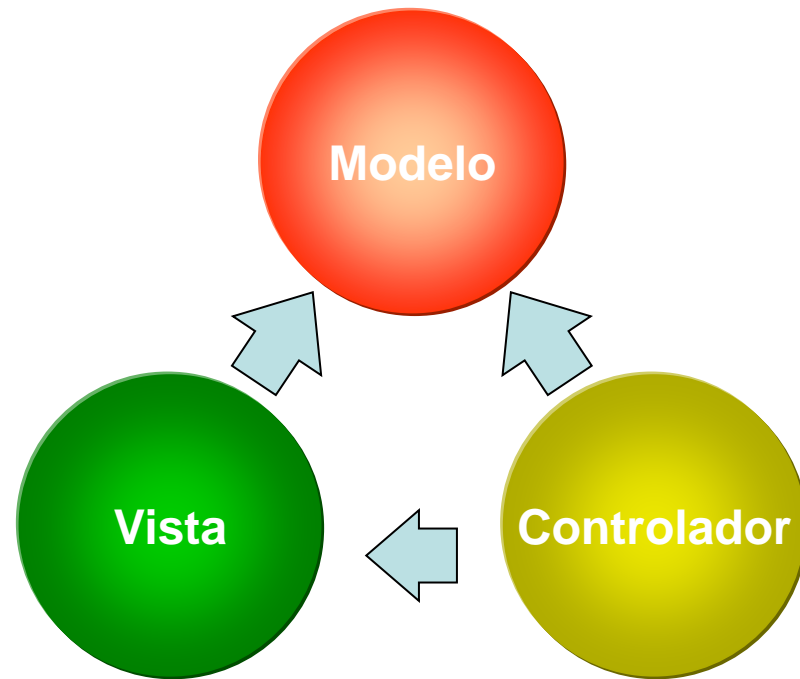
Capacitación Web – Conceptos Técnicos

El modelo Model-View-Controller (MVC) es un principio de diseño arquitectónico que separa los componentes de una aplicación web. Esta separación ofrece más control sobre las partes individuales de la aplicación, lo cual permite desarrollarlas, modificarlas y probarlas más fácilmente.

ASP.NET MVC forma parte del marco de trabajo ASP.NET. Desarrollar una aplicación ASP.NET MVC es una alternativa al desarrollo de páginas de formularios Web Forms de ASP.NET; no reemplaza el modelo de formularios Web Forms.

Fuente: [http://msdn.microsoft.com/es-es/library/gg416514\(v=vs.98\).aspx](http://msdn.microsoft.com/es-es/library/gg416514(v=vs.98).aspx)

**El Modelo MVC  
incluye los  
siguientes  
componentes:**



### 3. VENTAJAS DE UNA APLICACIÓN WEB BASADA EN MVC

- ✓ Resulta más fácil de administrar la complejidad dividiendo una aplicación en el modelo, la vista y el controlador.
- ✓ No utiliza el view state o server-based forms. Esto es ideal para quienes quieren control total sobre el comportamiento de una aplicación.
- ✓ Utiliza un patrón Front Controller que procesa las solicitudes de aplicación Web a través de un único controlador. Esto le permite diseñar una aplicación que soporta una rica infraestructura de enrutamiento.
- ✓ Proporciona mayor compatibilidad para test-driven development (TDD).

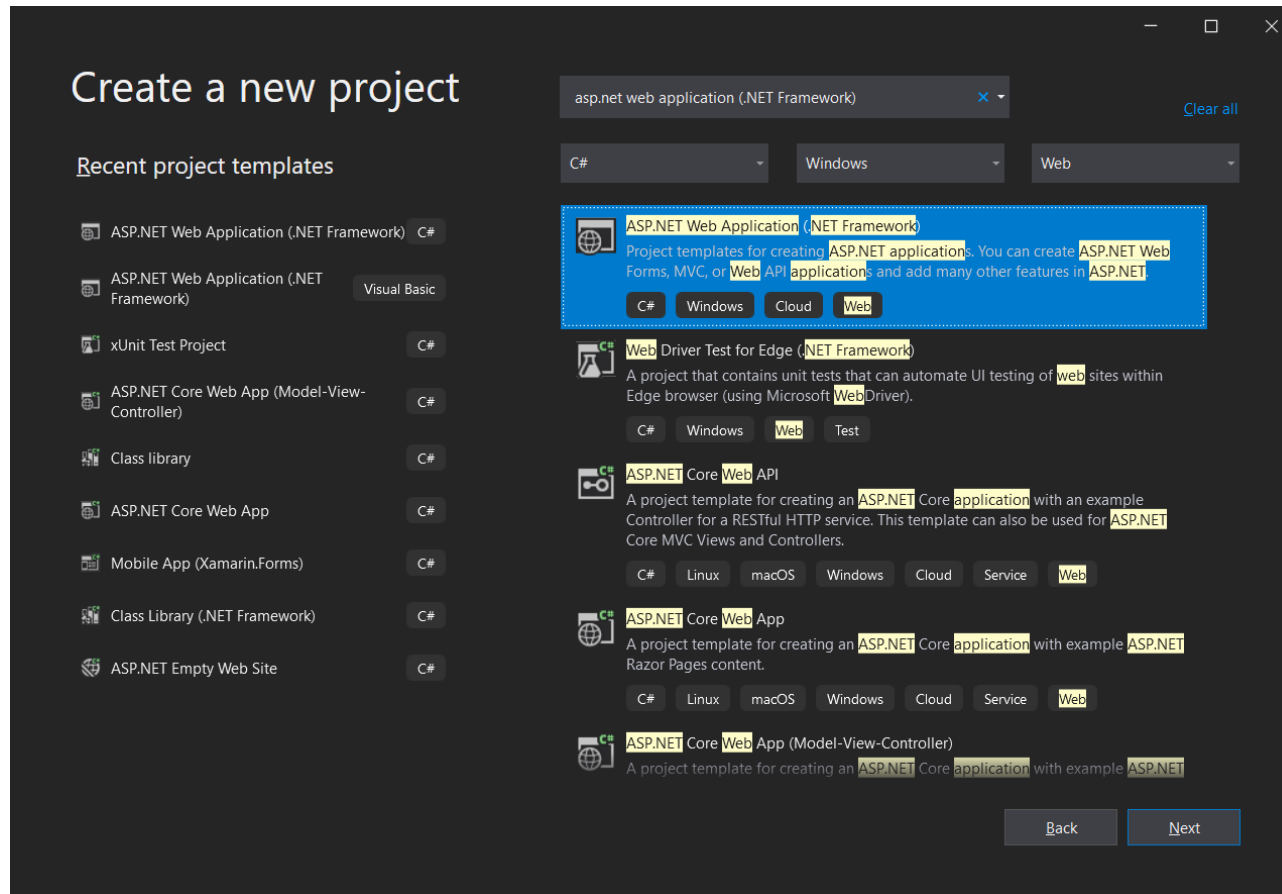
#### 4. CONSTRUYENDO UNA APLICACIÓN MVC

Para fortalecer esta presentación, y para no profundizar con aspectos técnicos del framework, construiremos una aplicación muy simple. La misma permitirá listar provincias, apoyando la creación, modificación y eliminación de las mismas.

Dispondremos de:

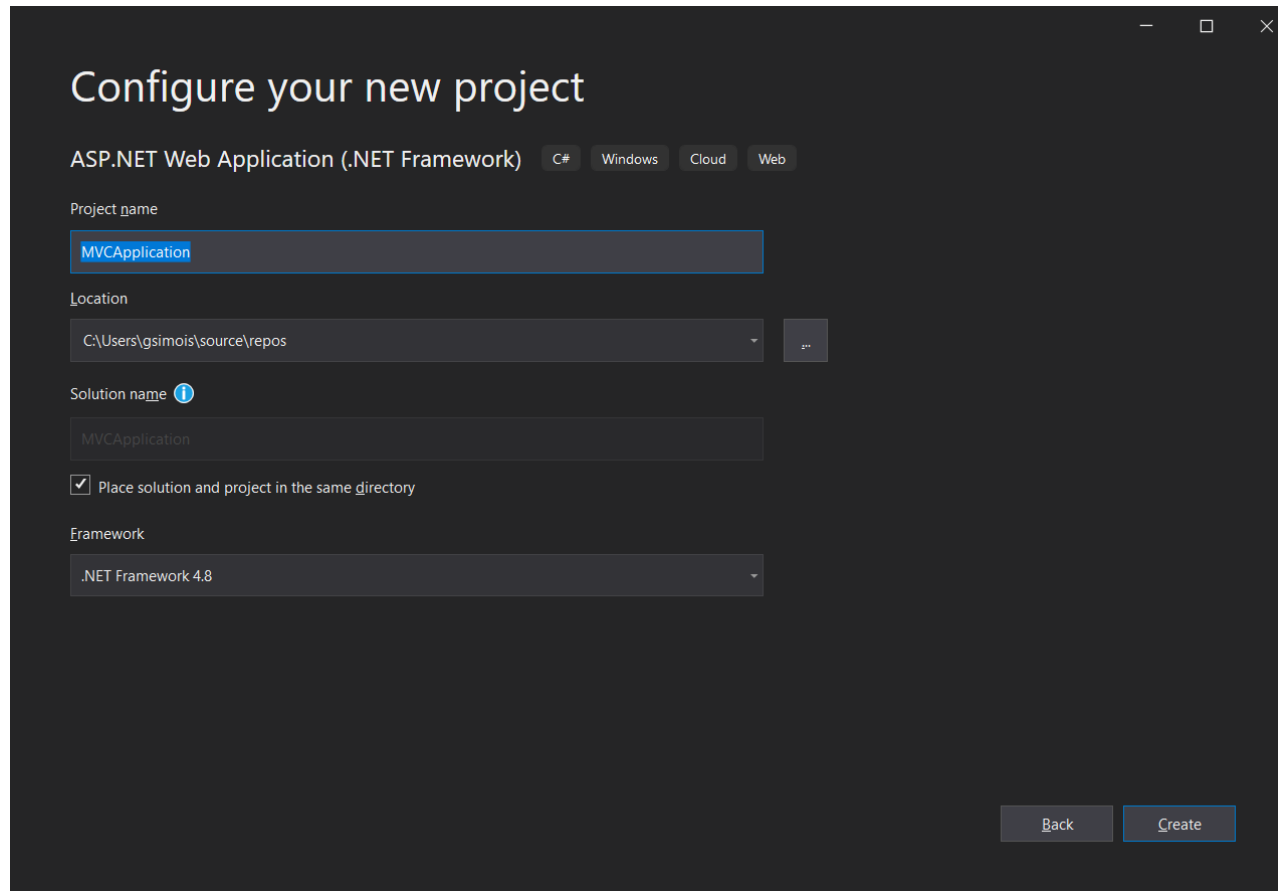
- Una página con el listado de provincias.
- Una página con el detalle de cada una de ellas.

## 5. CREANDO NUESTRA APLICACIÓN



Para crear nuestra aplicación, seleccionamos la opción de menú “Nuevo Proyecto”, Y seleccionamos el template ASP.NET Web Application (.NET Framework)

## 5. CREANDO NUESTRA APLICACIÓN



Configure your new project

ASP.NET Web Application (.NET Framework) C# Windows Cloud Web

Project name

MVCApplication

Location

C:\Users\gsimoi\source\repos

Solution name ⓘ

MVCApplication

☒ Place solution and project in the same directory

Framework


.NET Framework 4.8

Back Create


Luego especificamos el nombre de la aplicación y la versión de MVC. En este caso utilizaremos la versión 4.8

## 5. CREANDO NUESTRA APLICACIÓN


# Create a new ASP.NET Web Application

**Empty**


An empty project template for creating ASP.NET applications. This template does not have any content in it.

**Web Forms**


A project template for creating ASP.NET Web Forms applications. ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-driven model. A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access.

**MVC**

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

**Web API**

A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.

**Single Page Application**

A project template for creating rich client side JavaScript driven HTML5 applications using ASP.NET Web API. Single Page Applications provide a rich user experience which includes client-side interactions using HTML5, CSS3, and JavaScript.

**Authentication**

No Authentication  
[Change](#)

**Add folders & core references**

☐ Web Forms  
☒ MVC  
☐ Web API

**Advanced**

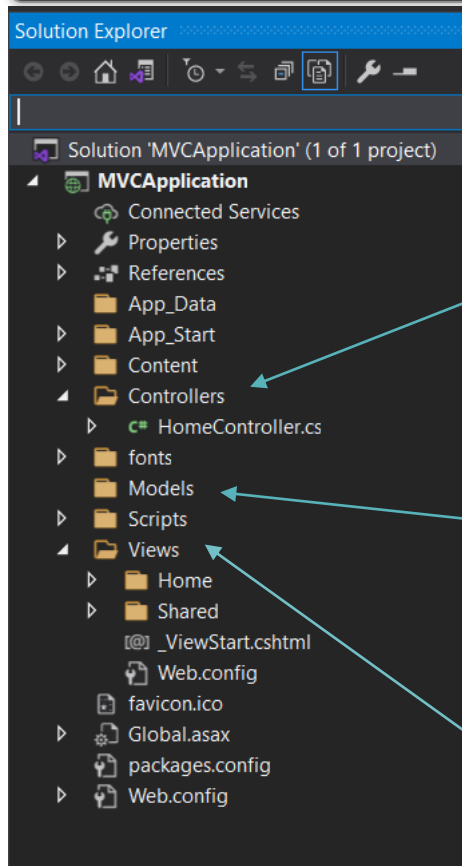
☒ Configure for HTTPS  
☐ Docker support  
(Requires [Docker Desktop](#))  
☐ Also create a project for unit tests

[Back](#) [Create](#)

Por último,  
seleccionamos el  
template MVC y  
seleccionamos la opción  
“Crear”.



## 6. ESTRUCTURA DE LAS APLICACIONES MVC



Carpeta con todos los controladores de la aplicación. Todo controlador debe respetar el nombre Xxx**Controller**. (que termine con la palabra Controller).

Carpeta con las clases que representan los distintos ViewModels

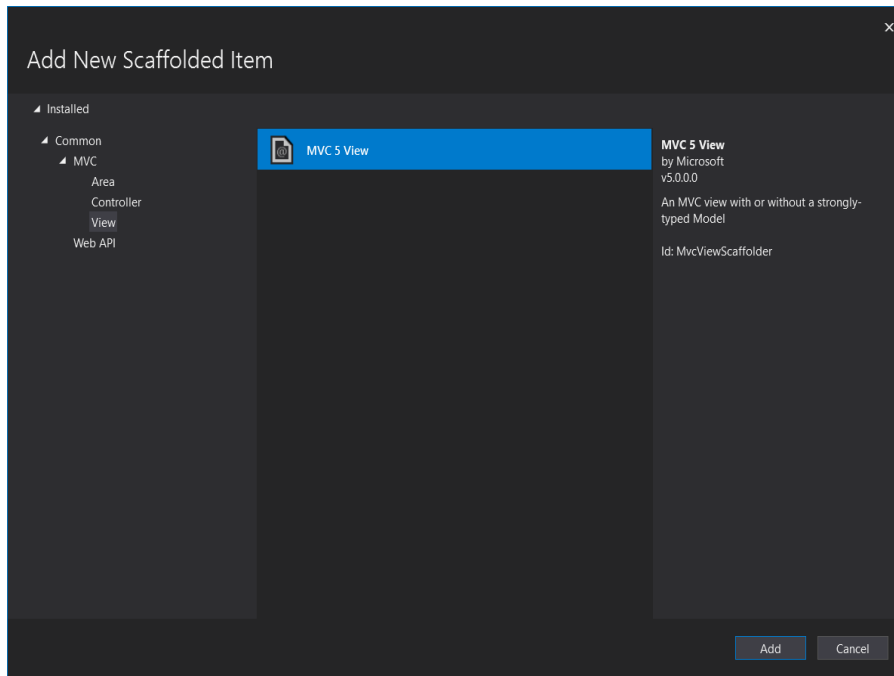
Carpeta con todas las vistas de la aplicación. La forma típica de contener las vistas para una pantalla en particular es creando una carpeta con el mismo nombre del controlador.

**En la carpeta Models del aplicativo, agregamos una clase que llamaremos “ProvinciaModel”:**

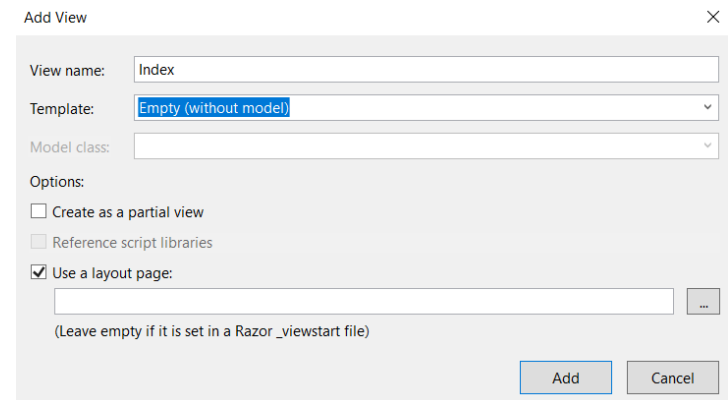
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace MVCApplication.Models
{
    public class ProvinciaModel
    {
        public int Id { get; set; }
        public string Descripcion { get; set; }
    }
}
```

**En la carpeta Views, creamos una subcarpeta que llamaremos Provincia, y dentro de ésta agregaremos una View llamada “Index”:**



**Utilizaremos Razor como el motor de vistas, y dejaremos el tilde “Use a layout or master page” activo (tomará el default layout).**

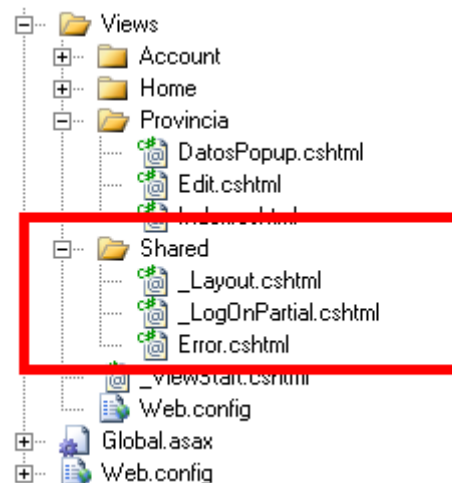


## 9. Agregando la View (II)

```
@model IList<MVCAApplication.Models.ProvinciaModel>
@{
    ViewBag.Title = "Index";
}

<h2>Index of Provincia</h2>
<table>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Id)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Descripcion)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { idProvincia=item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { idProvincia=item.Id })
        </td>
    </tr>
}
</table>
```

Hay contenido que es común a todas las páginas de nuestra aplicación (por ejemplo, el título, estilos, librerías JavaScript comunes al proyecto). Y este contenido es implementado en un único lugar en el proyecto. Por default, al crear un proyecto se crea el archivo `_Layout.cshtml` en la carpeta `/Views/Shared`.



## 11. Layout Page (II)

```
<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
  <link href="@Url.Content("~/Content/themes/base/jquery-ui.css")" rel="stylesheet" type="text/css" />
  <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript"></script>
  <script src="@Url.Content("~/Scripts/jquery-ui.js")" type="text/javascript"></script>
</head>
<body>
  <div class="page">
    <div id="header">
      <div id="title">
        <h1>My MVC Application</h1>
      </div>
      <div id="logindisplay">
        @Html.Partial("_LogOnPartial")
      </div>
      <div id="menucontainer">
        <ul id="menu">
          <li>@Html.ActionLink("Home", "Index", "Home")</li>
          <li>@Html.ActionLink("About", "About", "Home")</li>
        </ul>
      </div>
    </div>
    <div id="main">
      @RenderBody()
      <div id="footer">
      </div>
    </div>
  </div>
</body>
</html>
```

Librerías JavaScript y hojas de estilos comunes a todo el proyecto

**@RenderBody()** es la función que dibuja el contenido de una página.

## Por último, agregaremos el controller dentro de la carpeta “Controllers”.

```
namespace MVCApplication.Controllers
{
    public class ProvinciaController : Controller
    {
        //
        // GET: /Provincia/

        public ActionResult Index()
        {
            IList<ProvinciaModel> provincias = new List<ProvinciaModel>();

            provincias.Add(new ProvinciaModel() { Id = 1, Descripcion = "Buenos Aires" });
            provincias.Add(new ProvinciaModel() { Id = 2, Descripcion = "Córdoba" });
            provincias.Add(new ProvinciaModel() { Id = 3, Descripcion = "Entre Ríos" });

            return View(provincias);
        }
    }
}
```

**Nota:** La lista de provincias se pone a modo de ejemplo para la devolución de la misma en la vista “Index”.

## El espacio de nombres

System.ComponentModel.DataAnnotations dispone de varios atributos de validación que pueden ser agregados a nuestra ViewModel:

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

namespace MVCApplication.Models
{
    public class ProvinciaModel
    {
        [Required]
        public int Id { get; set; }

        [Required]
        [StringLength(20)]
        public string Descripcion { get; set; }
    }
}
```

[Required] es un atributo que obliga a la property siempre tener un valor.

[StringLength] limita el máximo de caracteres permitidos para el campo Descripcion.



Cuando especificamos los atributos de validación dentro del modelo, y éste es “bindeado” a una view en particular, el framework automáticamente (por default) realizará las validaciones sobre las properties de dicho modelo. Para poder hacer una visualización correcta de los mensajes de error, es muy común utilizar la siguiente “filosofía” para mostrar los errores en los archivos CSHTML:

```
@Html.TextBoxFor(x => x.IdTipoTrabajo)
```

```
@Html.ValidationMessageFor(x=> x.IdTipoTrabajo)
```

Para el textbox asociado a la property `IdTipoTrabajo`, se mostrará el mensaje de error “Required” cuando se haga submit del formulario y se deje este campo vacío.

## 15. Agregando la vista Edit a la solución

```
@model MVCApplication.Models.ProvinciaModel
@{
    ViewBag.Title = "Edit";
}

@using (Html.BeginForm()) {
    @Html.ValidationSummary()
    <fieldset>
        <legend>Provincia</legend>
        @Html.HiddenFor(model => model.Id)
        <div class="editor-label">
            @Html.LabelFor(model => model.Descripcion)
        </div>
        <div class="editor-field">
            @Html.TextBoxFor(model => model.Descripcion)
            @Html.ValidationMessageFor(model => model.Descripcion)
        </div>
        <p>
            <input type="submit" value="Grabar" />
        </p>
    </fieldset>
}
```

## 16. Modificando la clase ProvinciaController para contemplar la modificación

```
public ActionResult Edit(int idProvincia)
{
    IList<ProvinciaModel> provincias = ObtenerProvincias();

    //OJO! Esto se resolvería con la consulta al correspondiente
    //negocio. Es para poder ver en el ejemplo el ActionResult
    ProvinciaModel provincia =
        (from prov in provincias
         where prov.Id == idProvincia
         select prov).First();

    return View("Edit", provincia);
}

public ActionResult Edit(ProvinciaModel provincia)
{
    if (ModelState.IsValid) {
        //... Aquí va el código
        //... para almacenar los cambios

        return RedirectToAction("Index");
    }
    return View("Edit", provincia);
}
```

Para validar en el controller, disponemos de la property ModelState. Un código de ejemplo para realizar una correcta validación del modelo de vista podría ser:

```
public ActionResult Edit(ProvinciaModel provincia)
{
    if (ModelState.IsValid) {
        //... Aquí va el código
        //... para almacenar los cambios

        return RedirectToAction("Index");
    }
    return View("Edit", provincia);
}
```

**Lo siguiente que queremos probar con ASP.NET MVC es la creación de Popups invocados vía AJAX.**

**Para ello, estaremos necesitando:**

- **Dibujar el botón que invoque a la función JS que invoque al POPUP**
- **Crear la función que invoque al POPUP (en JavaScript)**
- **Crear los métodos en el Controller (ServerSide) para que devuelvan el HTML correspondiente y guarden los datos.**

**Para este ejemplo, podemos crear en el Index.cshtml el botón y el DIV que contenga el popup:**

```
<input type="button" value="Prueba View" onclick="CargarView()" />
<div id="popup"></div>
```

## Creamos el ViewModel que contendrá los datos del Popup

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace MVCApplication.Models
{
    public class DatosPopup
    {
        public bool InSitu { get; set; }
        public bool Tercerizado { get; set; }

        public IList<TipoTrabajo> TiposTrabajo { get; set; }
        public int IdTipoTrabajo { get; set; }
    }

    public class TipoTrabajo
    {
        public int Id { get; set; }
        public string Descripcion { get; set; }
    }
}
```

## Creamos la View que mostraremos en el Popup

```
@model MVCApplication.Models.DatosPopup
```

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

```
<h2>DatosPopup</h2>
```

```
<span class="ui-combobox">
```

```
    @Html.DropDownListFor(x => x.IdTipoTrabajo,  
        new SelectList(@Model.TiposTrabajo, "Id", "Descripcion"),  
        new { @Id = "ddlTiposTrabajo", @class="ui-autocomplete-input comboBox", })
```

```
    @Html.HiddenFor(x => x.IdTipoTrabajo)
```

```
</span>
```

```
<br />
```

```
@Html.CheckBoxFor(x => x.InSitu)
```

```
<br />
```

```
@Html.CheckBoxFor(x => x.Tercerizado)
```

```
@Html.TextBoxFor(x => x.IdTipoTrabajo)
```

```
@Html.ValidationMessageFor(x => x.IdTipoTrabajo)
```

## Programamos los Actions en el Controller correspondiente

```

public string CargarPopup() {
    DatosPopup datos = new DatosPopup();
    datos.InSitu = true;
    datos.Tercerizado = false;
    datos.TiposTrabajo = new List<TipoTrabajo>();
    datos.TiposTrabajo.Add(new TipoTrabajo() { Id = 1, Descripcion = "Tipo1" });
    datos.TiposTrabajo.Add(new TipoTrabajo() { Id = 2, Descripcion = "Tipo2" });
    datos.TiposTrabajo.Add(new TipoTrabajo() { Id = 3, Descripcion = "Tipo3" });

    System.IO.StringWriter sw = new System.IO.StringWriter();
    ViewEngineResult ver = ViewEngines.Engines.FindPartialView(this.ControllerContext, "DatosPopup");
    this.ViewData.Model = datos;
    ViewContext vc = new ViewContext(this.ControllerContext, ver.View, this.ViewData, this.TempData, sw);

    ver.View.Render(vc, sw);
    return sw.GetStringBuilder().ToString();
}

public string GuardarDatosPopup(DatosPopup datosPopup) {
    string respuesta = "";
    try {
        if (ModelState.IsValid) {
            respuesta = "TODO OK!!!"; //TODO: realizar las operaciones necesarias de la ViewModel
        }
    } catch (Exception ex) { respuesta = ex.Message; }
    return respuesta;
}

```



## 21. Usando Pop-ups con JQuery y Ajax

Finalmente, invocamos al Popup desde donde lo necesitemos. En este caso, lo invocamos desde otra vista mediante JS (crearemos un botón que invoque a «CargarView»)

```
<script type="text/javascript">
function CargarView() {
$.ajax({
  url: '@Url.Action("CargarPopup", "Provincia")',
  success: function (respuesta) {
    $("#popup").html(respuesta);
    $("#popup").dialog({
      modal: true,
      buttons: [{
        text: "Aceptar",
        id: "btnAceptarDetalle",
        click: function () {
          $.ajax({
            url: '@Url.Action("GuardarDatosPopup", "Provincia")',
            type: 'POST',
            data: ObtenerDatosPopup(),
            success: function (jsonResponse) {
              alert(jsonResponse);
              $("#popup").dialog("close");
            },
            error: function (e) { alert("Error: " + e); },
            dataType: 'text'
          });
        }
      }]
    });
  },
  error: function (error) { }
});
}
</script>
```

```
function ObtenerDatosPopup() {
  var datos = {
    'IdTipoTrabajo': $("#ddlTiposTrabajo").val(),
    'InSitu': $("#InSitu").attr("checked"),
    'Tercerizado': $("#Tercerizado").attr("checked")
  };
  return datos;
}
```

**El Popup se debería ver más o menos así:**

# My MVC Application

## Index of Provincia

1	Buenos Aires	<a href="#">Edit</a>   <a href="#">Delete</a>
2	Córdoba	<a href="#">Edit</a>   <a href="#">Delete</a>
3	Entre Ríos	<a href="#">Edit</a>   <a href="#">Delete</a>

VERJSON PruebaView

### DatosPopup

Tipo1 ▾

☒ ☐ 0

Aceptar

La utilización de DataAnnotations para validar las propiedades de las distintas viewModels es muy útil. Dichas DataAnnotations las encontraremos en el espacio de nombres `System.ComponentModel.DataAnnotations`.

Es posible definir atributos personalizados que nos permiten establecer restricciones como pueden ser rangos de valores correctos, longitudes máximas para una propiedad, evitar la introducción de nulos y diversas opciones más. Estas anotaciones se conocen comúnmente como restricciones.

Existen diversas formas de definir nuestras validaciones:

- Validación utilizando el atributo `[CustomValidation]`
- Validación mediante la creación de atributos personalizados

Definimos nuestra función (estática) declarando el método de validación:

```
public static class ProvinciaValidations
{
    public static ValidationResult IsValidIdProvincia(int value)
    {
        if (value > 0)
        {
            return ValidationResult.Success;
        }
        return new ValidationResult("El Id de la Provincia debe ser mayor a 0");
    }
}
```

Y utilizamos nuestra CustomValidation creada:

```
public class ProvinciaModel
{
    [Required]
    [CustomValidation(typeof(ProvinciaValidations), "IsValidIdProvincia")]
    public int Id { get; set; }

    [Required]
    [StringLength(20)]
    public string Descripcion { get; set; }
}
```

## Definimos nuestro atributo personalizado:

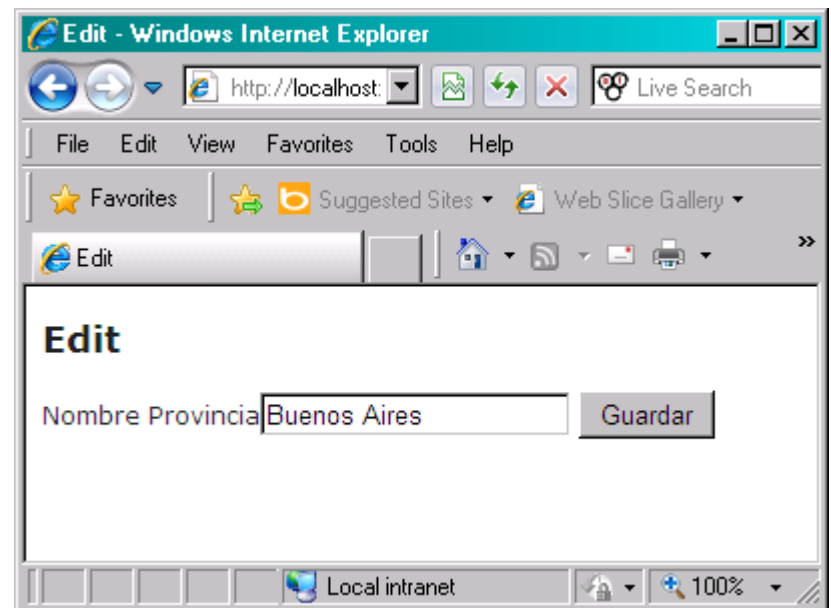
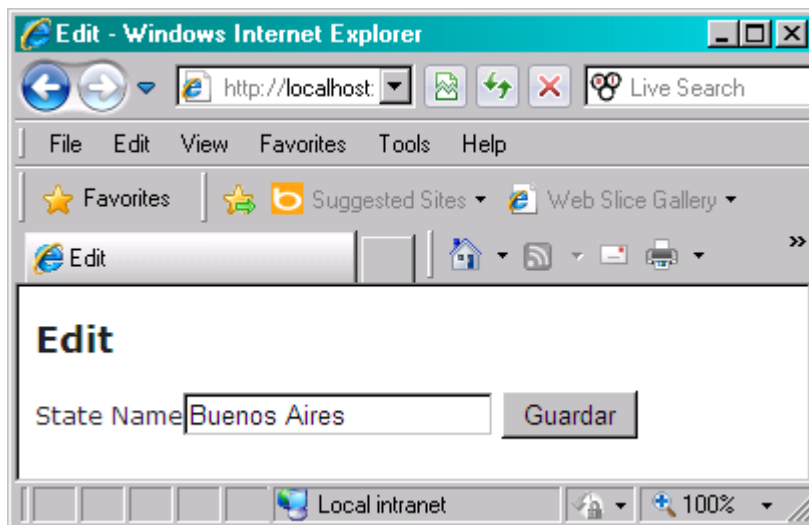
```
public class IsValidIdProvinciaAttribute
{
    public override bool IsValid(object value)
    {
        int id = Convert.ToInt32(value);
        if (id > 0)
        {
            return true;
        }
        return false;
    }
}
```

## Y utilizamos nuestro atributo creado:

```
public class ProvinciaModel
{
    [Required]
    [IsValidIdProvinciaAttribute(ErrorMessage="El Id de la provincia es inválido")]
    public int Id { get; set; }

    [Required]
    [StringLength(20)]
    public string Descripcion { get; set; }
}
```

Como sabemos, las aplicaciones Web pueden ser accedidas desde cualquier parte del mundo. Como tendremos Clientes que hablen un idioma distinto al nuestro, vamos a tener que implementar alguna estrategia para que todos los textos (excepto los que están almacenados en la BBDD) aparezcan traducidos al idioma correspondiente.



Primero que nada, nos conviene definir un idioma “default” para nuestra aplicación. Esto lo definimos en el archivo de configuración (web.config), dentro de la sección <system.web>

```
<globalization fileEncoding="utf-8" requestEncoding="utf-8"  
responseEncoding="utf-8" culture="es-ES" uiCulture="es-ES" />
```

Luego, tendremos que agregar un archivo de Recursos de Idioma. Este concepto es muy común entre las aplicaciones Web. Para agregar un archivo de recursos, dentro del proyecto y la carpeta que deseemos, desde el menú contextual, elegimos *Agregar Nuevo => Resource File*. A este archivo de recursos lo podemos llamar “Recursos”, o “Global”, o “Resources” (haciendo alusión a que contiene los recursos de idioma).

Pero para nosotros, ¿qué son los recursos de idioma? Son simplemente clases que contienen pares de nombre-descripción donde “nombre” es el ID del recurso, y “descripción” es el texto que se mostrará por pantalla cuando se haga referencia a esta entrada. Recordemos siempre que el alcance del recurso sea público (ya que accedemos a él desde la clase que lo representa).

	Name ▲	Value	Comment
►	Nombre	Nombre	
	NombreProvincia	Nombre Provincia	
*			



En nuestra aplicación definiremos tanto archivos de recursos como idiomas que querremos manejar. Para nuestro ejemplo definiremos dos: `GlobalResources.resx` y `GlobalResources.en-US.resx`. Nótese en que los nombres sólo varía parte de la extensión del mismo, indicando a qué lenguaje representa. El "default" no tiene este indicador.

	Name ▲	Value	Comment
▶	Nombre	Nombre	
	NombreProvincia	Nombre Provincia	
*			

**GlobalResources.resx**

	Name ▲	Value	Comment
▶	Nombre	Name	
	NombreProvincia	State Name	
*			

**GlobalResources.en-US.resx**

Para utilizar nuestros archivos de recursos en nuestra aplicación web utilizando MVC, utilizaremos la DataAnnotation Display en el modelo de la vista, de la siguiente manera:

```
public class ProvinciaModel
{
    [Required]
    public int Id { get; set; }

    [Display(Name = "NombreProvincia", ResourceType=typeof(Resources.GlobalResources))]
    [StringLength(5)]
    public string Descripcion { get; set; }
}
```

Y en la vista utilizaremos el HTML Helper LabelFor

```
@{
    ViewBag.Title = "Edit";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
@model MvcApplication4.Models.ProvinciaModel

<h2>Edit</h2>
@using (Html.BeginForm())
{
    @Html.HiddenFor(model => model.Id)

    @Html.LabelFor(model => model.Descripcion)
    @Html.TextBoxFor(model => model.Descripcion)
    @Html.ValidationMessageFor(model => model.Descripcion)
    <input type="submit" value="Guardar" />
}
```

Si queremos cambiar el idioma, tenemos dos opciones:

- Cambiar en el web.config la opción de "globalización".
- Por código, en tiempo de ejecución, forzar el cambio de la cultura:

```
System.Globalization.CultureInfo cultura = new System.Globalization.CultureInfo("en-US");  
System.Threading.Thread.CurrentThread.CurrentCulture = cultura;  
System.Threading.Thread.CurrentThread.CurrentUICulture = cultura;
```

# ÍNDICE

1. INTRODUCCIÓN MVC3

2. RAZOR ENGINE

## **Introducción**

El MVC de ASP.NET tiene soporte para lo que se denomina “view engines”, los cuales son librerías que implementan diferentes opciones de sintaxis, las cuales son las encargadas de “dibujar” el HTML que se entrega finalmente en el Response.

Hay varios motores que pueden utilizarse (el default es el de ASPX). Al momento de escribir esta presentación, nuestra opción fue utilizar Razor, por los siguientes motivos:

- La sintaxis de Razor es simple, por lo que nos fue fácil de aprender.
- Visual Studio incluye IntelliSense y cambios de color para la sintaxis de Razor.

Razor nos permite crear nuestras páginas escribiendo código HTML, y sobre ellas podemos agregar código del lado del servidor. Para agregar código, utilizamos el carácter @ (Razor no requiere que se cierre el bloque de código, a diferencia del <% %> de ASPX).  
Veamos un ejemplo:

```
<h1>Razor Example</h1>

<h3>
    Hello @name, the year is @DateTime.Now.Year
</h3>

<p>
    Checkout <a href="/Products/Details/@productId">this product</a>
</p>
```

El poder escribir código de esta manera sin la necesidad de agregar los marcadores de apertura y cierre de código (como en ASP), hace que escribir código del lado del servidor sea mucho más fluido:

```
<h1>Razor Example</h1>

<h3>
    Hello @name, the year is @DateTime.Now.Year
</h3>

<p>
    Checkout <a href="/Products/Details/@productId">this product</a>
</p>
```

```
<ul id="products">
    @foreach(var p in products) {
        <li>@p.Name ($@p.Price)</li>
    }
</ul>
```

```
@{
    int number = 1;
    string message = "Number is" + number;
}

<p>Your Message: @message</p>
```

```
@if(products.Count == 0) {
    <p>Sorry - no products in this category</p>
} else {
    <p>We have a products for you!</p>
}
```

## **HTML Helpers**

El MVC de ASP.NET incluye el concepto de “HTML Helpers”, los cuales son métodos que pueden ser invocados en páginas, los cuales generan HTML. Ejemplos:

```
<fieldset>
  <legend>Edit Product</legend>

  <div>
    @Html.LabelFor(m => m.ProductID)
  </div>
  <div>
    @Html.TextBoxFor(m => m.ProductID)
    @Html.ValidationMessageFor(m => m.ProductID)
  </div>
</fieldset>
```



## **HTML Helpers**

El MVC de ASP.NET incluye los siguientes "HTML Helpers":

Helper	HTML Element
Html.CheckBox	<input type="checkbox" />
Html.DropDownList	<select></select>
Html.Hidden	<input type="hidden" />
Html.Label	<label for="" />
Html.ListBox	<select></select> or <select multiple></select>
Html.Password	<input type="password" />
Html.Radio	<input type="radio" />
Html.TextArea	<textarea></textarea>
Html.TextBox	<input type="text" />

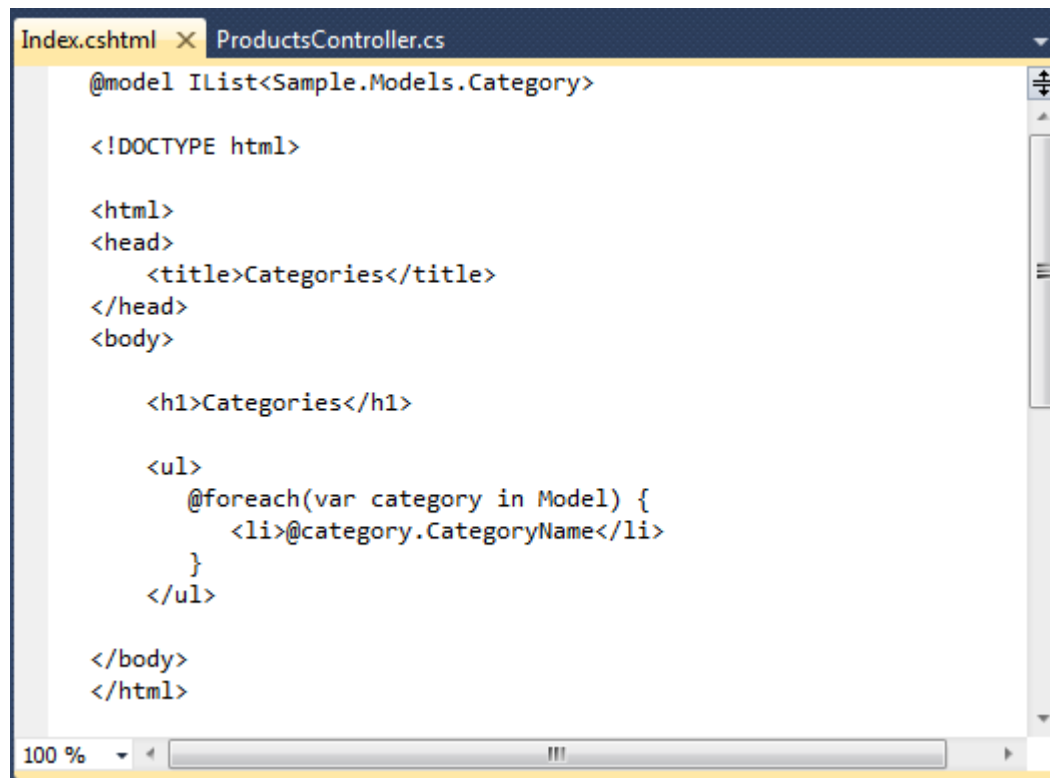
## **HTML Helpers**

Además, Razor agrega algunos “HTML Helpers” más. Ejemplos:

- WebGrid. Dibuja una grilla de datos, con funcionalidades de paginación y ordenado.
- WebImage. Dibuja una imagen.
- WebMail. Envía un mail.

## Directiva @model

Como vimos en el ejemplo inicial de esta PPT, esta directiva provee una forma de referenciar un modelo tipado desde vistas. Es una forma sencilla de acceder a los datos del "Modelo" previamente cargado en el controller.

A screenshot of a code editor window with two tabs: 'Index.cshtml' (active) and 'ProductsController.cs'. The 'Index.cshtml' tab contains the following code:

```
@model IList<Sample.Models.Category>

<!DOCTYPE html>

<html>
<head>
    <title>Categories</title>
</head>
<body>

    <h1>Categories</h1>

    <ul>
        @foreach (var category in Model) {
            <li>@category.CategoryName</li>
        }
    </ul>

</body>
</html>
```

The editor has a dark blue title bar, a light gray sidebar on the right, and a status bar at the bottom showing '100 %' and a scroll bar.