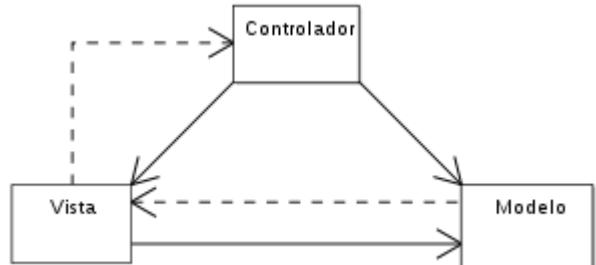


Modelo-vista-controlador

Modelo-vista-controlador (MVC) es un patrón de arquitectura de software, que separa los datos y principalmente lo que es la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el **modelo, la vista y el controlador**; es decir: por un lado define componentes para la representación de la información y, por otro lado, para la interacción del usuario.^{1 2} Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.^{3 4}



Un diagrama sencillo que muestra la relación entre el modelo, la vista y el controlador. Nota: las líneas sólidas indican una asociación directa, y principalmente las punteadas una indirecta (por ejemplo, patrón Observer).

Historia

El patrón MVC fue una de las primeras ideas en el campo de las interfaces gráficas de usuario y uno de los primeros trabajos en describir e implementar aplicaciones software en términos de sus diferentes funciones.⁵

MVC fue introducido por Trygve Reenskaug (web personal (<http://heim.ifi.uio.no/~trygver>)) en Smalltalk-76 durante su visita a Xerox Parc^{6 7} en los años 70, seguidamente, en los años 80, Jim Althoff y otros implementaron una versión de MVC para la biblioteca de clases de Smalltalk-80.⁸ Solo más tarde, en 1988, MVC se expresó como un concepto general en un artículo⁹ sobre Smalltalk-80.

En esta primera definición de MVC el controlador se definía como «el módulo que se ocupa de la entrada» (de forma similar a como la vista «se ocupa de la salida»). Esta definición no tiene cabida en las aplicaciones modernas en las que esta funcionalidad es asumida por una combinación de la 'vista' y algún framework moderno para desarrollo. El 'controlador', en las aplicaciones modernas de la década de 2000, es un módulo o una sección intermedia de código, que hace de intermediario de la comunicación entre el 'modelo' y la 'vista', y unifica la validación (utilizando llamadas directas o el «observer» para desacoplar el 'modelo' de la 'vista' en el 'modelo' activo¹⁰).

Algunos aspectos del patrón MVC han evolucionado dando lugar a ciertas variantes del concepto original, ya que «las partes del MVC clásico realmente no tienen sentido para los clientes actuales»:¹¹

- HMVC (MVC Jerárquico)
- MVA (Modelo-Vista-Adaptador)
- MVP (Modelo-Vista-Presentador)
- MVVM (Modelo-Vista Vista-Modelo)

- ... y otros que han adaptado MVC a diferentes contextos.

Descripción del patrón

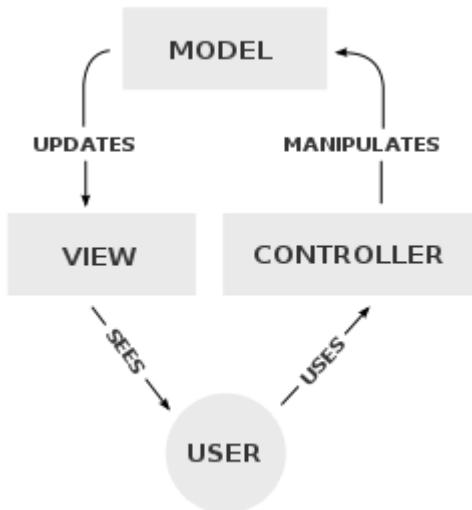
De manera genérica, los componentes de MVC se podrían definir como sigue:

- **El Modelo:** Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (*lógica de negocio*). Envía a la 'vista' aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'.¹²
- **El Controlador:** Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos). También puede enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta el 'modelo' (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo' (véase **Middleware**).
- **La Vista:** Presenta el 'modelo' (información y *lógica de negocio*) en un formato adecuado para interactuar (usualmente la interfaz de usuario), por tanto requiere de dicho 'modelo' la información que debe representar como salida.

Interacción de los componentes

Aunque se pueden encontrar diferentes implementaciones de MVC, el flujo de control que se sigue generalmente es el siguiente:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.)
2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler) o callback.
3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, se podría utilizar el patrón Observador para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio. Un objeto vista puede registrarse con el modelo y esperar a los cambios, pero aun así el modelo en sí mismo sigue sin saber nada de la vista. Este uso del patrón Observador no es posible en las aplicaciones Web puesto que las clases de la vista están desconectadas del modelo y del controlador. En general el controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice. *Nota: En algunas implementaciones la vista no tiene*



Una típica colaboración entre los componentes de un MVC

acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista. Por ejemplo en el MVC usado por Apple en su framework Cocoa. Suele citarse como **Modelo-Interface-Control**, una variación del MVC más puro

5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente...

MVC y bases de datos

Muchos sistemas [informáticos] utilizan un sistema de gestión de base de datos el cual gestiona los datos que debe utilizar la aplicación; en líneas generales del **MVC** dicha gestión corresponde al modelo. La unión entre *capa de presentación* y *capa de negocio* conocido en el paradigma de la Programación por capas representaría la integración entre la **Vista** y su correspondiente **Controlador** de eventos y acceso a datos, MVC no pretende discriminar entre capa de negocio y capa de presentación pero sí pretende separar la capa *visual gráfica* de su correspondiente *programación y acceso a datos*, algo que mejora el desarrollo y mantenimiento de la *Vista* y el *Controlador* en paralelo, ya que ambos cumplen ciclos de vida muy distintos entre sí.

Uso en aplicaciones Web

Aunque originalmente MVC fue desarrollado para aplicaciones de escritorio, ha sido ampliamente adaptado como arquitectura para diseñar e implementar aplicaciones web en los principales lenguajes de programación. Se han desarrollado multitud de frameworks, comerciales y no comerciales, que implementan este patrón (ver apartado siguiente "*Frameworks MVC*"); estos frameworks se diferencian básicamente en la interpretación de como las funciones MVC se dividen entre cliente y servidor.¹³

Los primeros frameworks MVC para desarrollo web planteaban un enfoque de cliente ligero en el que casi todas las funciones, tanto de la vista, el modelo y el controlador recaían en el servidor. En este enfoque, el cliente manda la petición de cualquier hiperenlace o formulario al controlador y después recibe de la vista una página completa y actualizada (u otro documento); tanto el modelo como el controlador (y buena parte de la vista) están completamente alojados en el servidor. Como

las tecnologías web han madurado, ahora existen frameworks como JavaScriptMVC, Backbone o jQuery¹⁴ que permiten que ciertos componentes MVC se ejecuten parcial o totalmente en el cliente (véase AJAX).

Frameworks MVC

Lenguaje	Licencia	Nombre
.NET	Castle Project (http://www.castleproject.org/index.html)	MonoRail (http://www.castleproject.org/projects/monorail/)
.NET	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	Spring.NET (http://www.springframework.net/)
.NET	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	Maverick.NET (https://mavnet.sourceforge.net/)
.NET	MS-PL (http://www.opensource.org/licenses/ms-pl.html)	ASP.NET MVC (https://www.asp.net/mvc)
.NET	Microsoft Patterns & Practices (http://msdn.microsoft.com/practices/)	User Interface Process (UIP) Application Block (https://web.archive.org/web/20070219022606/http://msdn.microsoft.com/practices/compcat/default.aspx?pull=%2Flibrary%2Fen-us%2Fdnpag%2Fhtml%2Fuiipab.asp)
C++	BSD license (https://web.archive.org/web/20170521001531/http://treefrogframework.github.io/treefrog-framework/user-guide/en/introduction/)	treefrog (http://www.treefrogframework.org)
Delphi/ Object Pascal	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	Delphi MVC Framework (https://github.com/danieleteti/delphimvcframework)
Delphi/ Object Pascal	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	mORMot Framework (https://synopse.info/foosil/wiki/Synopse+OpenSource)
Objective C	Apple (http://developer.apple.com)	Cocoa
Ruby	MIT (http://opensource.org/licenses/mit-license.php)	Ruby on Rails (http://rubyonrails.org/)
Ruby	MIT (http://opensource.org/licenses/mit-license.php)	Merb
Ruby	MIT (http://opensource.org/licenses/mit-license.php)	Ramaze
Ruby	MIT (http://opensource.org/licenses/mit-license.php)	Rhodes
Java	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	Grails
Java	GPL (http://www.opensource.org/licenses/gpl-licensing.php)	Interface Java Objects (http://nt.tusoporte.es/IjoProject) (enlace roto disponible en Internet Archive; véase el historial (https://web.archive.org/web/*http://nt.tusoporte.es/IjoProject), la primera versión (https://web.archive.org/web/1/http://nt.tusoporte.es/IjoProject) y la última (https://web.archive.org/web/2/http://nt.tusoporte.es/IjoProject)).
Java	LGPL (http://www.martincordova.com)	Framework Dinámica
Java	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	Struts
Java	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	Brutos framework
Java	Apache (http://beehive.apache.org/)	Beehive

Java	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	Spring
Java	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	Tapestry (http://tapestry.apache.org/)
Java	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	Aurora (https://web.archive.org/web/20131230132142/http://auroramvc.org/)
Java	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	JavaServerFaces (http://java.sun.com/javaee/javaserverfaces/)
Java	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	PrimeFaces (https://www.primefaces.org/#primefaces)
Java	Apache (http://www.apache.org/licenses/LICENSE-2.0.html)	Vaadin (https://web.archive.org/web/20170511162553/https://vaadin.com/home/)
JavaScript	GPLv3 (http://www.gnu.org/copyleft/gpl.html)	Sails.JS (https://web.archive.org/web/20161118215029/http://sailsjs.org/)
JavaScript	GPLv3 (http://www.gnu.org/copyleft/gpl.html)	ExtJS 4 (http://www.sencha.com/products/extjs/)
JavaScript	MIT (https://github.com/angular/angular.js/blob/master/LICENSE)	AngularJS (https://angularjs.org/)
JavaScript	MIT (https://github.com/kamilmysliwiec/nest/blob/master/LICENSE)	Nest (https://github.com/kamilmysliwiec/nest)
Perl	GPL (http://www.opensource.org/licenses/gpl-licence.php)	Mojolicious (http://mojolicio.us/)
Perl	GPL (http://www.opensource.org/licenses/gpl-licence.php)	Catalyst (http://www.catalystframework.org/)
Perl	GPL (http://www.opensource.org/licenses/gpl-licence.php)	CGI::Application (https://web.archive.org/web/20071229000621/http://cgiapp.erlbaum.net/)
Perl	GPL (http://www.opensource.org/licenses/gpl-licence.php)	Gantry Framework (https://web.archive.org/web/20071229000533/http://www.usegantry.org/)
Perl	GPL (http://www.opensource.org/licenses/gpl-licence.php)	Jifty (http://jifty.org/view/HomePage)
Perl	GPL (http://www.opensource.org/licenses/gpl-licence.php)	Maypole (https://web.archive.org/web/20091117133112/http://maypole.perl.org/)
Perl	GPL (http://www.opensource.org/licenses/gpl-licence.php)	OpenInteract2 (http://www.openinteract.org/)
Perl	Comercial	PageKit (http://pagekit.org/)
Perl	GPL (http://www.opensource.org/licenses/gpl-licence.php)	Cyclone 3 (https://web.archive.org/web/20071215103304/http://www.cyclone3.org/home)
Perl	GPL (http://www.opensource.org/licenses/gpl-licence.php)	CGI::Builder (http://search.cpan.org/perldoc?CGI::Builder)
PHP	GPL (http://www.opensource.org/licenses/gpl-licence.php)	BitPHP (https://github.com/bitphp)
PHP	BSD (https://github.com/phalcon/cphalcon/blob/master/LICENSE.txt)	phalcon (https://phalcon.io/en-us)
PHP	MIT (http://opensource.org/licenses/mit-license.php)	Laravel (http://laravel.com/)
PHP	GPL (http://www.opensource.org/licenses/gpl-licence.php)	Self Framework (php5, MVC, ORM, Templates, I18N, Múltiples DB) (https://web.archive.org/web/20111020013226/http://sites.google.com/site/phpframeworkpoo/download-framework)

PHP	LGPL (http://www.opensource.org/licenses/lgpl-licensed.php)	ZanPHP (https://web.archive.org/web/20110926095451/http://www.zanphp.com/)
PHP	LGPL (http://www.opensource.org/licenses/lgpl-licensed.php)	Tlalokes (https://web.archive.org/web/20090415200934/http://tlalokes.org/)
PHP	GPL (http://www.siaempresarial.com/mvc/mvc-licensed.php) (enlace roto disponible en Internet Archive; véase el historial (https://web.archive.org/web/*http://www.siaempresarial.com/mvc/mvc-license.php), la primera versión (https://web.archive.org/web/1/http://www.siaempresarial.com/mvc/mvc-licensed.php) y la última (https://web.archive.org/web/2/http://www.siaempresarial.com/mvc/mvc-license.php)).	SiaMVC (http://siaempresarial.com/siamvc/) (enlace roto disponible en Internet Archive; véase el historial (https://web.archive.org/web/*http://siaempresarial.com/siamvc/), la primera versión (https://web.archive.org/web/1/http://siaempresarial.com/siamvc) y la última (https://web.archive.org/web/2/http://siaempresarial.com/siamvc)).
PHP	LGPL (http://www.opensource.org/licenses/lgpl-licensed.php)	Agavi (https://web.archive.org/web/20080318151307/http://www.agavi.org/)
PHP	BSD (http://www.opensource.org/licensesbsd-licensed.php)	Zend Framework (http://framework zend.com), proyecto continuado como Laminas Framework (https://getlaminas.org/)
PHP	MIT (http://www.opensource.org/licenses/mit-license.php)	CakePHP (http://www.cakephp.org/)
PHP	GNU/GPL (http://www.opensource.org/licenses/gpl-licensed.php)	KumbiaPHP (http://www.kumbiaphp.com/)
PHP	MIT (http://www.symfony-project.org/license)	Symfony (http://www.symfony.com/)
PHP	MIT (http://www.symfony-project.com/licenses/mit-license.php) (enlace roto disponible en Internet Archive; véase el historial (https://web.archive.org/web/*http://www.symfony-project.com/licenses/mit-license.php), la primera versión (https://web.archive.org/web/1/http://www.symfony-project.com/licenses/mit-license.php) y la última (https://web.archive.org/web/2/http://www.symfony-project.com/licenses/mit-license.php)).	QCodo (http://qcodo.com/)
PHP	GNU/GPL (http://www.opensource.org/licenses/gpl-licensed.php)	CodeIgniter (http://codeigniter.com/)
PHP	GNU/GPL (http://www.opensource.org/licenses/gpl-licensed.php)	Polka-PHP (https://github.com/rotela/polka-php)
PHP	BSD (http://kohanaframework.org/)	Kohana (http://kohanaframework.org/)
PHP	MPL 1.1 (https://web.archive.org/web/20090103001154/http://php4e.codeman.cl/index.php?SECCION=msg&COD=59&MODULE=home)	PHP4ECore (https://web.archive.org/web/20080417032603/http://php4e.codeman.cl/)
PHP	GNU (http://www.practico.org/)	Practico (http://www.practico.org/)
PHP	GNU (https://web.archive.org/web/20100216021409/http://flavorphp.com/)	FlavorPHP (https://web.archive.org/web/20100216021409/http://flavorphp.com/)
PHP	Apache 2.0 (http://www.apache.org/licenses/LICENSE-2.0)	Yupp PHP Framework (https://code.google.com/p/yupp/)
PHP	BSD (http://www.yiiframework.com/license)	Yii PHP Framework (http://www.yiiframework.com/)
PHP	GPL (http://www.opensource.org/licenses/gpl-licensed.php)	Osezno PHP Framework
PHP	MIT (https://github.com/alrik11es/sPHPf/blob/master/license.txt) (enlace roto disponible en Internet Archive; véase el historial (https://web.archive.org/web/*https://github.com/alrik11es/sPHPf/blob/master/license.txt), la primera versión (https://web.archive.org/web/1/https://github.com/alrik11es/sPHPf/blob/master/license.txt) y la última (https://web.archive.org/web/2/https://github.com/alrik11es/sPHPf/blob/master/license.txt)).	(sPHPf) Simple PHP Framework (https://web.archive.org/web/20120111161846/http://sphpf.coldstarstudios.com/)

PHP	GNU/GPL (http://www.gvpontis.gva.es/cast/gvhidra-herramienta/gvhidra-que-es-gvhidra/) (enlace roto disponible en Internet Archive; véase el historial (https://web.archive.org/web/*/http://www.gvpontis.gva.es/cast/gvhidra-herramienta/gvhidra-que-es-gvhidra/), la primera versión (https://web.archive.org/web/1/http://www.gvpontis.gva.es/cast/gvhidra-herramienta/gvhidra-que-es-gvhidra/) y la última (https://web.archive.org/web/2/http://www.gvpontis.gva.es/cast/gvhidra-herramienta/gvhidra-que-es-gvhidra/)).	gvHidra (http://www.gvhidra.org) (enlace roto disponible en Internet Archive; véase el historial (https://web.archive.org/web/*/http://www.gvhidra.org/), la primera versión (https://web.archive.org/web/1/http://www.gvhidra.org/) y la última (https://web.archive.org/web/2/http://www.gvhidra.org/)).
Python	ZPL (https://web.archive.org/web/20060424193351/http://www.zope.org/Resources/License/)	Zope3 (https://web.archive.org/web/20070105081737/http://wiki.zope.org/zope3/MVC)
Python	Varias (http://docs.turbogears.org/1.0/License)	Turbogears (http://www.turbogears.org)
Python	GPL (http://web2py.com/examples/default/license)	Web2py (http://web2py.com)
Python	BSD (https://web.archive.org/web/20090525033300/http://pylonshq.com/project/pylonshq/browser/LICENSE)	Pylons (https://web.archive.org/web/20061205035643/http://pylonshq.com)
Python	BSD (https://code.djangoproject.com/browser/django/trunk/LICENSE)	Django
AS3	Adobe Open Source (https://web.archive.org/web/20080826010704/http://opensource.adobe.com/wiki/display/cairngorm/Cairngorm)	Cairngorm (https://web.archive.org/web/20080826010704/http://opensource.adobe.com/wiki/display/cairngorm/Cairngorm)
AS3 y Flex	MIT License (http://www.opensource.org/licenses/mit-license.php)	CycleFramework (https://code.google.com/p/cycleframework)

Véase también

- [Ingeniería Técnica en Informática de Gestión](#)
- [Ingeniería de Software](#)

Referencias

1. "More deeply, the framework exists to separate the representation of information from user interaction." [The DCI Architecture: A New Vision of Object-Oriented Programming](#) (http://www.artima.com/articles/dci_vision.html) Archivado (https://web.archive.org/web/20170929001352/http://www.artima.com/articles/dci_vision.html) el 29 de septiembre de 2017 en [Wayback Machine](#). - Trygve Reenskaug and James Coplien - March 20, 2009.
2. "... the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object." [Applications Programming in Smalltalk-80\(TM\):How to use Model-View-Controller \(MVC\)](#) (<https://web.archive.org/web/20120429161935/http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>).
3. «Simple Example of MVC (Model View Controller) Design Pattern for Abstraction» (<http://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design>).
4. «Best MVC Practices» (<http://www.yiiframework.com/doc/guide/1.1/en/basics.best-practices>).
5. <http://c2.com/cgi/wiki?ModelViewControllerHistory> Historia del Modelo-Vista-Controlador
6. Notes and Historical documents (<http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>) from Trygve Reenskaug, inventor of MVC.
7. "A note on DynaBook requirements", Trygve Reenskaug, 22 March 1979, [SysReq.pdf](#) (<http://folk.uio.no/trygver/1979/sysreq/SysReq.pdf>).
8. How to use Model-View-Controller (MVC) (<https://web.archive.org/web/20090801040629/http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>)
9. Krasner, Glenn E.; Stephen T. Pope (Aug/Sep de 1988). «A cookbook for using the model-view controller user interface paradigm in Smalltalk-80» (<http://dl.acm.org/citation.cfm?id=50757.50759>). [The JOT](#) (SIGS Publications). Also published as "A Description of the Model-View-

MVC

[Área personal](#) / Mis cursos / [MVC01](#) / [Introducción y conceptos](#) / [Introducción y conceptos #1](#)

Introducción y conceptos #1

Programar utilizando **MVC** consiste en separar la aplicación en tres partes principales.

El **modelo** representa los **datos de la aplicación**, la **vista** hace una **presentación del modelo de datos**, y el **controlador** **maneja y enruta las peticiones [requests]** hechas por los usuarios.

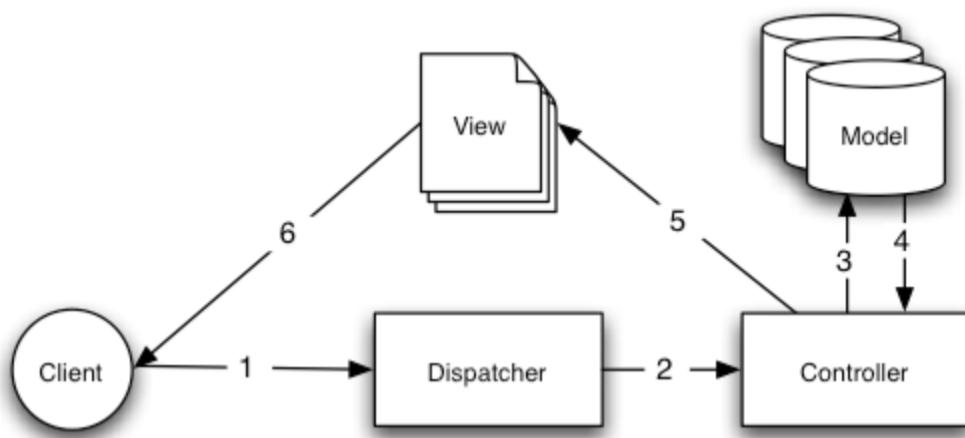


Figura 1

La figura 1 muestra un ejemplo sencillo de una petición [request] MVC.

A efectos ilustrativos, supongamos que un usuario llamado Ricardo acaba de hacer clic en el enlace “¡Comprar un pastel personalizado ahora!” de la página inicial de la aplicación.

1. Ricardo hace clic en el enlace apuntando a <http://www.ejemplo.com/pasteles/comprar>, y su navegador hace una petición al servidor web.
2. **El despachador** comprueba la URL de la petición (/pasteles/comprar), y **le pasa la petición al controlador adecuado**.
3. **El controlador realiza lógica de aplicación específica**. Por ejemplo, puede comprobar si Ricardo ha iniciado sesión.
4. El controlador también **utiliza modelos para acceder a los datos de la aplicación**. La mayoría de las veces los modelos representan tablas de **una base de datos**. En este ejemplo, el controlador utiliza un modelo para buscar la última compra de Ricardo en la base de datos.
5. Una vez que el controlador ha hecho su magia en los datos, **se los pasa a la vista**. La vista toma los datos y los deja listos para su presentación al usuario.
La mayoría de las veces **las vistas vienen en formato HTML**, pero una vista puede ser fácilmente un PDF, un documento XML, o un **objeto JSON**, dependiendo de tus necesidades.
6. Una vez que el objeto encargado de procesar vistas ha utilizado los datos del modelo para construir una vista completa, el contenido se devuelve al navegador de Ricardo.

Casi todas las peticiones a tu aplicación seguirán este patrón básico.

Beneficios

¿Por qué utilizar MVC? Porque **es un patrón de diseño de software probado y se sabe que funciona**. Con MVC **la aplicación se puede desarrollar rápidamente, de forma modular y mantenible**.

Separar las funciones de la aplicación en **modelos, vistas y controladores** hace que la aplicación sea muy ligera. Estas características nuevas se añaden fácilmente y las antiguas toman automáticamente una forma nueva.

El diseño modular permite a los diseñadores y a los desarrolladores trabajar conjuntamente, así como realizar rápidamente el prototipado. Esta separación también permite hacer cambios en una parte de la aplicación sin que las demás se vean afectadas.

MVC

[Área personal](#) / Mis cursos / [MVC01](#) / [Introducción y conceptos](#) / [Introducción y conceptos #2](#)

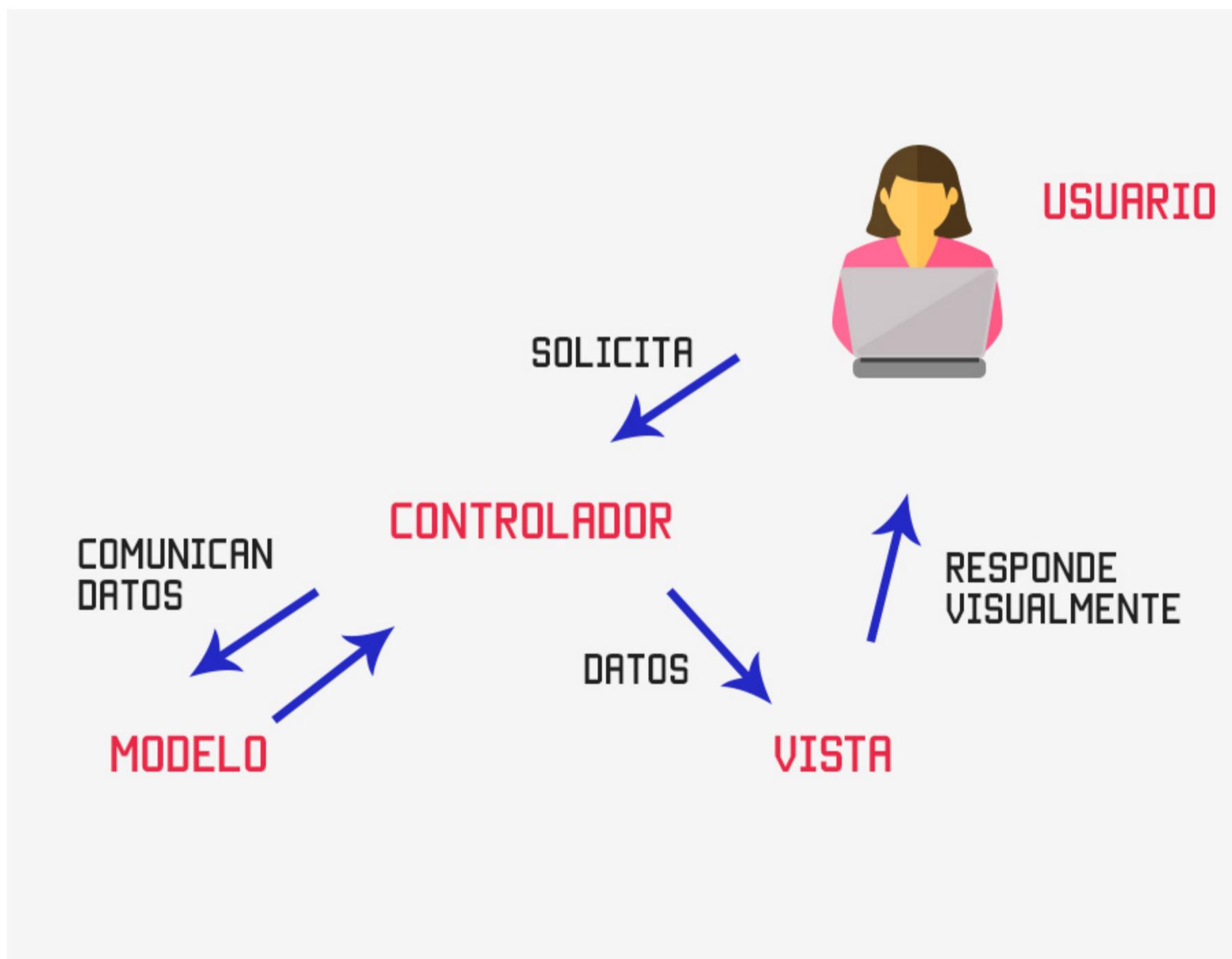
Introducción y conceptos #2

El MVC o Modelo-Vista-Controlador es un patrón de arquitectura de software que, utilizando 3 componentes (Vistas, Models y Controladores) separa la lógica de la aplicación de la lógica de la vista en una aplicación.

Es una arquitectura importante puesto que se utiliza tanto en componentes gráficos básicos hasta sistemas empresariales; la mayoría de los frameworks modernos utilizan MVC (o alguna adaptación del MVC) para la arquitectura.

En este pequeño artículo intentamos introducirte a los conceptos del MVC.

UNA IMAGEN



UNA ANALOGÍA

Una que me gusta mucho es la de la televisión. En tu televisión puedes ver distintos canales distribuidos por tu proveedor de cable o televisión (que representa al modelo), todos los canales que puedes ver son la vista, y tú cambiando de canal, controlando qué ves representas al controlador.

LA EXPLICACIÓN

Los puntos anteriores sólo para proveer background, y que ojalá puedas utilizar las referencias ahora que vamos a explicar qué es.

Antes que nada, me gustaría mencionar por qué se utiliza el **MVC**, la razón es que nos permite separar los componentes de nuestra aplicación dependiendo de la responsabilidad que tienen, esto significa que cuando hacemos un cambio en alguna parte de nuestro código, esto no afecte otra parte del mismo. Por ejemplo, si modificamos nuestra Base de Datos, sólo deberíamos modificar el modelo que es quién se encarga de los

datos y el resto de la aplicación debería permanecer intacta. Esto respeta el principio de la responsabilidad única. Es decir, una parte de tu código no debe de saber qué es lo que hace toda la aplicación, sólo debe de tener una responsabilidad.

En web, el MVC funcionaría así: Cuando el usuario manda una petición al navegador, digamos quiere ver el [curso de AngularJS](#), el controlador responde a la solicitud, porque él es el que controla la lógica de la aplicación, una vez que el controlador nota que el usuario solicitó el curso de Angular, le pide al modelo la información del curso.

El modelo, que se encarga de los datos de la aplicación, consulta la base de datos y digamos, obtiene todos los videos del curso de AngularJS, la información del curso y el título, el modelo responde al controlador con los datos que pidió (nota como en la imagen las flechas van en ambos sentidos, porque el controlador pide datos, y el modelo responde con los datos solicitados).

Una vez el controlador tiene los datos del curso de AngularJS, se los manda a la vista, la vista aplica los estilos, organiza la información y construye la página que vez en el navegador.

Resumamos entonces los conceptos.

MODELO

Se encarga de los datos, generalmente (pero no obligatoriamente) consultando la base de datos. Actualizaciones, consultas, búsquedas, etc. todo eso va aquí, en el modelo.

CONTROLADOR

Se encarga de... controlar, recibe las órdenes del usuario y se encarga de solicitar los datos al modelo y de comunicárselos a la vista.

VISTAS

Son la representación visual de los datos, todo lo que tenga que ver con la interfaz gráfica va aquí. Ni el modelo ni el controlador se preocupan de cómo se verán los datos, esa responsabilidad es únicamente de la vista.

Última modificación: Monday, 25 de March de 2019, 09:18

◀ Introducción y conceptos #1

Ir a...

Presentación MVC ►

Usted se ha identificado como Leandro Sandoval (Cerrar sesión)

MVC01

Resumen de retención de datos

Descargar la app para dispositivos móviles

CODES

Capacitación Web – Conceptos Técnicos

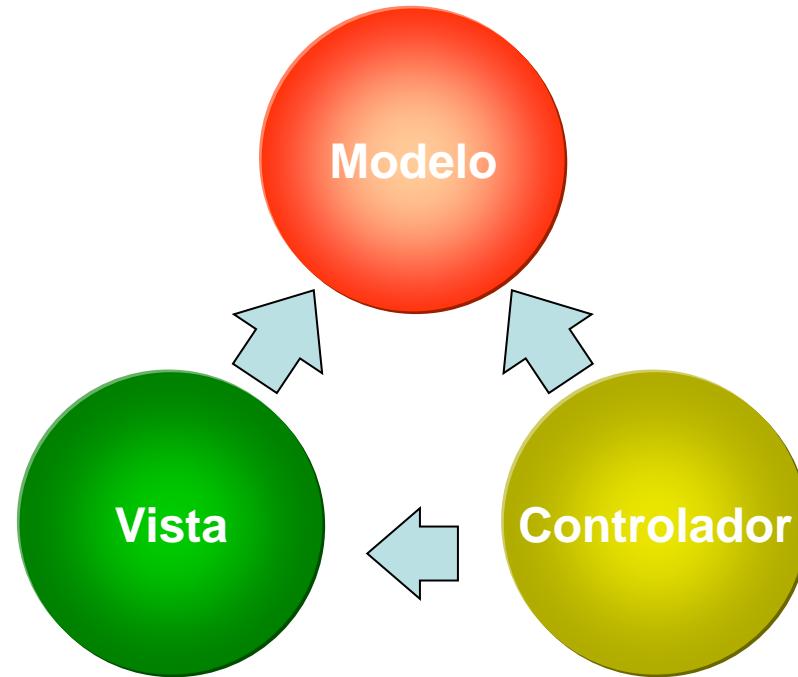
1. INTRODUCCIÓN ASP.NET MVC

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).

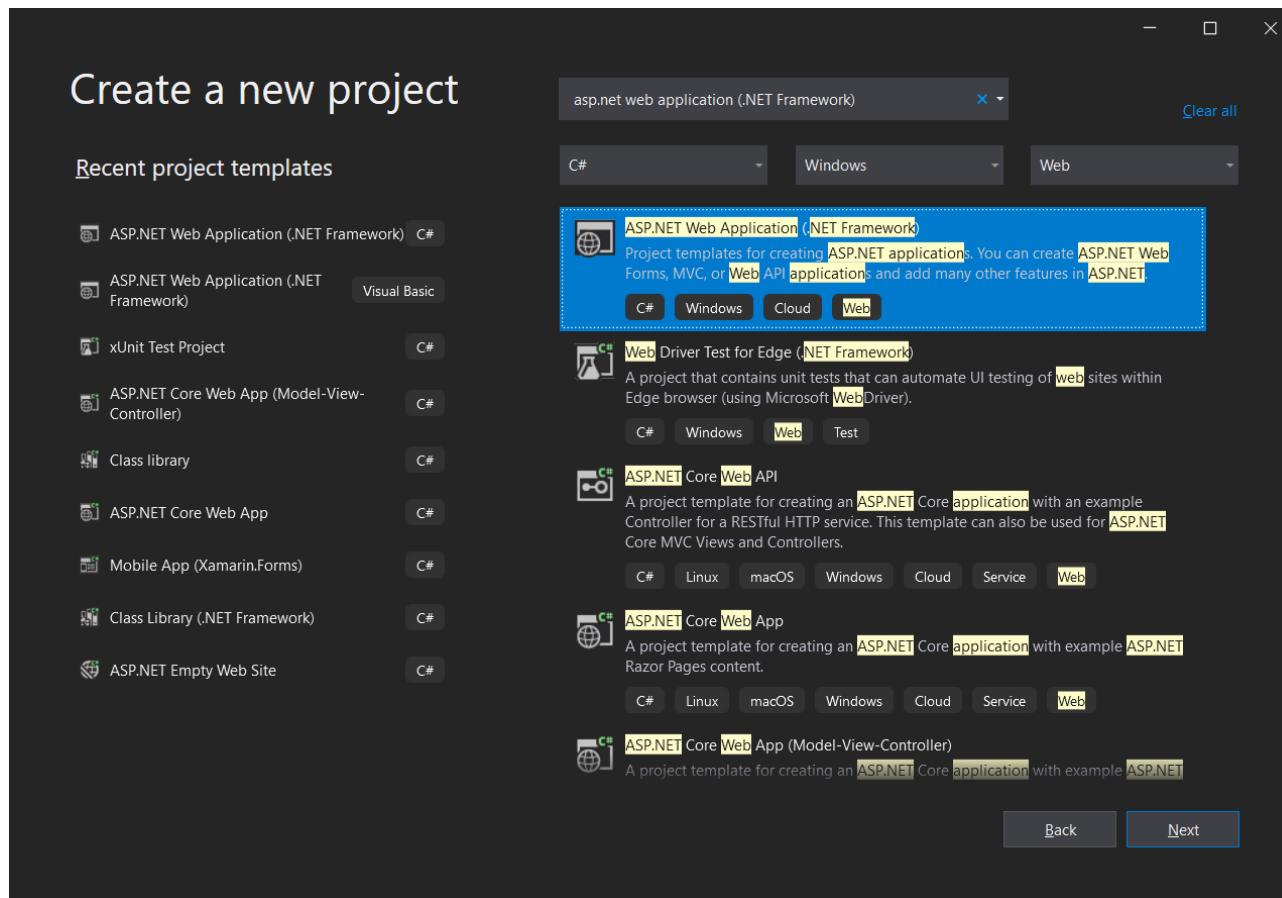
4. CONSTRUYENDO UNA APLICACIÓN MVC

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:

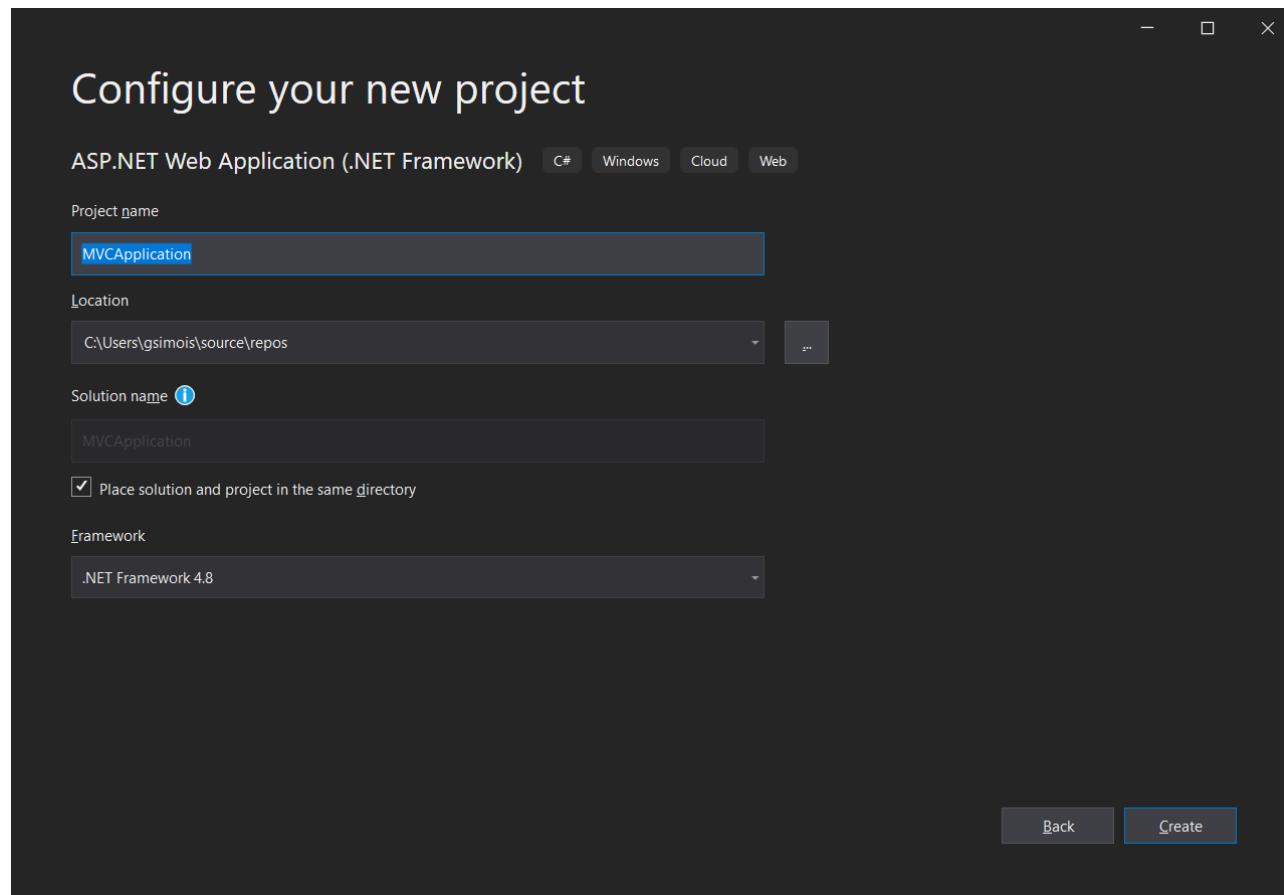
- Una página con el listado de provincias.
- Una página con el detalle de cada una de ellas.

5. CREANDO NUESTRA APLICACIÓN



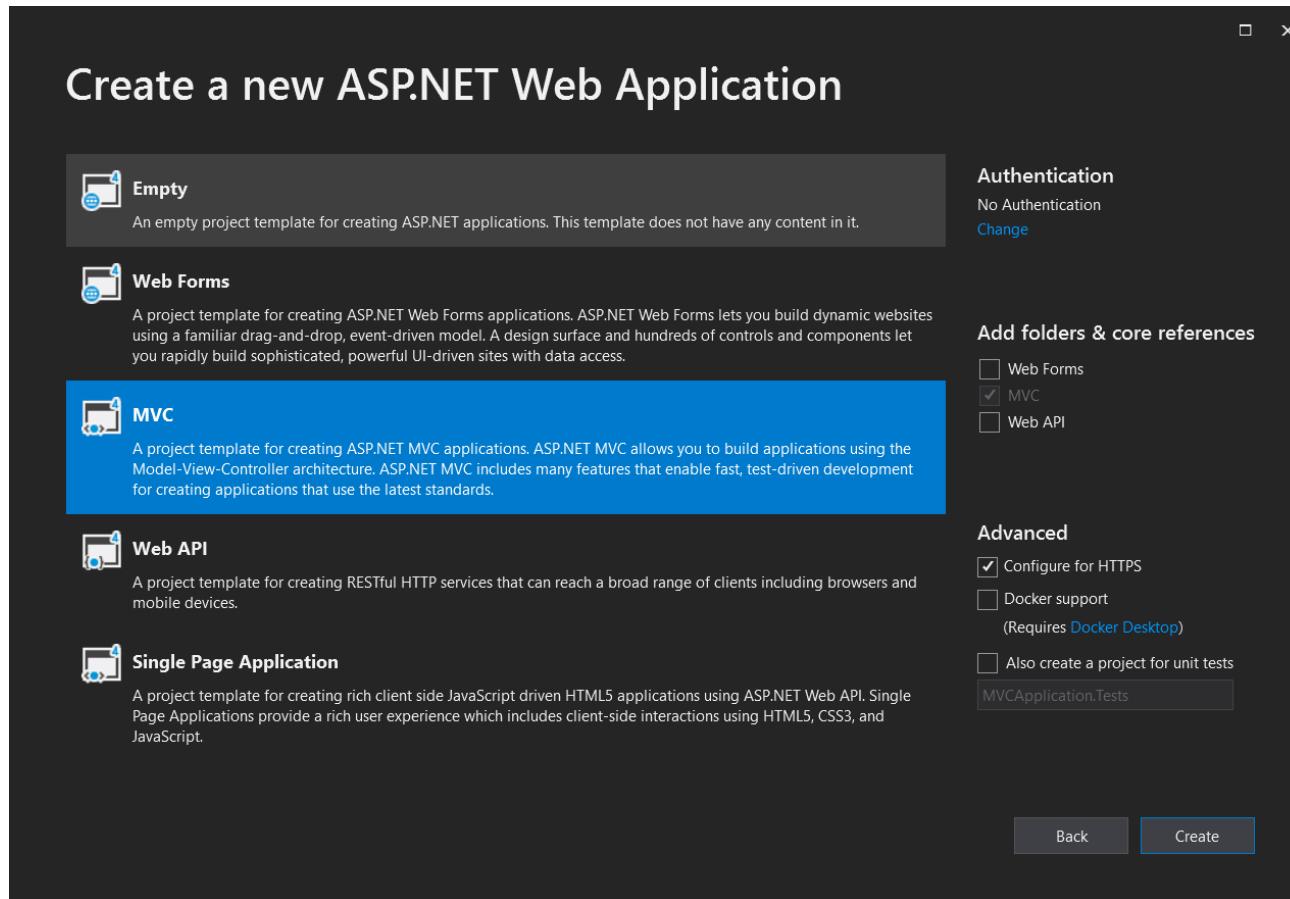
Para crear nuestra aplicación, seleccionamos la opción de menú “Nuevo Proyecto”, Y seleccionamos el template **ASP.NET Web Application (.NET Framework)**

5. CREANDO NUESTRA APLICACIÓN



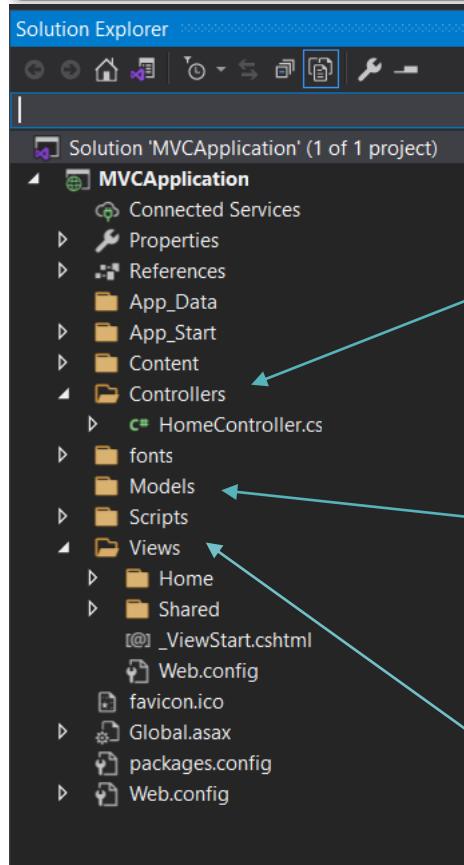
Luego especificamos el nombre de la aplicación y la versión de MVC. En este caso utilizaremos la versión 4.8

5. CREANDO NUESTRA APLICACIÓN



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 **XxxxController**. (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.

7. Agregando la clase ProvinciaModel

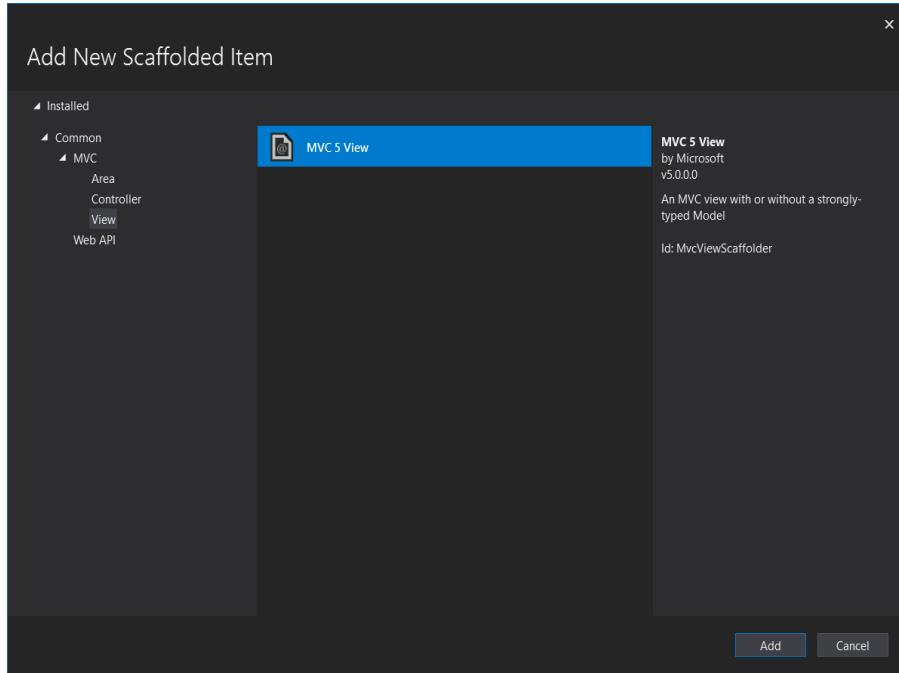
En la carpeta Models del aplicativo, agregamos una clase que llamaremos “ProvinciaModel”:

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

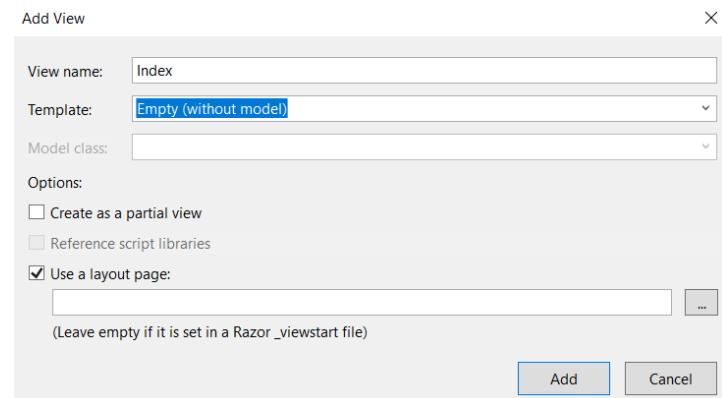
namespace MVCAplication.Models
{
    public class ProvinciaModel
    {
        public int Id { get; set; }
        public string Descripcion { get; set; }
    }
}
```

8. Agregando la View

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).

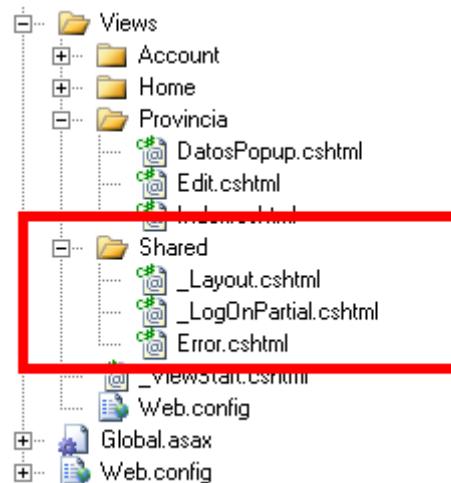


9. Agregando la View (II)

```
@model IList<MVCAccount.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 MVCAplication.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.

14. Agregando validaciones al modelo (II)

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);
}
```

17. Agregando validaciones al modelo (III)

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 MVCAplication.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;
}
```

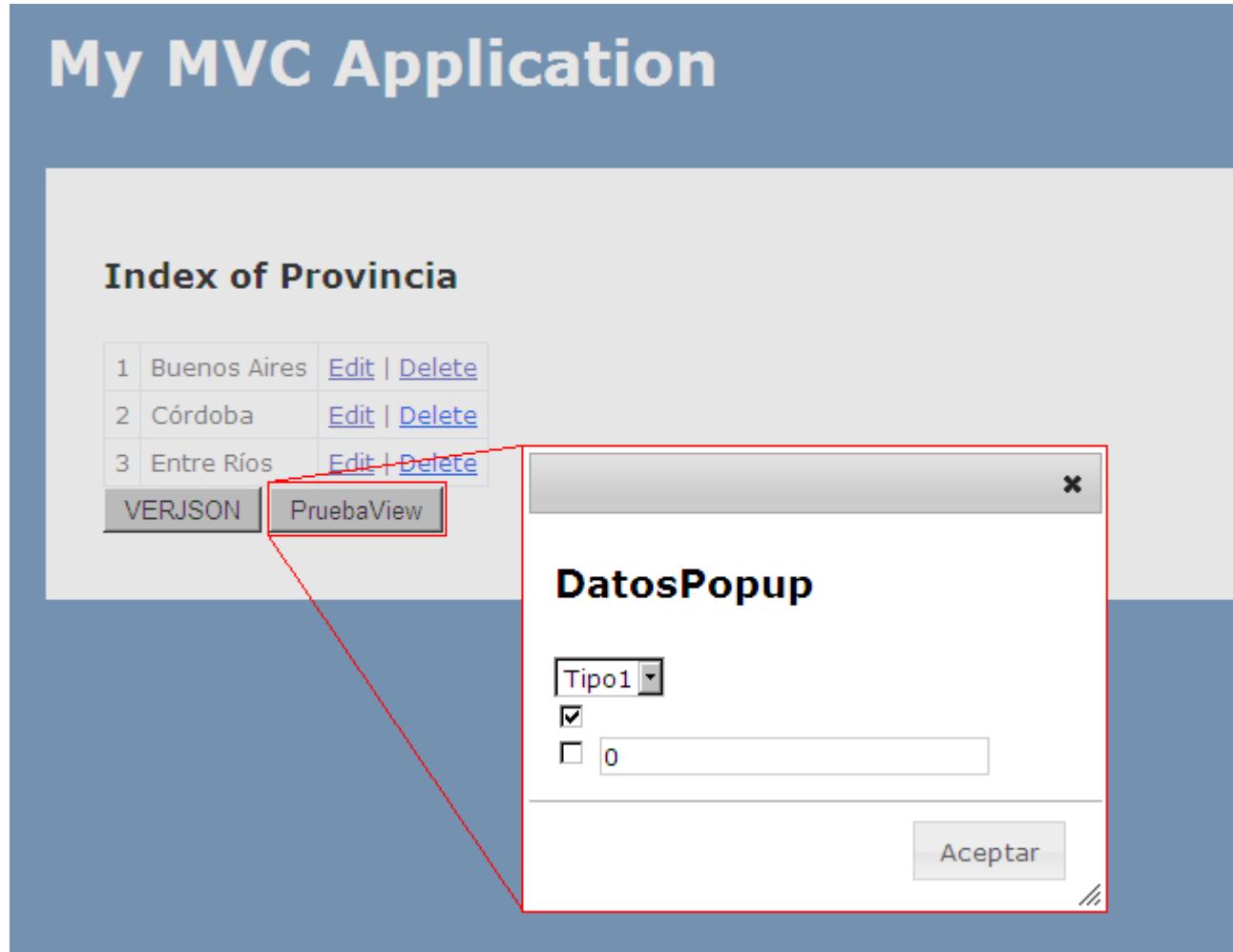
21. Usando Pop-ups con JQuery y Ajax

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:



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

24. Validación utilizando el atributo [CustomValidation]

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; }

}
```

25. Validación mediante la creación de atributos personalizados

Definimos nuestro atributo personalizado:

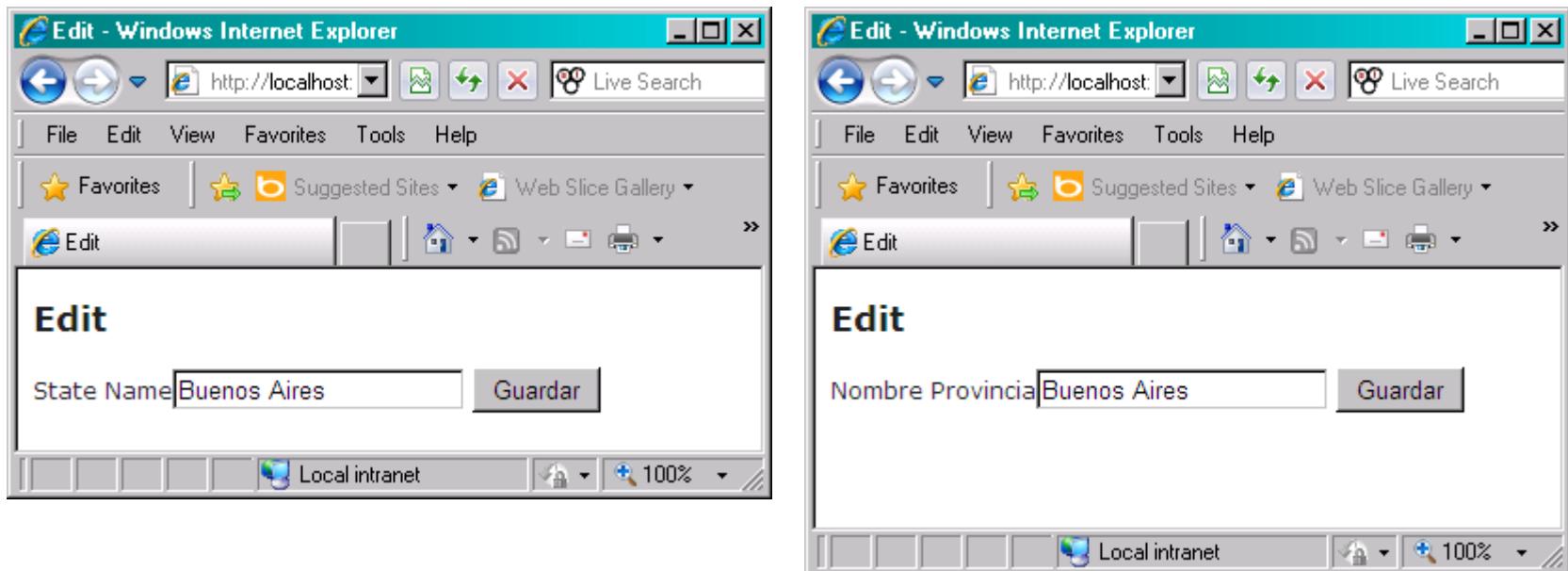
```
public class IsValidIdProvinciaAttributte
{
    public override bool IsValid(object value)
    {
        int id = Convert.ToInt32(value);
        if (id > 0)
        {
            return true;
        }
        return false;
    }
}
```

Y utilizamos nuestro atributo creado:

```
public class ProvinciaModel
{
    [Required]
    [IsValidIdProvinciaAttributte(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 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

30. Regionalización

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" />

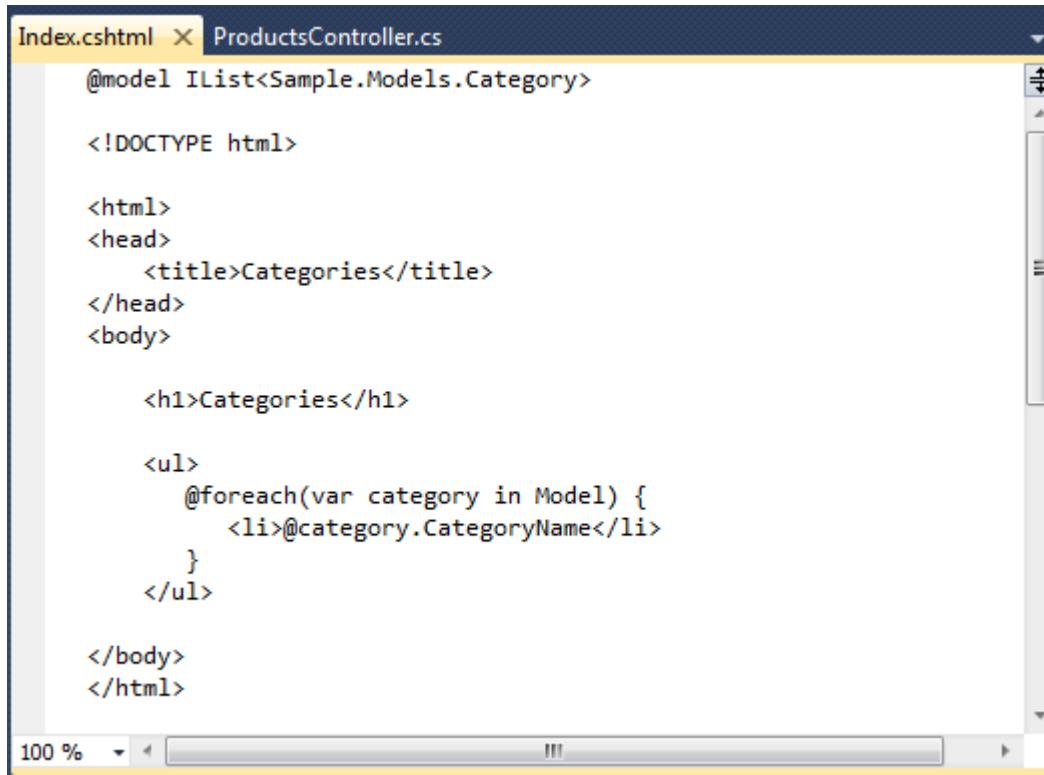
HTML Helpers

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.



The screenshot shows a code editor window with two tabs: "Index.cshtml" and "ProductsController.cs". The "Index.cshtml" tab is active, displaying the following Razor code:

```
Index.cshtml × ProductsController.cs
@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 code uses the `@model` directive to specify the `IList<Sample.Models.Category>` type for the view model. It then uses a `foreach` loop to iterate over the `Category` items and output their names in an unordered list.

System.Web.Mvc Namespace

Reference

Classes

AcceptVerbsAttribute	Represents an attribute that specifies which HTTP verbs an action method will respond to.
ActionDescriptor	Provides information about an action method, such as its name, controller, parameters, attributes, and filters.
ActionExecutedContext	Provides the context for the ActionExecuted method of the ActionFilterAttribute class.
ActionExecutingContext	Provides the context for the ActionExecuting method of the ActionFilterAttribute class.
ActionFilterAttribute	Represents the base class for filter attributes.
ActionMethodSelectorAttribute	Represents an attribute that is used to influence the selection of an action method.
ActionNameAttribute	Represents an attribute that is used for the name of an action.
ActionNameSelectorAttribute	Represents an attribute that affects the selection of an action method.
ActionResult	Represents the result of an action method.
AdditionalMetadataAttribute	Provides a class that implements the IMetadataAware interface in order to support additional metadata.
AjaxHelper	Represents support for rendering HTML in AJAX scenarios within a view.
AjaxHelper<TModel>	Represents support for rendering HTML in AJAX scenarios within a strongly typed view.
AjaxRequestExtensions	Represents a class that extends the HttpRequestBase class by adding the ability to determine whether an HTTP request is an AJAX request.
AllowAnonymousAttribute	Represents an attribute that marks controllers and actions to skip the AuthorizeAttribute during authorization.
AllowHtmlAttribute	Allows a request to include HTML markup during model binding by skipping request validation for the property. (It is strongly recommended that your application explicitly check all models where you disable request validation in order to prevent script exploits.)
AreaRegistration	Provides a way to register one or more areas in an ASP.NET MVC application.
AreaRegistrationContext	Encapsulates the information that is required in order to register an area within an ASP.NET MVC application.

AssociatedMetadataProvider	Provides an abstract class to implement a metadata provider.
AssociatedValidatorProvider	Provides an abstract class for classes that implement a validation provider.
AsyncController	Provided for backward compatibility with ASP.NET MVC 3.
AsyncTimeoutAttribute	Represents an attribute that is used to set the timeout value, in milliseconds, for an asynchronous method.
AuthorizationContext	Encapsulates the information that is required for using an AuthorizeAttribute attribute.
AuthorizeAttribute	Specifies that access to a controller or action method is restricted to users who meet the authorization requirement.
BindAttribute	Represents an attribute that is used to provide details about how model binding to a parameter should occur.
BuildManagerCompiledView	Represents the base class for views that are compiled by the BuildManager class before being rendered by a view engine.
BuildManagerViewEngine	Provides a base class for view engines.
ByteArrayModelBinder	Maps a browser request to a byte array.
CachedAssociatedMetadataProvider<TModelMetadata>	Provides an abstract class to implement a cached metadata provider.
CachedDataAnnotationsMetadataAttributes	Provides a container to cache System.ComponentModel.DataAnnotations attributes.
CachedDataAnnotationsModelMetadata	Provides a container to cache DataAnnotationsModelMetadata .
CachedDataAnnotationsModelMetadataProvider	Implements the default cached model metadata provider for ASP.NET MVC.
CachedModelMetadata<TPrototypeCache>	Provides a container for cached metadata.
CancellationTokenModelBinder	Provides a mechanism to propagate notification that model binder operations should be canceled.
ChildActionOnlyAttribute	Represents an attribute that is used to indicate that an action method should be called only as a child action.
ChildActionValueProvider	Represents a value provider for values from child actions.
ChildActionValueProviderFactory	Represents a factory for creating value provider objects for child actions.
ClientDataTypeModelValidatorProvider	Returns the client data-type model validators.
CompareAttribute	Provides an attribute that compares two properties of a model.

ContentResult	Represents a user-defined content type that is the result of an action method.
Controller	Provides methods that respond to HTTP requests that are made to an ASP.NET MVC Web site.
ControllerActionInvoker	Represents a class that is responsible for invoking the action methods of a controller.
ControllerBase	Represents the base class for all MVC controllers.
ControllerBuilder	Represents a class that is responsible for dynamically building a controller.
ControllerContext	Encapsulates information about an HTTP request that matches specified RouteBase and ControllerBase instances.
ControllerDescriptor	Encapsulates information that describes a controller, such as its name, type, and actions.
ControllerInstanceFilterProvider	Adds the controller to the FilterProviderCollection instance.
CustomModelBinderAttribute	Represents an attribute that invokes a custom model binder.
DataAnnotationsModelMetadata	Provides a container for common metadata, for the DataAnnotationsModelMetadataProvider class, and for the DataAnnotationsModelValidator class for a data model.
DataAnnotationsModelMetadataProvider	Implements the default model metadata provider for ASP.NET MVC.
DataAnnotationsModelValidator	Provides a model validator.
DataAnnotationsModelValidator<TAttribute>	Provides a model validator for a specified validation type.
DataAnnotationsModelValidatorProvider	Implements the default validation provider for ASP.NET MVC.
ModelErrorInfoModelValidatorProvider	Provides a container for the error-information model validator.
DefaultControllerFactory	Represents the controller factory that is registered by default.
DefaultModelBinder	Maps a browser request to a data object. This class provides a concrete implementation of a model binder.
DefaultViewLocationCache	Represents a memory cache for view locations.
DependencyResolver	Provides a registration point for dependency resolvers that implement IDependencyResolver or the Common Service Locator IServiceLocator interface.
DependencyResolverExtensions	Provides a type-safe implementation of GetService(Type) and GetServices(Type) .

DictionaryValueProvider<TValue>	Represents the base class for value providers whose values come from a collection that implements the IDictionary< TKey, TValue > interface.
EmptyModelMetadataProvider	Provides an empty metadata provider for data models that do not require metadata.
EmptyModelValidatorProvider	Provides an empty validation provider for models that do not require a validator.
EmptyResult	Represents a result that does nothing, such as a controller action method that returns nothing.
ExceptionContext	Provides the context for using the HandleErrorAttribute class.
ExpressionHelper	Provides a helper class to get the model name from an expression.
FieldValidationMetadata	Provides a container for client-side field validation metadata.
FileContentResult	Sends the contents of a binary file to the response.
FilePathResult	Sends the contents of a file to the response.
FileResult	Represents a base class that is used to send binary file content to the response.
FileStreamResult	Sends binary content to the response by using a Stream instance.
Filter	Represents a metadata class that contains a reference to the implementation of one or more of the filter interfaces, the filter's order, and the filter's scope.
FilterAttribute	Represents the base class for action and result filter attributes.
FilterAttributeFilterProvider	Defines a filter provider for filter attributes.
FilterInfo	Encapsulates information about the available action filters.
FilterProviderCollection	Represents the collection of filter providers for the application.
FilterProviders	Provides a registration point for filters.
FormCollection	Contains the form value providers for the application.
FormContext	Encapsulates information that is required in order to validate and process the input data from an HTML form.
FormValueProvider	Represents a value provider for form values that are contained in a NameValueCollection object.
FormValueProviderFactory	Represents a class that is responsible for creating a new instance of a form-value provider object.
GlobalFilterCollection	Represents a class that contains all the global filters.
GlobalFilters	Represents the global filter collection.

HandleErrorAttribute	Represents an attribute that is used to handle an exception that is thrown by an action method.
HandleErrorInfo	Encapsulates information for handling an error that was thrown by an action method.
HiddenInputAttribute	Represents an attribute that is used to indicate whether a property or field value should be rendered as a hidden input element.
HtmlHelper	Supports the rendering of HTML controls in a view.
HtmlHelper<TModel>	Represents support for rendering HTML controls in a strongly typed view.
HttpAntiForgeryException	This type/member supports the .NET Framework infrastructure and is not intended to be used directly from your code.
HttpDeleteAttribute	Represents an attribute that is used to restrict an action method so that the method handles only HTTP DELETE requests.
HttpFileCollectionValueProvider	Represents a value provider to use with values that come from a collection of HTTP files.
HttpFileCollectionValueProviderFactory	Represents a class that is responsible for creating a new instance of an HTTP file collection value provider object.
HttpGetAttribute	Represents an attribute that is used to restrict an action method so that the method handles only HTTP GET requests.
HttpHeadAttribute	Specifies that the HTTP request must be the HTTP HEAD method.
HttpNotFoundResult	Defines an object that is used to indicate that the requested resource was not found.
HttpOptionsAttribute	Represents an attribute that is used to restrict an action method so that the method handles only HTTP OPTIONS requests.
HttpPatchAttribute	Represents an attribute that is used to restrict an action method so that the method handles only HTTP PATCH requests.
HttpPostAttribute	Represents an attribute that is used to restrict an action method so that the method handles only HTTP POST requests.
HttpPostedFileBaseModelBinder	Binds a model to a posted file.
HttpPutAttribute	Represents an attribute that is used to restrict an action method so that the method handles only HTTP PUT requests.
HttpRequestExtensions	Extends the HttpRequestBase class that contains the HTTP values that were sent by a client during a Web request.
StatusCodeResult	Provides a way to return an action result with a specific HTTP response status code and description.
UnauthorizedResult	Represents the result of an unauthorized HTTP request.

JavaScriptResult	Sends JavaScript content to the response.
JQueryFormValueProvider	The JQuery Form Value provider is used to handle JQuery formatted data in request Forms.
JQueryFormValueProviderFactory	Provides the necessary ValueProvider to handle JQuery Form data.
JsonResult	Represents a class that is used to send JSON-formatted content to the response.
JsonValueProviderFactory	Enables action methods to send and receive JSON-formatted text and to model-bind the JSON text to parameters of action methods.
LinqBinaryModelBinder	Maps a browser request to a LINQ Binary object.
MaxLengthAttributeAdapter	Provides an adapter for the MaxLengthAttribute attribute.
MinLengthAttributeAdapter	Provides an adapter for the MinLengthAttribute attribute.
ModelBinderAttribute	Represents an attribute that is used to associate a model type to a model-builder type.
ModelBinderDictionary	Represents a class that contains all model binders for the application, listed by binder type.
ModelBinderProviderCollection	No content here will be updated; please do not add material here.
ModelBinderProviders	Provides a container for model binder providers.
ModelBinders	Provides global access to the model binders for the application.
ModelBindingContext	Provides the context in which a model binder functions.
ModelClientValidationEqualToRule	This type/member supports the .NET Framework infrastructure and is not intended to be used directly from your code.
ModelClientValidationRangeRule	This type/member supports the .NET Framework infrastructure and is not intended to be used directly from your code.
ModelClientValidationRegexRule	This type/member supports the .NET Framework infrastructure and is not intended to be used directly from your code.
ModelClientValidationRemoteRule	Represents the remote rule for the validation of the model client.
ModelClientValidationRequiredRule	Represents the required rule for the validation of the model client.
ModelClientValidationRule	This type/member supports the .NET Framework infrastructure and is not intended to be used directly from your code.
ModelClientValidationStringLengthRule	This type/member supports the .NET Framework infrastructure and is not intended to be used directly from your code. Represents a length of the validation rule of the model client.
ModelError	Represents an error that occurs during model binding.

ModelErrorCollection	A collection of ModelError instances.
ModelMetadata	Provides a container for common metadata, for the ModelMetadataProvider class, and for the ModelValidator class for a data model.
ModelMetadataProvider	Provides an abstract base class for a custom metadata provider.
ModelMetadataProviders	Provides a container for the current ModelMetadataProvider instance.
ModelState	Encapsulates the state of model binding to a property of an action-method argument, or to the argument itself.
ModelStateDictionary	Represents the state of an attempt to bind a posted form to an action method, which includes validation information.
ModelValidationResult	Provides a container for a validation result.
ModelValidator	Provides a base class for implementing validation logic.
ModelValidatorProvider	Provides a list of validators for a model.
ModelValidatorProviderCollection	No content here will be updated; please do not add material here.
ModelValidatorProviders	Provides a container for the current validation provider.
MultiSelectList	Represents a list of items that users can select more than one item from.
MvcFilter	When implemented in a derived class, provides a metadata class that contains a reference to the implementation of one or more of the filter interfaces, the filter's order, and the filter's scope.
MvcHandler	Selects the controller that will handle an HTTP request.
MvcHtmlString	Represents an HTML-encoded string that should not be encoded again.
MvcHttpHandler	Verifies and processes an HTTP request.
MvcRouteHandler	Creates an object that implements the IHttpHandler interface and passes the request context to it.
MvcWebRazorHostFactory	Creates instances of <code>System.Web.Mvc.MvcWebPageRazorHost</code> files.
NameValueCollectionExtensions	Extends a NameValueCollection object so that the collection can be copied to a specified dictionary.
NameValuePair	Represents the base class for value providers whose values come from a NameValueCollection object.
NoAsyncTimeoutAttribute	Provides a convenience wrapper for the AsyncTimeoutAttribute attribute.
NonActionAttribute	Represents an attribute that is used to indicate that a controller method is not an action method.
OutputCacheAttribute	Represents an attribute that is used to mark an action method whose

	output will be cached.
OverrideActionFiltersAttribute	Represents the attributes associated with the override filter.
OverrideAuthenticationAttribute	Represents the attributes associated with the authentication.
OverrideAuthorizationAttribute	Represents the attributes associated with the authorization.
OverrideExceptionFiltersAttribute	Represents the attributes associated with the exception filter.
OverrideResultFiltersAttribute	Represents the attributes associated with the result filter.
ParameterBindingInfo	Encapsulates information for binding action-method parameters to a data model.
ParameterDescriptor	Contains information that describes a parameter.
PartialViewResult	Represents a base class that is used to send a partial view to the response.
PreApplicationStartCode	Provides a registration point for ASP.NET Razor pre-application start code.
QueryStringValueProvider	Represents a value provider for query strings that are contained in a NameValueCollection object.
QueryStringValueProviderFactory	Represents a class that is responsible for creating a new instance of a query-string value-provider object.
RangeAttributeAdapter	Provides an adapter for the RangeAttribute attribute.
RazorView	Represents the class used to create views that have Razor syntax.
RazorViewEngine	Represents a view engine that is used to render a Web page that uses the ASP.NET Razor syntax.
RedirectResult	Controls the processing of application actions by redirecting to a specified URI.
RedirectToRouteResult	Represents a result that performs a redirection by using the specified route values dictionary.
ReflectedActionDescriptor	Contains information that describes a reflected action method.
ReflectedControllerDescriptor	Contains information that describes a reflected controller.
ReflectedParameterDescriptor	Contains information that describes a reflected action-method parameter.
RegularExpressionAttributeAdapter	Provides an adapter for the RegularExpressionAttribute attribute.
RemoteAttribute	Provides an attribute that uses the jQuery validation plug-in remote validator.
RequiredAttributeAdapter	Provides an adapter for the RequiredAttributeAttribute attribute.

RequireHttpsAttribute	Represents an attribute that forces an unsecured HTTP request to be resent over HTTPS.
ResultExecutedContext	Provides the context for the OnResultExecuted(ResultExecutedContext) method of the ActionFilterAttribute class.
ResultExecutingContext	Provides the context for the OnResultExecuting(ResultExecutingContext) method of the ActionFilterAttribute class.
RouteAreaAttribute	Defines the area to set for all the routes defined in this controller.
RouteAttribute	Place on a controller or action to expose it directly via a route. When placed on a controller, it applies to actions that do not have any System.Web.Mvc.RouteAttribute 's on them.
RouteCollectionAttributeRoutingExtensions	Provides routing extensions for route collection attribute.
RouteCollectionExtensions	Extends a RouteCollection object for MVC routing.
RouteDataProvider	Represents a value provider for route data that is contained in an object that implements the IDictionary<TKey,TValue> interface.
RouteDataProviderFactory	Represents a factory for creating route-data value provider objects.
RoutePrefixAttribute	Annotates a controller with a route prefix that applies to all actions within the controller.
SelectList	Represents a list that lets users select one item.
SelectListGroup	Represents the optgroup HTML element and its attributes. In a select list, multiple groups with the same name are supported. They are compared with reference equality.
SelectListItem	Represents the selected item in an instance of the SelectList class.
SessionStateAttribute	Specifies the session state of the controller.
SessionStateTempDataProvider	Provides session-state data to the current TempDataDictionary object.
StringLengthAttributeAdapter	Provides an adapter for the StringLengthAttribute attribute.
TagBuilder	Contains classes and properties that are used to create HTML elements. This class is used to write helpers, such as those found in the System.Web.Helpers namespace.
TempDataDictionary	Represents a set of data that persists only from one request to the next.
TemplateInfo	Encapsulates information about the current template context.
UrlHelper	Contains methods to build URLs for ASP.NET MVC within an application.
UrlParameter	Represents an optional parameter that is used by the MvcHandler class during routing.
ValidatableObjectAdapter	Provides an object adapter that can be validated.

ValidateAntiForgeryTokenAttribute	Represents an attribute that is used to prevent forgery of a request.
ValidateInputAttribute	Represents an attribute that is used to mark action methods whose input must be validated.
ValueProviderCollection	Represents the collection of value-provider objects for the application.
ValueProviderDictionary	Note: This API is now obsolete. Represents a dictionary of value providers for the application.
ValueProviderFactories	Represents a container for value-provider factory objects.
ValueProviderFactory	Represents a factory for creating value-provider objects.
ValueProviderFactoryCollection	Represents the collection of value-provider factories for the application.
ValueProviderResult	Represents the result of binding a value (such as from a form post or query string) to an action-method argument property, or to the argument itself.
ViewContext	Encapsulates information that is related to rendering a view.
ViewDataDictionary	Represents a container that is used to pass data between a controller and a view.
ViewDataDictionary<TModel>	Represents a container that is used to pass strongly typed data between a controller and a view.
ViewDataInfo	Encapsulates information about the current template content that is used to develop templates and about HTML helpers that interact with templates.
ViewEngineCollection	Represents a collection of view engines that are available to the application.
ViewEngineResult	Represents the result of locating a view engine.
ViewEngines	Represents a collection of view engines that are available to the application.
ViewMasterPage	Represents the information that is needed to build a master view page.
ViewMasterPage<TModel>	Represents the information that is required in order to build a strongly typed master view page.
ViewPage	Represents the properties and methods that are needed to render a view as a Web Forms page.
ViewPage<TModel>	Represents the information that is required in order to render a strongly typed view as a Web Forms page.
ViewResult	Represents a class that is used to render a view by using an IView instance that is returned by an IViewEngine object.
ViewResultBase	Represents a base class that is used to provide the model to the view and then render the view to the response.

ViewStartPage	Provides an abstract class that can be used to implement a view start (master) page.
ViewTemplateUserControl	Provides a container for TemplateInfo objects.
ViewTemplateUserControl<TModel>	Provides a container for TemplateInfo objects.
ViewType	Represents the type of a view.
ViewUserControl	Represents the information that is needed to build a user control.
ViewUserControl<TModel>	Represents the information that is required in order to build a strongly typed user control.
VirtualPathProviderViewEngine	Represents an abstract base-class implementation of the IViewEngine interface.
WebFormView	Represents the information that is needed to build a Web Forms page in ASP.NET MVC.
WebFormViewEngine	Represents a view engine that is used to render a Web Forms page to the response.
WebViewPage	Represents the properties and methods that are needed in order to render a view that uses ASP.NET Razor syntax.
WebViewPage<TModel>	Represents the properties and methods that are needed in order to render a view that uses ASP.NET Razor syntax.

Interfaces

IActionFilter	Defines the methods that are used in an action filter.
IActionInvoker	Defines the contract for an action invoker, which is used to invoke an action in response to an HTTP request.
IActionInvokerFactory	Used to create an IActionInvoker instance for the current request.
IAuthorizationFilter	Defines the methods that are required for an authorization filter.
IClientValidatable	Provides a way for the ASP.NET MVC validation framework to discover at run time whether a validator has support for client validation.
IController	Defines the methods that are required for a controller.
IControllerActivator	Provides fine-grained control over how controllers are instantiated using dependency injection.
IControllerFactory	Defines the methods that are required for a controller factory.
IDependencyResolver	Defines the methods that simplify service location and dependency resolution.

IEnumerableValueProvider	Represents a special IValueProvider that has the ability to be enumerable.
IExceptionFilter	Defines the methods that are required for an exception filter.
IFilterProvider	Provides an interface for finding filters.
IMetadataAware	Provides an interface for exposing attributes to the AssociatedMetadataProvider class.
MethodInfoActionDescriptor	An optional interface for ActionDescriptor types which provide a MethodInfo .
IModelBinder	Defines the methods that are required for a model binder.
IModelBinderProvider	Defines methods that enable dynamic implementations of model binding for classes that implement the IModelBinder interface.
IMvcFilter	Defines members that specify the order of filters and whether multiple filters are allowed.
IResultFilter	Defines the methods that are required for a result filter.
IRouteWithArea	Associates a route with an area in an ASP.NET MVC application.
ITempDataProvider	Defines the contract for temporary-data providers that store data that is viewed on the next request.
ITempDataProviderFactory	Used to create an ITempDataProvider instance for the controller.
IUnvalidatedValueProvider	Represents an IValueProvider interface that can skip request validation.
IValueProvider	Defines the methods that are required for a value provider in ASP.NET MVC.
IView	Defines the methods that are required for a view.
I ViewDataContainer	Defines the methods that are required for a view data dictionary.
IViewEngine	Defines the methods that are required for a view engine.
IViewLocationCache	Defines the methods that are required in order to cache view locations in memory.
IViewPageActivator	Provides fine-grained control over how view pages are created using dependency injection.

Enums

AreaReference	Controls interpretation of a controller name when constructing a RemoteAttribute .
FilterScope	Defines values that specify the order in which ASP.NET MVC filters run within the same filter type and filter order.

FormMethod	Enumerates the HTTP request types for a form.
Html5DateRenderingMode	Enumerates the date rendering mode for HTML5.
HttpVerbs	Enumerates the HTTP verbs.
InputType	Enumerates the types of input controls.
JsonRequestBehavior	Specifies whether HTTP GET requests from the client are allowed.
TagRenderMode	Enumerates the modes that are available for rendering HTML tags.

Delegates

ActionSelector	Represents a delegate that contains the logic for selecting an action method.
DataAnnotationsModelValidationFactory	Represents the method that creates a DataAnnotationsModelValidatorProvider instance.
DataAnnotationsValidatableObjectAdapterFactory	Provides a factory for validators that are based on IValidatableObject .

Feedback

Was this page helpful?

 Yes

 No

ValidationExtensions.ValidationSummary Method

Reference

Definition

Namespace: [System.Web.Mvc.Html](#)

Assembly: System.Web.Mvc.dll

Package: Microsoft.AspNet.Mvc v5.2.6

Overloads

ValidationSummary(HtmlHelper, String, Object, String)	
ValidationSummary(HtmlHelper, String, IDictionary<String,Object>, String)	
ValidationSummary(HtmlHelper, Boolean, String, String)	
ValidationSummary(HtmlHelper, Boolean, String, Object)	Returns an unordered list (ul element) of validation messages that are in the ModelStateDictionary object and optionally displays only model-level errors.
ValidationSummary(HtmlHelper, Boolean, String, IDictionary<String,Object>)	Returns an unordered list (ul element) of validation messages that are in the ModelStateDictionary object and optionally displays only model-level errors.
ValidationSummary(HtmlHelper, String, String)	
ValidationSummary(HtmlHelper, String, Object)	Returns an unordered list (ul element) of validation messages in the ModelStateDictionary object.
ValidationSummary(HtmlHelper, String, IDictionary<String,Object>)	Returns an unordered list (ul element) of validation messages that are in the ModelStateDictionary object.
ValidationSummary(HtmlHelper, Boolean, String)	Returns an unordered list (ul element) of validation messages that are in the ModelStateDictionary object and optionally displays only model-level errors.
ValidationSummary(HtmlHelper, String)	Returns an unordered list (ul element) of validation messages that are in the ModelStateDictionary object.
ValidationSummary(HtmlHelper, Boolean)	Returns an unordered list (ul element) of validation messages that are in the ModelStateDictionary object and optionally displays only model-level errors.

level errors.

ValidationSummary(HtmlHelper)	Returns an unordered list (ul element) of validation messages that are in the ModelStateDictionary object.
ValidationSummary(HtmlHelper, Boolean, String, IDictionary<String, Object>, String)	
ValidationSummary(HtmlHelper, Boolean, String, Object, String)	

ValidationSummary(HtmlHelper, String, Object, String)

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, string message, object htmlAttributes,  
string headingTag);
```

Parameters

htmlHelper [HtmlHelper](#)

message [String](#)

htmlAttributes [Object](#)

headingTag [String](#)

Returns

[MvcHtmlString](#)

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, String, IDictionary<String, Object>, String)

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, string message,  
System.Collections.Generic.IDictionary<string,object> htmlAttributes, string  
headingTag);
```

Parameters

htmlHelper [HtmlHelper](#)

message [String](#)

htmlAttributes [IDictionary<String, Object>](#)

headingTag [String](#)

Returns

[MvcHtmlString](#)

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, Boolean, String, String)

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, bool excludePropertyErrors, string message,  
string headingTag);
```

Parameters

htmlHelper [HtmlHelper](#)

excludePropertyErrors Boolean

message String

headingTag String

Returns

[MvcHtmlString](#)

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, Boolean, String, Object)

Returns an unordered list (ul element) of validation messages that are in the [ModelStateDictionary](#) object and optionally displays only model-level errors.

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, bool excludePropertyErrors, string message,  
object htmlAttributes);
```

Parameters

htmlHelper [HtmlHelper](#)

The HTML helper instance that this method extends.

excludePropertyErrors Boolean

true to have the summary display model-level errors only, or false to have the summary display all errors.

message String

The message to display with the validation summary.

htmlAttributes Object

An object that contains the HTML attributes for the element.

Returns

[MvcHtmlString](#)

A string that contains an unordered list (ul element) of validation messages.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, Boolean, String, IDictionary<String, Object>)

Returns an unordered list (ul element) of validation messages that are in the [ModelStateDictionary](#) object and optionally displays only model-level errors.

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, bool excludePropertyErrors, string message,  
System.Collections.Generic.IDictionary<string,object> htmlAttributes);
```

Parameters

htmlHelper [HtmlHelper](#)

The HTML helper instance that this method extends.

excludePropertyErrors [Boolean](#)

true to have the summary display model-level errors only, or false to have the summary display all errors.

message [String](#)

The message to display with the validation summary.

htmlAttributes [IDictionary<String, Object>](#)

A dictionary that contains the HTML attributes for the element.

Returns

[MvcHtmlString](#)

A string that contains an unordered list (ul element) of validation messages.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, String, String)

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, string message, string headingTag);
```

Parameters

htmlHelper [HtmlHelper](#)

message [String](#)

headingTag [String](#)

Returns

[MvcHtmlString](#)

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, String, Object)

Returns an unordered list (ul element) of validation messages in the [ModelStateDictionary](#) object.

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, string message, object htmlAttributes);
```

Parameters

htmlHelper [HtmlHelper](#)

The HTML helper instance that this method extends.

message [String](#)

The message to display if the specified field contains an error.

htmlAttributes [Object](#)

An object that contains the HTML attributes for the element.

Returns

[MvcHtmlString](#)

A string that contains an unordered list (ul element) of validation messages.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, String, IDictionary<String, Object>)

Returns an unordered list (ul element) of validation messages that are in the [ModelStateDictionary](#) object.

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, string message,  
System.Collections.Generic.IDictionary<string,object> htmlAttributes);
```

Parameters

htmlHelper [HtmlHelper](#)

The HTML helper instance that this method extends.

message [String](#)

The message to display if the specified field contains an error.

htmlAttributes [IDictionary<String, Object>](#)

A dictionary that contains the HTML attributes for the element.

Returns

[MvcHtmlString](#)

A string that contains an unordered list (ul element) of validation messages.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, Boolean, String)

Returns an unordered list (ul element) of validation messages that are in the [ModelStateDictionary](#) object and optionally displays only model-level errors.

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, bool excludePropertyErrors, string message);
```

Parameters

htmlHelper [HtmlHelper](#)

The HTML helper instance that this method extends.

excludePropertyErrors [Boolean](#)

true to have the summary display model-level errors only, or false to have the summary display all errors.

message `String`

The message to display with the validation summary.

Returns

`MvcHtmlString`

A string that contains an unordered list (ul element) of validation messages.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, String)

Returns an unordered list (ul element) of validation messages that are in the [ModelStateDictionary](#) object.

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, string message);
```

Parameters

htmlHelper `HtmlHelper`

The HMTL helper instance that this method extends.

message `String`

The message to display if the specified field contains an error.

Returns

`MvcHtmlString`

A string that contains an unordered list (ul element) of validation messages.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, Boolean)

Returns an unordered list (ul element) of validation messages that are in the [ModelStateDictionary](#) object and optionally displays only model-level errors.

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, bool excludePropertyErrors);
```

Parameters

htmlHelper [HtmlHelper](#)

The HTML helper instance that this method extends.

excludePropertyErrors [Boolean](#)

true to have the summary display model-level errors only, or false to have the summary display all errors.

Returns

[MvcHtmlString](#)

A string that contains an unordered list (ul element) of validation messages.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper)

Returns an unordered list (ul element) of validation messages that are in the [ModelStateDictionary](#) object.

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper);
```

Parameters

htmlHelper [HtmlHelper](#)

The HTML helper instance that this method extends.

Returns

[MvcHtmlString](#)

A string that contains an unordered list (ul element) of validation messages.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, Boolean, String, IDictionary<String, Object>, String)

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, bool excludePropertyErrors, string message,  
System.Collections.Generic.IDictionary<string,object> htmlAttributes, string  
headingTag);
```

Parameters

htmlHelper [HtmlHelper](#)

excludePropertyErrors [Boolean](#)

message [String](#)

htmlAttributes [IDictionary<String, Object>](#)

headingTag [String](#)

Returns

[MvcHtmlString](#)

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

ValidationSummary(HtmlHelper, Boolean, String, Object, String)

C#

```
public static System.Web.Mvc.MvcHtmlString ValidationSummary (this  
System.Web.Mvc.HtmlHelper htmlHelper, bool excludePropertyErrors, string message,  
object htmlAttributes, string headingTag);
```

Parameters

htmlHelper [HtmlHelper](#)

excludePropertyErrors Boolean

message String

htmlAttributes Object

headingTag String

Returns

[MvcHtmlString](#)

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

MVC

[Dashboard](#) / My courses / [MVC01](#) / [MVC Model State](#) / [Preparación ejemplo ModelState](#)

Preparación ejemplo ModelState

ViewModels / Home / AddUserVM.cs

```
public class AddUserVM
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }
}
```

Vistas / Inicio / Add.cshtml

```
@model ModelStateDemo.ViewModels.Home.AddUserVM
<h2>Add</h2>

@using(Html.BeginForm())
{
    <div>
        <div>
            @Html.TextBoxFor(x => x.FirstName)
        </div>
        <div>
            @Html.TextBoxFor(x => x.LastName)
        </div>
        <div>
            @Html.TextBoxFor(x => x.EmailAddress)
        </div>
        <div>
            <input type="submit" value="Save" />
        </div>
    </div>
}
```

Controladores / HomeController.cs

```
...
[HttpGet]
public ActionResult Add()
{
    AddUserVM model = new AddUserVM();
    return View(model);
}

[HttpPost]
public ActionResult Add(AddUserVM model)
{
    if(!ModelState.IsValid)
    {
        return View(model);
    }
    return RedirectToAction("Index");
}
```

Cuando enviamos el formulario a la acción POST, todos los valores que ingresamos se mostrarán en la instancia AddUserVM.

MVC

[Dashboard](#) / My courses / [MVC01](#) / [MVC Model State](#) / [La clase ModelStateDictionary](#)

La clase ModelStateDictionary

Veamos el formulario HTML representado para la página Agregar:

```
<form action="/Home/Add" method="post">
    <div>
        <div>
            <label for="FirstName">First Name:</label>
            <input id="FirstName" name="FirstName" type="text" value="">
        </div>
        <div>
            <label for="LastName">Last Name:</label>
            <input id="LastName" name="LastName" type="text" value="">
        </div>
        <div>
            <label for="EmailAddress">Email Address:</label>
            <input id="EmailAddress" name="EmailAddress" type="text" value="">
        </div>
        <div>
            <input type="submit" value="Save">
        </div>
    </div>
</form>
```

En un POST, todos los valores de las `<input>` etiquetas se envían al servidor como pares clave-valor.

Cuando MVC recibe un POST, toma todos los parámetros de publicación y los agrega a una instancia de [ModelStateDictionary](#).

Al depurar la acción POST del controlador en Visual Studio, podemos usar la ventana Locales para investigar este diccionario:

Name	Type
this	ModelStateDemo.Controllers.HomeController
base	System.Object
base	System.Object
ActionInvoker	System.Web.Mvc.Async.AsyncControllerActionInvoker
AsyncManager	System.Web.Mvc.Async.AsyncManager
Binders	System.Web.Mvc.ModelBinderDictionary
DisableAsyncSupport	bool
HttpContext	System.Web.HttpContextWrapper
ModelState	System.Web.Mvc.ModelStateDictionary
Count	int
IsReadOnly	bool
IsValid	bool
Keys	System.Collections.Generic.IEnumerable<string>
[0]	string
[1]	string
[2]	string
Raw View	
Values	System.Collections.Generic.IEnumerable<System.Web.Mvc.ModelState>
[0]	System.Web.Mvc.ModelState
[1]	System.Web.Mvc.ModelState
[2]	System.Web.Mvc.ModelState
Non-Public members	
Results View	Expanding the Results View will enumerate the IEnumerable<System.Web.Profile.DefaultProfile>
Profile	System.Object

La propiedad Valores del ModelStateDictionary contiene instancias que son del tipo [System.Web.Mvc.ModelState](#). ¿Qué contiene realmente un ModelState?

¿Qué hay en un ModelState?

Aquí es cómo se ven esos valores, desde la misma sesión del depurador:

Locals		
Name	Value	Type
Keys	Count = 3 [0] "FirstName" [1] "LastName" [2] "EmailAddress"	System.Collections.Generic.IEnumerable<System.String>
Raw View		
Values	Count = 3 [0] {System.Web.Mvc.ModelState} Errors Count = 0 Value {System.Web.Mvc.ValueProviderResult}	System.Collections.Generic.IEnumerable<System.Object>
[0]	AttemptedValue "Matthew" Culture {en-US} RawValue {string[1]}	System.String
Errors		System.Collections.Generic.IEnumerable<System.Object>
Value	{System.Web.Mvc.ValueProviderResult}	System.Object
[1]	AttemptedValue "Jones" Culture {en-US} RawValue {string[1]}	System.String
Errors		System.Collections.Generic.IEnumerable<System.Object>
Value	{System.Web.Mvc.ValueProviderResult}	System.Object
[2]	AttemptedValue "test@test.com" Culture {en-US} RawValue {string[1]}	System.String
Errors		System.Collections.Generic.IEnumerable<System.Object>
Value	{System.Web.Mvc.ValueProviderResult}	System.Object

Cada una de las propiedades tiene una instancia de [ValueProviderResult](#) que contiene los [valores reales enviados al servidor](#).

MVC crea todas estas instancias automáticamente para nosotros [cuando enviamos un POST con datos](#), y la acción POST tiene entradas que se asignan a los [valores enviados](#). Esencialmente, MVC está envolviendo las entradas del usuario en clases más amigables con el servidor (ModelState y ValueProviderResult) para un uso más sencillo.

Sin embargo, todavía hay dos propiedades importantes que no hemos discutido: la propiedad [ModelState.Errors](#) y la propiedad [ModelStateDictionary.IsValid](#). Se utilizan para la segunda función de ModelState: [para almacenar los errores encontrados en los valores enviados](#).

Last modified: Monday, 1 April 2019, 8:29 AM

◀ Preparación ejemplo ModelState

Jump to...

▶ Errores de validación en ModelState ►

You are logged in as [Leandro Sandoval](#) ([Log out](#))

[MVC01](#)

[Data retention summary](#)

[Get the mobile app](#)

ModelStateDictionary Class

Reference

Definition

Namespace: [System.Web.Mvc](#)

Assembly: System.Web.Mvc.dll

Package: Microsoft.AspNet.Mvc v5.2.6

Represents the state of an attempt to bind a posted form to an action method, which includes validation information.

C#

```
[System.Serializable]
public class ModelStateDictionary :
    System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePa-
    ir<string, System.Web.Mvc.ModelState>>,
    System.Collections.Generic.IDictionary<string, System.Web.Mvc.ModelState>,
    System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePa-
    ir<string, System.Web.Mvc.ModelState>>
```

Inheritance [Object](#) → [ModelStateDictionary](#)

Attributes [SerializableAttribute](#)

Implements [ICollection<KeyValuePair<String,ModelState>>](#) ,
[ICollection<KeyValuePair<TKey,TValue>>](#) ,
[IDictionary<String,ModelState>](#) ,
[IEnumerable<KeyValuePair<String,ModelState>>](#) ,
[IEnumerable<KeyValuePair<TKey,TValue>>](#) , [IEnumerable<T>](#) ,
[IEnumerable](#)

Constructors

ModelStateDictionary()	Initializes a new instance of the ModelStateDictionary class.
ModelStateDictionary(ModelStateDictionary)	Initializes a new instance of the ModelStateDictionary class by using values that are copied from the specified model-state

dictionary.

Properties

Count	Gets the number of key/value pairs in the collection.
IsReadOnly	Gets a value that indicates whether the collection is read-only.
IsValid	Gets a value that indicates whether this instance of the model-state dictionary is valid.
Item[String]	Gets or sets the value that is associated with the specified key.
Keys	Gets a collection that contains the keys in the dictionary.
Values	Gets a collection that contains the values in the dictionary.

Methods

Add(KeyValue Pair<String,ModelState>)	Adds the specified item to the model-state dictionary.
Add(String, ModelState)	Adds an element that has the specified key and value to the model-state dictionary.
AddModelError(String, Exception)	Adds the specified model error to the errors collection for the model-state dictionary that is associated with the specified key.
AddModelError(String, String)	Adds the specified error message to the errors collection for the model-state dictionary that is associated with the specified key.
Clear()	Removes all items from the model-state dictionary.
Contains(KeyValue Pair<String,ModelState>)	Determines whether the model-state dictionary contains a specific value.
ContainsKey(String)	Determines whether the model-state dictionary contains the specified key.
CopyTo(KeyValue Pair<String,ModelState>[], Int32)	Copies the elements of the model-state dictionary to an array, starting at a specified index.
GetEnumerator()	Returns an enumerator that can be used to iterate through the collection.
IsValidField(String)	Determines whether there are any ModelError objects that are associated with or prefixed with the specified key.

Merge(ModelStateDictionary)	Copies the values from the specified ModelStateDictionary object into this dictionary, overwriting existing values if keys are the same.
Remove(KeyValuePair<String,ModelState>)	Removes the first occurrence of the specified object from the model-state dictionary.
Remove(String)	Removes the element that has the specified key from the model-state dictionary.
SetModelValue(String, ValueProviderResult)	Sets the value for the specified key by using the specified value provider dictionary.
TryGetValue(String, Model State)	Attempts to gets the value that is associated with the specified key.

Explicit Interface Implementations

IEnumerable.GetEnumerator()	Returns an enumerator that can be used to iterate through the collection.
---	---

Applies to

Product	Versions
ASP.NET MVC	5.2

Feedback

Was this page helpful?

 Yes

 No

MVC

[Dashboard](#) / My courses / [MVC01](#) / [MVC Model State](#) / [Errores de validación en ModelState](#)

Errores de validación en ModelState

Vamos a cambiar nuestra clase AddUserVM:

```
public class AddUserVM
{
    [Required(ErrorMessage = "Please enter the user's first name.")]
    [StringLength(50, ErrorMessage = "The First Name must be less than {1} characters.")]
    [Display(Name = "First Name")]
    public string FirstName { get; set; }

    [Required(ErrorMessage = "Please enter the user's last name.")]
    [StringLength(50, ErrorMessage = "The Last Name must be less than {1} characters.")]
    [Display(Name = "Last Name")]
    public string LastName { get; set; }

    [EmailAddress(ErrorMessage = "The Email Address is not valid")]
    [Required(ErrorMessage = "Please enter an email address.")]
    [Display(Name = "Email Address")]
    public string EmailAddress { get; set; }
}
```

Hemos agregado atributos de validación, específicamente [Requerido](#), [StringLength](#) y [EmailAddress](#).

También hemos establecido los mensajes de error que se mostrarán si se producen los errores de validación correspondientes.

Con los cambios anteriores en su lugar, modifiquemos la vista Agregar para mostrar los mensajes de error si ocurren:

```
@model ModelStateDemo.ViewModels.Home.AddUserVM

<h2>Add</h2>

@using(Html.BeginForm())
{
    @Html.ValidationSummary()
    <div>
        <div>
            @Html.LabelFor(x => x.FirstName)
            @Html.TextBoxFor(x => x.FirstName)
            @Html.ValidationMessageFor(x => x.FirstName)
        </div>
        <div>
            @Html.LabelFor(x => x.LastName)
            @Html.TextBoxFor(x => x.LastName)
            @Html.ValidationMessageFor(x => x.LastName)
        </div>
        <div>
            @Html.LabelFor(x => x.EmailAddress)
            @Html.TextBoxFor(x => x.EmailAddress)
            @Html.ValidationMessageFor(x => x.EmailAddress)
        </div>
        <div>
            <input type="submit" value="Save" />
        </div>
    </div>
}
```

Observe los dos ayudantes que estamos usando ahora, [ValidationSummary](#) y [ValidationMessageFor](#).

- [ValidationSummary](#) lee todos los errores del estado del modelo y los muestra en una lista con viñetas.
- [ValidationMessageFor](#) muestra solo errores para la propiedad especificada.

Veamos qué sucede cuando intentamos enviar un POST no válido que falta la dirección de correo electrónico. Cuando llegamos a la acción POST durante la depuración, tenemos los siguientes valores en nuestro ModelStateDictionary:

Name	Type
ModelState	System.Web.Mvc.ModelStateDictionary
Count	int
IsReadOnly	bool
IsValid	bool
Keys	System.Collections.Generic.IEnumerable<string>
[0]	string
[1]	string
[2]	string
Raw View	
Values	System.Collections.Generic.IEnumerable<System.Web.Mvc.ValueProviderResult>
[0]	System.Web.Mvc.ValueProviderResult
Errors	System.Collections.Generic.IEnumerable<System.Web.Mvc.ModelError>
Value	System.Web.Mvc.ValueProviderResult
Non-Public members	
[1]	System.Web.Mvc.ValueProviderResult
Errors	System.Collections.Generic.IEnumerable<System.Web.Mvc.ModelError>
Value	System.Web.Mvc.ValueProviderResult
Non-Public members	
[2]	System.Web.Mvc.ValueProviderResult
Errors	System.Collections.Generic.IEnumerable<System.Web.Mvc.ModelError>
Value	System.Web.Mvc.ValueProviderResult
Non-Public members	
Raw View	
Non-Public members	

Tenga en cuenta que la instancia de ModelState para la dirección de correo electrónico ahora tiene un error en la colección de Errores.

Cuando MVC crea el estado del modelo para las propiedades enviadas, también pasa por cada propiedad en el Modelo de Vista y valida la propiedad usando atributos asociados a ella. Si se encuentra algún error, se agrega a la colección de Errores en el ModelState de la propiedad.

También tenga en cuenta que IsValid es falso ahora. Eso es porque existe un error; IsValid es falso si alguna de las propiedades enviadas tiene algún mensaje de error adjunto.

Lo que todo esto significa es que al configurar la validación de esta manera, permitimos que MVC funcione de la forma en que fue diseñado.

ModelState almacena los valores enviados, permite que se asignen a propiedades de clase (o simplemente como parámetros a la acción) y mantiene una colección de mensajes de error para cada propiedad. En escenarios simples, esto es todo lo que necesitamos, ¡y todo esto está sucediendo detrás de escena!

Last modified: Monday, 1 April 2019, 8:32 AM

[◀ La clase ModelStateDictionary](#)

[Jump to...](#)

[Validación personalizada ►](#)

You are logged in as [Leandro Sandoval \(Log out\)](#)

[MVC01](#)

[Data retention summary](#)

[Get the mobile app](#)

MVC

[Dashboard](#) / My courses / [MVC01](#) / [MVC Model State](#) / [Validación personalizada](#)

Validación personalizada

De hecho, podemos añadir errores al estado a través del modelo de [AddModelError](#) método de ModelStateDictionary:

```
[HttpPost]
public ActionResult Add(AddUserVM model)
{
    if(model.FirstName == model.LastName)
    {
        ModelState.AddModelError("LastName", "The last name cannot be the same as the first name.");
    }
    if(!ModelState.IsValid)
    {
        return View(model);
    }
    return RedirectToAction("Index");
}
```

El primer parámetro del método AddModelError es el nombre de la propiedad a la que se aplica el error. En este caso, lo configuramos como Apellido. También puede establecerlo en nada (o en un nombre falso) si solo desea que aparezca en el [resumen](#) de validación y no en un mensaje de validación.

Last modified: Monday, 1 April 2019, 8:34 AM

◀ [Errores de validación en ModelState](#)

Jump to...



[ModelState.Clear\(\) ¿Qué es? ►](#)

MVC

[Dashboard](#) / My courses / [MVC01](#) / [MVC Model State](#) / [ModelState.Clear\(\)...¿Que es?](#)

ModelState.Clear() ¿Que es?

Si obtiene su modelo de un formulario y desea manipular los datos que provienen del formulario del cliente y escribirlos en una vista, debe llamar a `ModelState.Clear()` para limpiar los valores de `ModelState`.

El motivo es que, normalmente, desea devolver al cliente el formulario con todos los errores. Entonces, cuando devuelve el parámetro que contiene su modelo a la vista que se devolverá, este usará el valor de `ModelState`.

Entonces, por ejemplo, si cambio una propiedad y la envío al cliente:

```
1 [HttpPost]
2 public ActionResult Edit(MyObject objModel)
3 {
4     objModel.Property1 = "NEW VALUE";
5     //...
6     return View(objModel)
7 }
```

Esto no pondrá en la interfaz de usuario el nuevo valor para la propiedad1 porque los valores de `ModelState` no contienen este valor sino el que ingresó el usuario.

Para poder anular el estado del modelo, debe eliminar todos los datos del mismo.

```
1 [HttpPost]
2 public ActionResult Edit(MyObject objModel)
3 {
4     objModel.Property1 = "NEW VALUE";
5     //...
6     ModelState.Clear();
7     return View(objModel)
8 }
```

Para borrar la memoria del estado del modelo, debe usar `ModelState.Clear()`. También puede eliminar solo el campo deseado utilizando el método de `ModelState`.

```
1 ModelState.Remove("Property1");
```

Además, si siempre desea no usar `ModelState`, es posible que no quiera usar `HtmlHelper` y directamente use el modelo con código `Html`.

```
1 My Property: <input type="text" name="Property1" value="@Model.Property1" />
```

En cualquier situación, lo que debe recordar es que `ModelState` es el mecanismo predeterminado y, por defecto, será el que se usará para mostrar la información al formulario.

MVC

[Dashboard](#) / My courses / [MVC01](#) / [MVC Model State](#) / [Resumen](#)

Resumen

- 1) El ModelState representa los valores enviados y los errores en dichos valores durante un **POST**.
- 2) El proceso de validación respeta los atributos como Required y EmailAddress (y otros), y podemos agregar errores personalizados a la validación si así lo deseamos.
- 3) Utilizamos las clases **ValidationSummary** and **ValidationMessage** para leer directamente desde ModelState para mostrar los errores al usuario.
- 4) Cuando MVC crea el estado del modelo para las propiedades enviadas, también pasa por cada propiedad en el Modelo de Vista y valida la propiedad usando atributos asociados a ella. Si se encuentra algún error, se agrega a la colección de Errores en el ModelState de la propiedad.
- 5) Cada una de las propiedades tiene una instancia de **ValueProviderResult** que contiene los valores reales enviados al servidor. MVC crea todas estas instancias automáticamente para nosotros cuando enviamos un POST con datos, y la acción POST tiene entradas que se asignan a los valores enviados. Esencialmente, **MVC está envolviendo las entradas del usuario en clases más amigables con el servidor**(ModelState y ValueProviderResult) para un uso más sencillo.
- 6) Si obtiene su modelo de un formulario y desea manipular los datos que provienen del formulario del cliente y escribirlos en una vista, debe llamar a ModelState.Clear () para limpiar los valores de ModelState.
- 7) [Html.ActionLink](#) puede ser utilizado para navegar a diferentes controladores, con y sin parámetros.

Last modified: Thursday, 4 April 2019, 1:57 PM

[◀ ModelState.Clear\(\) ¿Que es?](#)

Jump to...

[View\(\) vs RedirectToAction\(\) vs Redirect\(\)](#) ►

MVC

[Dashboard](#) / My courses / [MVC01](#) / [MVC Routing](#) / [View\(\) vs RedirectToAction\(\) vs Redirect\(\)](#)

View() vs RedirectToAction() vs Redirect()

"...[View() vs RedirectToAction() vs Redirect()]..."

Hay muchas formas de devolver o representar una vista en ASP.NET MVC.

Muchos desarrolladores se confundieron cuando usaron los métodos View (), RedirectToAction (), Redirect () y RedirectToRoute ().

View()

Este método **se utiliza para generar el contenido HTML que se muestra para la vista especificada y lo envía al navegador**

RedirectToAction()

Este método se utiliza para **redirigir a la acción especificada en lugar de representar el HTML.**

En este caso, **el navegador recibe la notificación de redirección y realiza una nueva solicitud para la acción especificada.** (Esto actúa como Response.Redirect () en ASP.NET WebForm)

Además, RedirectToAction **construye una url de redireccionamiento a una acción / controlador específico en su aplicación** y usa la tabla de rutas para generar la URL correcta.

RedirectToAction hace que el navegador reciba un redireccionamiento 302 dentro de su aplicación y le brinda una manera más fácil de trabajar con su tabla de rutas.

Redirect()

Este método **se utiliza para redirigir a la URL especificada en lugar de representar HTML.** En este caso, **el navegador recibe la notificación de redireccionamiento y realiza una nueva solicitud para la URL especificada.** (Esto también actúa como Response.Redirect () en Asp.Net WebForm)

En este caso, **debe especificar la URL completa para redirigir**

Además, Redirect también hace que el navegador reciba un redireccionamiento 302 dentro de su aplicación, pero **usted mismo debe crear las URL.**

Controller.Redirect(String) Method

Reference

Definition

Namespace: [System.Web.Mvc](#)

Assembly: System.Web.Mvc.dll

Package: Microsoft.AspNet.Mvc v5.2.6

Creates a [RedirectResult](#) object that redirects to the specified URL.

C#

```
protected internal virtual System.Web.Mvc.RedirectResult Redirect (string url);
```

Parameters

url [String](#)

The URL to redirect to.

Returns

[RedirectResult](#)

The redirect result object.

Applies to

Product	Versions
ASP.NET MVC	5.2

Feedback

Was this page helpful?

 Yes

 No

Controller.RedirectToAction Method

Reference

Definition

Namespace: [System.Web.Mvc](#)

Assembly: System.Web.Mvc.dll

Package: Microsoft.AspNet.Mvc v5.2.6

Overloads

RedirectToAction(String)	Redirects to the specified action using the action name.
RedirectToAction(String, Object)	Redirects to the specified action using the action name and route values.
RedirectToAction(String, String)	Redirects to the specified action using the action name and controller name.
RedirectToAction(String, RouteValueDictionary)	Redirects to the specified action using the action name and route dictionary.
RedirectToAction(String, String, Object)	Redirects to the specified action using the action name, controller name, and route dictionary.
RedirectToAction(String, String, RouteValueDictionary)	Redirects to the specified action using the action name, controller name, and route values.

RedirectToAction(String)

Redirects to the specified action using the action name.

C#

```
protected internal System.Web.Mvc.RedirectToRouteResult RedirectToAction(string actionName);
```

Parameters

actionName [String](#)

The name of the action.

Returns

[RedirectToRouteResult](#)

The redirect result object.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

RedirectToAction(String, Object)

Redirects to the specified action using the action name and route values.

C#

```
protected internal System.Web.Mvc.RedirectToRouteResult RedirectToAction(string actionName, object routeValues);
```

Parameters

actionName [String](#)

The name of the action.

routeValues [Object](#)

The parameters for a route.

Returns

[RedirectToRouteResult](#)

The redirect result object.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

RedirectToAction(String, String)

Redirects to the specified action using the action name and controller name.

C#

```
protected internal System.Web.Mvc.RedirectToRouteResult RedirectToAction(string actionName, string controllerName);
```

Parameters

actionName `String`

The name of the action.

controllerName `String`

The name of the controller.

Returns

`RedirectToRouteResult`

The redirect result object.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

RedirectToAction(String, RouteValueDictionary)

Redirects to the specified action using the action name and route dictionary.

C#

```
protected internal System.Web.Mvc.RedirectToRouteResult RedirectToAction(string actionName, System.Web.Routing.RouteValueDictionary routeValues);
```

Parameters

actionName [String](#)

The name of the action.

routeValues [RouteValueDictionary](#)

The parameters for a route.

Returns

[RedirectToRouteResult](#)

The redirect result object.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

RedirectToAction(String, String, Object)

Redirects to the specified action using the action name, controller name, and route dictionary.

C#

```
protected internal System.Web.Mvc.RedirectToRouteResult RedirectToAction(string actionName, string controllerName, object routeValues);
```

Parameters

actionName [String](#)

The name of the action.

controllerName [String](#)

The name of the controller.

routeValues [Object](#)

The parameters for a route.

Returns

[RedirectToRouteResult](#)

The redirect result object.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

RedirectToAction(String, String, RouteValueDictionary)

Redirects to the specified action using the action name, controller name, and route values.

C#

```
protected internal virtual System.Web.Mvc.RedirectToRouteResult  
RedirectToAction (string actionName, string controllerName,  
System.Web.Routing.RouteValueDictionary routeValues);
```

Parameters

actionName [String](#)

The name of the action.

controllerName [String](#)

The name of the controller.

routeValues [RouteValueDictionary](#)

The parameters for a route.

Returns

[RedirectToRouteResult](#)

The redirect result object.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

Feedback

Was this page helpful?



Controller.View Method

Reference

Definition

Namespace: [System.Web.Mvc](#)

Assembly: System.Web.Mvc.dll

Package: Microsoft.AspNet.Mvc v5.2.6

Overloads

View(String, String, Object)	Creates a ViewResult object using the view name, master-page name, and model that renders a view.
View()	Creates a ViewResult object that renders a view to the response.
View(Object)	Creates a ViewResult object by using the model that renders a view to the response.
View(String)	Creates a ViewResult object by using the view name that renders a view.
View(IView)	Creates a ViewResult object that renders the specified IView object.
View(String, Object)	Creates a ViewResult object that renders the specified IView object.
View(String, String)	Creates a ViewResult object using the view name and master-page name that renders a view to the response.
View(IView, Object)	Creates a ViewResult object that renders the specified IView object.

View(String, String, Object)

Creates a [ViewResult](#) object using the view name, master-page name, and model that renders a view.

C#

```
protected internal virtual System.Web.Mvc.ViewResult View (string view-
```

```
Name, string masterName, object model);
```

Parameters

viewName [String](#)

The name of the view that is rendered to the response.

masterName [String](#)

The name of the master page or template to use when the view is rendered.

model [Object](#)

The model that is rendered by the view.

Returns

[ViewResult](#)

The view result.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

View()

Creates a [ViewResult](#) object that renders a view to the response.

C#

```
protected internal System.Web.Mvc.ViewResult View();
```

Returns

[ViewResult](#)

The [View\(\)](#) result that renders a view to the response.

Applies to

- ▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

View(Object)

Creates a [ViewResult](#) object by using the model that renders a view to the response.

C#

```
protected internal System.Web.Mvc.ViewResult View (object model);
```

Parameters

model Object

The model that is rendered by the view.

Returns

[ViewResult](#)

The view result.

Applies to

- ▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

View(String)

Creates a [ViewResult](#) object by using the view name that renders a view.

C#

```
protected internal System.Web.Mvc.ViewResult View (string viewName);
```

Parameters

viewName [String](#)

The name of the view that is rendered to the response.

Returns

[ViewResult](#)

The view result.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

View(IView)

Creates a [ViewResult](#) object that renders the specified IView object.

C#

```
protected internal System.Web.Mvc.ViewResult View (System.Web.Mvc.IView  
view);
```

Parameters

view [IView](#)

The view that is rendered to the response.

Returns

[ViewResult](#)

The view result.

Applies to

- ▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

View(String, Object)

Creates a [ViewResult](#) object that renders the specified `IView` object.

C#

```
protected internal System.Web.Mvc.ViewResult View (string viewName, object model);
```

Parameters

viewName `String`

The view that is rendered to the response.

model `Object`

The model that is rendered by the view.

Returns

[ViewResult](#)

The view result.

Applies to

- ▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

View(String, String)

Creates a [ViewResult](#) object using the view name and master-page name that renders a view to the response.

C#

```
protected internal System.Web.Mvc.ViewResult View (string viewName,  
string masterName);
```

Parameters

viewName [String](#)

The name of the view that is rendered to the response.

masterName [String](#)

The name of the master page or template to use when the view is rendered.

Returns

[ViewResult](#)

The view result.

Applies to

▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

View(IView, Object)

Creates a [ViewResult](#) object that renders the specified [IView](#) object.

C#

```
protected internal virtual System.Web.Mvc.ViewResult View  
(System.Web.Mvc.IView view, object model);
```

Parameters

view [IView](#)

The view that is rendered to the response.

model [Object](#)

The model that is rendered by the view.

Returns

[ViewResult](#)

The view result.

Applies to

- ▼ ASP.NET MVC 5.2

Product	Versions
ASP.NET MVC	5.2

Feedback

Was this page helpful?

 Yes

 No

MVC

[Dashboard](#) / My courses / [MVC01](#) / [MVC Routing](#) / [Html.ActionLink](#)

Html.ActionLink

Aquí se muestra como incorporar un link (tag <A> de html) mediante la clase Html helper de MVC Razor.

Para enviar parámetros al controlador, es posible hacerlo definiendo en el controlador los mismos nombres de parámetros que tiene el Html Link; o bien utilizando en el Html Link los mismos nombres de propiedades de alguna clase modelo utilizada por la acción.

Parámetro por parámetro....

EN LA VISTA.....

```
@Html.ActionLink("Este es el link con parámetros", "MiAccion2", new { parametro1 = "Hola Mundo!!!", parametro2 = "¿Todo bien?" })
```

EN EL CONTROLADOR

```
[HttpGet]
public ActionResult MiAccion2(string parametro1, string parametro2)
{
    var mensaje = parametro1 + " " + parametro2;
    mensaje // "Hola Mundo!!! ¿Todo bien?"

    /* ... a partir de aquí puedo retornar con o sin modelo, a la misma vista
     * o redireccionar hacia otra
     */
    return View();
}
```

Reutilizando una clase de modelo

EN LA VISTA.....

```
@Html.ActionLink("Este es el link con parámetros", "MiAccion2", new { parametro1 = "Hola Mundo!!!", parametro2 = "¿Todo bien?" })
```

EN EL CONTROLADOR

```
[HttpGet]
public ActionResult MiAccion2(MiAccion2Modelo modelo)
{
    var mensaje = modelo.parametro1 + " " + modelo.parametro2;
    mensaje // "Hola Mundo!!! ¿Todo bien?"

    /* ... a partir de aquí puedo retornar con o sin modelo, a la misma vista
     * o redireccionar hacia otra
     */
    return View();
}
```

MVC

[Dashboard](#) / My courses / [MVC01](#) / [MVC Routing](#) / [Resumen MVC Rounting](#)

Resumen MVC Rounting

IMPORTANTE

- 1) El método `View()` no realiza nuevas solicitudes, simplemente representa la vista sin cambiar las URL en la barra de direcciones del navegador.
- 2) El método `RedirectToAction()` realiza una nueva solicitud y la URL en la barra de direcciones del navegador se actualiza con la URL generada por MVC.
- 3) El método `Redirect()` también realiza una nueva solicitud y se actualiza la URL en la barra de direcciones del navegador, pero debe especificar la URL completa para redirigir.
- 4) Entre los métodos `RedirectToAction()` y `Redirect()`, la mejor práctica es usar `RedirectToAction()` para cualquier cosa relacionada con las acciones / controladores de su aplicación.
- 5) Si usa `Redirect()` y proporciona la URL, deberá modificar esas URL manualmente cuando cambie la tabla de ruta.

Last modified: Monday, 1 April 2019, 9:00 AM

◀ [Html.ActionLink](#)

Jump to...



[Conceptos MVC ▶](#)