

Introducción a ASP.NET MVC 4

El **ASP.NET MVC Framework** es un framework de aplicaciones web que implementa el patrón modelo-vista-controlador (**MVC**).

Basado en ASP.NET, permite a los desarrolladores de software construir una aplicación web como una composición de tres funciones: Modelo, Vista y Controlador.

En marzo de 2009 se hizo pública la primera versión de ASP.NET MVC. El patrón de arquitectura MVC (model-view-controller) no es nuevo (data de 1979) ni es algo que haya inventado Microsoft. Existen muchos frameworks de desarrollo web populares que utilizan MVC, como por ejemplo Ruby on Rails, Spring o Apache Struts. MVC es un patrón de arquitectura que ayuda a crear una separación lógica entre el modelo (la lógica de acceso a datos), la vista (la lógica de presentación) y el controlador (la lógica de negocio).

Uno de los pilares básicos de ASP.NET MVC es el concepto de enrutamiento (routing), lo que permite a las aplicaciones aceptar peticiones a URLs que no se corresponden con ficheros físicos en el servidor. Por ejemplo, en ASP.NET Web Forms las URLs tienen el siguiente formato "<http://website/products.aspx?category=dvd>" en el que físicamente existe un fichero products.aspx en la raíz del sitio web.

En MVC la misma URL tendría el siguiente aspecto "<http://website/products/dvd>" sin que el servidor web necesariamente contenga una carpeta products con una subcarpeta dvd. De forma predeterminada, ASP.NET MVC enruta las peticiones al controlador y a la vista adecuada en función de la URL. Es decir, en el ejemplo anterior, nos devolverá la vista dvd del controlador products.

ASP.NET MVC es una implementación del patrón de diseño *MVC*, el cual separa la aplicación en tres componente, el modelo la vista y el controlador.

Partes

Vista: Es la representación de los datos, el ejemplo más claro en una aplicación web es el *HTML* de la pantalla, pero podría ser tranquilamente un *JSON* que expone una *API*.

Modelo: Es todo el dominio de la aplicación, con sus datos, su comportamiento y todo lo necesario para solucionar el problema de negocios, sin necesidad de ninguno de los otros dos componentes.

Controlador: Simplemente vincula la vista y el modelo, se encarga de recibir las peticiones del usuario (en una aplicación web serían las peticiones *HTTP*, por ejemplo cuando un usuario coloca una dirección URL o presiona un link, en el ejemplo de la *API* la invocación de un método de la misma) y retornar el resultado del modelo a la vista, de esto podemos sacar conclusiones importantes:

- El modelo puede (y debería) funcionar sin la vista ni el controlador.
- La vista no conoce el modelo, sino que recibe datos del controlador. (no siempre los datos que recibe una vista son objetos del modelo)
- El modelo no recibe nunca una petición directa de la vista, todo llega a través del controlador.

- El modelo representa todo el negocio.
- Nunca debería existir lógica de negocios en el controlador.

Otras características

Gran extensibilidad

Vamos a ver que el *framework* está pensado de una manera en que todo se puede cambiar, si bien casi siempre los componentes que trae resuelven nuestro problema, para aquellos casos en que no sea así o queramos algo especial podemos hacerlo gracias a su capacidad de extensibilidad.

Convención sobre configuración

En general se busca minimizar la configuración necesaria y muchas del comportamiento se define por convención, por ejemplo: cuando se escribe la *URL*.

<http://sitio.com/usuarios/list>

ASP.NET MVC sabe que tiene que instanciar el controlador *UsuariosController* e invocar el método *List*, ya que así está definido por convención.

Hay muchas convenciones, las vamos a ir viendo a su tiempo, de todos modos casi siempre es posible modificar este comportamiento por defecto.

Motor de vistas

Una de las tres partes del patrón es la vista, existe una sintaxis especial programar en la vista (además del *HTML*) en el caso de *ASP.NET MVC* es posible cambiar el motor que interpreta lo que escribimos en la vista, por defecto tenemos dos, el mismo de *Webforms* que se llama *ASP.NET* y uno especialmente pensado para *MVC* que se llama *Razor*.

Motor de rutas

La forma de saber qué controlador será invocado cuando el usuario escriba una *URL* en el navegador es definida por el motor de ruteo, es muy flexible y permite hacer cosas muy interesantes que ayudan a la organización y a [*SEO*](#)

HTML Helpers

Dentro de la vista no tenemos la posibilidad de usar controles como en *Webforms* pero podemos utilizar los llamados *HTMLHelpers* que son algo similar y permiten hacer cosas como generar el *HTML* de un combo (elemento select de *HTML*) a partir de un *IEnumerable<T>*, también podemos crear nuestros propios *HTMLHelpers*.

Action Filters

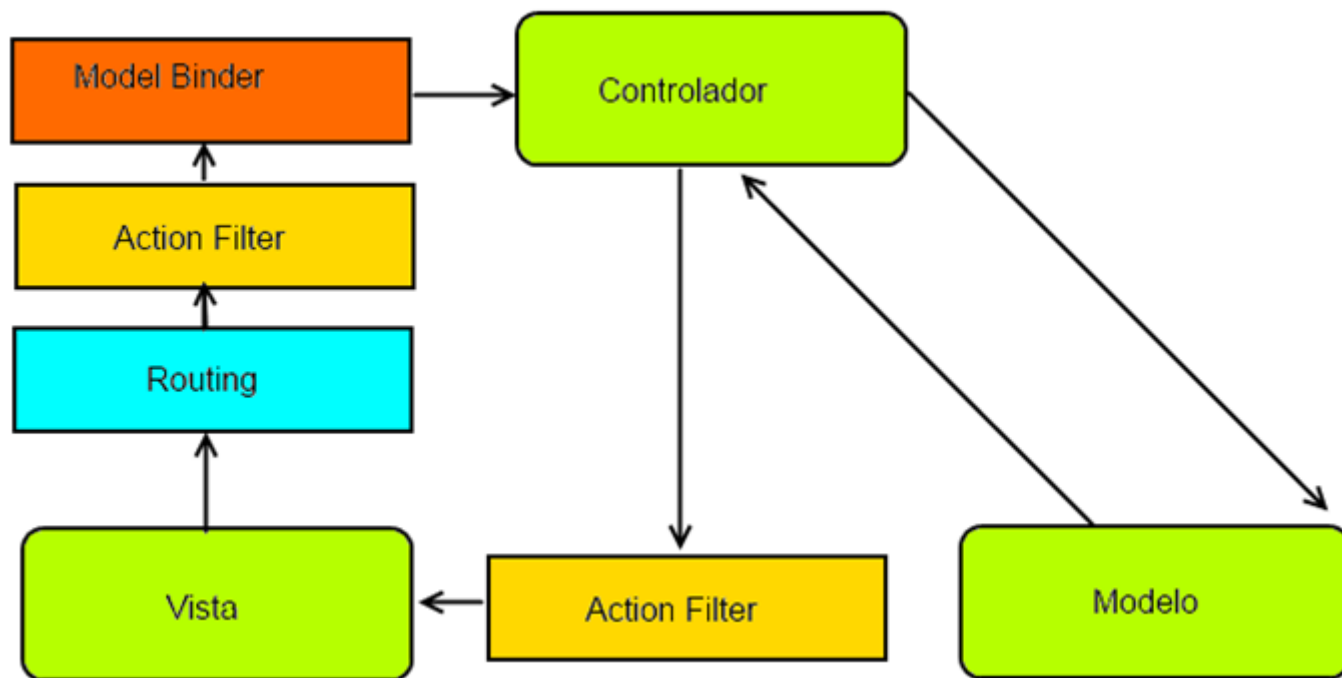
Las acciones del usuario llegan al controlador, esto lo sabemos, pero existe otro componente que recibe antes las invocaciones y también puede modificar las respuestas del controlador, se llaman *intercepting filters* según la definición del patrón *MVC*, en el caso de *ASP.NET MVC* se los llama *Action Filters* y permite hacer cosas como decorar un método del un controlador con una atributo y que esto impida que se llame al método sin tener una cookie dada ya que la llamada es recibida primero por el *Action Filter*, también podemos manipular la respuesta del controlador y agregarle un *META TAG* al *HTML* resultante o cualquier otra cosa que necesitemos.

Model Binder

Este componente se encarga de recibir los datos enviados por el usuario (tanto por *GET* como por *POST*, etc.) y generar un objeto a partir de ellos, es decir, si un controlador espera que el usuario le envíe un objeto "Mensaje" el *model binder* va a intentar crear uno y llenarlo con los datos que se enviaron en el request *HTTP*, vamos a verlo en más detalle después pero es casi mágico.

Action Result

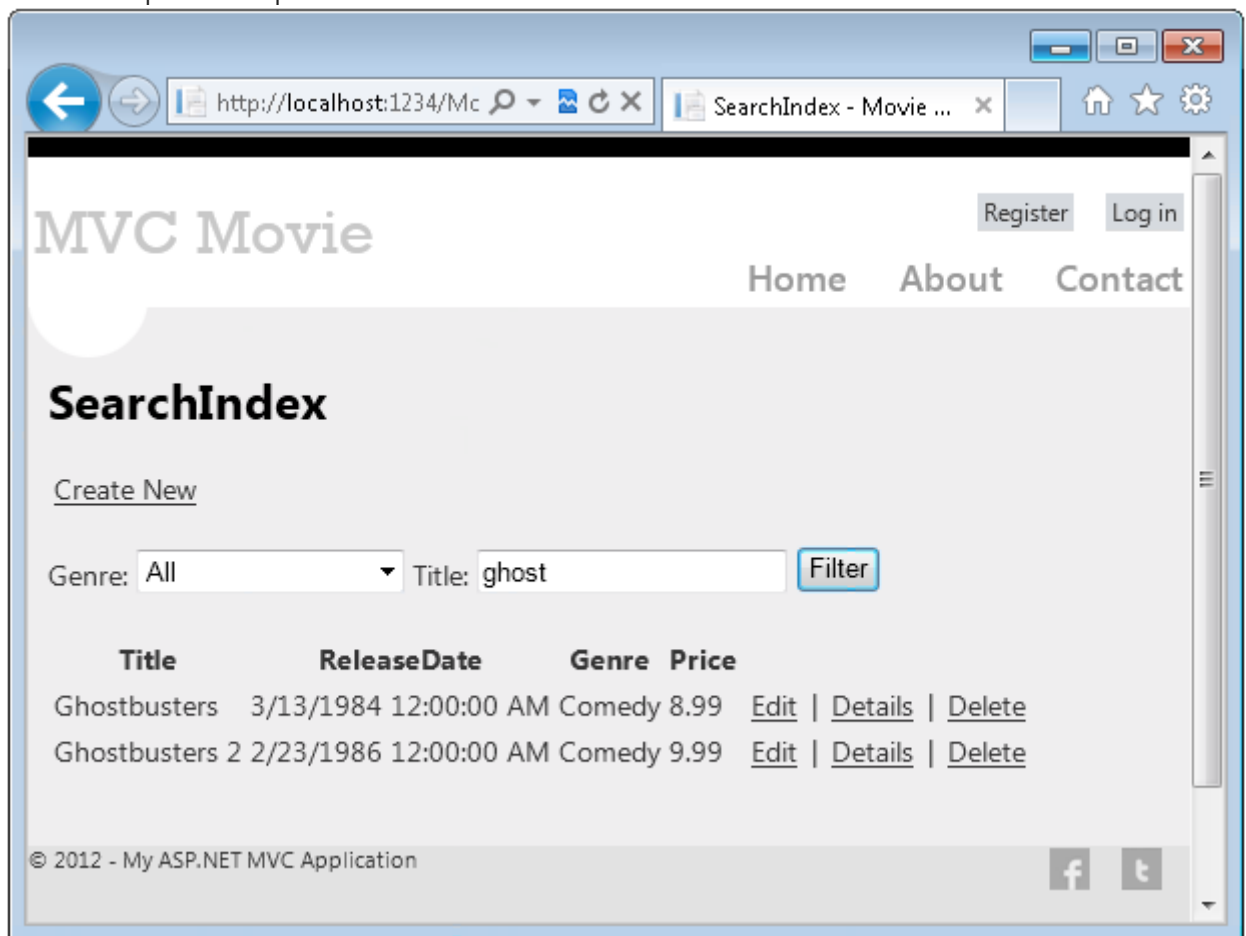
Cuando un controlador consulta al modelo y éste devuelve un resultado, está listo para mostrarlo al usuario, la respuesta del controlador siempre se basa en un *ActionResult*, que no es más que un tipo que representa una respuesta *HTTP*, todos los tipos de resultados (o casi) heredan de *ActionResult*, el más común es el *ViewResult* que retorna una representación *HTML*, pero hay otros como por ejemplo *JSONResult* que devuelve un *JSON*, de nuevo, vamos a verlo más en detalle después.



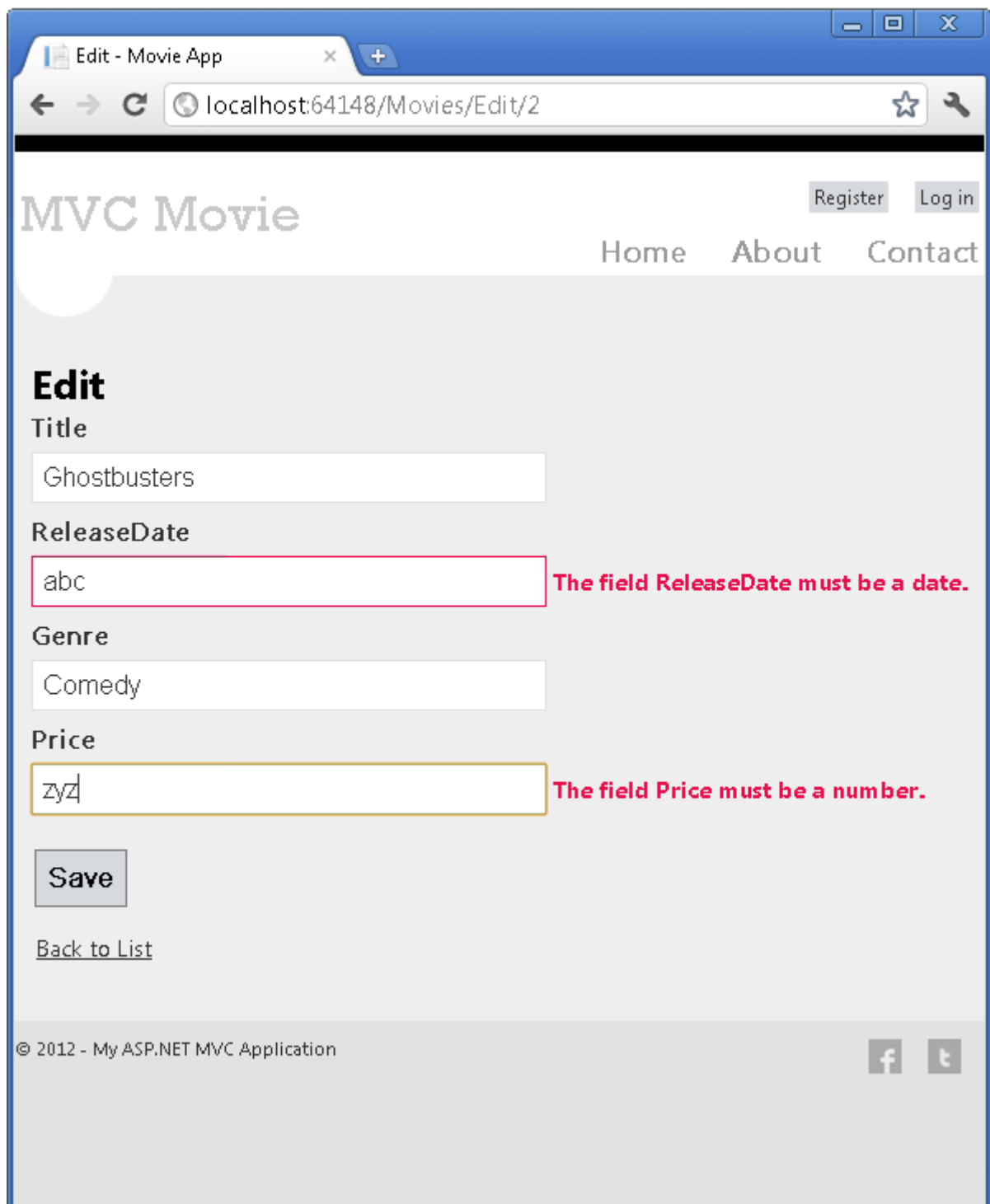
El modelo quedaría más o menos así (simplificado y levemente modificado para mejor comprensión)

Pasos para construir una aplicación Asp net MVC

Vamos a implementar una aplicación sencilla película-lista que apoya la creación, edición, búsqueda y una lista de películas a partir de una base de datos. A continuación se presentan dos capturas de pantalla de la aplicación que va a construir. Incluye una página que muestra una lista de películas a partir de una base de datos:



La aplicación también te permite añadir, editar y borrar películas, así como ver los detalles sobre los individuales. Todos los escenarios de entrada de datos incluyen la validación para asegurar que los datos almacenados en la base de datos es correcta.

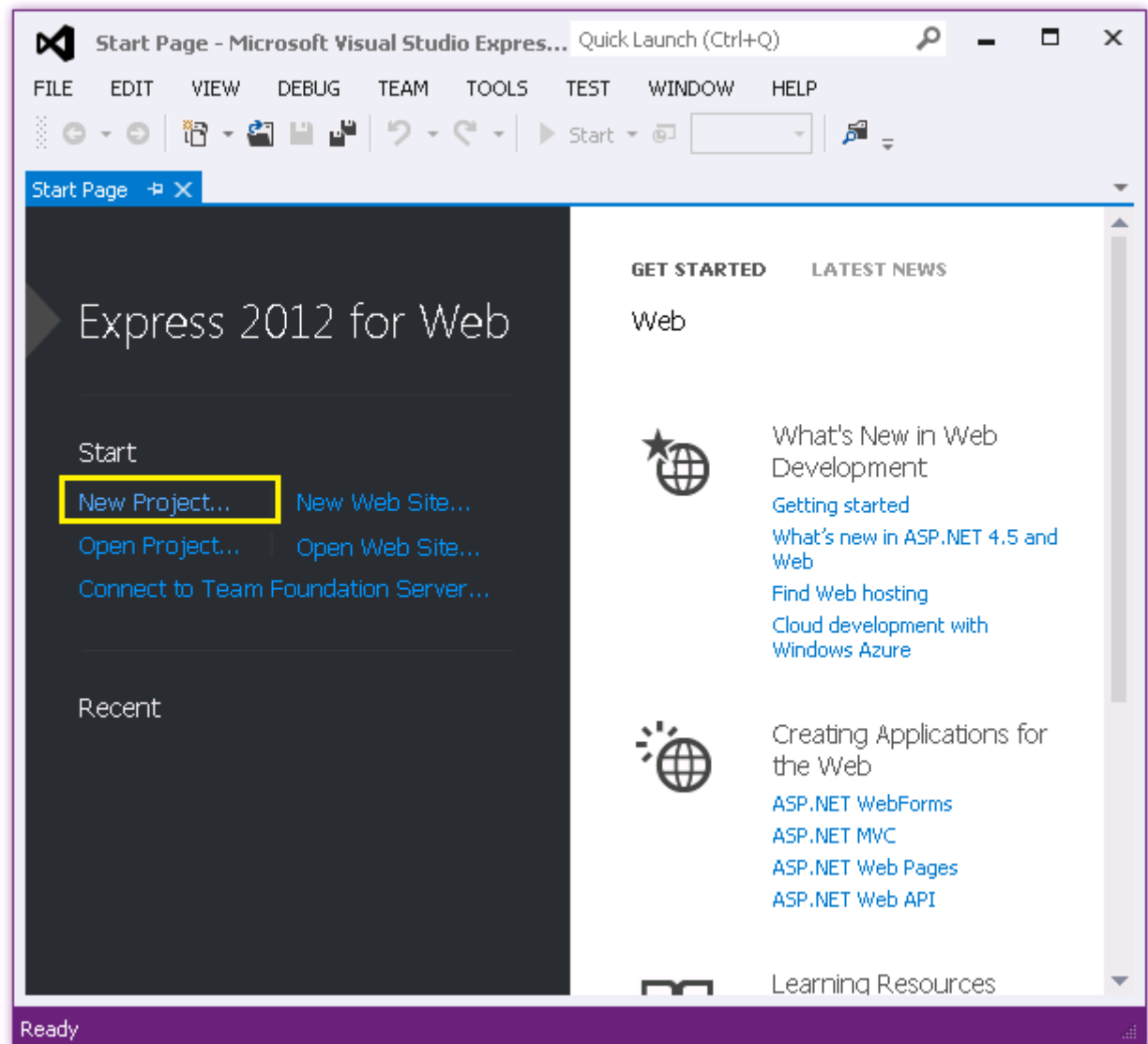


Empezando

Comience por ejecutar Visual Studio Express 2012 o Visual Web Developer 2010 Express. La mayor parte de las capturas de pantalla de esta serie utilizan Visual Studio Express 2012, pero se puede completar este tutorial con Visual Studio 2010 / SP1, Visual Studio 2012, Visual Studio Express 2012 o Visual Web Developer 2010 Express. Seleccione **Nuevo proyecto** en la página de **inicio**.

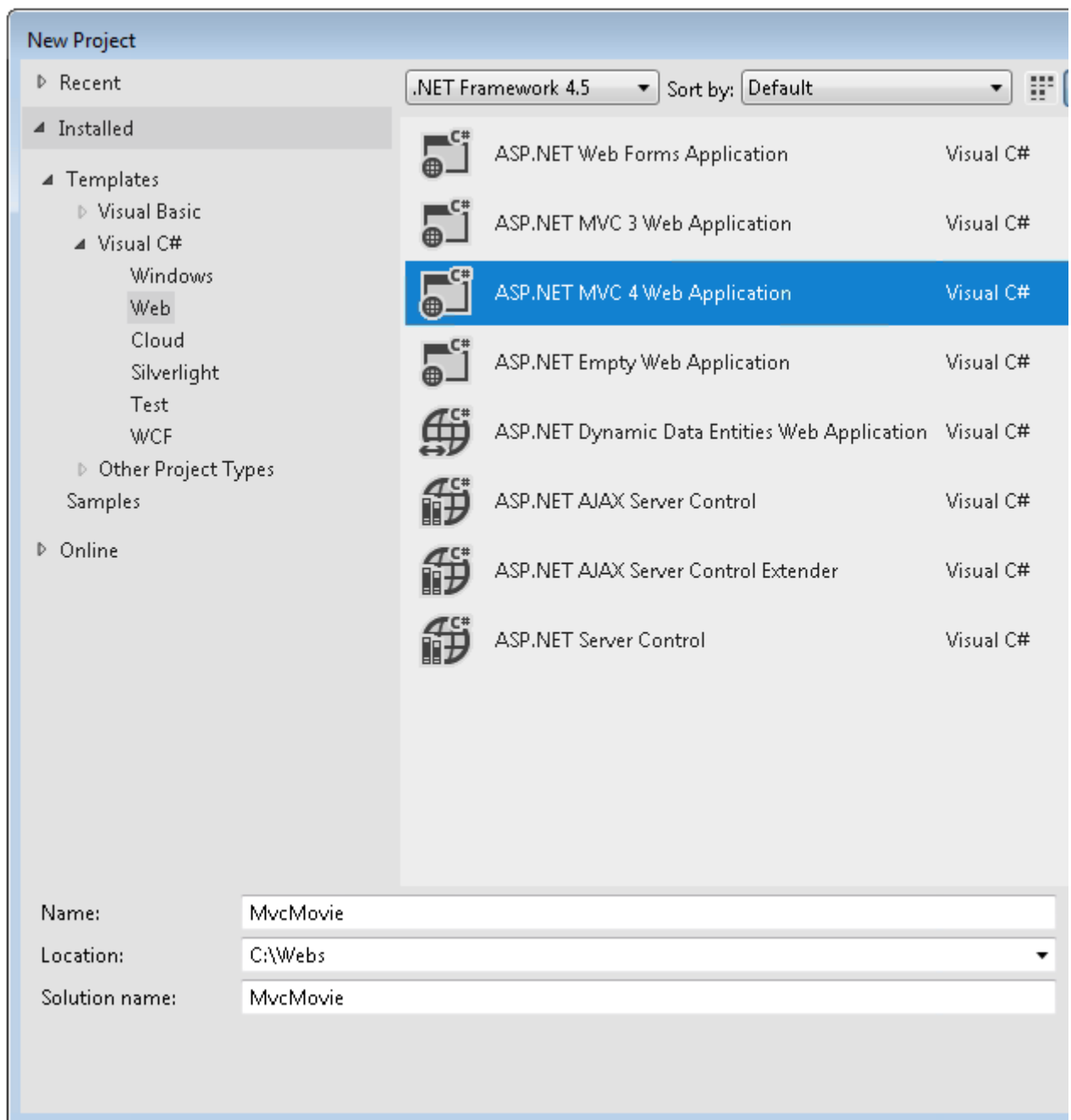
Visual Studio es un IDE o entorno de desarrollo integrado. Al igual que usted utiliza Microsoft Word para escribir documentos, vamos a usar un IDE para crear aplicaciones. En Visual Studio hay una barra de herramientas en la parte superior que muestra varias opciones disponibles

para usted. También hay un menú que ofrece otra manera de realizar tareas en el IDE. (Por ejemplo, en lugar de seleccionar **Nuevo proyecto** en la página **de inicio**, puede utilizar el menú y seleccione **Archivo> Nuevo proyecto.**)

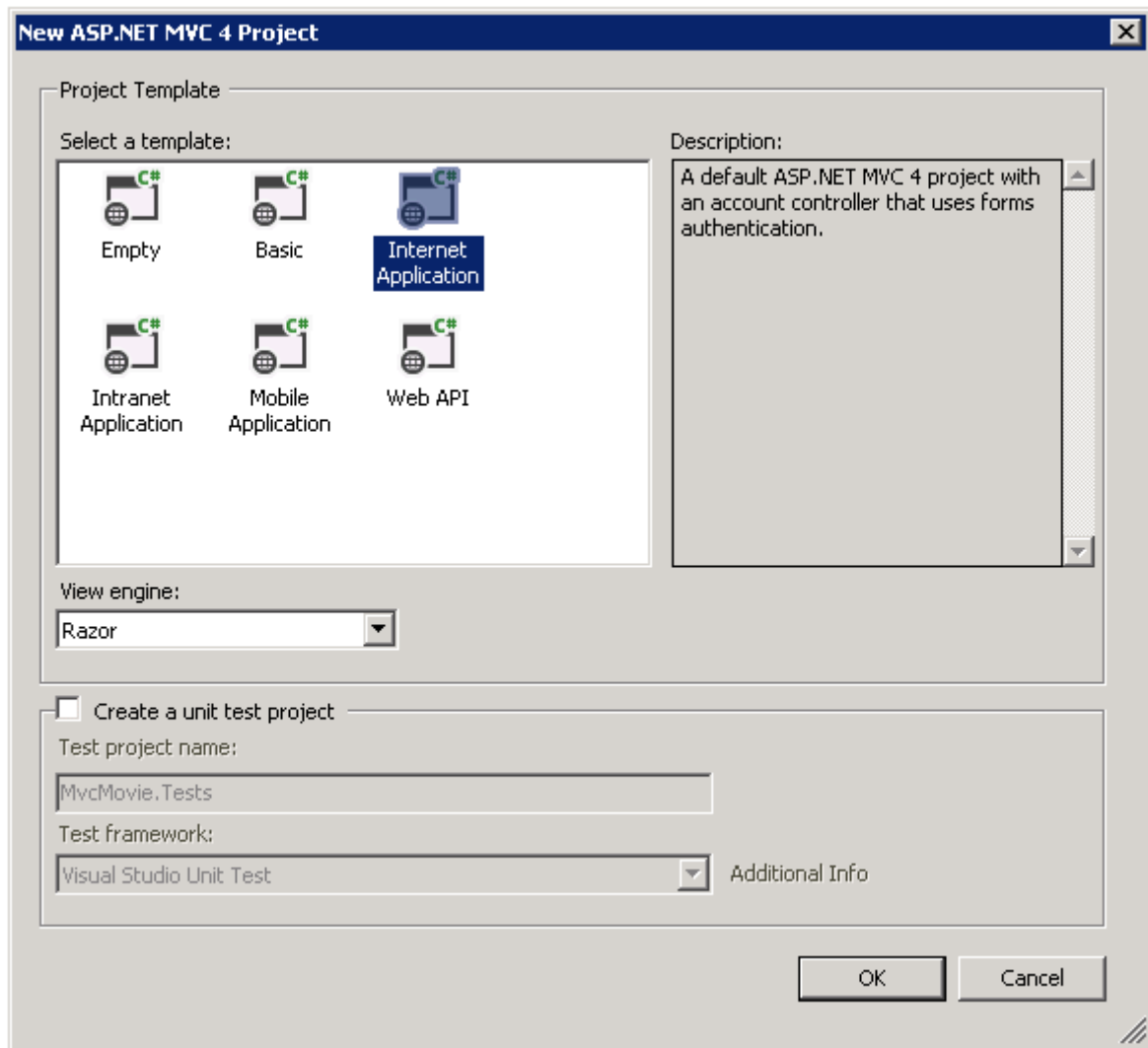


Creación de su primera aplicación

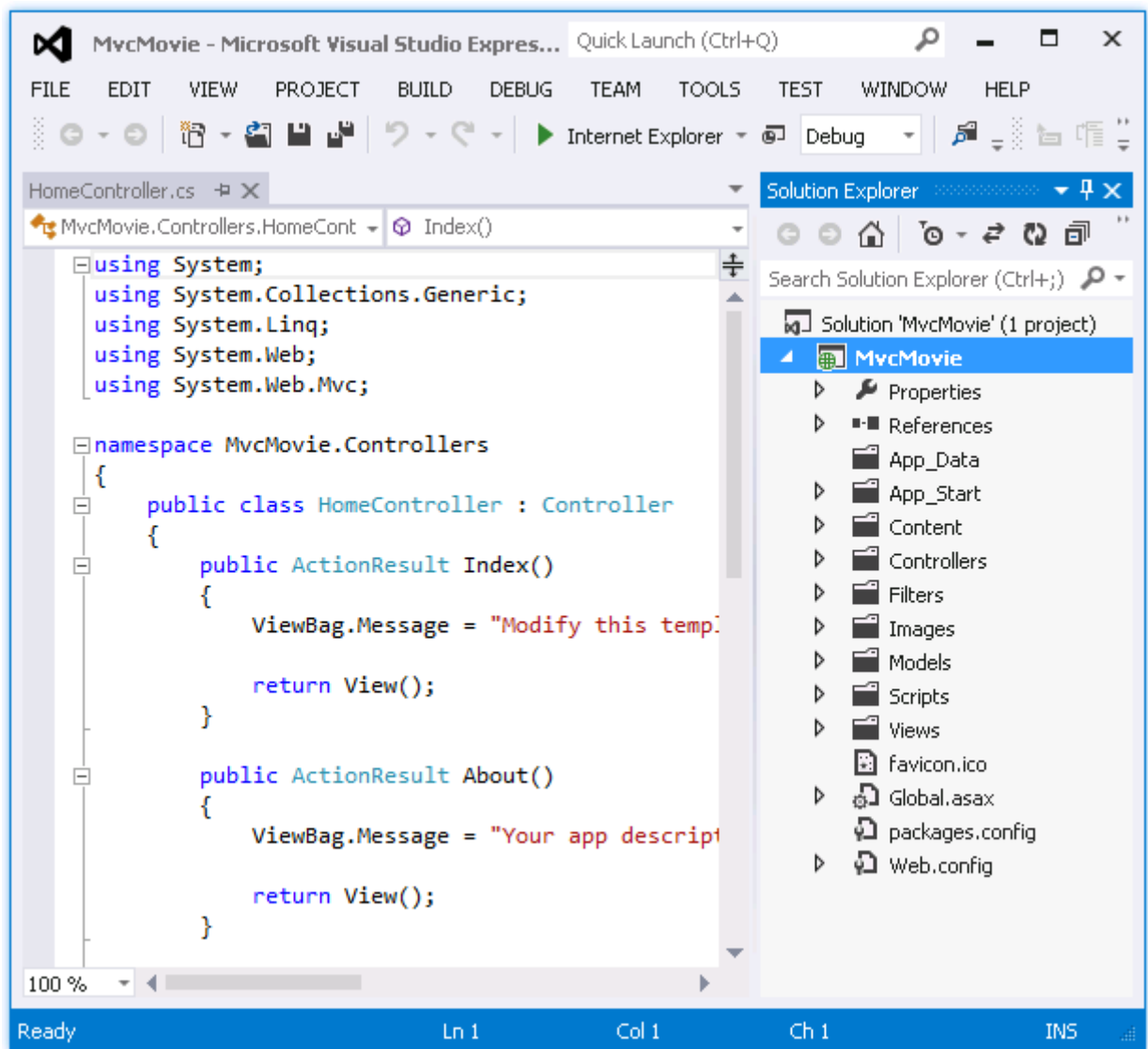
Puede crear aplicaciones utilizando Visual Basic o Visual C # como lenguaje de programación. Seleccione Visual C # a la izquierda y luego seleccione **ASP.NET MVC 4 Aplicación Web**. El nombre de su proyecto "MvcMovie" y luego haga clic **en Aceptar**.



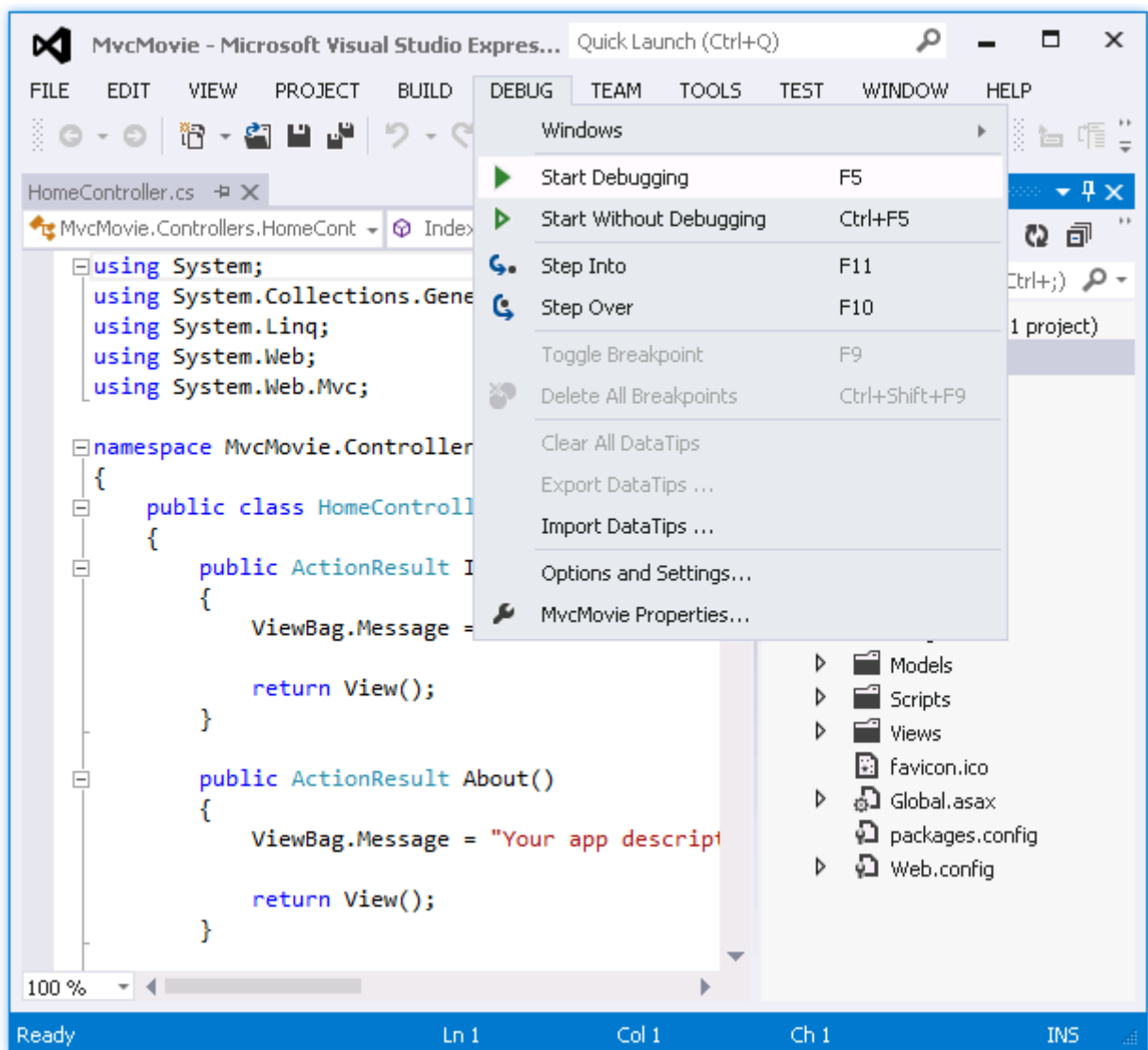
En el cuadro de diálogo **Nuevo proyecto 4 ASP.NET MVC**, seleccione **Aplicación de Internet**. Deja **Razor** como motor vista por defecto.



Haga clic **en Aceptar**. Visual Studio utiliza una plantilla predeterminada para el proyecto ASP.NET MVC que acaba de crear, por lo que tiene una aplicación de trabajo en este momento sin hacer nada! Este es un simple "Hello World!" proyecto, y es un buen lugar para comenzar su aplicación.

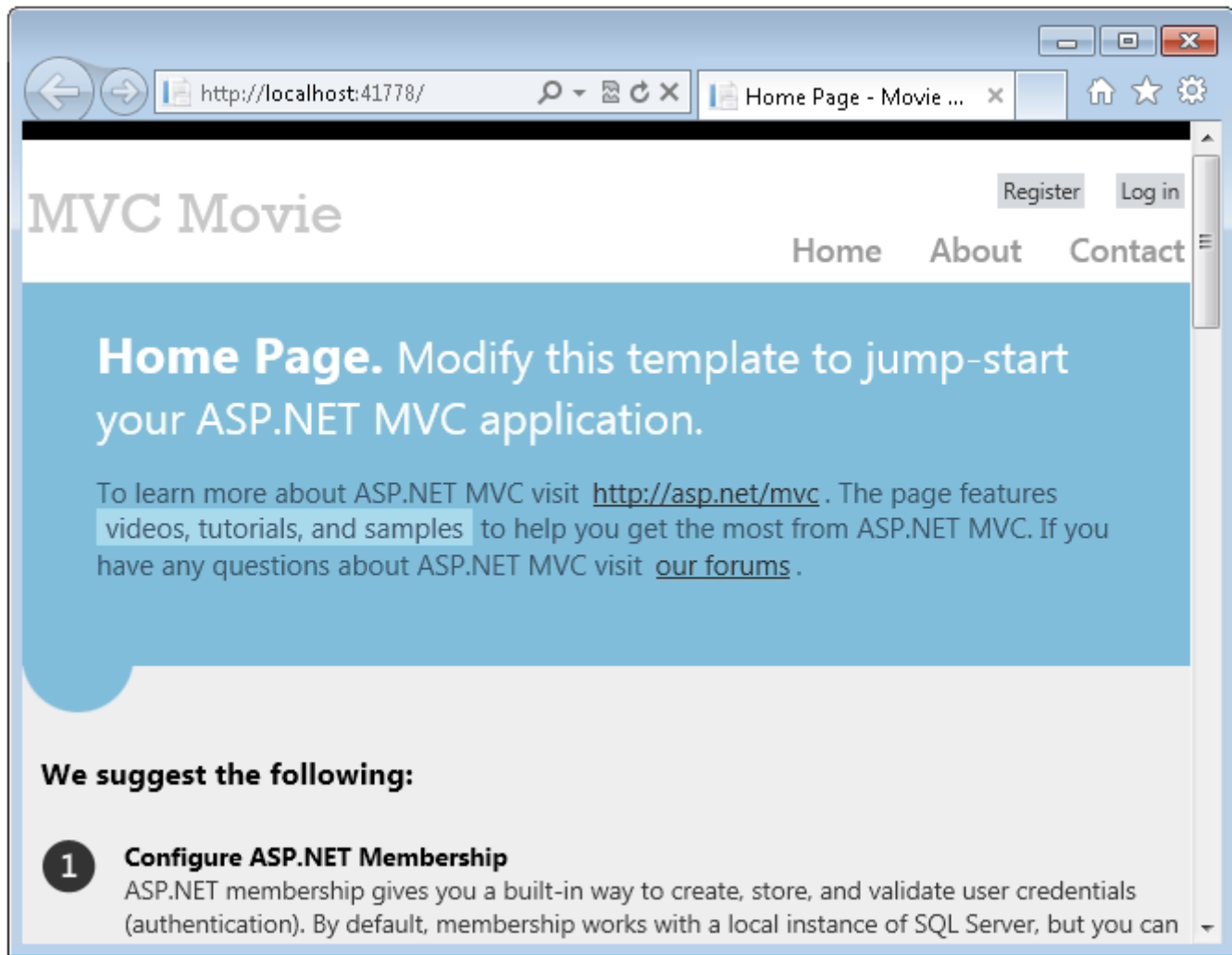


En el menú **Depurar**, seleccione **Iniciar depuración**.



Observe que el atajo de teclado para iniciar la depuración es F5.

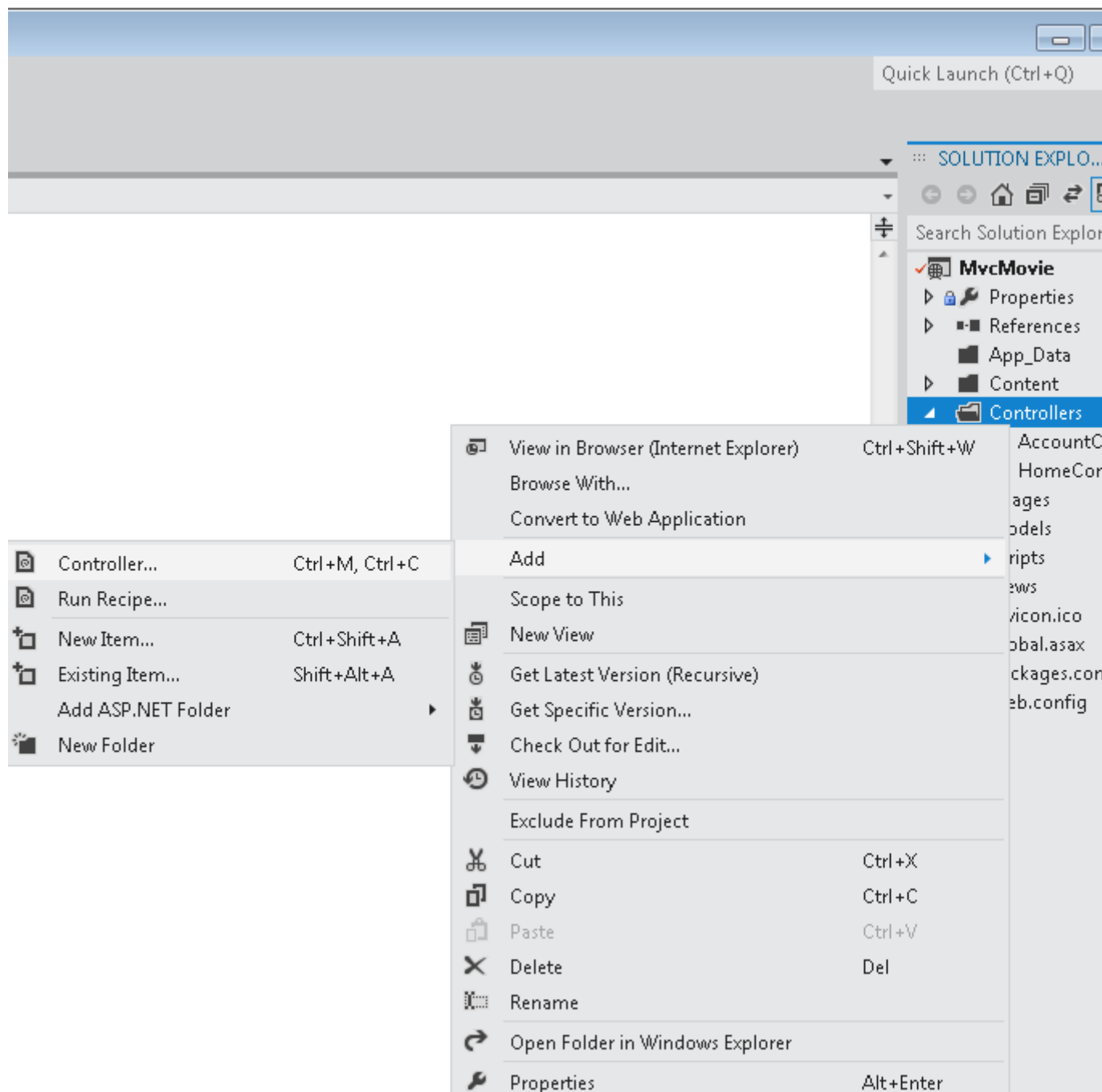
F5, Visual Studio para iniciar IIS Express y ejecutar la aplicación web. Visual Studio y luego lanza un navegador y abre la página principal de la aplicación. Observe que la barra de direcciones del navegador dice **localhost** y no algo como **example.com**. Esto se debe a **localhost** siempre apunta a su propio equipo local, que en este caso se está ejecutando la aplicación que acaba de construir. Cuando Visual Studio se ejecuta un proyecto web, un puerto aleatorio se utiliza para el servidor web. En la imagen de abajo, el número de puerto es 41788. Al ejecutar la aplicación, es probable que vea un número de puerto diferente.



Nada más sacarlo de la caja de esta plantilla predeterminada que da inicio, Contacto y Acerca de las páginas. También proporciona soporte para registrarse y conectarse, y enlaces a Facebook y Twitter. El siguiente paso es cambiar cómo funciona esta aplicación y aprender un poco acerca de ASP.NET MVC. Cierre el navegador y vamos a cambiar algo de código.

Adición de un controlador

Vamos a comenzar por la creación de una clase controlador. En **Explorador de soluciones**, haga clic en el *Controladores* carpeta y luego seleccione **Añadir Controller**.



El nombre de su nuevo controlador "HelloWorldController". Deje la plantilla predeterminada como **Controlador MVC Empty** y haga clic en **Añadir**.

Add Controller

Controller name:
HelloWorldController

Scaffolding options

Template:
Empty MVC controller

Model class:

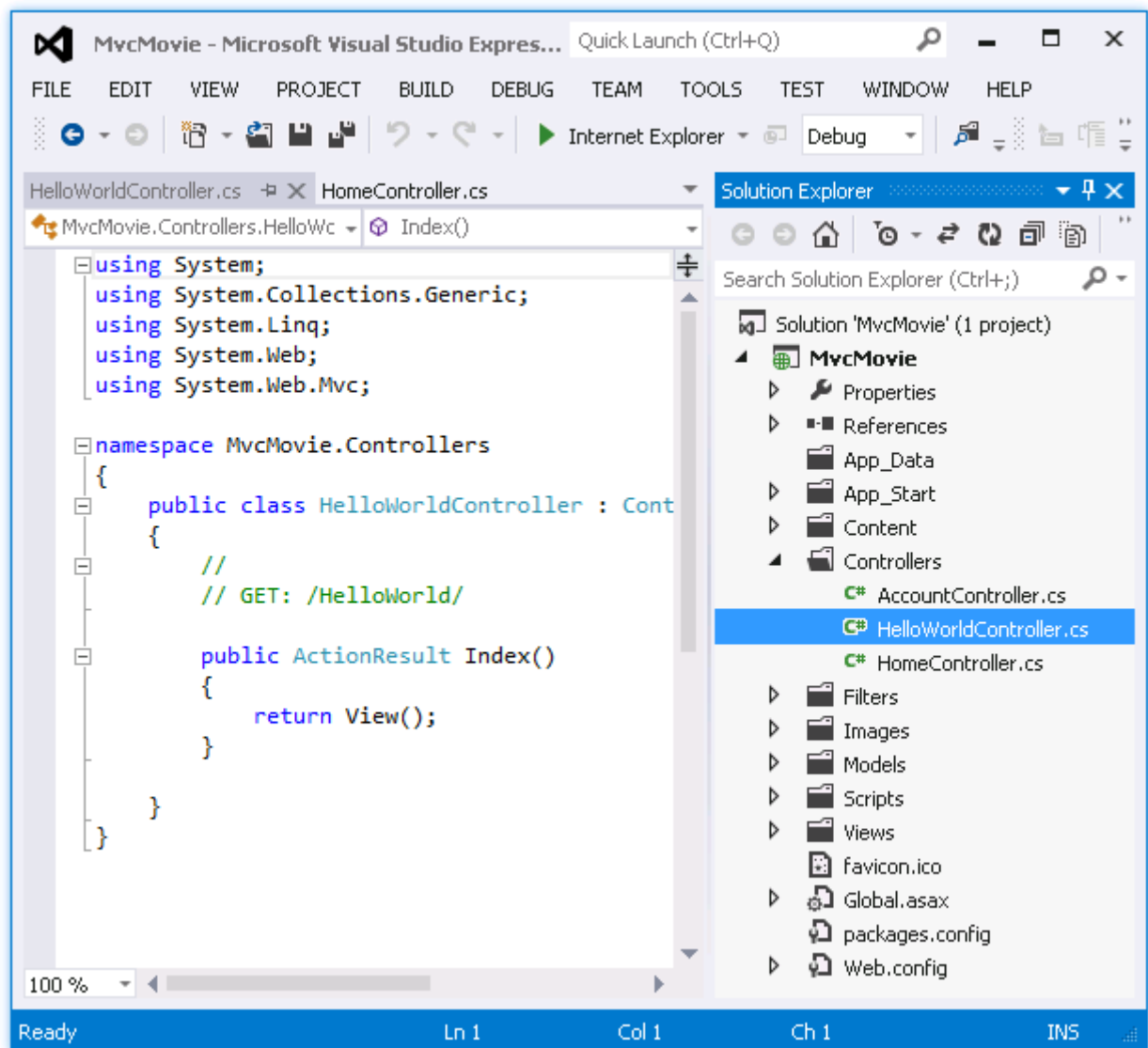
Data context class:

Views:
None

Advanced Options...

Add Cancel

Fíjese en **Explorador de soluciones** que un nuevo archivo ha sido creado llamado *HelloWorldController.cs*. El archivo está abierto en el IDE.



Reemplace el contenido del archivo con el siguiente código.

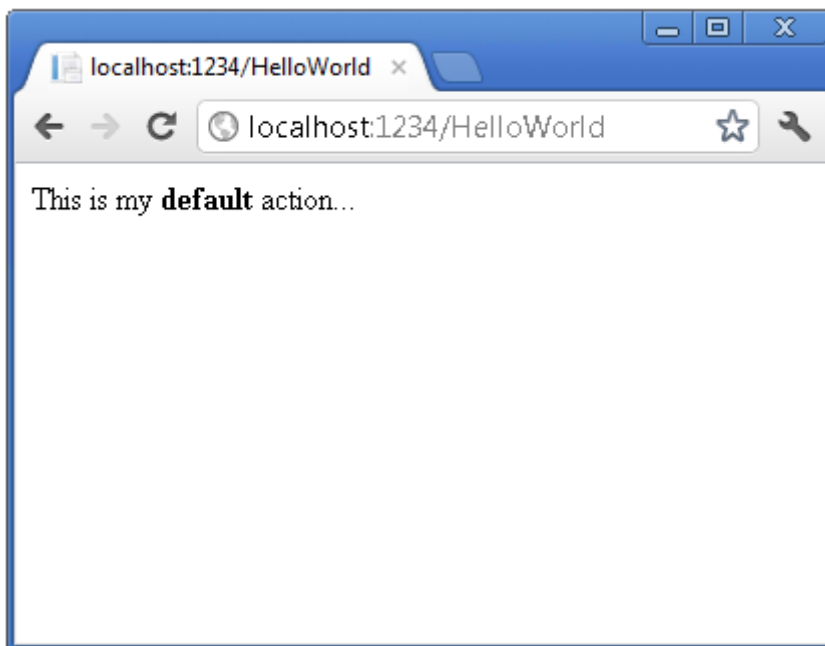
```
utilizando . Sistema Web;  
utilizando .. Sistema Web Mvc;  
  
espacio de nombres . MvcMovie Controladores  
{  
    pública clase HelloWorldController : Controlador  
    {  
        //  
        // GET: / HelloWorld /  
  
        pública string Índice ()  
        {  
            retorno "Este es mi <b> por defecto </ b> la acción ...";  
        }  
  
        //  
        // GET: / HelloWorld / Welcome /  
    }  
}
```

```

    pública    string    Bienvenido ()
    {
        retorno    "Este es el método de acción Bienvenido ...";
    }
}

```

Los métodos de controlador volverá una cadena de HTML como un ejemplo. El controlador es el nombre **HelloWorldController** y el primer método anterior se denomina **Index**. Invoquemos desde un navegador. Ejecutar la aplicación (presione F5 o Ctrl + F5). En el navegador, añadir "HelloWorld" a la ruta en la barra de direcciones. (Por ejemplo, en la ilustración de abajo, es `http://localhost:1234/HelloWorld`) La página en el navegador se verá como la siguiente captura de pantalla. En el método anterior, el código devuelve una cadena directamente. Le dijiste que el sistema simplemente regresar algo de HTML, y lo hizo!

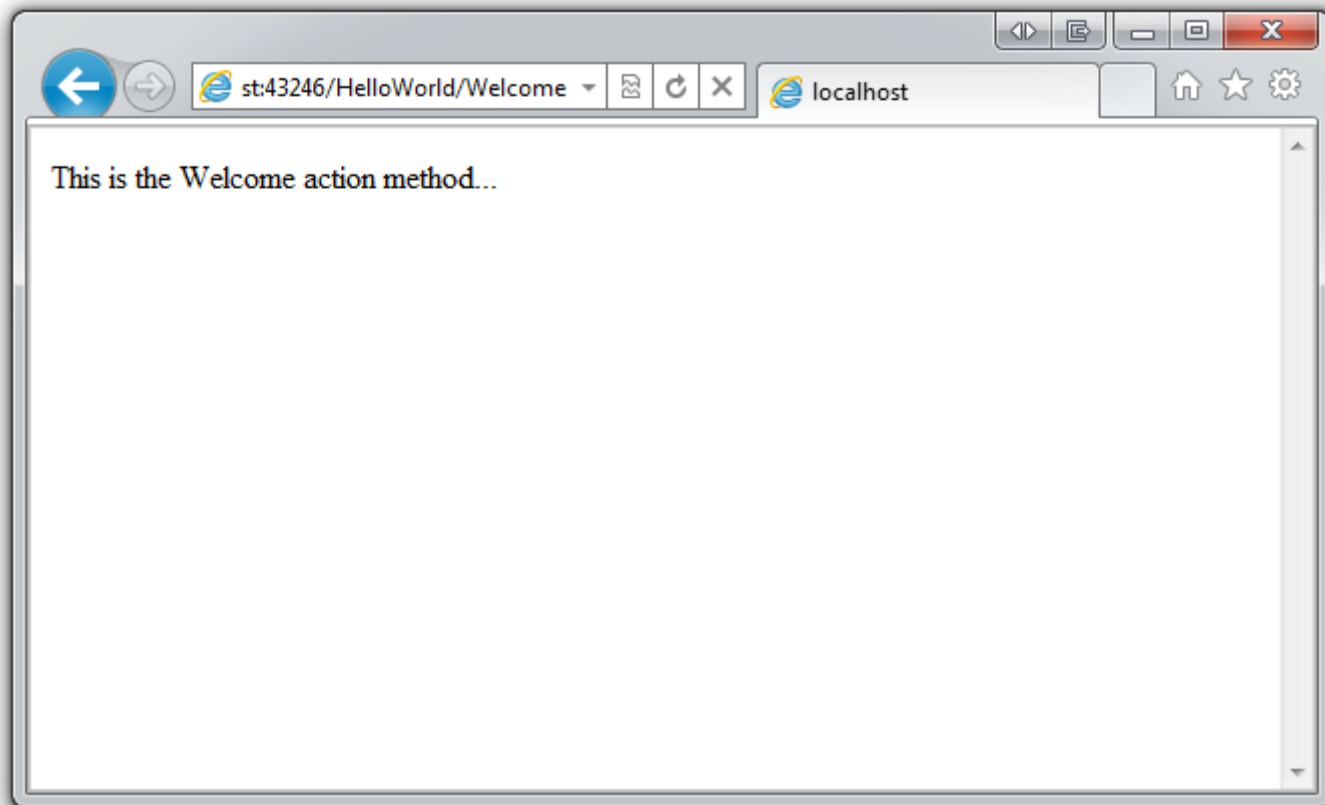


ASP.NET MVC invoca diferentes clases de controlador (y diferentes métodos de acción dentro de ellas) en función de la dirección URL entrante. La lógica de enrutamiento URL por defecto utilizado por ASP.NET MVC utiliza un formato como éste para determinar qué código para invocar:

/[Controller]/[ActionName]/[Parameters]

La primera parte de la URL determina la clase del controlador para ejecutar. Así `/HelloWorld` mapas a la **HelloWorldController** clase. La segunda parte de la URL determina el método de acción de la clase a ejecutar. Así `/HelloWorld/Índice` haría que el **Index** método de la **HelloWorldController** clase a ejecutar. Tenga en cuenta que sólo tuvimos que navegar a `/HelloWorld` y el **Index** método se utiliza por defecto. Esto es porque un método denominado **Index** es el método por defecto que se llama en un controlador si no se especifica de forma explícita.

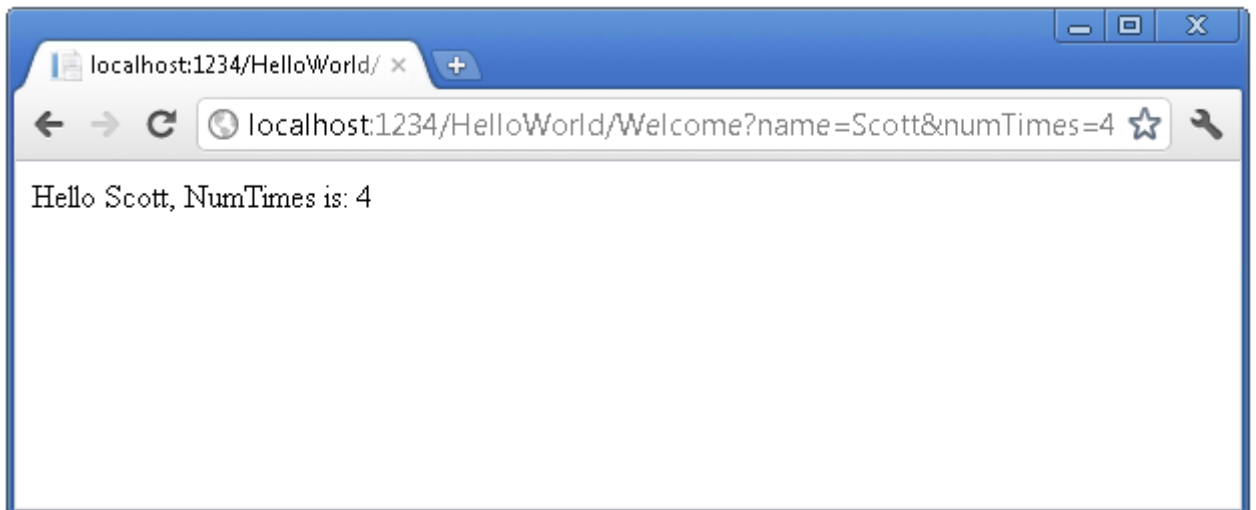
Busque `http://localhost:xxxx/HelloWorld/Welcome`. El **Welcome** Método de carreras y devuelve la cadena "Este es el método de acción Bienvenido ...". El MVC asignación predeterminada es **/[Controller]/[ActionName]/[Parameters]**. Por esta URL, el controlador es **HelloWorld** y **Welcome** es el método de acción. No ha utilizado la **[Parameters]** parte de la URL todavía.



Vamos a modificar un poco el ejemplo para que pueda pasar un poco de información de los parámetros de la URL para el controlador (por ejemplo, `/HelloWorld/Welcome?Name=Scott&numtimes=4`). Cambiar su `Welcome` método para incluir dos parámetros como se muestra a continuación. Observe que el código utiliza la función de C# opcional-parámetro para indicar que los `numTimes` parámetro debe defecto a 1 si no hay valor se pasa para ese parámetro.

```
pública string Bienvenido (nombre de la cadena, int numTimes =  
1) {  
    retorno HttpUtility. HtmlEncode ("Hola" + Nombre + "  
NumTimes es:" + numTimes);  
}
```

Ejecute la aplicación y vaya a la dirección URL de ejemplo (`http://localhost:xxxx/HelloWorld/Welcome name = Scott & numtimes = 4`). Puede probar diferentes valores para `name` y `numtimes` en la URL. El [sistema de encuadernación modelo ASP.NET MVC](#) mapea automáticamente los parámetros con nombre de la cadena de consulta en la barra de direcciones a los parámetros de su método.



En estos dos ejemplos, el controlador ha estado haciendo la parte "VC" de MVC - es decir, la visión y el trabajo del controlador. El controlador está volviendo HTML directamente. Por lo general no desea controladores que regresan directamente HTML, ya que se vuelve muy engorroso código. En su lugar, normalmente usaremos un archivo de vista de plantilla independiente para ayudar a generar la respuesta HTML. Echemos un vistazo al lado de lo que podemos hacer esto.

Adición de una vista

En esta sección vas a modificar el `HelloWorldController` clase para utilizar archivos de vista de la plantilla para encapsular correctamente el proceso de generación de respuestas HTML a un cliente.

Vamos a crear un archivo de vista de plantilla mediante el [motor de vistas Razor](#) introducido con plantillas de vista basados en Razor. ASP.NET MVC 3 tienen una extensión de archivo `.cshtml`, y proporciona una manera elegante de crear una salida HTML utilizando C#. Razor minimiza el número de personajes y pulsaciones necesarias para escribir una plantilla de vista, y permite un flujo de trabajo de codificación ayuno líquido.

Actualmente el `Index` método devuelve una cadena con un mensaje que se codifica duro en la clase del controlador. Cambie el `Index` método para devolver un `View` objeto, como se muestra en el siguiente código:

```
pública ActionResult Índice ()
{
    retorno Ver ();
}
```

El `Index` método anterior utiliza una plantilla de vista para generar una respuesta HTML al navegador. Métodos Controlador (también conocidos como [métodos de acción](#)), como el `Index` método anterior, en general, devolver un [ActionResult](#) (o una clase derivada de [ActionResult](#)), no tipos primitivos como cadena.

En el proyecto, agregue una plantilla de vista que se puede utilizar con el `Index` método. Para ello, haga clic dentro del `Index` método y haga clic en **Añadir vista**.

```

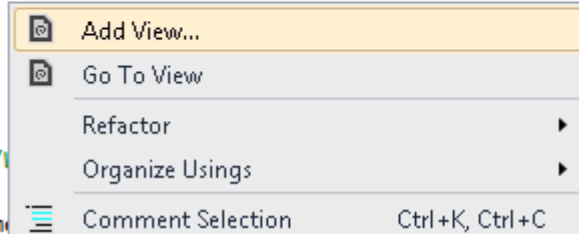
public class HelloWorldController : Controller
{
    //
    // GET: /HelloWorld/

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

    //
    // GET: /HelloWorld/

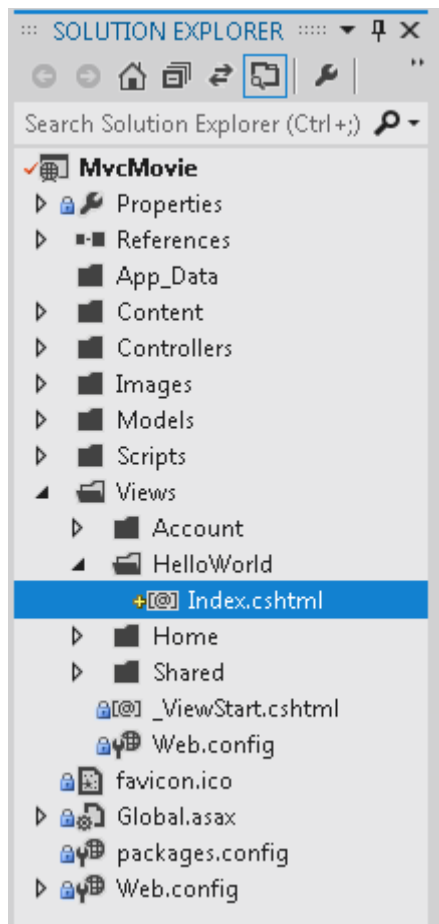
    public string Welcome

```

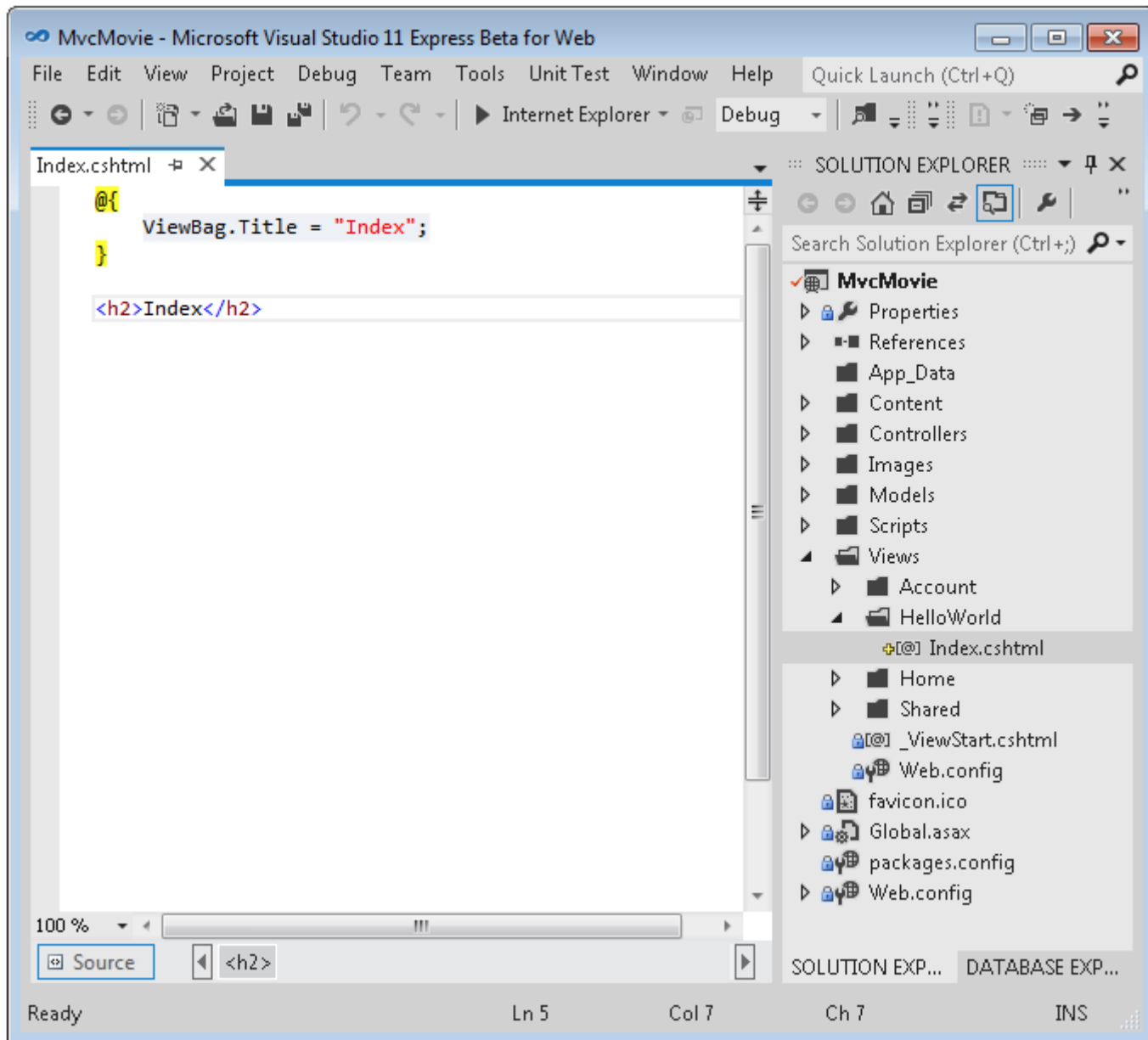


Aparecerá el cuadro de diálogo **Agregar vista**. Conserve los valores predeterminados como están y haga clic en el botón **Agregar**:

Se crean las vistas \ carpeta *HelloWorld MvcMovie* \ y los *MvcMovie* \ \ Vistas archivo *HelloWorld* \ *Index.cshtml*. Se los puede ver en **el Explorador de soluciones**:



A continuación se muestra el archivo *Index.cshtml* que se creó:



Agregue el siguiente código HTML en el **<h2>** etiqueta.

```
<P> Hola de nuestra plantilla de vista! </ P>
```

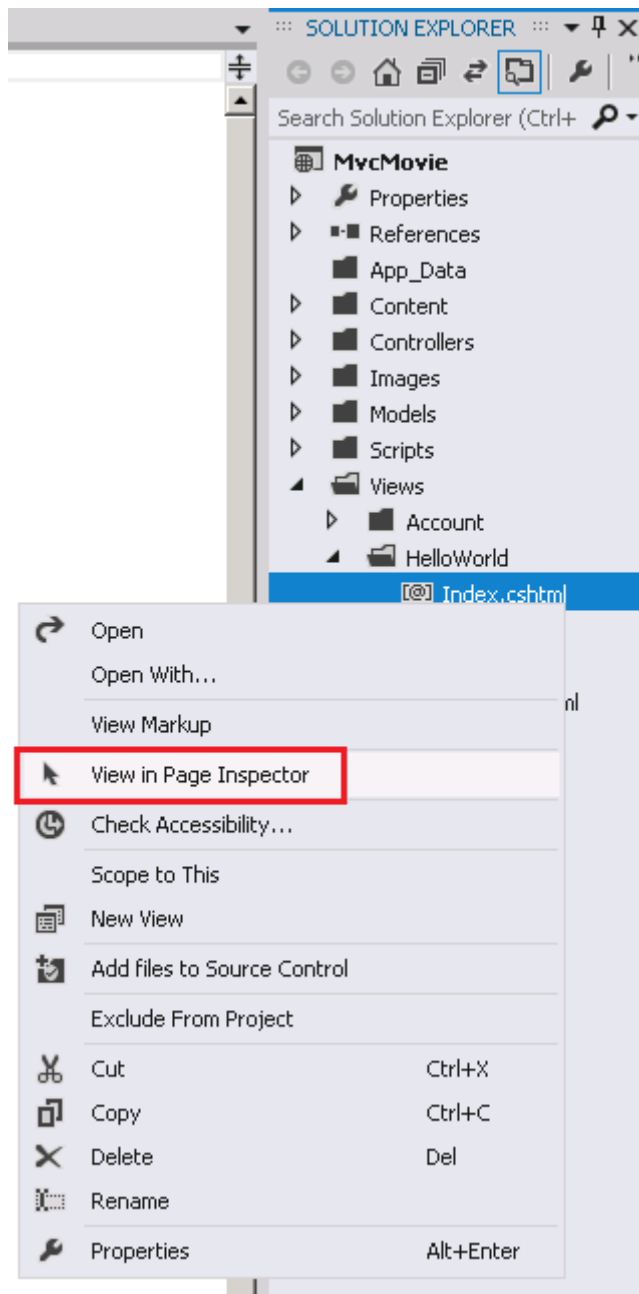
Los completos *MvcMovie \ \ Vistas* archivo *HelloWorld \ Index.cshtml* se muestra a continuación.

```
@ {
    ViewBag. Título    =    "Índice";
}
```

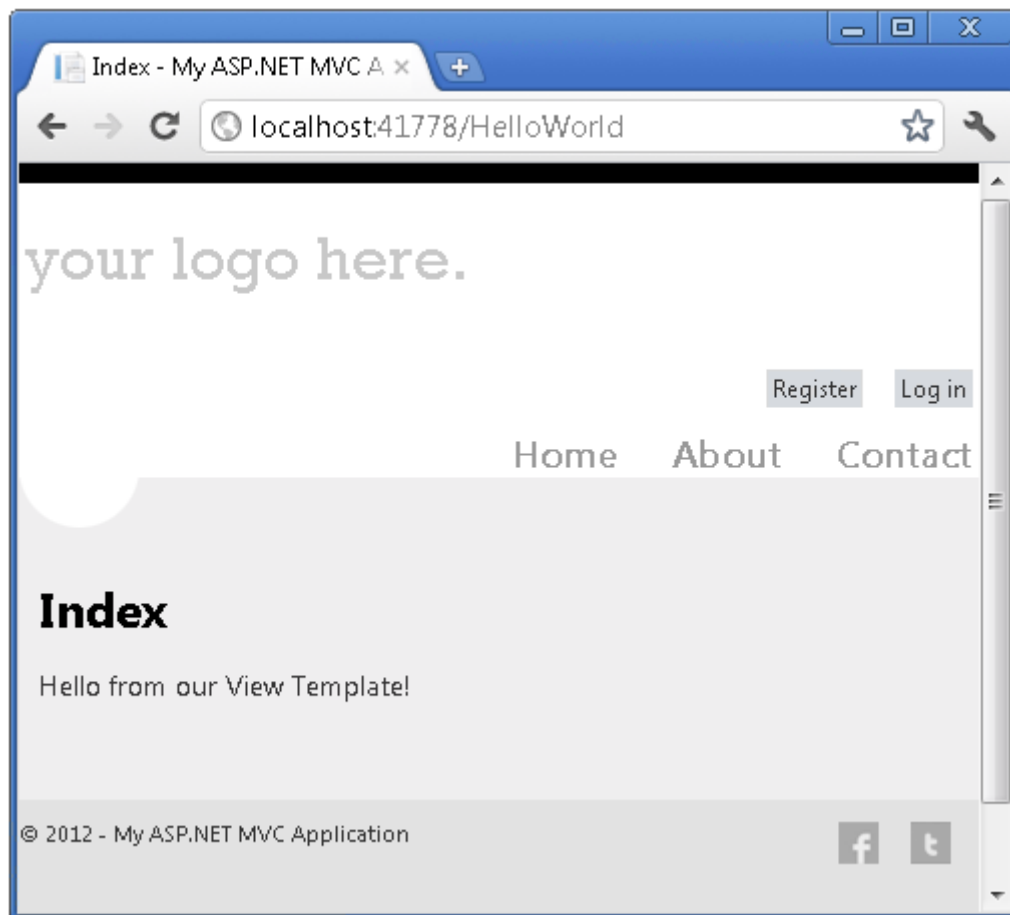
```
<H2> Índice </ h2>
```

```
<P> Hola de nuestra plantilla de vista! </ P>
```

Si está utilizando Visual Studio 2012, en el Explorador de soluciones, haga clic derecho en el archivo *Index.cshtml* y seleccione **Ver en Inspector de la página**.



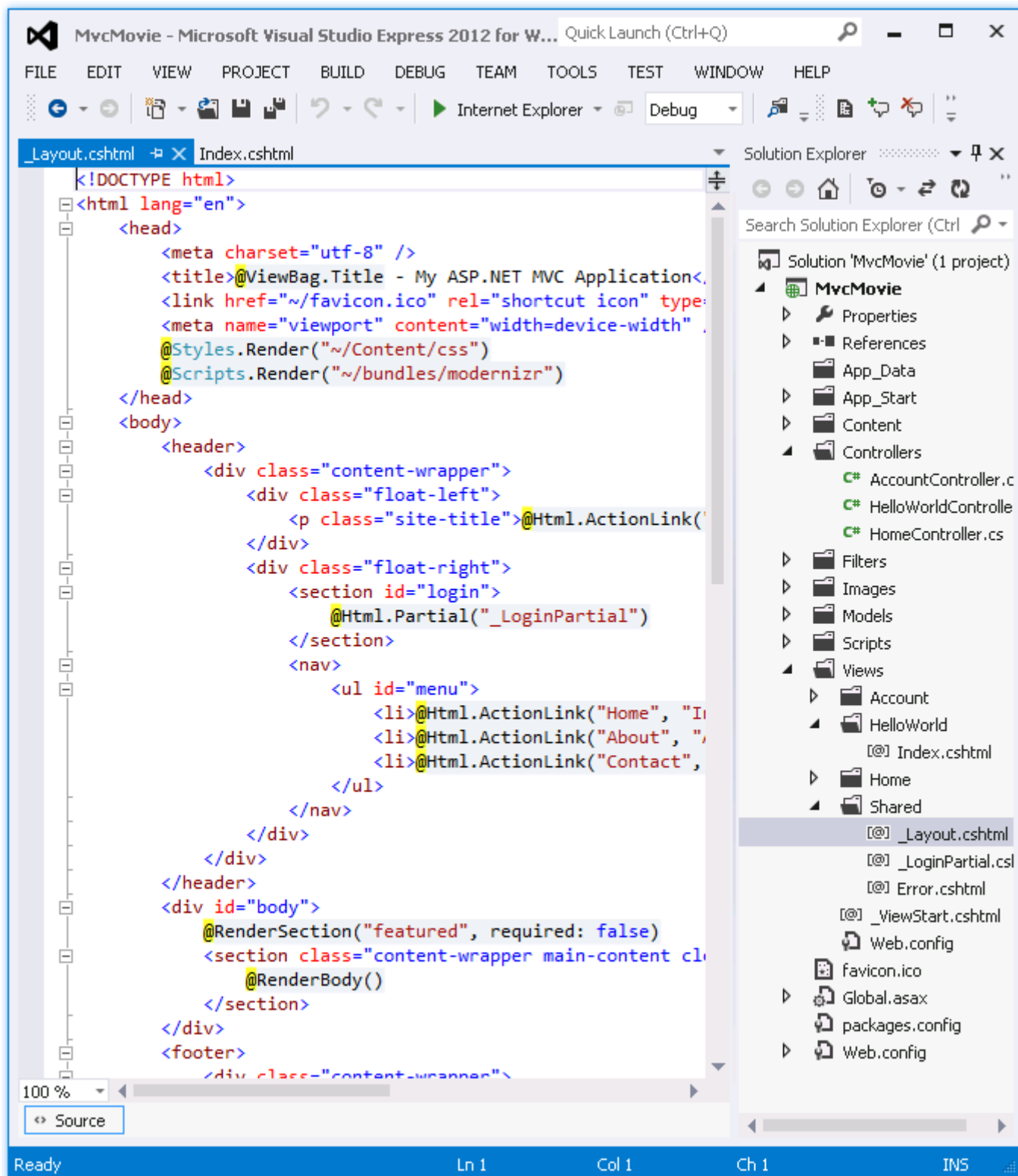
El [tutorial Inspector de la página](#) tiene más información acerca de esta nueva herramienta. Alternativamente, r ONU la aplicación y busque la **HelloWorld** controlador (<http://localhost:xxxx/HelloWorld>). El **Index** método en el controlador no hizo mucho trabajo; simplemente corrió la declaración **return View()**, que especifica que el método debe utilizar un archivo de vista de plantilla para hacer una respuesta al navegador. Debido a que no se ha especificado de forma explícita el nombre del archivo de plantilla de vista de utilizar, ASP.NET MVC utilizando por defecto en el archivo de vista *Index.cshtml* en la carpeta *HelloWorld \ Views*. La imagen de abajo muestra la cadena "Hola desde nuestra plantilla View!" modificable en la vista.



Se ve muy bien. Sin embargo, observe que la barra de título del navegador muestra "Índice Mi ASP.NET A" y el gran enlace en la parte superior de la página dice "su logotipo aquí." Por debajo de la "su logotipo aquí." enlace son de registro e ingrese enlaces, y por debajo que los enlaces a páginas de inicio, Acerca de y Contacto. Vamos a cambiar algunas de ellas.

Cambiar las vistas y Diseño Páginas

En primer lugar, usted quiere cambiar el "su logotipo aquí." título en la parte superior de la página. Ese texto es común a todas las páginas. Se aplica realmente en un solo lugar en el proyecto, a pesar de que aparece en cada página de la aplicación. Ir al *Vistas / carpeta / compartida* en **el Explorador de soluciones** y abra el archivo *_Layout.cshtml*. Este archivo se llama un *diseño de página* y es la "shell" compartido que todas las demás páginas utilizan.



Plantillas de diseño le permiten especificar el formato contenedor HTML de su sitio en un lugar y luego lo aplica a través de múltiples páginas en su sitio. Encuentra el `@RenderBody()` línea. `RenderBody` es un marcador de posición en todas las páginas específicos de vista que cree se presenta, "envuelto" en la página de diseño. Por ejemplo, si selecciona el enlace Acerca de, las Vistas \ Vista Home \ `About.cshtml` se representa dentro de la `RenderBody` método.

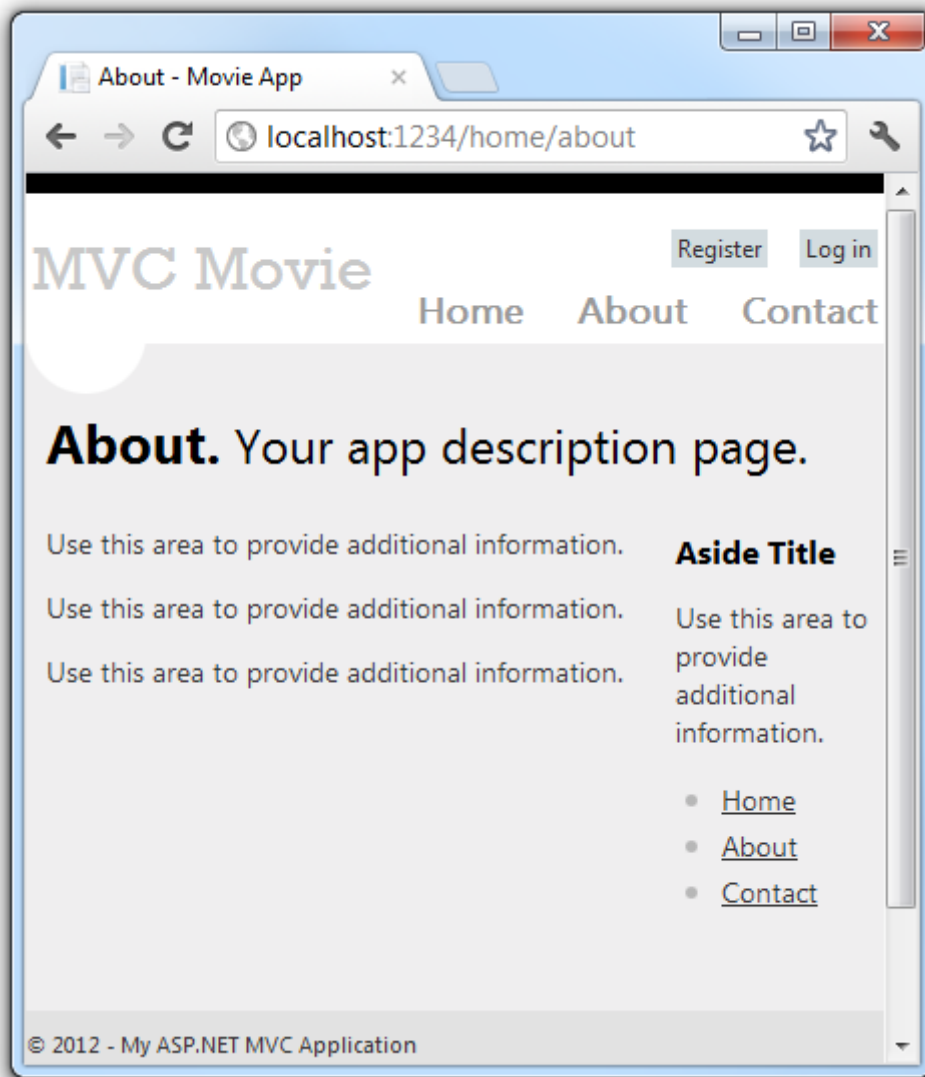
Cambiar el rumbo sitio-título en la plantilla de diseño de "su logotipo aquí" a "MVC Movie".

```
<Div class = "float-left">
    <P class = "sitio-title"> @ Html.ActionLink ("MVC Movie", "Índice",
"Home") </ p>
</ Div>
```

Reemplace el contenido del elemento de título con el siguiente marcado:

```
<Title> @ ViewBag.Title - Movie App </ title>
```

Ejecutar la aplicación y observe que ahora dice "MVC Movie". Haga clic en el enlace **Acerca de**, y ves cómo esa página muestra "MVC Movie", también. Hemos sido capaces de hacer el cambio una vez en la plantilla de diseño y cuentan con todas las páginas en el sitio reflejan el nuevo título.



Ahora, vamos a cambiar el título de la vista Index.

Abiertas *MvcMovie \ \ HelloWorld Vistas \ Index.cshtml*. Hay dos lugares para hacer un cambio: en primer lugar, el texto que aparece en el título del navegador, y luego en la cabecera secundaria (el `<h2>` elemento). Usted va a hacer de ellos un poco diferente para que pueda ver que poco de los cambios que parte de la aplicación del código.

```
@ {
    ViewBag. Título    =    "Lista de películas";
```

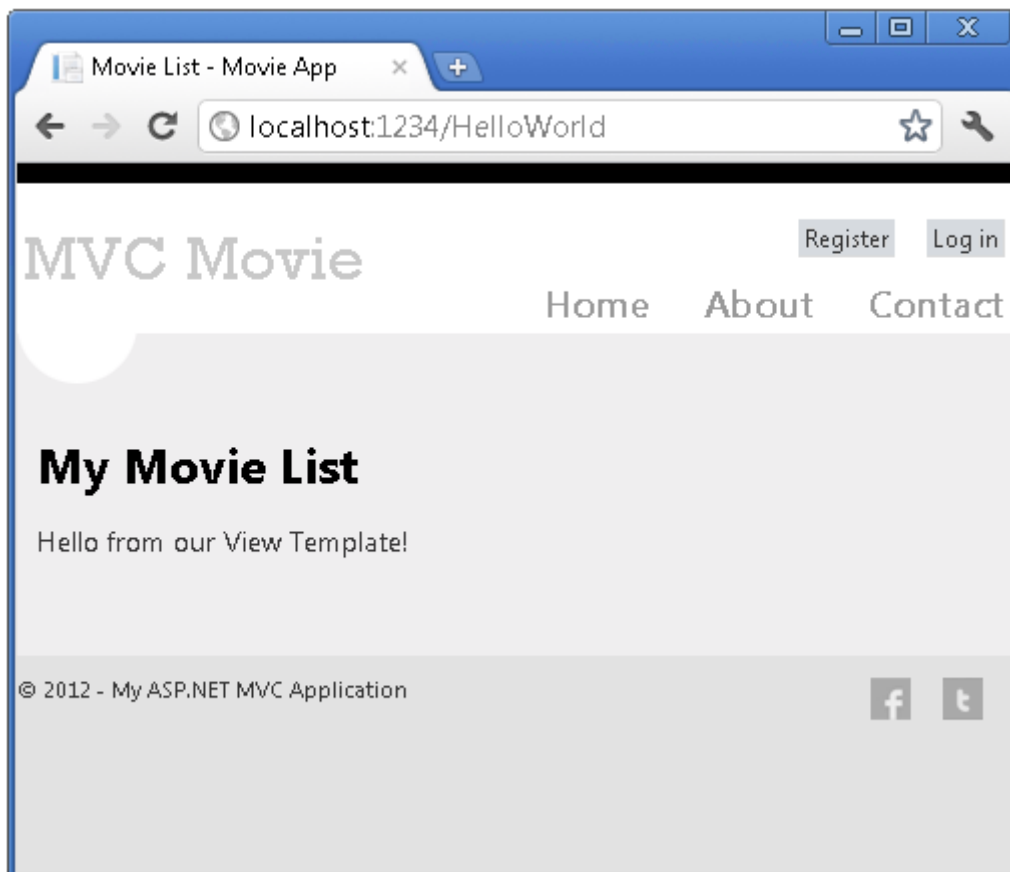


```
}  
  
<H2> Mi Película Lista </ h2>  
  
<P> Hola de nuestra plantilla de vista! </ P>
```

Para indicar el título HTML para mostrar, el código anterior establece un **Title** de propiedad del **ViewBag** objeto (que se encuentra en la plantilla de vista *Index.cshtml*). Si uno mira hacia atrás en el código fuente de la plantilla de diseño, te darás cuenta de que la plantilla utiliza este valor en el **<title>** elemento como parte de la **<head>** sección del HTML que hemos modificado anteriormente. El uso de este **ViewBag** enfoque, puede pasar fácilmente otros parámetros entre su plantilla de vista y su archivo de diseño.

Ejecutar la aplicación y vaya a *http://localhost:xx/HelloWorld*. Tenga en cuenta que el título del navegador, el título principal y de las líneas secundarias han cambiado. (Si no ve los cambios en el navegador, es posible que esté viendo el contenido en caché. Presione Ctrl + F5 en su navegador para forzar la respuesta del servidor a cargar.) El título del navegador se crea con el **ViewBag.Title** establecimos en la plantilla de vista *Index.cshtml* y el adicional "- Película App", agregó en el archivo de diseño.

Observe también cómo se fusionó el contenido en la plantilla de vista *Index.cshtml* con la plantilla de vista *Layout.cshtml* y una sola respuesta HTML se envía al navegador. Plantillas de diseño hacen que sea muy fácil para hacer los cambios que se aplican en todas las páginas de la aplicación.



Nuestro poco de "datos" (en este caso el "Hola de nuestra plantilla de vista!" Mensaje) no es modificable, sin embargo. La aplicación MVC tiene una "V" (ver) y usted tiene una "C" (controlador), pero no "M" (modelo) todavía. En breve, vamos a caminar a través de cómo crear una base de datos y recuperar datos de los modelos de la misma.

Pasar datos del Controlador a la Vista

Antes de ir a una base de datos y hablamos de modelos, sin embargo, primero vamos a hablar de la transmisión de información desde el controlador a una vista. Clases de controlador se invocan en respuesta a una solicitud de URL entrante. Una clase controlador es donde se escribe el código que se encarga de las peticiones del navegador entrantes, recupera datos de una base de datos, y en última instancia decide qué tipo de respuesta para enviar de vuelta al navegador. Ver las plantillas se pueden utilizar desde un controlador para generar y dar formato a una respuesta HTML al navegador.

Los controladores son responsables de proporcionar cualquier dato u objetos se requiere para que una plantilla de vista para hacer una respuesta al navegador. Una mejor práctica: **Una plantilla de vista nunca debe realizar la lógica de negocio o interactuar con una base de datos directamente.** En cambio, una plantilla de vista debe trabajar sólo con los datos que se proporciona a la misma por parte del controlador. El mantenimiento de esta "separación de intereses" ayuda a mantener el código limpio, comprobable y más fácil de mantener.

Actualmente, la `Welcome` método de acción en el `HelloWorldController` clase toma un `name` y un `numTimes` parámetro y luego envía los valores directamente en el navegador. En lugar de tener el controlador render esta respuesta como una cadena, vamos a cambiar el controlador al utilizar una plantilla de vista en su lugar. La plantilla de vista va a generar una respuesta dinámica, lo que significa que usted necesita para pasar bits apropiados de los datos del controlador a la vista con el fin de generar la respuesta. Usted puede hacer esto haciendo que el controlador de poner los datos dinámicos (parámetros) que la plantilla de vista necesita en un `ViewBag` objeto de que la plantilla de vista entonces puede acceder.

Volver al archivo `HelloWorldController.cs` y cambiar el `Welcome` método para agregar un `Message` y `NumTimes` valor al `ViewBag` objeto. `ViewBag` es un objeto dinámico, lo que significa que puedes poner lo que quieras a ella; la `ViewBag` objeto tiene propiedades no definido hasta que pongas algo en su interior. El [sistema de encuadernación modelo ASP.NET MVC](#) mapea automáticamente los parámetros con nombre (`name` y `numTimes`) de la cadena de consulta en la barra de direcciones a los parámetros de su método. El archivo `HelloWorldController.cs` completa se parece a esto:

```
utilizando    . Sistema Web;
utilizando    .. Sistema Web Mvc;

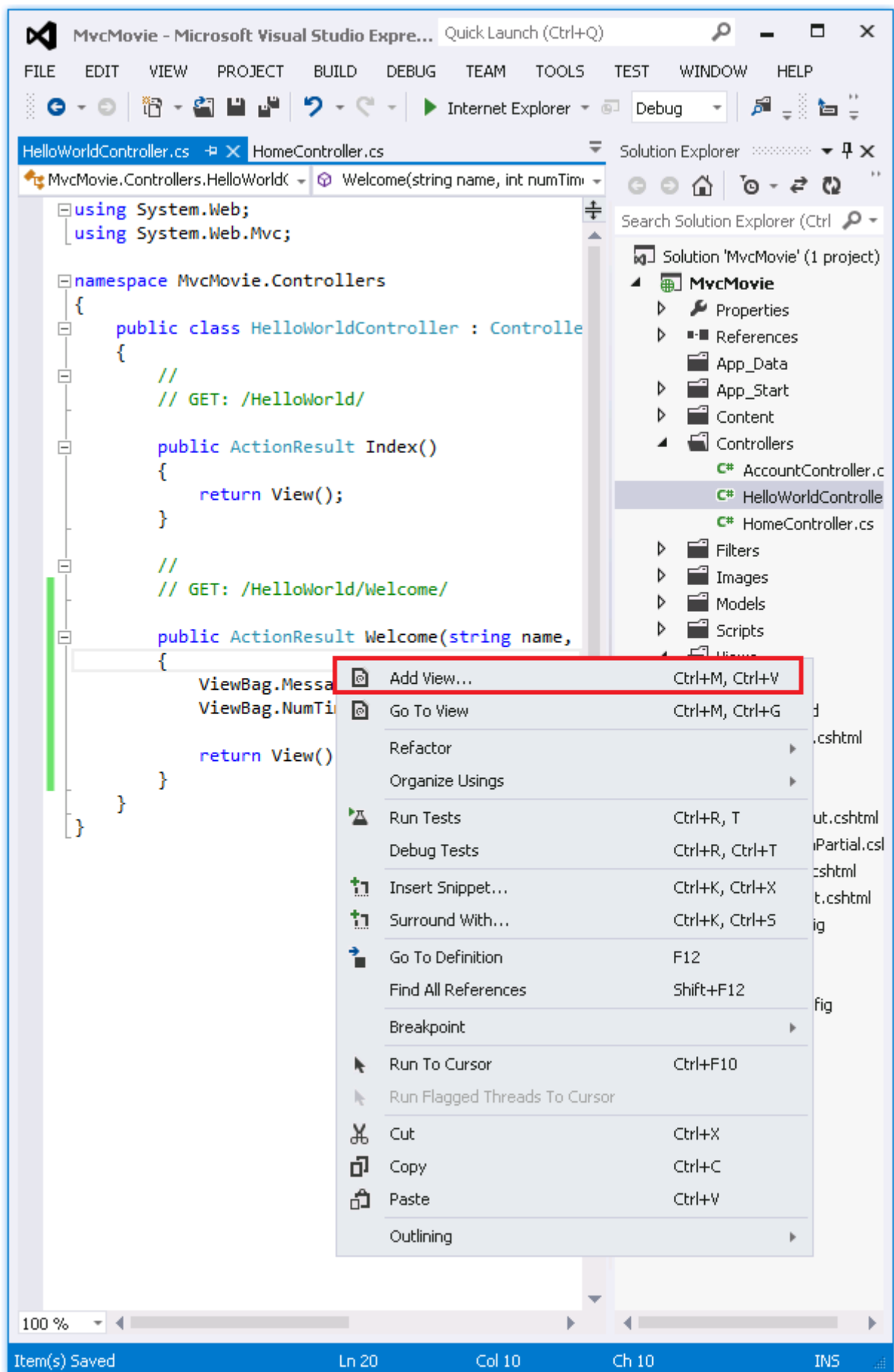
espacio de nombres    . MvcMovie Controladores
{
    pública     clase     HelloWorldController     :     Controlador
    {
        pública     ActionResult     Índice ()
        {
            retorno     Ver ();
        }

        pública     ActionResult     Bienvenido (nombre de la cadena,     int
numTimes =     1)
        {
            . ViewBag Mensaje     =     "Hola"     + Nombre;
            ViewBag. NumTimes     = numTimes;

            retorno     Ver ();
        }
    }
}
```

```
}
```

Ahora el **ViewBag** objeto contiene datos que se pasa a la vista de forma automática. Después, usted necesita una visión plantilla Bienvenido! En el menú **Generar**, seleccione **Generar MvcMovie** para asegurarse de que el proyecto se compila. A continuación, haga clic dentro de la **Welcome** método y haga clic en **Añadir vista**.



Esto es lo que el cuadro de diálogo **Agregar vista** parece:

View name:
Welcome

View engine:
Razor (CSHTML)

☐ Create a strongly-typed view

Model class:

Scaffold template:
Empty

☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:
MainContent

Add Cancel

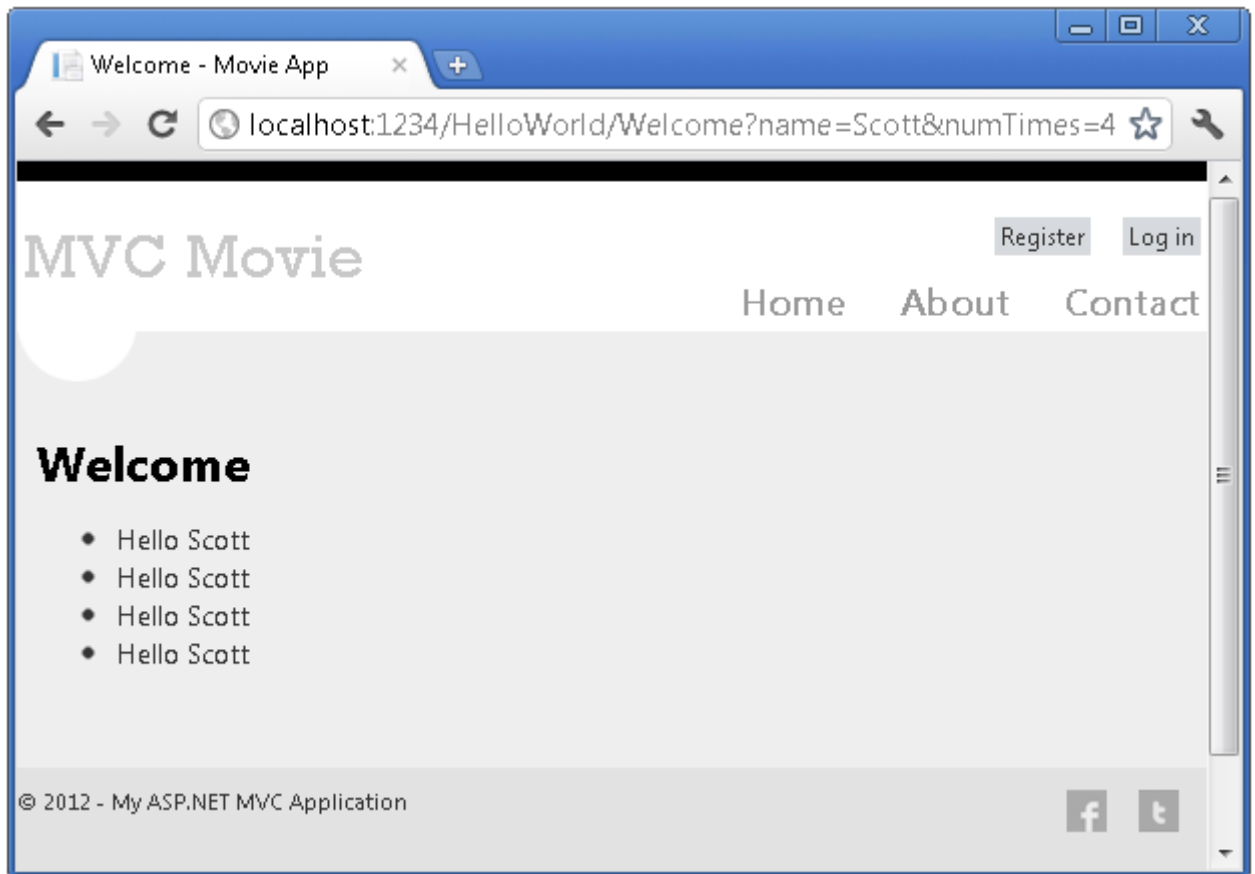
Haga clic **en Agregar** y, a continuación, agregue el código siguiente bajo el **<h2>** elemento en el nuevo archivo *Welcome.cshtml*. Vamos a crear un lazo que dice "Hola" tantas veces como el usuario dice que debería. El archivo *Welcome.cshtml* completa se muestra a continuación.

```
@ {  
    ViewBag.Título    =    "Bienvenido";  
}  
  
<H2> Bienvenido </ h2>  
  
<U1>  
    for (int i = 0; i <ViewBag.NumTimes; i ++ ) {  
        <Li> @ ViewBag.Message </ li>  
    }  
</ U1>
```

Ejecutar la aplicación y vaya a la siguiente URL:

http://localhost:xx/HelloWorld/Welcome name = Scott & numtimes = 4

Ahora los datos se obtienen a partir de la URL y se pasa al controlador utilizando el [ligante modelo](#) . El controlador empaqueta los datos en un **ViewBag** objeto y pasa ese objeto a la vista. La vista a continuación, muestra los datos como HTML al usuario.



En el ejemplo anterior, se utilizó una **ViewBag** objeto para pasar datos desde el controlador a una vista. Últimos en el tutorial, vamos a utilizar un modelo de vista para pasar datos de un controlador a una vista. El enfoque de vista del modelo de datos que pasa es generalmente mucho más preferible al enfoque de ver la bolsa. Véase la entrada de blog [dinámico V fuertemente tipado Vistas](#) para más información.

Bueno, eso fue una especie de una "M" para el modelo, pero no el tipo de base de datos. Vamos a tomar lo que hemos aprendido y crear una base de datos de películas.

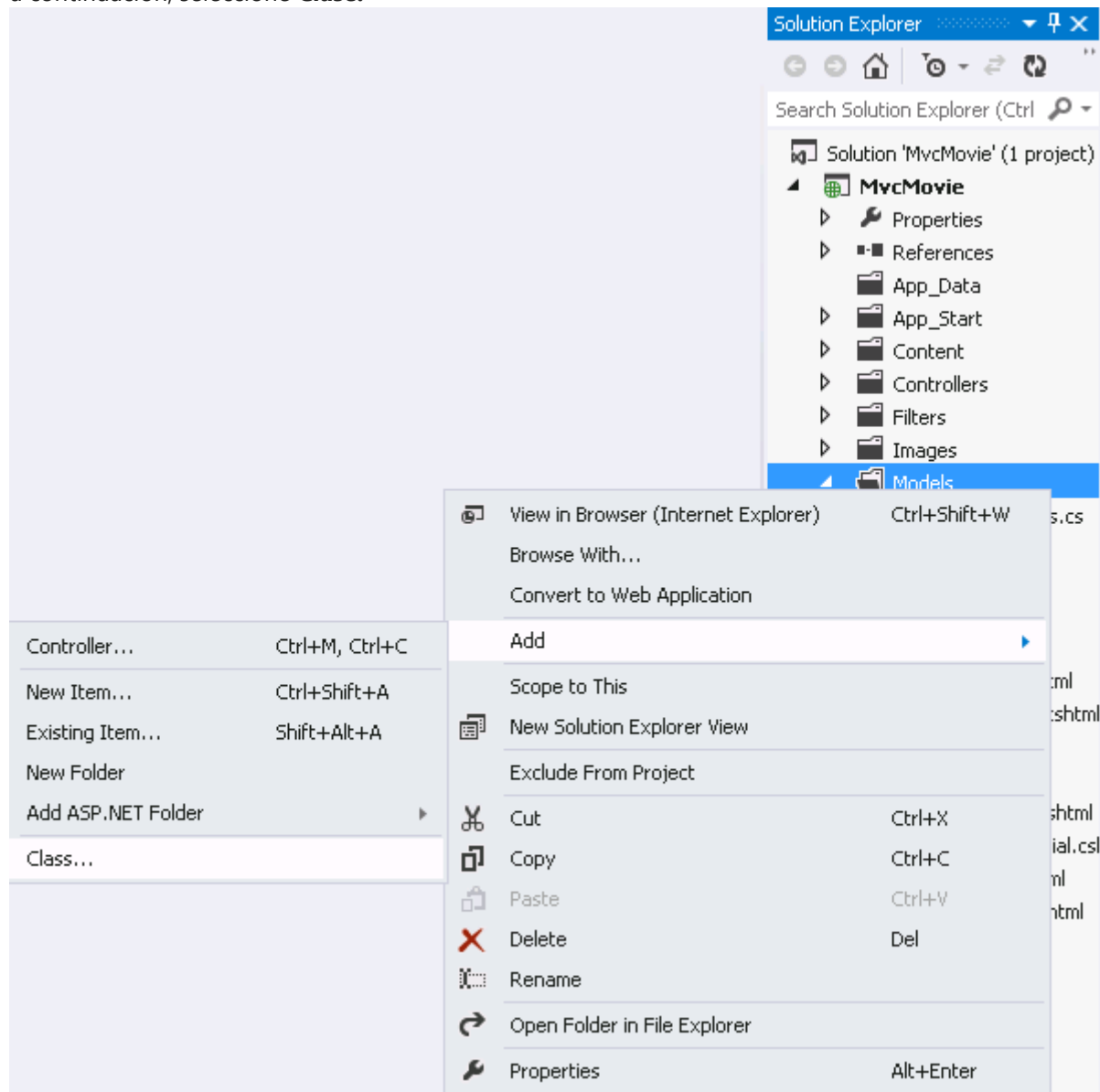
Adición de un modelo

En esta sección vamos a añadir algunas clases para el manejo de las películas en una base de datos. Estas clases serán la parte de "modelo" de la aplicación ASP.NET MVC.

Vamos a usar una tecnología de acceso a datos NET Framework conocido como el [Marco de la entidad](#) para definir y trabajar con estas clases del modelo. Entity Framework (a menudo referida como EF) apoya un paradigma de desarrollo llamado *Primer Código*. Primer Código le permite crear objetos del modelo escribiendo clases simples. (Estos son también conocidos como clases POCO, de "plain-viejos objetos CLR."), Entonces usted puede tener la base de datos creada sobre la marcha de sus clases, lo que permite un flujo de trabajo de desarrollo muy limpia y rápida.

Adición de Clases de modelos

En el **Explorador de soluciones**, haga clic derecho en la carpeta *Modelos*, seleccione **Agregar** y, a continuación, seleccione **Clase**.



Introduzca el nombre de la clase "Movie".

Agregue los siguientes cinco propiedades a la **Movie** de clase:

```
pública    clase    Película
{
    pública    int ID {    conseguir;    establecido;    }
    pública    string Título {    conseguir;    establecido;    }
    pública    DateTime ReleaseDate {    conseguir;    establecido;
}
    pública    string Género {    conseguir;    establecido;    }
    pública    decimal Precio {    conseguir;    establecido;    }
}
```

Usaremos la **Movie** clase para representar las películas en una base de datos. Cada instancia de una **Movie** objeto se corresponderá con una fila dentro de una tabla de base de datos, y cada propiedad de la **Movie** clase mapear a una columna en la tabla.

En el mismo archivo, agregue la siguiente **MovieDbContext** clase:

```
pública     clase     MovieDbContext     :     DbContext
{
    pública     DbSet <Película>     Películas     {     conseguir;
establecido;     }
}
```

El **MovieDbContext** clase representa el contexto de la base de la película de Entity Framework, que se ocupa de ir a buscar, almacenar y actualizar **Movie** instancias de clase en una base de datos. El **MovieDbContext** deriva del **DbContext** clase base proporcionada por el Marco de la entidad.

Con el fin de poder hacer referencia **DbContext** y **DbSet** , es necesario agregar la siguiente **using** declaración en la parte superior del archivo:

```
utilizando     .. Sistema de Datos de Entidad;
```

El archivo completo *Movie.cs* se muestra a continuación. (Varios usando declaraciones que no son necesarios se han eliminado.)

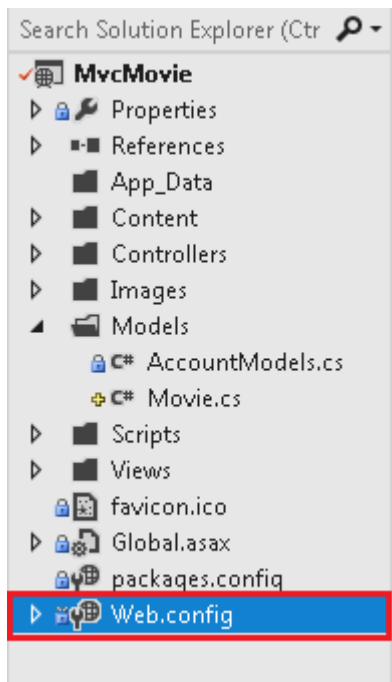
```
using System;
utilizando System.Data.Entity;

MvcMovie.Models de espacio de nombres
{
    public class Movie
    {
        int ID público {get; establecido; }
        string Título pública {get; establecido; }
        pública DateTime ReleaseDate {get; establecido; }
        public string Género {get; establecido; }
        Precio público decimal {get; establecido; }
    }

    MovieDbContext clase pública: DbContext
    {
        públicas DbSet <Película> Películas {get; establecido; }
    }
}
```

Crear una cadena de conexión y trabajar con SQL Server LocalDB

El **MovieDbContext** clase que ha creado se encarga de la tarea de conectar a las bases de datos y cartografía **Movie** objetos a los registros de base de datos. Una pregunta que podría preguntar, sin embargo, es cómo especificar qué base de datos se va a conectar. Que va a hacer que mediante la adición de información de conexión en el archivo *Web.config* de la aplicación. Abra el archivo *Web.config* raíz de la aplicación. (No el archivo *Web.config* en la carpeta *Views*.) Abra el archivo *Web.config* se indica en rojo.



Agregue la siguiente cadena de conexión a la `<connectionStrings>` elemento en el archivo *Web.config*.

```
<Add name = "MovieDBContext"
      connectionString = "Data Source = (LocalDB) \ v11.0; AttachDBFileName = |
DataDirectory | \ Movies.mdf; Integrated Security = True"
      providerName = "System.Data.SqlClient"
/>
```

El siguiente ejemplo muestra una parte del archivo *Web.config* con la nueva cadena de conexión añadido:

```
<ConnectionStrings>
  <Add name = "DefaultConnection"
        connectionString = "Data Source = (LocalDB) \ v11.0; Initial Catalog
= aspnet-MvcMovie-2012213181139; Integrated Security = true"
        providerName = "System.Data.SqlClient"
  />
  <Add name = "MovieDBContext"
        connectionString = "Data Source = (LocalDB) \ v11.0; AttachDBFileName
= | DataDirectory | \ Movies.mdf; Integrated Security = True"
        providerName = "System.Data.SqlClient"
  />
</ ConnectionStrings>
```

Esta pequeña cantidad de código y XML es todo lo que necesita para escribir con el fin de representar y almacenar los datos de la película en una base de datos.

A continuación, usted construirá un nuevo **MoviesController** clase que se puede utilizar para mostrar los datos de la película y permitir a los usuarios crear nuevas listas de películas.

Acceso a los datos de su modelo desde un controlador

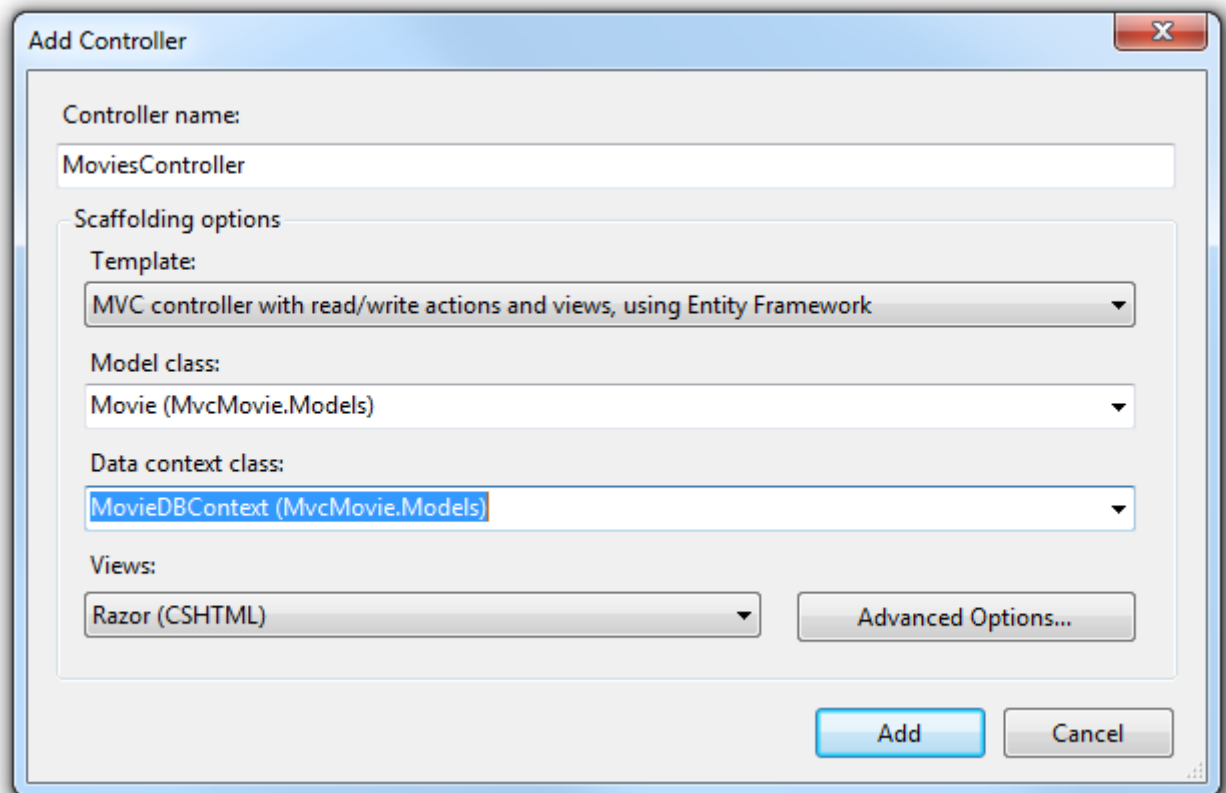
En esta sección, vamos a crear un nuevo **MoviesController** clase y escribir código que recupera los datos de la película y lo muestra en el navegador utilizando una plantilla de vista.

Generar la aplicación antes de pasar al siguiente paso.

Haga clic derecho en la carpeta de *controladores* y crear un

nuevo **MoviesController** controlador. Las siguientes opciones no aparecerán hasta generar la aplicación. Seleccione las siguientes opciones:

- Nombre del controlador: **MoviesController**. (Este es el valor predeterminado.)
- **Acciones y puntos de vista del controlador MVC con lectura / escritura, utilizando Entity Framework:**Plantilla.
- Clase del modelo: **Película (MvcMovie.Models)**.
- Datos de clase de contexto: **MovieDbContext (MvcMovie.Models)**.
- Vistas: **Razor (CSHTML)**. (El valor predeterminado.)

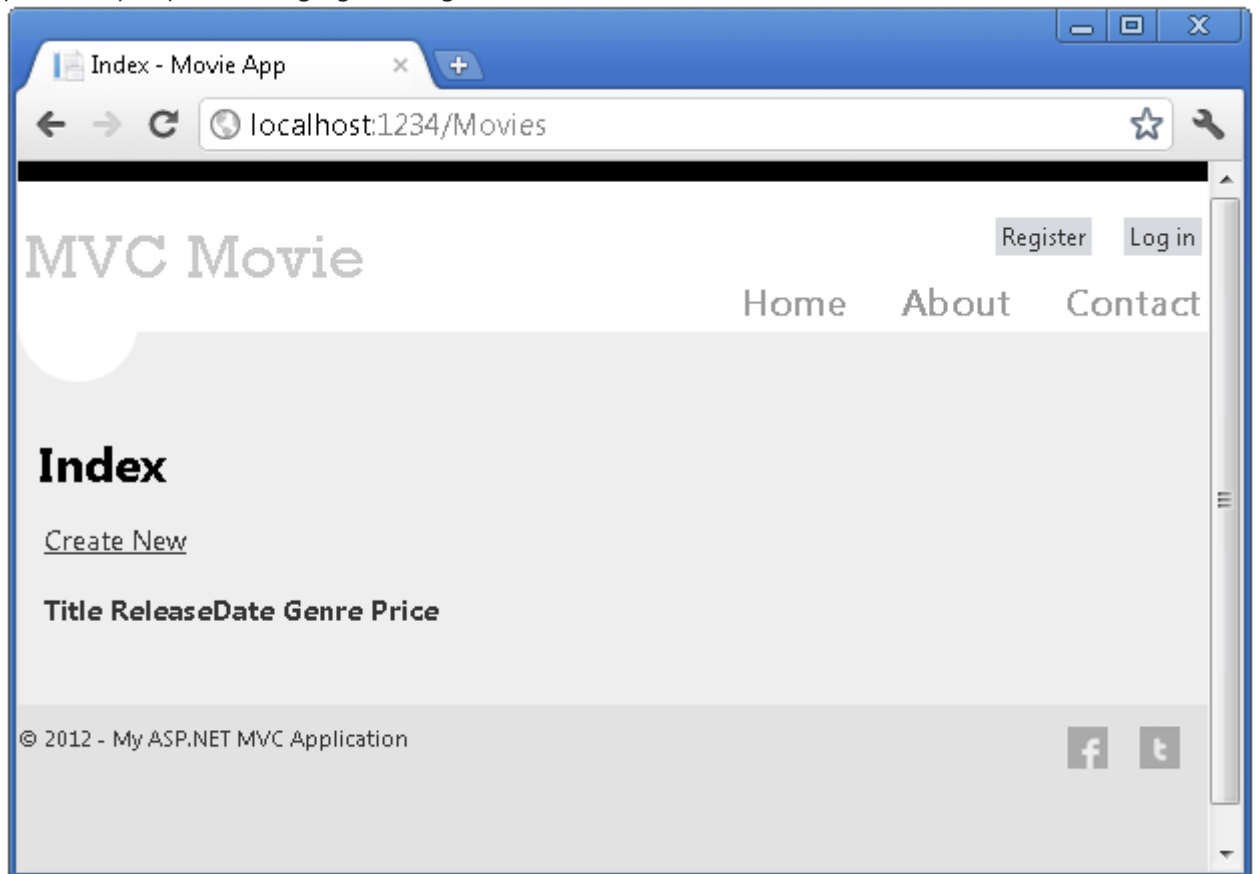


Haga clic en **Agregar**. Visual Studio Express crea los siguientes archivos y carpetas:

- Un archivo *MoviesController.cs* en la carpeta *Controllers* del proyecto.
- A carpeta *Películas* en la carpeta *Vistas* del proyecto.
- *Create.cshtml*, *Delete.cshtml*, *Details.cshtml*, *Edit.cshtml* y *Index.cshtml* en las nuevas vistas \ carpeta *Películas*.

ASP.NET MVC 4 crea automáticamente el CRUD (crear, leer, actualizar y eliminar) los métodos de acción y puntos de vista para que (la creación automática de los métodos y puntos de vista de acción CRUD se conoce como andamios).Ahora tiene una aplicación web completamente funcional que te permite crear, listar, editar y eliminar entradas de cine.

Ejecutar la aplicación y busque el **Movies** controlador añadiendo */ Películas* a la URL en la barra de direcciones de su navegador. Debido a que la aplicación se basa en la ruta por defecto (definido en el archivo *Global.asax*), el navegador petición *http: // localhost: xxxx / Películas* se encamina al default **Index** método de acción del **Movies** controlador. En otras palabras, el navegador petición *http: // localhost: xxxx / Películas* es efectivamente el mismo que el navegador petición *http: // localhost: xxxx / Cine / Index*. El resultado es una lista vacía de películas, porque no ha agregado ninguna todavía.



Crear una película

Seleccione el enlace **crear nuevo**. Introduce algunos detalles sobre una película y luego haga clic en el botón **Crear**.

Create - Movie App

localhost:1234/Movies/Create

MVC Movie

Register Log in

Home About Contact

Create

Title

When Harry Met Sally

ReleaseDate

1/11/1989

Genre

Comedy

Price

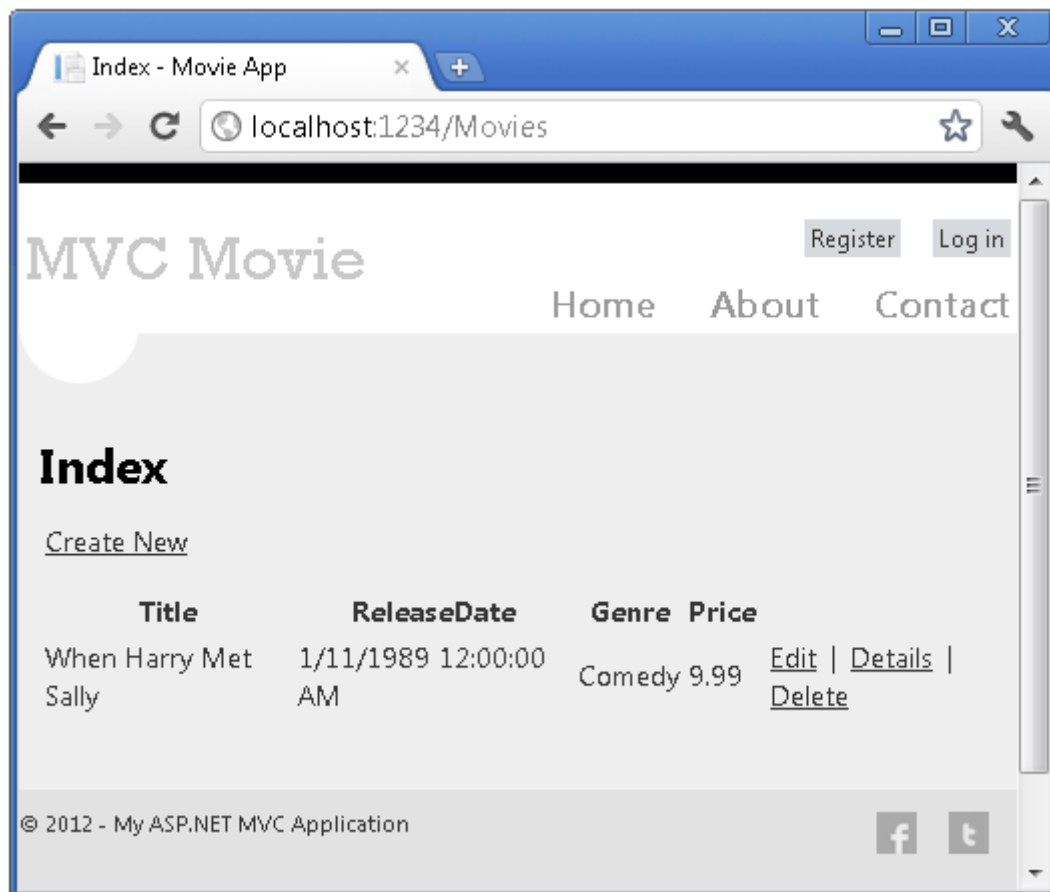
9.99

Create

[Back to List](#)

© 2012 - My ASP.NET MVC Application

Al hacer clic en el botón **Crear** causa la forma que se ha publicado en el servidor, donde la información de la película se guarda en la base de datos. Usted está entonces redirigido a la URL / *Películas*, donde se puede ver la película que acaba de crear en el anuncio.



Crear un par de entradas de cine. Prueba el **Editar**, **Detalles** y **Borrar** vínculos, que son todos funcionales.

Evaluación del código generado

Abra los *Controladores* \ archivo *MoviesController.cs* y examinar la generada **Index** método. Una parte del controlador de la película con el **Index** método se muestra a continuación.

```
pública    clase    MoviesController    :    Controlador
{
    privado    MovieDbContext db =    nuevo    MovieDbContext ();

    //
    // GET: / Cine /

    Índice ActionResult pública ()
    {
        retorno    Ver (db Películas ToList ());
    }
}
```

La siguiente línea de la **MoviesController** clase crea una instancia del contexto de la base de la película, como se describió anteriormente. Usted puede utilizar el contexto de la base de datos de películas para consultar, editar y borrar películas.

```
privado    MovieDbContext db =    nuevo    MovieDbContext ();
```

Una petición al **Movies** controlador devuelve todas las entradas en el **Movies** tabla de la base de datos de la película y luego pasa los resultados al **Index** vista.

Modelos inflexible de tipos y la palabra clave Model

Anteriormente en este tutorial, usted vio cómo un controlador puede pasar los datos u objetos de una plantilla de vista utilizando el **ViewBag** objeto. El **ViewBag** es un objeto dinámico que proporciona una forma de enlace en tiempo conveniente para pasar información a una vista. ASP.NET MVC también ofrece la posibilidad de pasar los datos u objetos con establecimiento inflexible de una plantilla de vista. Este enfoque inflexible de tipos permite una mejor comprobación de su código y IntelliSense más rico en el editor de Visual Studio en tiempo de compilación. El mecanismo de andamios en Visual Studio utiliza este enfoque con los **MoviesController** plantillas de clase y ver cuando creó los métodos y puntos de vista. En los controladores \ *MoviesController.cs* archivo examinar la generada **Details** método. Una porción de la película con el controlador de **Details** método se muestra a continuación.

```
pública ActionResult detalles (int id = 0)
{
    Movie movie = db.Movies.Find (id);
    if (película == null)
    {
        volver HttpNotFound ();
    }
    volver Ver ( película );
}
```

Si una **Movie** se encuentra, una instancia de la **Movie** se pasa al modelo de la vista Detalles. Examine el contenido de las Vistas \ *Movies* \ archivo *Details.cshtml*.

Al incluir un **@model** declaración en la parte superior del archivo de vista de plantilla, puede especificar el tipo de objeto que la opinión espera. Cuando creó el controlador de la película, Visual Studio incluye automáticamente los siguientes **@model** declaración en la parte superior del archivo *Details.cshtml*:

```
Model MvcMovie.Models.Movie
```

Esta **@model** directiva le permite acceder a la película que el controlador pasa a la vista mediante un **Model** de objeto que está fuertemente tipado. Por ejemplo, en la plantilla *Details.cshtml*, el código pasa cada campo de película a los **DisplayNameFor** y **DisplayFor** HTML Helpers con el establecimiento inflexible **Model** objeto. El Cree y métodos de edición y visualización de plantillas también pasar un objeto de modelo de película.

Examine la plantilla de vista *Index.cshtml* y el **Index** método en el archivo *MoviesController.cs*. Observe cómo el código crea una **List** de objetos cuando se llama a la **View** método de ayuda en el **Index** método de acción. El código a continuación, pasa esta **Movies** lista desde el controlador a la vista:

```
Índice ActionResult pública ()
{
    volver Ver ( db.Movies.ToList () );
}
```

Cuando creó el controlador de la película, Visual Studio Express incluye automáticamente los siguientes **@model** declaración en la parte superior del archivo *Index.cshtml*:

```
Model IEnumerable <MvcMovie. Modelos. Movie>
```

Esta **@model** directiva le permite acceder a la lista de películas que el controlador pasa a la vista mediante un **Model** de objeto que está fuertemente tipado. Por ejemplo, en la plantilla *Index.cshtml*, el código recorre las películas por hacer un **foreach** declaración sobre el establecimiento inflexible **Model** de objetos:

```

foreach ( elemento var en el Modelo ) {
    <Tr>
        <Td>
            @ Html.DisplayFor (ModelItem => item.Title )
        </ Td>
        <Td>
            @ Html.DisplayFor (ModelItem => item.ReleaseDate )
        </ Td>
        <Td>
            @ Html.DisplayFor (ModelItem => item.Genre )
        </ Td>
        <Td>
            @ Html.DisplayFor (ModelItem => item.Price )
        </ Td>
        <Th>
            @ Html.DisplayFor (ModelItem => item.Rating )
        </ Th>
        <Td>
            @ Html.ActionLink ("Edit", "Editar", nuevo { id = item.ID })
|
            @ Html.ActionLink ("Detalles", "Detalles", { id = item.ID })
|
            @ Html.ActionLink ("Borrar", "Borrar", { id = item.ID })
        </ Td>
    </ Tr>
}

```

Debido a que el **Model** de objeto es de tipo fuerte (como un **IEnumerable<Movie>** objeto), cada **item** objeto en el bucle se escribe como **Movie** . Entre otras ventajas, esto significa que usted obtiene en tiempo de compilación comprobación del código y la compatibilidad con IntelliSense completa en el editor de código:

MvcMovie - Microsoft Visual Studio Express 2012 for... Quick Launch (Ctrl+Q)

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST WINDOW HELP

Index.cshhtml* Details.cshhtml MoviesController.cs Movie.cs

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Genre)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Price)
        </th>
        <th></th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
                @Html.ActionLink("Detail", "Details", new { id=item.ID }) |
                @Html.ActionLink("Delete", "Delete", new { id=item.ID })
            </td>
        </tr>
    }
</table>
```

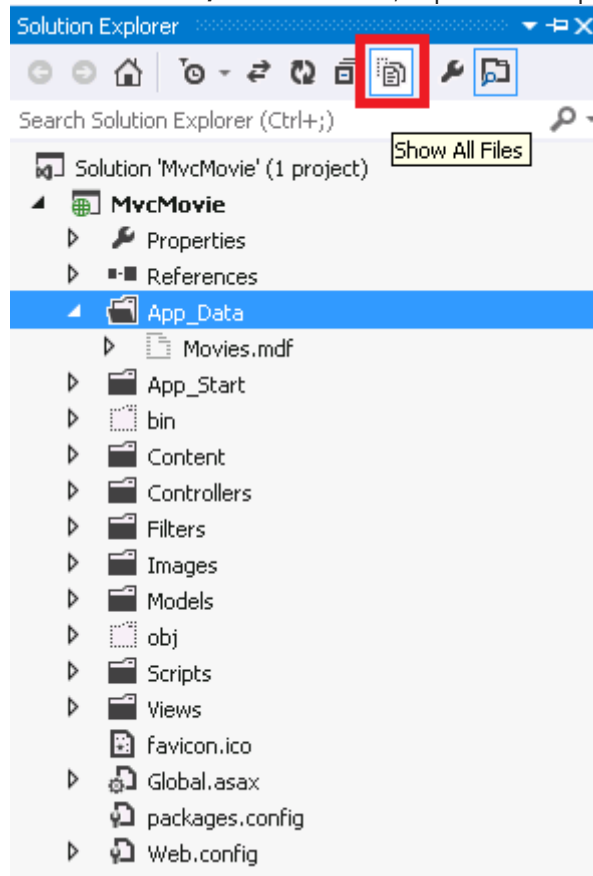
100 %

Source <table> <tr> <td>

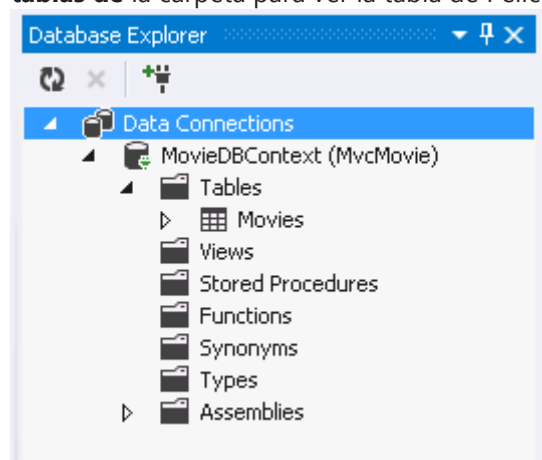
Ready Ln 41 Col 38 Ch 38 INS

Trabajar con SQL Server LocalDB

Entity Framework Code Primera detectó que la cadena de conexión de base de datos que se proporcionó señaló una **Movies** base de datos que no existía todavía, así Primer Código creó la base de datos de forma automática. Usted puede comprobar que se ha creado por mirar en la carpeta *App_Data*. Si no ve el archivo *Movies.mdf*, haga clic en el botón **Mostrar todos los archivos** en la barra de herramientas **Explorador de soluciones**, haga clic en el botón **Actualizar**, a continuación, expanda la carpeta *App_Data*.



Haga doble clic para abrir *Movies.mdf* **BASE DE DATOS EXPLORER**, a continuación, expanda **las tablas de** la carpeta para ver la tabla de Películas.



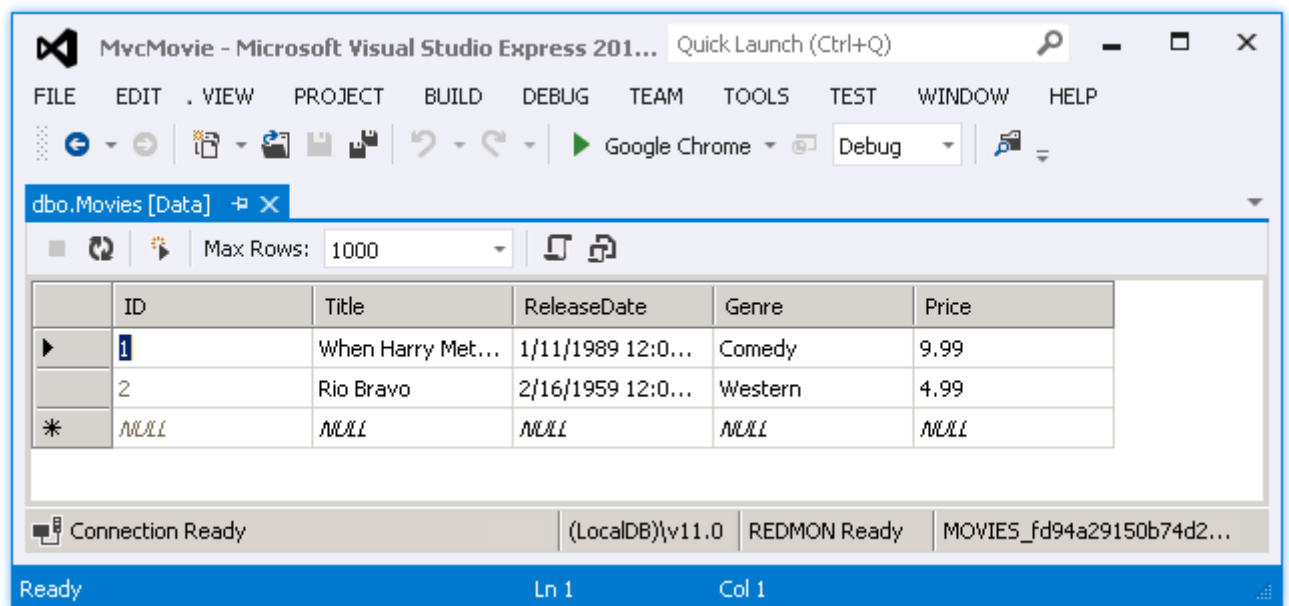
Nota: Si el explorador de base de datos no aparece, en el menú **Herramientas**, seleccione **Conectar con base de datos**, a continuación, se interrumpe el diálogo **Elegir origen de datos**. Esto obligará a abrir el explorador de base de datos.

Nota: Si está utilizando VWD o Visual Studio 2010 y recibo un error similar a cualquiera de las siguientes situaciones siguientes:

- La base de datos 'C:\ Webs \ MVC4 \ MVCMOVIE \ MVCMOVIE \ App_Data \ MOVIES.MDF' no se puede abrir porque es la versión 706 de este servidor es compatible con la versión 655 y anteriores. Un camino rebaja no es compatible.
- "Excepción no controlada InvalidOperationException era por código de usuario" El SqlConnection suministrado no especifica un catálogo inicial.

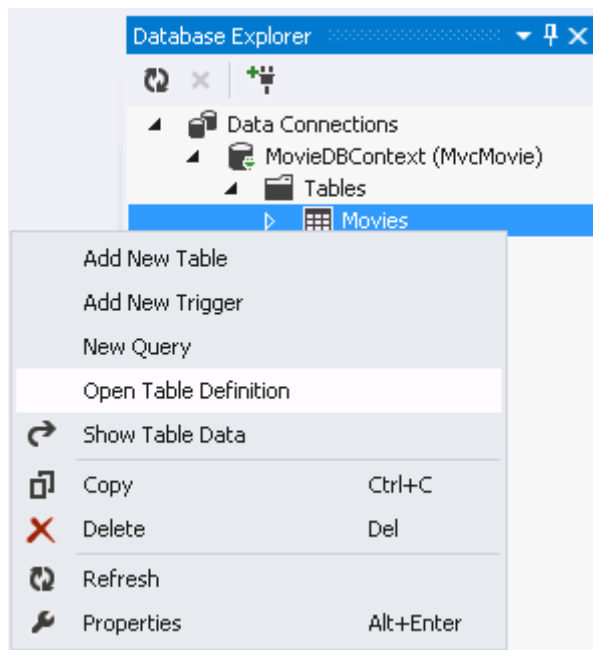
Es necesario instalar las [herramientas de datos de SQL Server](#) y [LocalDB](#) . Verifique la **MovieDbContext** cadena de conexión especificada en la página anterior.

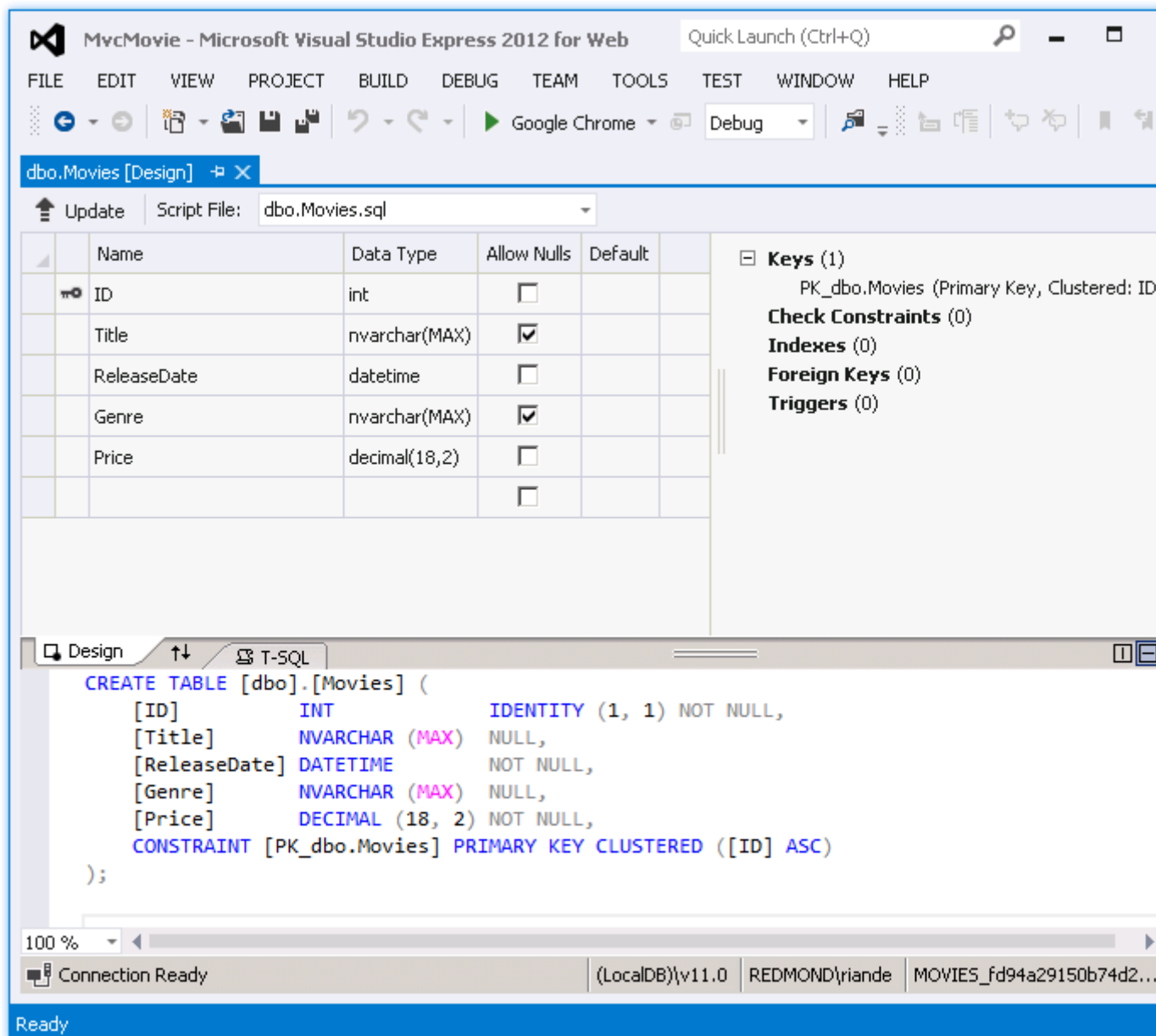
Haga clic con el **Movies** tabla y seleccione **Mostrar datos de tabla** para ver los datos que ha creado.



ID	Title	ReleaseDate	Genre	Price
1	When Harry Met...	1/11/1989 12:0...	Comedy	9.99
2	Rio Bravo	2/16/1959 12:0...	Western	4.99
NULL	NULL	NULL	NULL	NULL

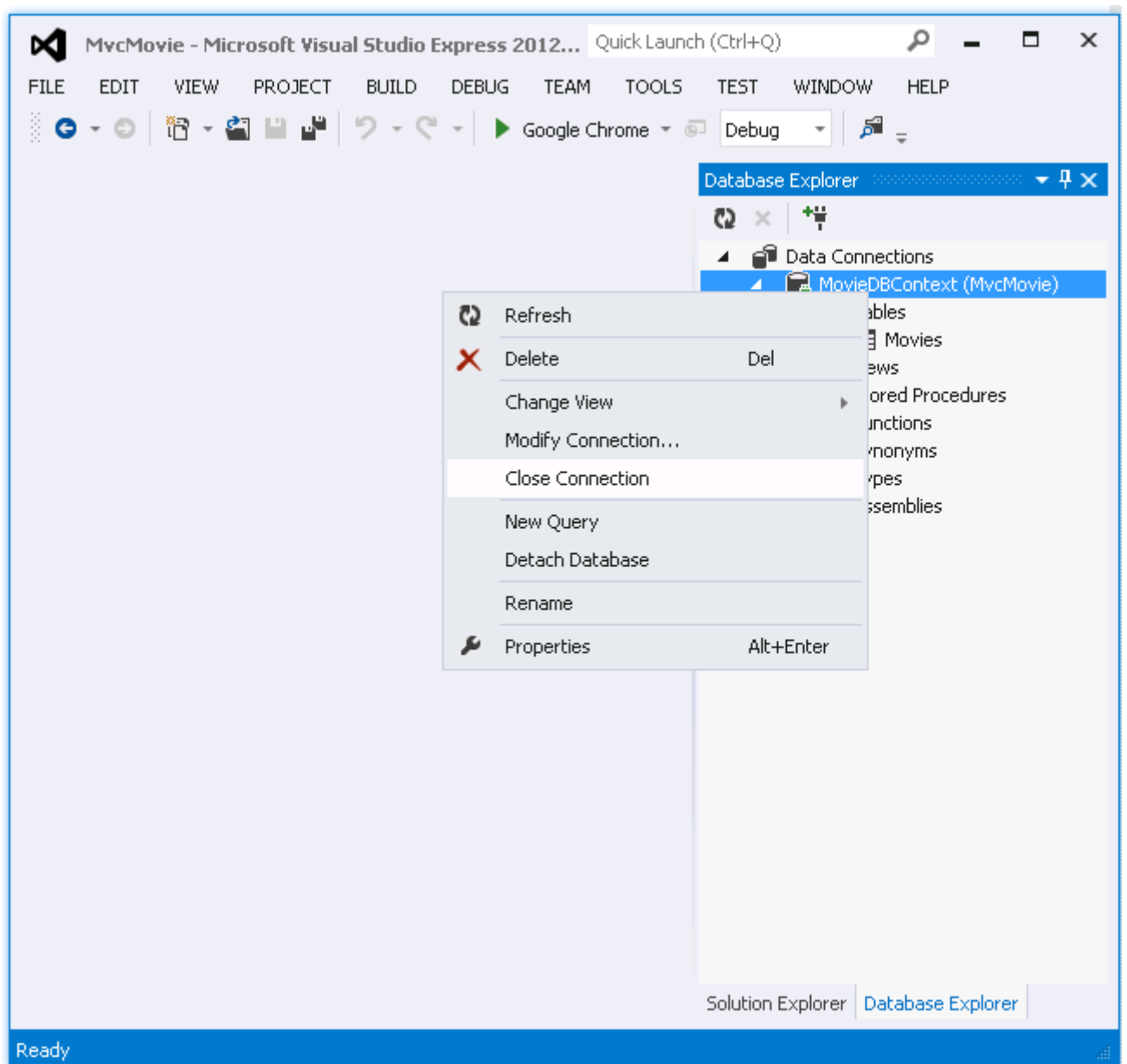
Haga clic con el **Movies** tabla y seleccione **Abrir tabla de definición** para ver la estructura de la tabla que el Código Entity Framework Primera creado para usted.





Observe cómo el esquema del **Movies** tabla asigna a la **Movie** de clase que creó anteriormente. Entity Framework Código Primero crea automáticamente este esquema para usted basado en su **Movie** clase.

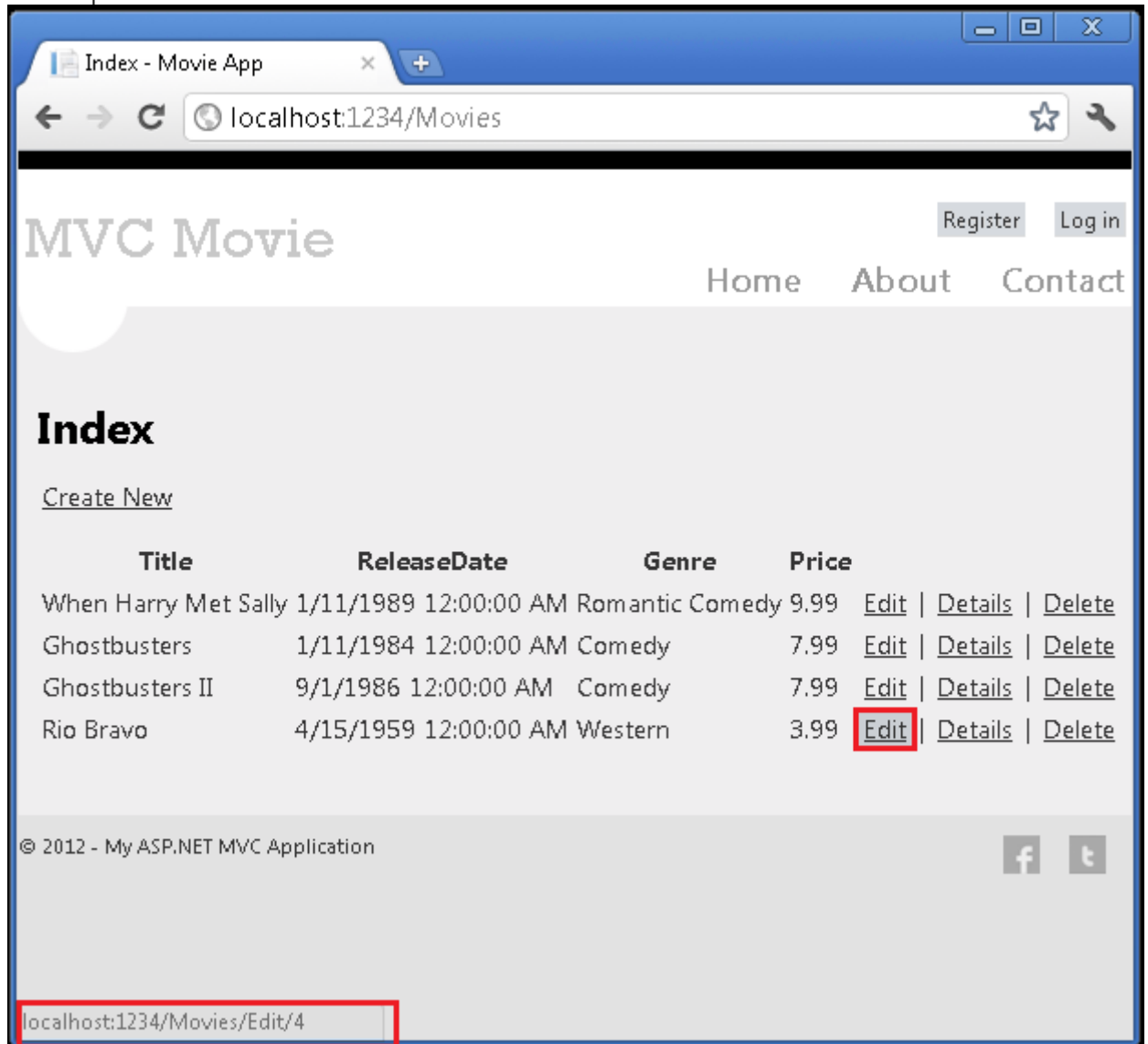
Cuando haya terminado, cierre la conexión haciendo clic derecho y seleccionando **MovieDbContext Cerrar conexión**. (Si no cierra la conexión, es posible que obtenga un error la próxima vez que se ejecuta el proyecto).



Ahora tiene la base de datos y una página simple enumeración para mostrar el contenido de la misma. En el siguiente tutorial, vamos a examinar el resto del código de andamiaje y añadimos un **SearchIndex** método y un **SearchIndex** vista que le permite buscar películas en esta base de datos.

Examinar los métodos de edición y vista de edición

En esta sección, podrás examinar los métodos de acción generados y puntos de vista para el controlador de la película. Entonces vamos a añadir una página de búsqueda personalizada. Ejecutar la aplicación y busque el **Movies** controlador añadiendo */ Películas* a la URL en la barra de direcciones de su navegador. Mantenga el puntero del ratón sobre un enlace **Editar** para ver la URL que enlaza con.



El enlace **Editar** fue generada por la `Html.ActionLink` método en las *Vistas \ Movies* \ vista *Index.cshtml*:

```
@ Html.ActionLink ("Edit", "Editar", nueva {id = item.ID})
```

<td>	
<code>@Html.ActionLink("Edit Me", "Edit", new { id=item.ID }) </code>	
<code>@Html.</code>	(extension) MvcHtmlString HtmlHelper.ActionLink(string linkText, string actionName, object routeValues)
<code>@Html.</code>	Returns an anchor element (a element) that contains the virtual path of the specified action.
<code>@Html.</code>	Exceptions:
	System.ArgumentException

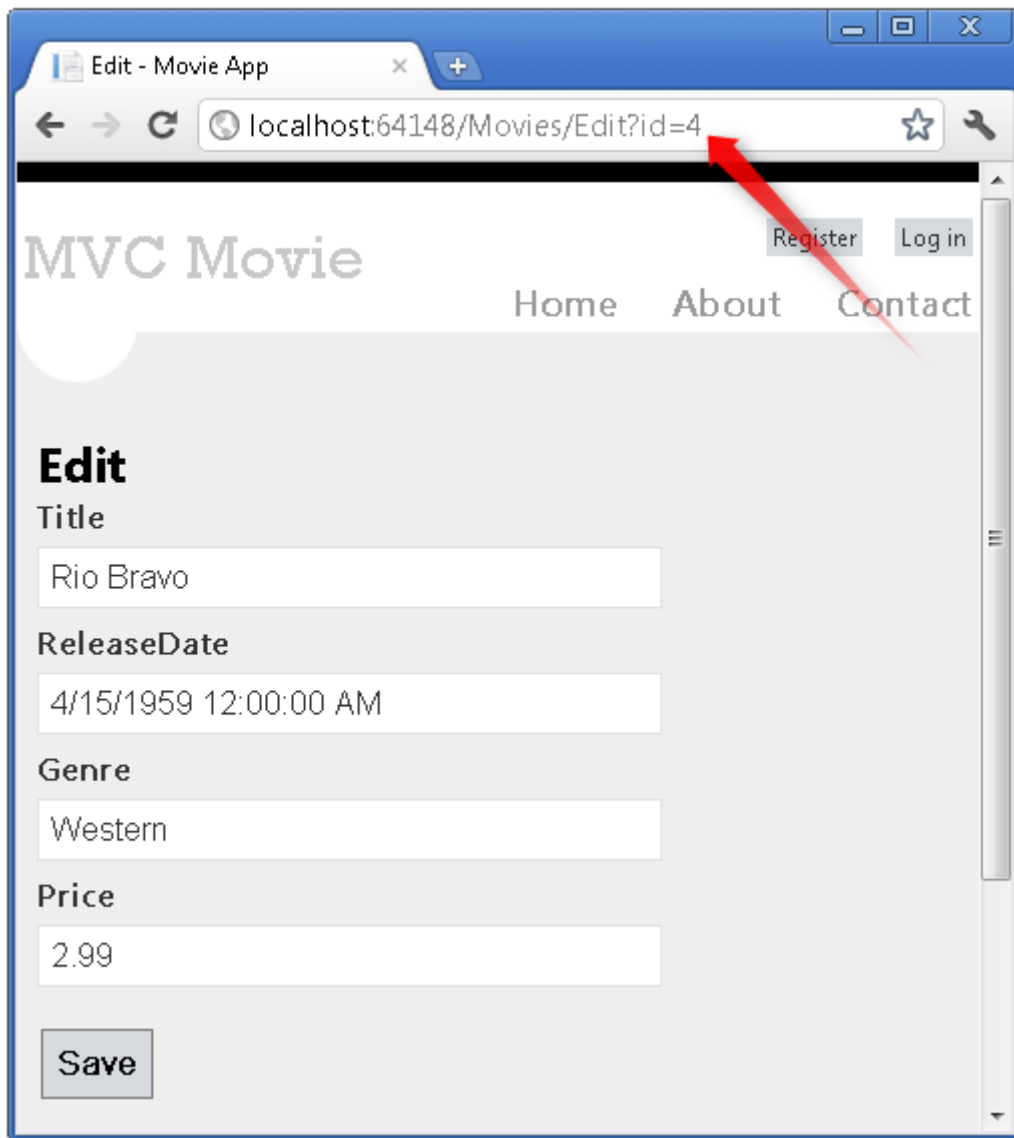
El **Html** objeto es un ayudante que está expuesta utilizando una propiedad en la [System.Web.Mvc.WebViewPage](#) clase base. El **ActionLink** método del ayudante hace que sea fácil de generar dinámicamente HTML hipervínculos que enlazan con métodos de acción en los controladores. El primer argumento de la **ActionLink** método es el texto del enlace a render (por ejemplo, `<a>Edit Me`). El segundo argumento es el nombre del método de acción a invocar. El último argumento es un [objeto anónimo](#) que genera los datos de la ruta (en este caso, el ID de 4).

El enlace generado se muestra en la imagen anterior es `http://localhost:xxxxx/Cine/Editar/4`. La ruta por defecto (establecido en `App_Start \ RouteConfig.cs`) realiza el patrón de URL `{controller}/{action}/{id}`. Por lo tanto, ASP.NET traduce `http://localhost:xxxxx/Cine/Editar/4` en una petición al **Edit** método de acción del **Movies** controlador con el parámetro **ID** igual a 4 Examine el siguiente código de la `App_Start \ RouteConfig.cs` presentar.

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

    routes.MapRoute (
        nombre: "Default",
        url: "{controlador} / {acción} / {id}",
        predeterminados: nuevo controlador = {"Home", action = "Índice",
            id = UrlParameter.Optional}
    );
}
```

También puede pasar parámetros del método de acción utilizando una cadena de consulta. Por ejemplo, la URL `http://localhost:xxxxx/Cine/EditarID=4` también pasa el parámetro **ID** del 4 al **Edit** método de acción del **Movies** controlador.



Abra el **Movies** controlador. Los dos **Edit** métodos de acción se muestran a continuación.

```
//  
// GET: / Cine / Editar / 5  
  
pública ActionResult Edit (int id = 0)  
{  
    Movie movie = db.Movies.Find (id);  
    if (película == null)  
    {  
        volver HttpNotFound ();  
    }  
    volver Ver (película);  
}  
  
//  
// POST: / Cine / Editar / 5  
  
[HttpPost]
```



```

pública ActionResult Edit (película Película)
{
    if (ModelState.IsValid)
    {
        db.Entry (película) .Estado = EntityState.Modified;
        db.SaveChanges ();
        volver RedirectToAction ("Índice");
    }
    volver Ver (película);
}

```

Observe la segunda **Edit** método de acción es precedida por la **HttpPost** atributo. Este atributo especifica que que la sobrecarga de la **Edit** método se puede invocar sólo para peticiones POST. Se podría aplicar el **HttpGet** atributo al primer método de edición, pero eso no es necesario porque es el valor predeterminado. (Nos referimos a los métodos de acción que se asignan implícitamente la **HttpGet** atributo como **HttpGet** métodos.)

El **HttpGet Edit** método toma el parámetro ID película, busca la película con Entity Framework **Find** método, y devuelve la película seleccionada a la vista de edición. El parámetro ID especifica un valor por defecto de cero si la **Edit** se llama al método sin parámetros. Si una película no se puede encontrar, HttpNotFound se devuelve. Cuando el sistema de andamiaje creado la vista de edición, se examinó la **Movie** clase y crea código para renderir **<label>** y **<input>** elementos para cada propiedad de la clase. El siguiente ejemplo muestra la vista de edición que se ha generado:

```

Model MvcMovie.Models.Movie

@ {
    ViewBag.Title = "Editar";
}

<H2> Editar </ h2>

using (Html.BeginForm ()) {
    @ Html.ValidationSummary (true)

    <Fieldset>
        <Legend> Película </ legend>

        @ Html.HiddenFor (model => model.ID)

        <Div class = "editor de etiqueta">
            @ Html.LabelFor (model => Model.Title)
        </ Div>
        <Div class = "editor-campo">
            @ Html.EditorFor (model => Model.Title)
            @ Html.ValidationMessageFor (model => Model.Title)
        </ Div>

        <Div class = "editor de etiqueta">
            @ Html.LabelFor (model => model.ReleaseDate)
        </ Div>
        <Div class = "editor-campo">

```

```

        @ Html.EditorFor (model => model.ReleaseDate)
        @ Html.ValidationMessageFor (model => model.ReleaseDate)
    </ Div>

    <Div class = "editor de etiqueta">
        @ Html.LabelFor (model => model.Genre)
    </ Div>
    <Div class = "editor-campo">
        @ Html.EditorFor (model => model.Genre)
        @ Html.ValidationMessageFor (model => model.Genre)
    </ Div>

    <Div class = "editor de etiqueta">
        @ Html.LabelFor (model => model.Price)
    </ Div>
    <Div class = "editor-campo">
        @ Html.EditorFor (model => model.Price)
        @ Html.ValidationMessageFor (model => model.Price)
    </ Div>

    <P>
        <Input type = "submit" value = "Guardar" />
    </ P>
</ Fieldset>
}

<Div>
    @ Html.ActionLink ("Volver a la lista", "Índice")
</ Div>

Scriptssection {
    @ Scripts.Render ("~/ paquetes / jqueryval")
}

```

Observe cómo la plantilla de vista tiene una `@model MvcMovie.Models.Movie` declaración en la parte superior del archivo - especifica que la opinión espera que el modelo de la plantilla de vista sea de tipo `Movie`.

El código scaffolded utiliza varios *métodos de ayuda* para agilizar el marcado

HTML. El `Html.LabelFor` helper muestra el nombre del campo ("Título", "ReleaseDate", "Género", o "Precio"). El `Html.EditorFor` ayudante hace un

HTML `<input>` elemento. El `Html.ValidationMessageFor` helper muestra ningún mensaje de validación asociados con esa propiedad.

Ejecutar la aplicación y vaya a la `/ Películas` URL. Haga clic en un enlace **Editar**. En el navegador, ver el código fuente de la página. El código HTML para el elemento de formulario se muestra a continuación.

```

<Form action = "/ Películas / Editar / 4" method = "post"> <fieldset>
    <Legend> Película </ legend>

    <Input data-val = "true"-val-número de datos = "El ID de campo debe
ser un número." requiere datos-val = "Se requiere que el campo ID." id =
name = tipo "ID" "ID" = valor "oculto" = "4" />

```

```

    <Div class = "editor de etiqueta">
        <Label for = "Title"> Título </ label>
    </ Div>
    <Div class = "editor-campo">
        <Input class = "text-box de una sola línea" id = "Título" name
= "Title" type = "text" value = "Rio Bravo" />
        <Span class = "field-validación-válido" datos-valmsg-for =
"Título" datos-valmsg-replace = "true"> </ span>
    </ Div>

    <Div class = "editor de etiqueta">
        <Label for = "ReleaseDate"> ReleaseDate </ label>
    </ Div>
    <Div class = "editor-campo">
        <Input class = "text-box de una sola línea"-val de datos =
"true"-val-fecha data = "El ReleaseDate campo debe ser una fecha." requiere
datos-val-= "Se requiere que el campo ReleaseDate." id = "ReleaseDate" name
= "ReleaseDate" type = "text" value = "4/15/1959 12:00:00 AM" />
        <Span class = "field-validación-válido" datos-valmsg-for =
"ReleaseDate" datos-valmsg-replace = "true"> </ span>
    </ Div>

    <Div class = "editor de etiqueta">
        <Label for = "Género"> Género </ label>
    </ Div>
    <Div class = "editor-campo">
        <Input class = "text-box de una sola línea" id = name = tipo
"Género" "Género" = "text" value = "occidental" />
        <Span class = "field-validación-válido" datos-valmsg-for =
"Género" datos-valmsg-replace = "true"> </ span>
    </ Div>

    <Div class = "editor de etiqueta">
        <Label for = "Precio"> Precio </ label>
    </ Div>
    <Div class = "editor-campo">
        <= clase de entrada de datos-val = "text-box de una sola línea"
"verdaderos" data-val-cuenta = "El campo Precio debe ser un número."
requiere datos-val-= "Se requiere que el campo Precio." id = "Precio" name =
"Precio" type = "text" value = "2.99" />
        <Span class = "field-validación-válido" datos-valmsg-for =
"Precio" datos-valmsg-replace = "true"> </ span>
    </ Div>

    <P>
        <Input type = "submit" value = "Guardar" />
    </ P>
</ Fieldset>
</ Form>

```

Los `<input>` elementos se encuentran en un archivo HTML `<form>` elemento cuya `action` atributo está establecido para publicar a la `/Cine/Editar` URL. Los datos del formulario se publicarán en el servidor cuando se hace clic en el botón **Editar**.

El procesamiento de la solicitud POST

El siguiente listado muestra la `HttpPost` versión de la `Edit` método de acción.

```
[HttpPost]
pública ActionResult Edit (película Película)
{
    if (ModelState.IsValid)
    {
        db.Entry (película) .Estado = EntityState.Modified;
        db.SaveChanges ();
        volver RedirectToAction ("Índice");
    }
    volver Ver (película);
}
```

El [ligante modelo ASP.NET MVC](#) toma los valores del formulario publicado y crea una `Movie` objeto que se pasa como la `movie` de parámetros. El `ModelState.IsValid` método verifica que los datos presentados en el formulario se pueden utilizar para modificar (editar o actualizar) un `Movie` objeto. Si los datos son válidos, los datos del vídeo se guarda en el `Movies` colección de la `db(MovieDBContext` ejemplo). Los nuevos datos de la película se guarda en la base de datos llamando al `SaveChanges` método de `MovieDBContext` . Después de guardar los datos, el código redirige al usuario al `Index` método de acción de la `MoviesController` clase, que muestra el de la colección de películas, incluyendo los cambios que acaba de hacer.

Si los valores publicados no son válidos, se vuelven a mostrar en el formulario. Los `Html.ValidationMessageFor` ayudantes en la plantilla de vista `Edit.cshtml` cuidan de mostrar mensajes de error adecuados.

Edit - Movie App

localhost:64148/Movies/Edit/2

MVC Movie

Register Log in

Home About Contact

Edit

Title

Ghostbusters

ReleaseDate

abc

The field ReleaseDate must be a date.

Genre

Comedy

Price

zyz

The field Price must be a number.

Save

[Back to List](#)

© 2012 - My ASP.NET MVC Application

f t

Nota para apoyar la validación jQuery para las configuraciones regionales distintos del inglés que utilizan una coma (",") para el punto decimal, debe incluir *globalize.js* y sus *culturas* específicas / archivo *globalize.cultures.js* (de <https://github.com/jquery/globalize>) y Javascript para usar **Globalize.parseFloat**. El siguiente código muestra las modificaciones de las Vistas \ Movies \ Edit.cshtml archivo para trabajar con la cultura "fr-FR":

```
Scriptssection {
    @ Scripts.Render ("~/ paquetes / jqueryval")
    <Script src = "~/ scripts / globalize.js"> </ script>
    <Script src = "~/ Scripts / globalize.culture.fr-FR.js"> </ script>
```

```

<Script>
    $.validator.methods.number = Function (valor, elemento) {
        volver this.optional (elemento) ||
            ! isNaN (Globalize.parseFloat (valor));
    }
    $(Document) .ready (function () {
        Globalize.culture ("fr-FR");
    });
</ Script>
<Script>
    jQuery.extend (jQuery.validator.methods, {
        rango: function (valor, elemento, param) {
            // Utilice el plugin Globalización para analizar el valor
            var val = $.global.parseFloat (valor);
            volver this.optional (elemento) || (
                val > parametro [0] && val <= parametro [1]);
        }
    });
</ Script>
}

```

El campo decimal puede requerir una coma, no un punto decimal. Como solución temporal, puede agregar el elemento de la globalización en el archivo web.config raíz proyectos. El siguiente código muestra el elemento de la globalización con la cultura se establece en Estados Unidos Inglés.

```

<System.web>
  <Cultura globalización = "en-US" />
  <-! Elementos eliminados para mayor claridad ->
</system.web>

```

Todos los **HttpGet** métodos siguen un patrón similar. Consiguen un objeto de película (o una lista de objetos, en el caso del **Index**), y pasan el modelo a la vista. El **Create** método pasa un objeto de película vacío a la vista Crear. Todos los métodos que crear, editar, borrar o modificar los datos lo hacen en el **HttpPost** sobrecarga del método. Modificación de datos en un método HTTP GET es un riesgo de seguridad, tal como se describe en la entrada de blog [MVC ASP.NET Tip # 46 - No utilice Eliminar Vínculos porque crean agujeros de seguridad](#). Modificación de datos en un método GET HTTP también viola las mejores prácticas y la arquitectura [REST](#) patrón, que especifica que las peticiones GET no deberían cambiar el estado de su solicitud. En otras palabras, la realización de una operación GET debe ser una operación segura de que no tiene efectos secundarios y no modifica sus datos persistido.

Adición de un método de búsqueda y de búsqueda Ver

En esta sección vamos a añadir un **SearchIndex** método de acción que le permite buscar películas por género o nombre. Este estará disponible usando los `/Cine/SearchIndex` URL. La solicitud se mostrará un formulario HTML que contiene los elementos de entrada que puede introducir un usuario con el fin de buscar una película. Cuando un usuario envía el formulario, el método de acción obtendrá los valores de búsqueda enviados por el usuario y utilizar los valores para buscar en la base de datos.

Viendo el Formulario SearchIndex

Comience agregando un **SearchIndex** método de acción para el vigente **MoviesController** clase. El método devolverá una vista que contiene un formulario HTML. Aquí está el código:

```
pública ActionResult SearchIndex (searchstring cadena)
{
    películas var = de m en db.Movies
        seleccione m;

    if (! String.IsNullOrEmpty (searchstring))
    {
        películas = movies.Where (s => s.Title.Contains (searchstring));
    }

    Regresar ver (películas);
}
```

La primera línea de la **SearchIndex** método crea el siguiente [LINQ](#) consulta para seleccionar las películas:

```
películas var = de m en db.Movies
    seleccione m;
```

La consulta se define en este punto, pero aún no se ha ejecutado en el almacén de datos.

Si el **searchString** parámetro contiene una cadena, la consulta películas se modifica para filtrar en el valor de la cadena de búsqueda, usando el siguiente código:

```
if (! String.IsNullOrEmpty (searchstring))
{
    películas = movies.Where (s => s.Title.Contains (searchstring));
}
```

El **s => s.Title** código anterior es una [expresión lambda](#) . Lambdas se utilizan en basado en métodos [de LINQ](#) consultas como argumentos a los métodos estándar de operador de consulta, como el [caso de](#) método utilizado en el código anterior. Consultas LINQ no se ejecutan cuando se definen o cuando se modifican llamando a un método como el **Where** o **OrderBy** . En su lugar, la ejecución de consultas se difiere, lo que significa que la evaluación de una expresión se retrasa hasta su valor dado cuenta es en realidad repiten a lo largo o el **ToList** método se llama. En el **SearchIndex** muestra, la consulta se ejecuta en la vista SearchIndex. Para obtener más información acerca de la ejecución de consultas diferidas, vea [Ejecución de consultas](#) .

Ahora usted puede poner en práctica el **SearchIndex** vista que mostrará el formulario al usuario. Haga clic dentro del **SearchIndex** método y luego haga clic en **Añadir vista**. En el cuadro de diálogo **Agregar vista**, especifica que se va a pasar una **Movie** objeto a la plantilla de la vista como su clase del modelo. En la lista **de plantillas Andamios**, elija **lista**, haga clic en **Agregar**.

Add View

View name:
SearchIndex

View engine:
Razor (CSHTML)

☒ Create a strongly-typed view
Model class:
Movie (MvcMovie.Models)

Scaffold template:
List ☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:
...
(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:
MainContent

Add Cancel

Al hacer clic en el botón **Agregar**, se crearon las *Vistas \ Movies \ SearchIndex.cshtml* plantilla de vista. Debido a que la **lista** seleccionada en la lista **de plantillas Andamios**, Visual Studio genera automáticamente (andamiaje) algunas marcas por defecto en la vista. El andamiaje creado un formulario HTML. Examinó el **Movie** clase y creado código para rendir **<label>** elementos para cada propiedad de la clase. La lista muestra la vista Cree que se ha generado:

```
Model IEnumerable <MvcMovie.Models.Movie>

@ {
    ViewBag.Title = "SearchIndex";
}

<H2> SearchIndex </ h2>

<P>
    @ Html.ActionLink ("Crear nuevo", "Crear")
</ P>
<Table>
    <Tr>
        <Th>
            Título
        </ Th>
        <Th>
```



```

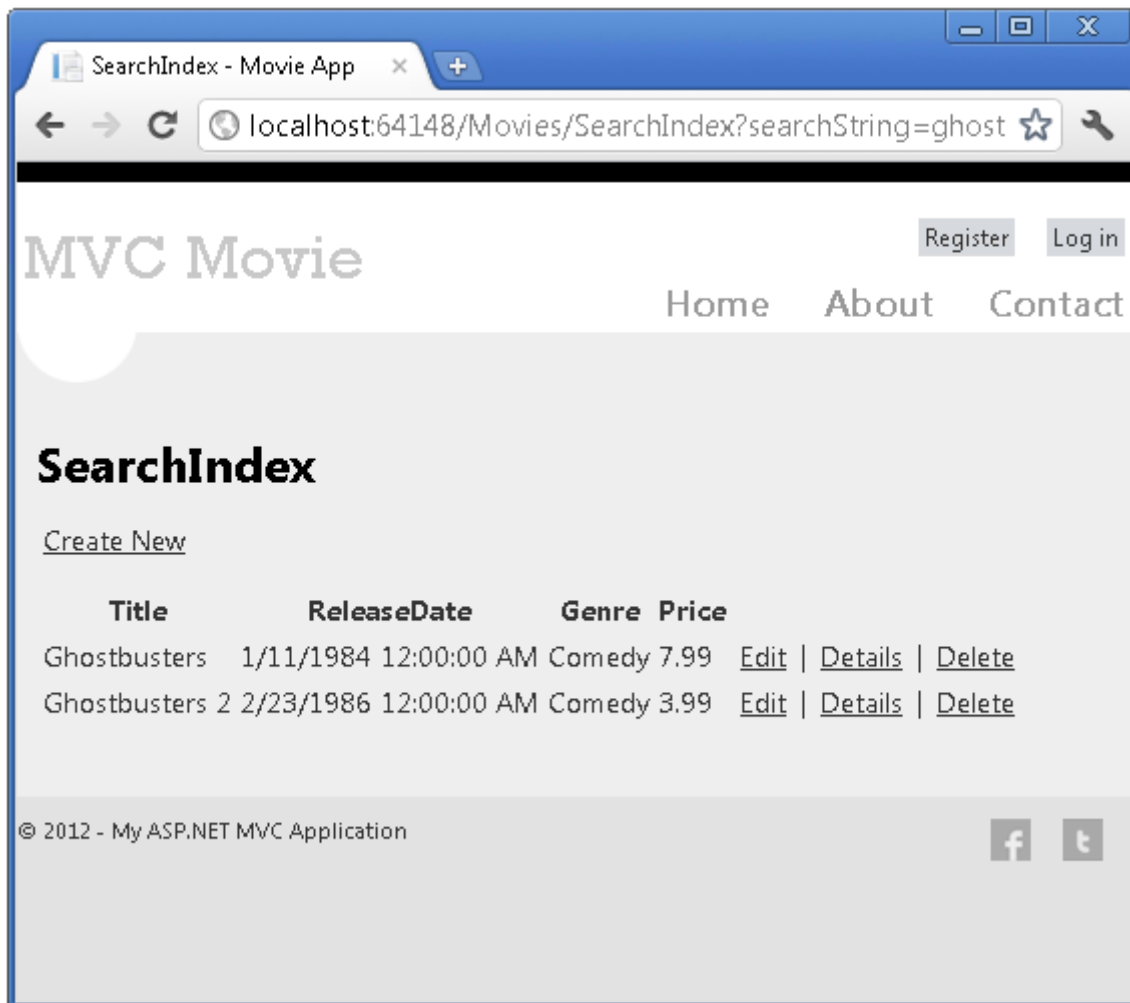
        ReleaseDate
    </ Th>
    <Th>
        Género
    </ Th>
    <Th>
        Precio
    </ Th>
    <Th> </ th>
</ Tr>

foreach (elemento var en el Modelo) {
    <Tr>
        <Td>
            @ Html.DisplayFor (ModelItem => item.Title)
        </ Td>
        <Td>
            @ Html.DisplayFor (ModelItem => item.ReleaseDate)
        </ Td>
        <Td>
            @ Html.DisplayFor (ModelItem => item.Genre)
        </ Td>
        <Td>
            @ Html.DisplayFor (ModelItem => item.Price)
        </ Td>
        <Td>
            @ Html.ActionLink ("Edit", "Editar", nueva {id = item.ID}) |
            @ Html.ActionLink ("Detalles", "Detalles", nueva {id = item.ID})
        |
            @ Html.ActionLink ("Borrar", "Borrar", nueva {id = item.ID})
        </ Td>
    </ Tr>
}

</ Table>

```

Ejecutar la aplicación y vaya a */ Películas / SearchIndex*. Anexar una cadena de consulta como *?searchString=ghost* de la URL. Se muestran las películas filtrados.



Si cambiar la firma del **SearchIndex** método para tener un parámetro denominado **id**, la **id** de parámetros coincidirá con el **{id}** marcador de posición para las rutas por defecto establecidos en el archivo *Global.asax*.

{Controlador} / {acción} / {id}

El original **SearchIndex** método es el siguiente ::

```
pública ActionResult SearchIndex (searchstring cadena)
{
    películas var = de m en db.Movies
        seleccione m;

    if (! String.IsNullOrEmpty (searchstring))
    {
        películas = movies.Where (s => s.Title.Contains (searchstring));
    }

    Regresar ver (películas);
}
```

La modificación **SearchIndex** método se vería de la siguiente manera:

```
pública ActionResult SearchIndex ( string id )
{
```

```

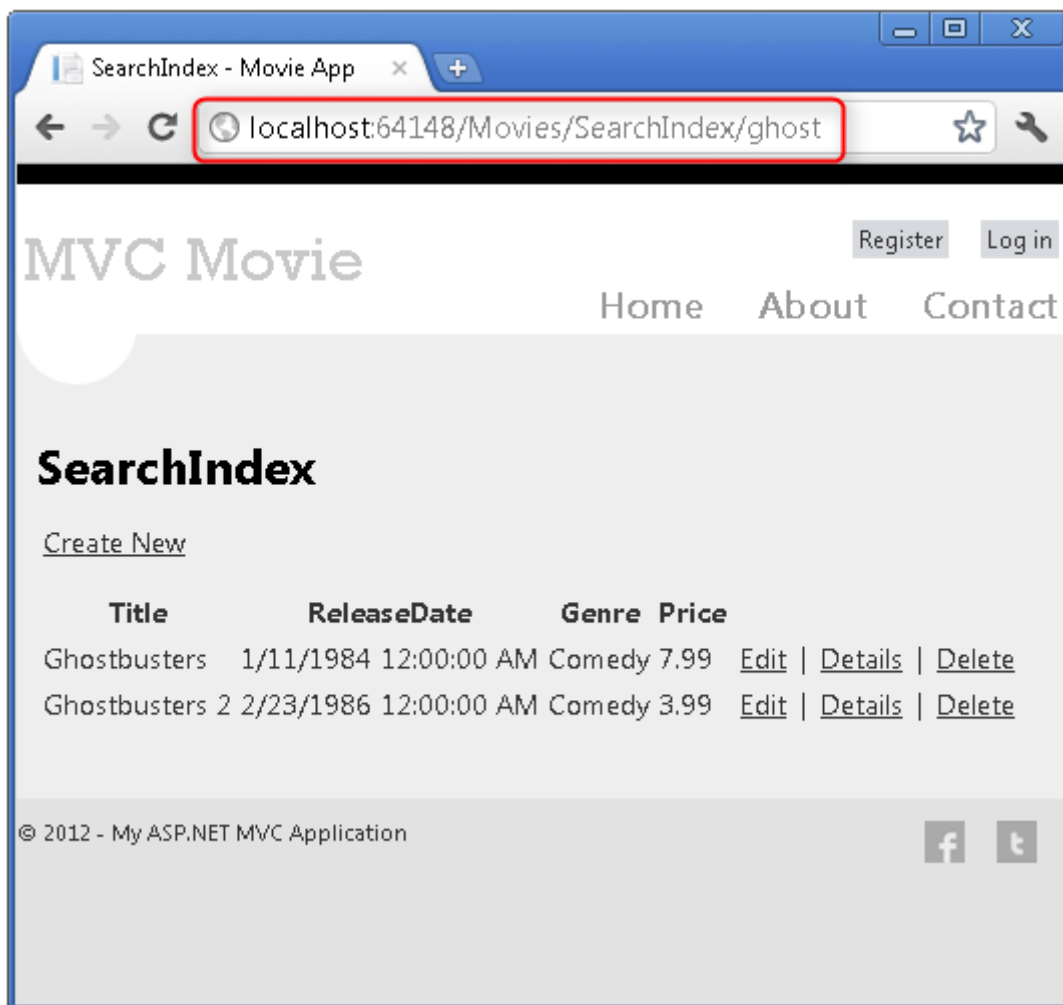
        cadena searchstring = Identificación;
        películas var = de m en db.Movies
            seleccione m;

        if (! String.IsNullOrEmpty (searchstring))
        {
            películas = movies.Where (s => s.Title.Contains (searchstring));
        }

        Regresar ver (películas);
    }

```

Ahora puede pasar el título de búsqueda como datos de la ruta (un segmento URL) en lugar de como un valor de cadena de consulta.



Sin embargo, no se puede esperar que los usuarios modifiquen la dirección cada vez que quieren buscar una película. Así que ahora vamos a añadir la interfaz de usuario para ayudarles a películas de filtro. Si ha cambiado la firma del **SearchIndex** método para probar cómo pasar el parámetro ID ruta determinada, cambie de nuevo para que su **SearchIndex** método toma un parámetro de cadena denominada **searchString**:

```

pública ActionResult SearchIndex (searchstring cadena)
{
    películas var = de m en db.Movies

```

```

        seleccione m;

        if (! String.IsNullOrEmpty (searchstring))
        {
            películas = movies.Where (s => s.Title.Contains (searchstring));
        }

        Regresar ver (películas);
    }

```

Abra las *Reproducciones \ Movies \ SearchIndex.cshtml* archivo, y justo después de `@Html.ActionLink("Create New", "Create")`, agregue lo siguiente:

```

using (Html.BeginForm ()) {
    <P> Título: @ Html.TextBox ("SearchString") <br />
    <Input type = "submit" value = "Filtro" /> </ p>
}

```

El siguiente ejemplo muestra una parte de las *vistas \ Movies \ SearchIndex.cshtml* archivo con el marcado de filtración añadido.

```

Model IEnumerable <MvcMovie.Models.Movie>

@ {
    ViewBag.Title = "SearchIndex";
}

<H2> SearchIndex </ h2>

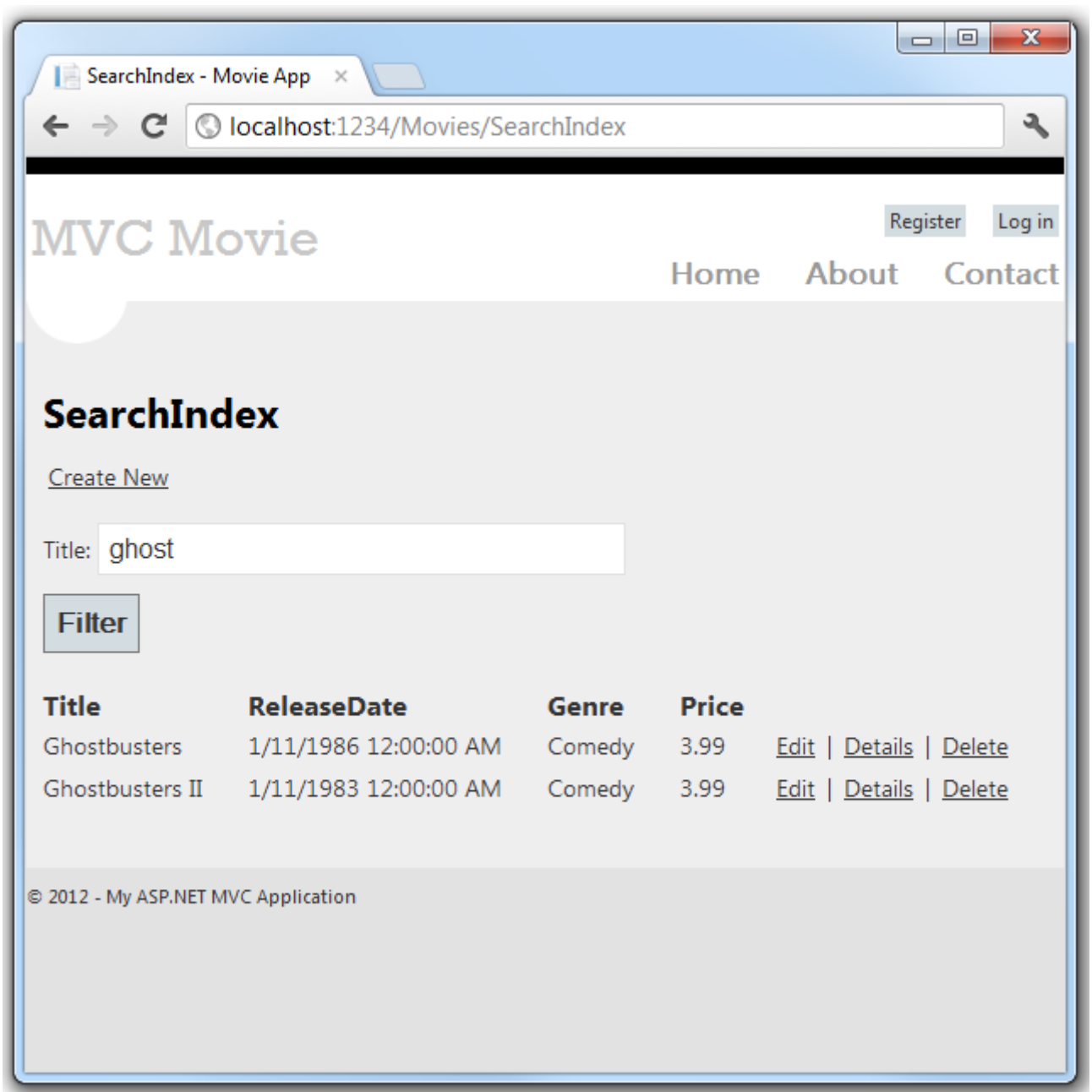
<P>
    @ Html.ActionLink ("Crear nuevo", "Crear")

    using (Html.BeginForm ()) {
        <P> Título: @ Html.TextBox ("SearchString") <br />
        <Input type = "submit" value = "Filtro" /> </ p>
    }
</ P>

```

El `Html.BeginForm` ayudante crea una abertura `<form>` tag. El `Html.BeginForm` ayudante hace que la forma para publicar a sí mismo cuando el usuario envía el formulario haciendo clic en el botón **Filtro**.

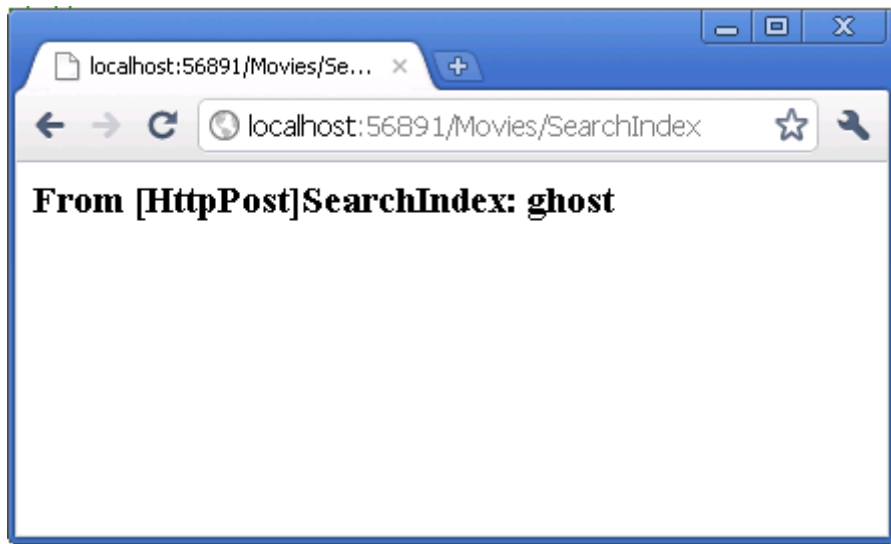
Ejecutar la aplicación y trate de buscar una película.



No hay **HttpPost** sobrecarga del **SearchIndex** método. Usted no lo necesita, ya que el método no cambia el estado de la aplicación, el filtrado de datos.

Se podría añadir el siguiente **HttpPost SearchIndex** método. En ese caso, el invocador de acción se correspondería con la **HttpPost SearchIndex** método, y el **HttpPost SearchIndex** método funcionaría como se muestra en la imagen de abajo.

```
[HttpPost]
SearchIndex cadena pública (FormCollection fc, searchstring cadena)
{
    retorno "<h3> Desde [HttpPost] SearchIndex:" + searchstring + "</ h3>";
}
```



Sin embargo, incluso si se agrega este **HttpPost** versión del **SearchIndex** método, hay una limitación en la manera en que todo esto ha sido implementado. Imagine que usted quiere marcar una búsqueda particular o si desea enviar un enlace a tus amigos que pueden hacer clic con el fin de ver la misma lista filtrada de películas. Observe que la dirección URL de la solicitud HTTP POST es la misma que la dirección URL de la solicitud GET (localhost: xxxxx / Cine / SearchIndex) - no hay información de búsqueda en la propia dirección URL. En este momento, la información de la cadena de búsqueda se envía al servidor como un valor de campo de formulario. Esto significa que no puede captar esa información de búsqueda de marcar o enviar a los amigos en una URL.

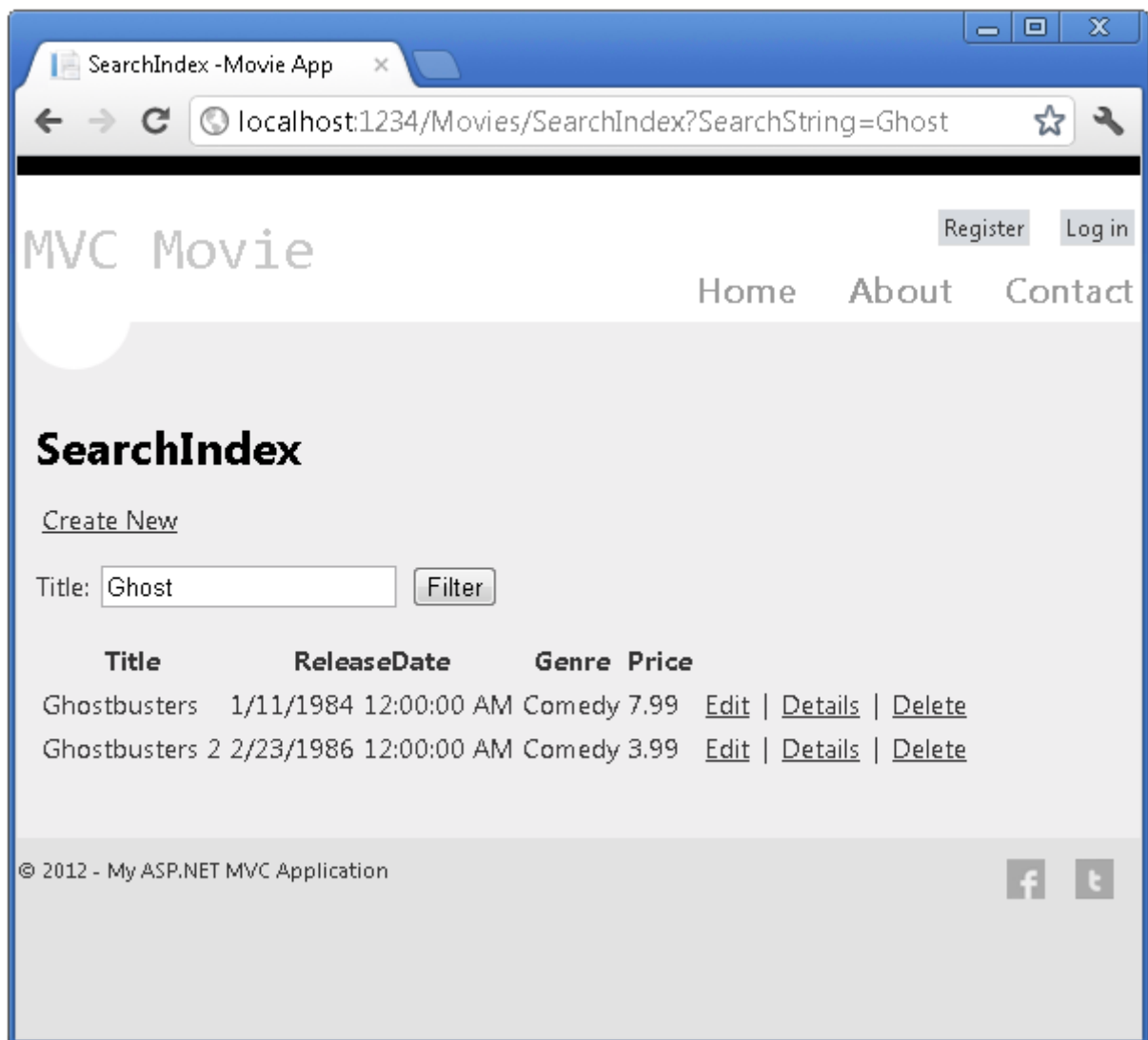
La solución es utilizar una sobrecarga de **BeginForm** que especifica que la solicitud POST debe agregar la información de búsqueda de la URL y que debe ser enviado a la **HttpGet** versión del **SearchIndex** método. Vuelva a colocar la sin parámetros existentes **BeginForm** método con la siguiente:

```
using (Html.BeginForm ("SearchIndex", "Películas" caseras, FormMethod.Get))
```

```
@Html.ActionLink("Create New", "Create")
@using (Html.BeginForm("SearchIndex", "Movies", FormMethod.Get))
{
    <p>
    @*
    <in
    </in
}
```

▲ 5 of 13 ▼ (extension) MvcForm HtmlHelper.BeginForm(string actionName, string controllerName, FormMethod method)
Writes an opening <form> tag to the response. When the user submits the form, the request will be processed by an action meth
method: The HTTP method for processing the form, either GET or POST.

Ahora, cuando usted envía una búsqueda, la URL contiene una cadena de consulta de búsqueda. La búsqueda también se destinará a la **HttpGet SearchIndex** método de acción, incluso si usted tiene un **HttpPost SearchIndex** método.



Adición Buscar por Género

Si ha añadido el **HttpPost** versión del **SearchIndex** método, borrarlos ahora.

A continuación, vamos a añadir una función para permitir a los usuarios buscar películas por género. Vuelva a colocar la **SearchIndex** método con el siguiente código:

```
pública ActionResult SearchIndex (cadena movieGenre, searchstring cadena)
{
    var GenreLst = new List <string> ();

    var = GenreQry de d en db.Movies
        orderby d.Genre
        seleccione d.Genre;
    GenreLst.AddRange (GenreQry.Distinct ());
    ViewBag.movieGenre = new SelectList (GenreLst);

    películas var = de m en db.Movies
        seleccione m;

    if (! String.IsNullOrEmpty (searchstring))
    {
```

```

        películas = movies.Where (s => s.Title.Contains (searchstring));
    }

    if (string.IsNullOrEmpty (movieGenre))
        Regresar ver (películas);
    demás
    {
        volver Ver (movies.Where (x => x.Genre == movieGenre));
    }
}

```

Esta versión de la **SearchIndex** método toma un parámetro adicional, a saber **movieGenre** . Las primeras líneas de código crean una **List** de objetos para mantener géneros cinematográficos, desde la base de datos.

El código siguiente es una consulta LINQ que recupera todos los géneros, desde la base de datos.

```

var = GenreQry de d en db.Movies
        orderby d.Genre
        seleccione d.Genre;

```

El código utiliza el **AddRange** método de la genérica **List** colección para agregar todos los géneros distintos a la lista. (Sin la **Distinct** modificador, se añadirían géneros duplicados - por ejemplo, se añadiría la comedia en dos ocasiones en nuestra muestra). El código a continuación, almacena la lista de los géneros en la **ViewBag** objeto.

El código siguiente muestra cómo comprobar el **movieGenre** parámetro. Si no está vacío, el código restringe aún más las películas de consulta para limitar las películas seleccionadas para el género especificado.

```

if (string.IsNullOrEmpty (movieGenre))
    Regresar ver (películas);
demás
{
    volver Ver (movies.Where (x => x.Genre == movieGenre));
}

```

Adición de marcado para la SearchIndex Ver para Apoyar Buscar por Género

Añadir un **Html.DropDownList** ayudante para las Vistas \ Movies

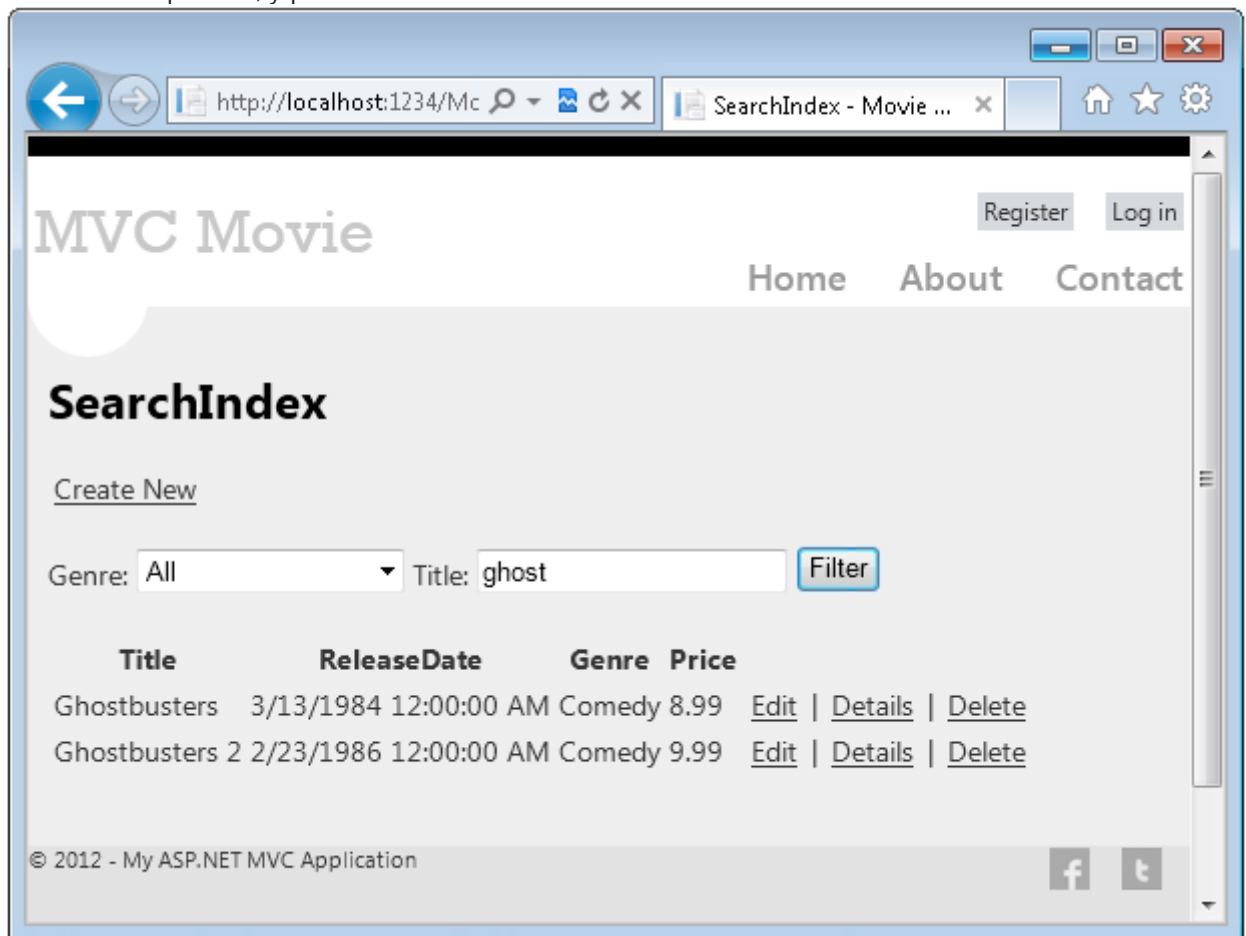
\ archivo *SearchIndex.cshtml*, justo antes del **TextBox** ayudante. El marcado completado se muestra a continuación:

```

<P>
    @ Html.ActionLink ("Crear nuevo", "Crear")
    using (Html.BeginForm ("SearchIndex", "Películas" caseras,
FormMethod.Get)) {
        <P> Género: @ Html.DropDownList ("movieGenre", "Todos")
            Título: @ Html.TextBox ("SearchString")
            <Input type = "submit" value = "Filtro" /> </ p>
    }
</ P>

```


Ejecutar la aplicación y vaya a / Cine / SearchIndex. Intente una búsqueda por género, por nombre de la película, y por tanto los criterios.



En esta sección usted examinó los métodos de acción CRUD y opiniones generadas por el marco. Ha creado un método de acción de búsqueda y la vista que permiten a los usuarios buscar por título de la película y el género. En la siguiente sección, veremos cómo añadir un alojamiento a la **Movie** de modelo y de cómo agregar un inicializador que creará automáticamente una base de datos de prueba.

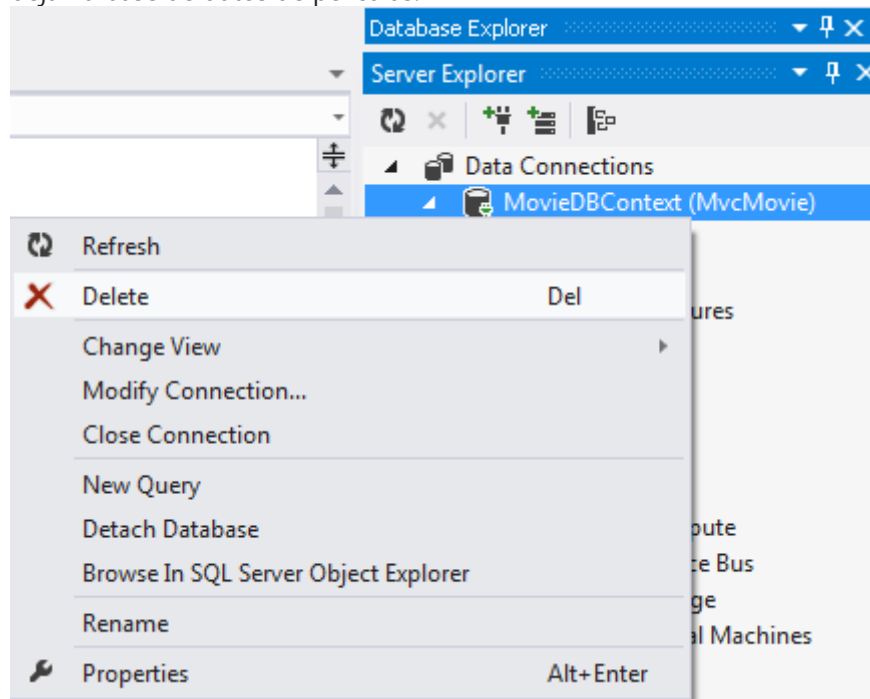
Adición de un nuevo campo para el modelo de la película y en la Tabla

En esta sección vamos a usar Entity Framework Código primeras migraciones de migrar algunos cambios en las clases del modelo de manera que se aplique el cambio a la base de datos. Por defecto, cuando se utiliza Código Entity Framework Primera para crear automáticamente una base de datos, como lo hizo anteriormente en este tutorial, Primer Código agrega una tabla a la base de datos para ayudar a rastrear si el esquema de la base de datos está en sintonía con el modelo de clases era generada a partir de. Si no están en sincronía, Entity Framework genera un error. Esto hace que sea más fácil de rastrear problemas en el tiempo de desarrollo que de otro modo sólo podría encontrar (por errores oscuros) en tiempo de ejecución.

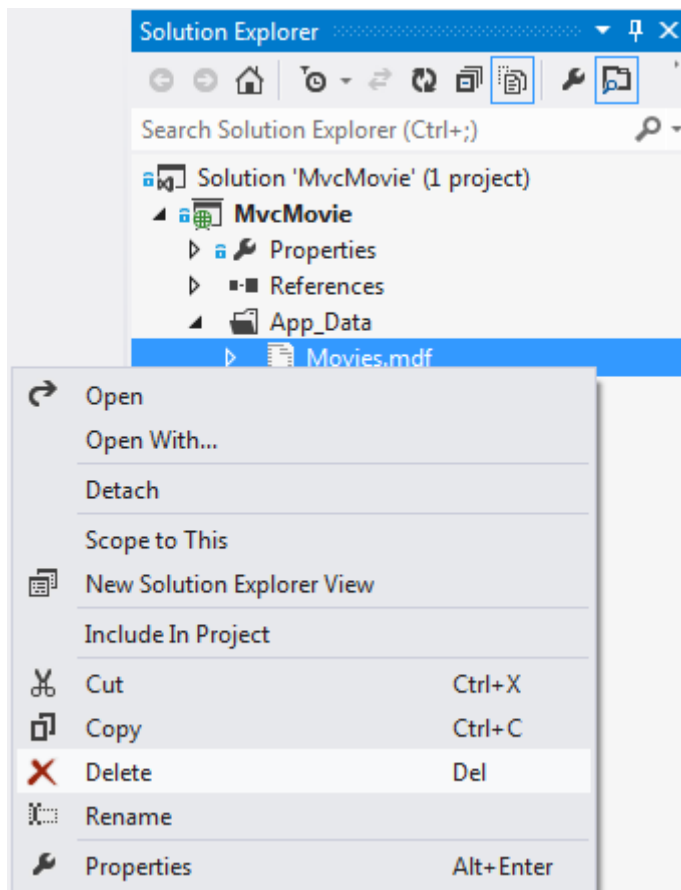
Configuración de Código primeras migraciones de los cambios de modelo

Si está utilizando Visual Studio 2012, haga doble clic en el archivo *Movies.mdf* desde el Explorador de soluciones para abrir la herramienta de base de datos. Visual Studio Express para Web mostrará el Explorador de base de datos, Visual Studio 2012 mostrará el Explorador de servidores. Si está utilizando Visual Studio 2010, utilice SQL Server Explorer Object.

En la herramienta de base de datos (Database Explorer, Explorador de servidores o el Explorador de objetos de SQL Server), haga clic derecho en **MovieDBContext** y seleccione **Eliminar** para dejar la base de datos de películas.

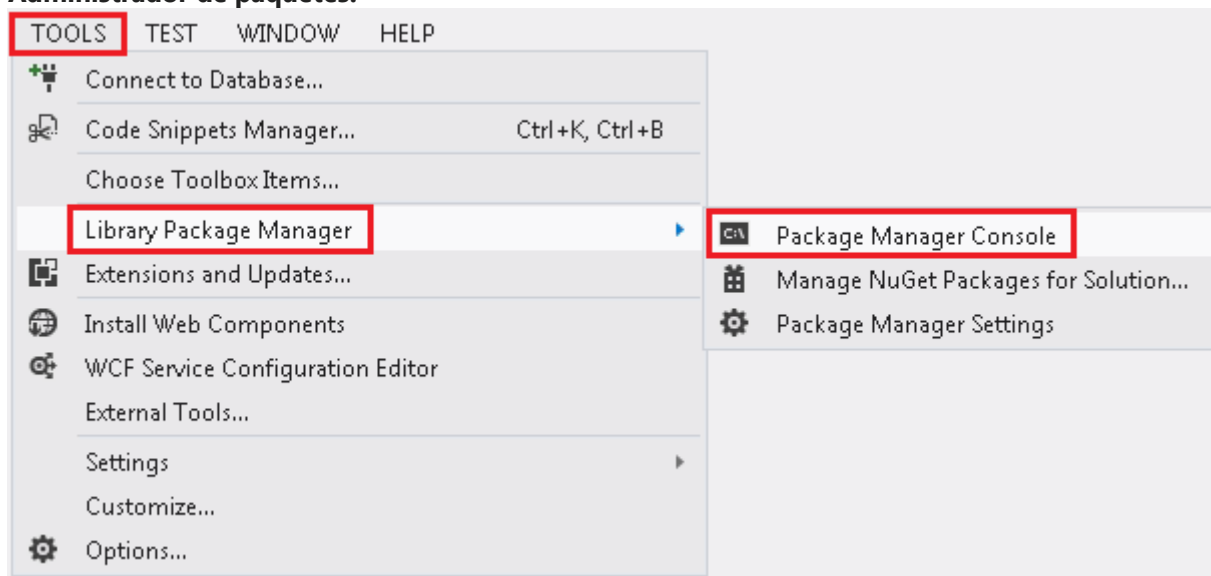


Regrese al Explorador de soluciones. Haga clic derecho sobre el archivo y seleccione *Movies.mdf* **Suprimir** para eliminar la base de datos de películas.

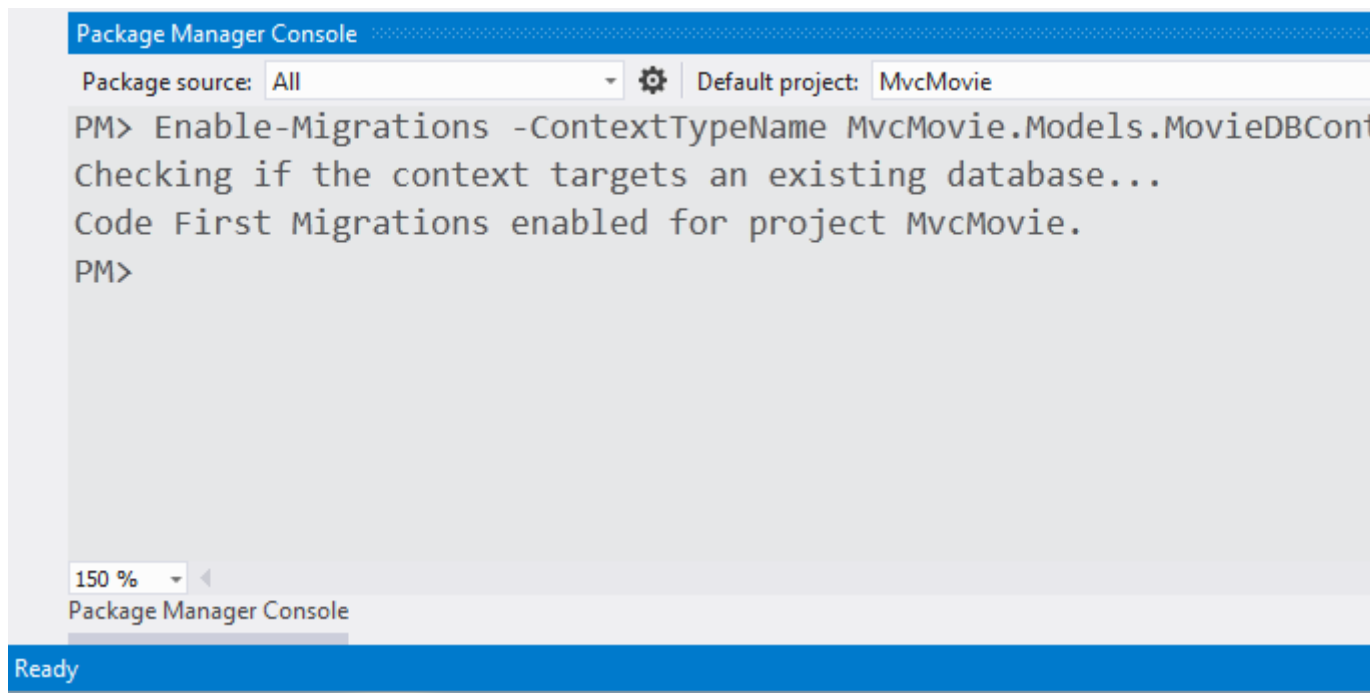


Genere la aplicación para asegurarse de que no hay errores.

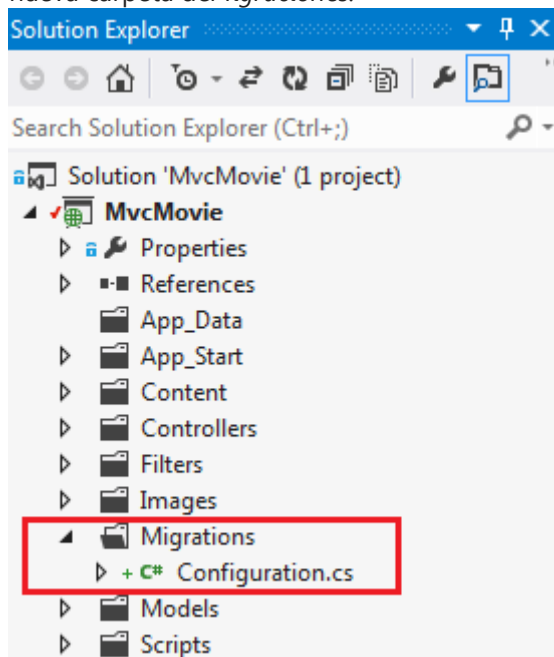
En el menú **Herramientas**, haga clic en **Biblioteca Package Manager** y luego **consola Administrador de paquetes**.



En la ventana **de la consola del Administrador de paquetes** en el **PM>** teclee "Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDbContext".



El comando **Enable-Migrations** (ver imagen superior) crea un archivo *Configuration.cs* en una nueva carpeta de *Migraciones*.



Visual Studio abre el archivo *Configuration.cs*. Vuelva a colocar la **Seed** método en el archivo *Configuration.cs* con el siguiente código:

```
protected override void Seed (contexto MvcMovie.Models.MovieDbContext)
{
    context.Movies.AddOrUpdate (i => I.Título,
        nueva película
        {
            Title = "Cuando Harry encontró a Sally",
            ReleaseDate = DateTime.Parse ("11/01/1989"),
            Género = "comedia romántica",
        })
}
```

```

        Precio = 7.99m
    },

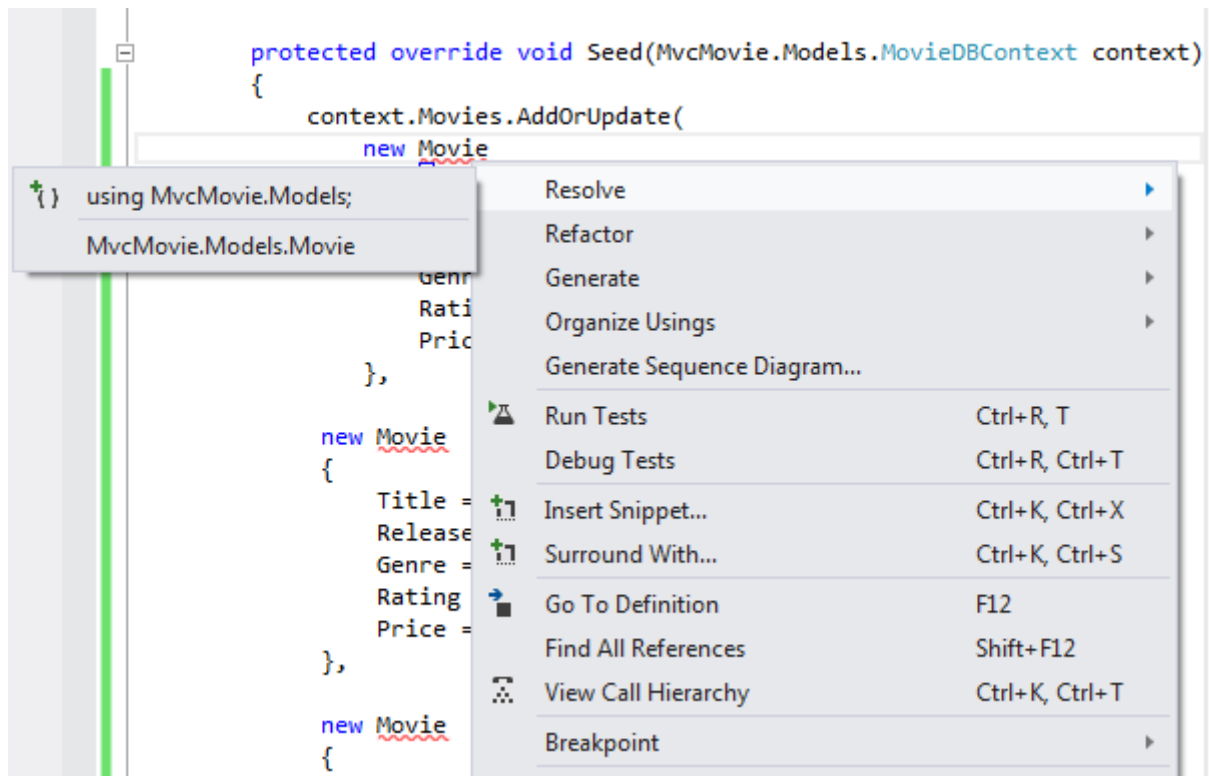
    nueva película
    {
        Title = "Ghostbusters",
        ReleaseDate = DateTime.Parse ("13/03/1984"),
        Género = "Comedia",
        Precio = 8.99M
    },

    nueva película
    {
        Title = "Ghostbusters 2",
        ReleaseDate = DateTime.Parse ("02/23/1986"),
        Género = "Comedia",
        Precio = 9.99m
    },

    nueva película
    {
        Title = "Rio Bravo",
        ReleaseDate = DateTime.Parse ("15/04/1959"),
        Género = "occidental",
        Precio = 3.99m
    }
);
}

```

Haga clic derecho sobre la línea roja ondulada bajo **Movie** y seleccione **Resolver** luego usando **MvcMovie.Models**;



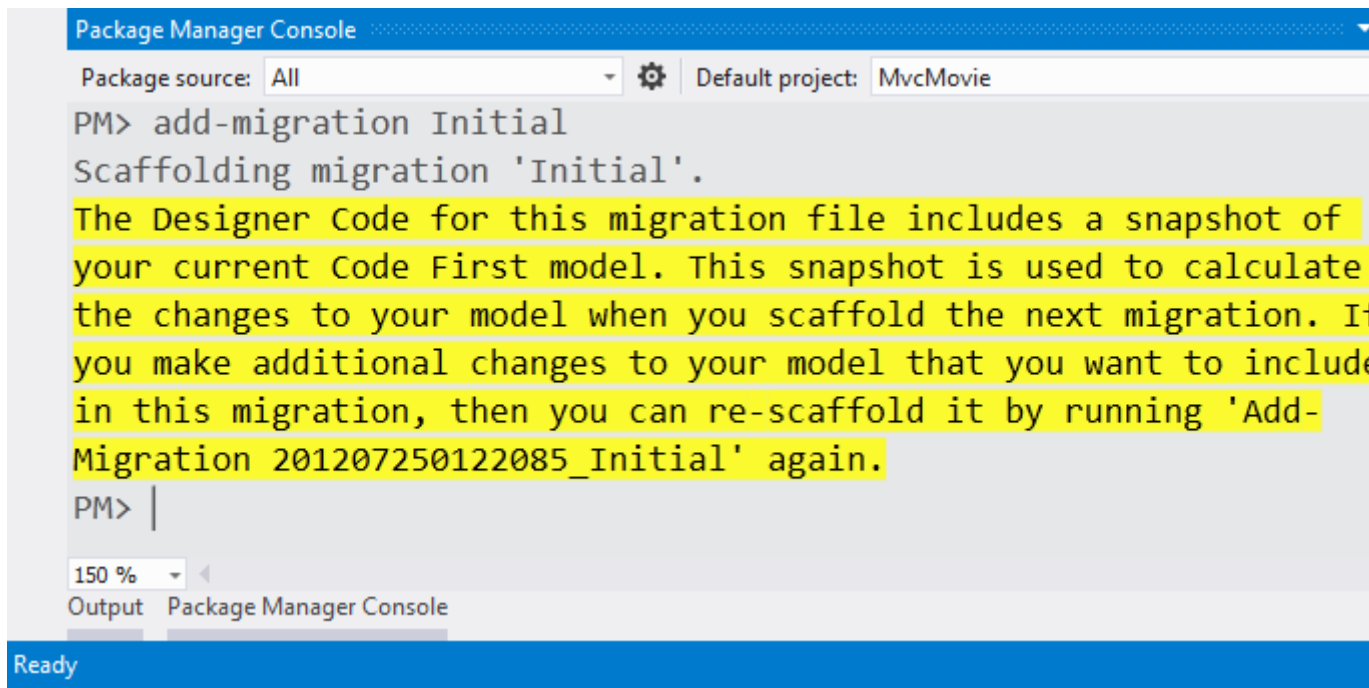
Si lo hace, añada la siguiente instrucción using:

utilizando `MvcMovie.Models;`

Código primeras migraciones llama a la **Seed** método después de cada migración (es decir, llamando a **update-base de datos** en la consola de Administrador de paquetes), y este método actualiza filas que ya se han insertado, o los inserta si no existen todavía.

Presione Ctrl-Shift-B para construir el proyecto. (Los siguientes pasos se producirá un error si su no construyen en este punto.)

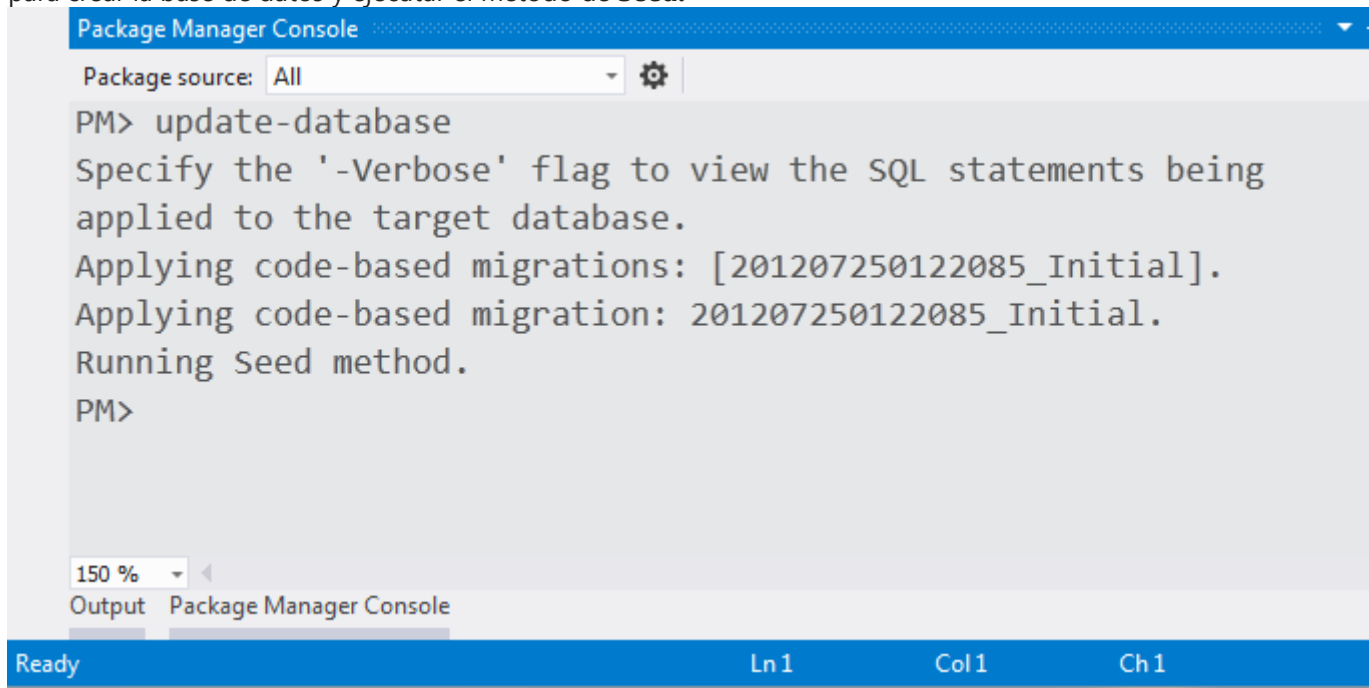
El siguiente paso es crear un **DbMigration** clase para la migración inicial. Esta migración a crea una nueva base de datos, es por eso que ha borrado el archivo *movie.mdf* en un paso anterior. En la ventana **de la consola del Administrador de paquetes**, escriba el comando "add-migración inicial" para crear la migración inicial. El nombre de "inicial" es arbitrario y se utiliza para nombrar el archivo de migración creado.



```
Package Manager Console
Package source: All [v] [g] Default project: MvcMovie
PM> add-migration Initial
Scaffolding migration 'Initial'.
The Designer Code for this migration file includes a snapshot of
your current Code First model. This snapshot is used to calculate
the changes to your model when you scaffold the next migration. If
you make additional changes to your model that you want to include
in this migration, then you can re-scaffold it by running 'Add-
Migration 201207250122085_Initial' again.
PM> |
150 % [v] [g]
Output Package Manager Console
Ready
```

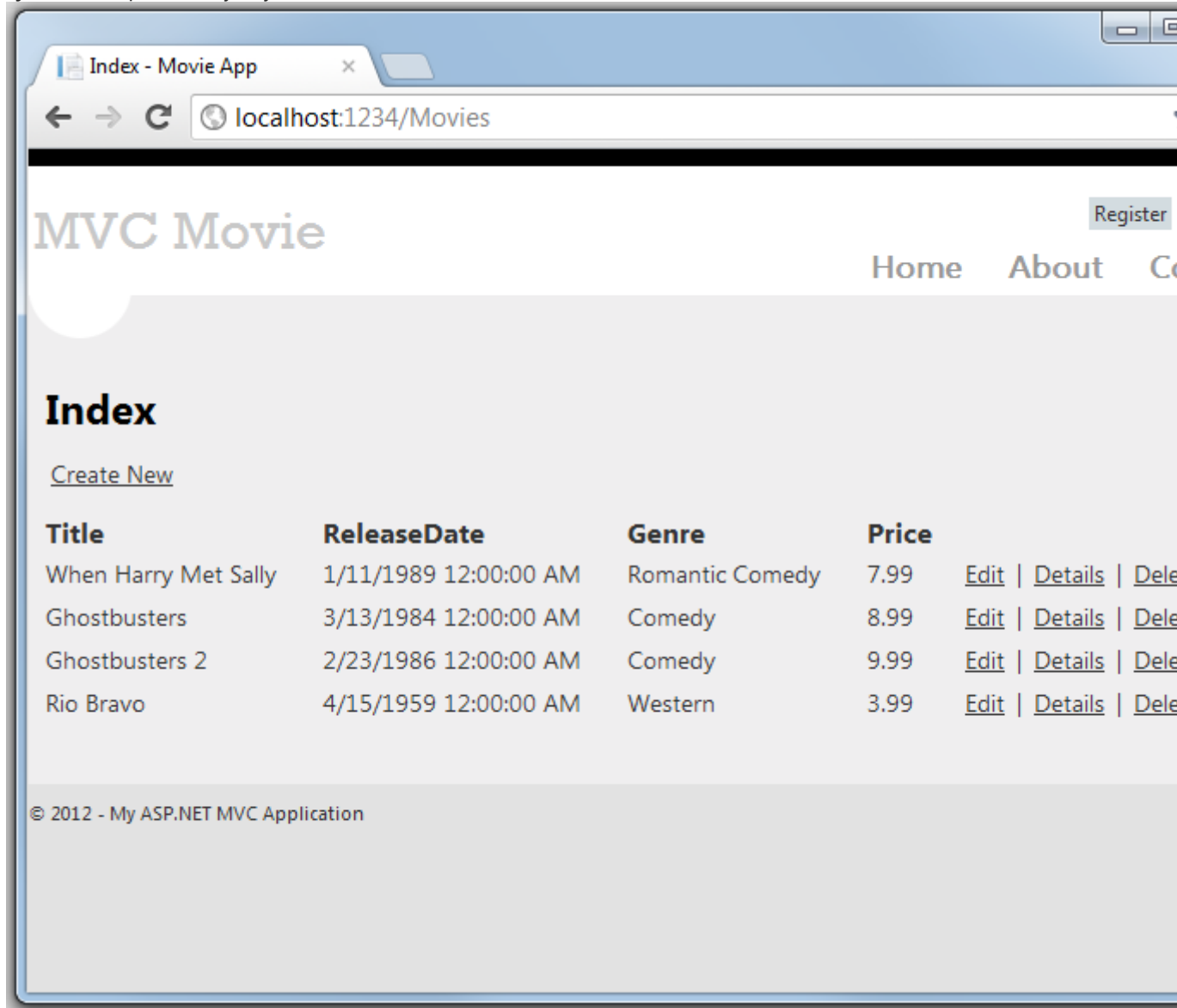
Código primeras migraciones crea otro archivo de clase en la carpeta de *Migraciones* (con el nombre `{ DateStamp _Initial.cs}`), y esta clase contiene código que crea el esquema de base de datos. El nombre del archivo de migración está pre-fijado con una marca de tiempo para ayudar con el pedido. Examine el `{ DateStamp _Initial.cs}` archivo, que contiene las instrucciones para crear la tabla Películas para la película DB. Cuando se actualiza la base de datos en las instrucciones siguientes, este `{ DateStamp _Initial.cs}` archivo se ejecutará y crear el esquema de base de datos. A continuación, el método de **Seed** se ejecutará para poblar la base de datos con datos de prueba.

En la **consola de Administrador de paquetes**, escriba el comando "update-base de datos" para crear la base de datos y ejecutar el método de **Seed**.



```
Package Manager Console
Package source: All [v] [g]
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being
applied to the target database.
Applying code-based migrations: [201207250122085_Initial].
Applying code-based migration: 201207250122085_Initial.
Running Seed method.
PM>
150 % [v] [g]
Output Package Manager Console
Ready Ln1 Col1 Ch1
```

Si obtiene un error que indica una tabla ya existe y no se puede crear, es probablemente porque se le terminó la aplicación después de que ha borrado la base de datos y antes de ejecutar `update-database`. En ese caso, elimine el archivo `Movies.mdf` de nuevo y vuelva a intentar la `update-database` del sistema. Si sigue apareciendo un error, elimine la carpeta migraciones y contenido a continuación, comenzar con las instrucciones en la parte superior de esta página (es decir borrar el archivo `Movies.mdf` luego proceder a Enable-Migraciones). Ejecutar la aplicación y vaya a la `/ Películas` URL. Se muestran los datos de la semilla.



Adición de una Clasificación de propiedades para el Modelo Movie

Comience agregando una nueva `Rating` alojamiento hasta el existente `Movie` clase. Abra los `Modelos \ archivoMovie.cs` y añadir la `Rating` propiedad como esta:

```
pública string Puntuación { conseguir; establecido; }
```

El total `Movie` clase ahora se ve como el siguiente código:

```
public class Movie  
{
```



```

    int ID público {get; establecido; }
    string Título pública {get; establecido; }
    pública DateTime ReleaseDate {get; establecido; }
    public string Género {get; establecido; }
    Precio público decimal {get; establecido; }
    cadena Clasificación pública {get; establecido; }
}

```

Generar la aplicación usando el **Build**> comando de menú **Movie Construir** o pulsando CTRL-SHIFT-B.

Ahora que usted ha actualizado el **Model** de clase, también es necesario actualizar los \ *Vistas* \ *Movies* \ \ *Index.cshtml* y *Vistas* \ *Movies* \ *view Create.cshtml* plantillas con el fin de mostrar la nueva **Rating** la propiedad en la vista del navegador.

Abra la carpeta \ *Vistas* \ *Movies* \ archivo *Index.cshtml* y añadir

un **<th>Rating</th>** encabezado de columna justo después de la columna **Precio**. A continuación, agregue un **<td>** columna cerca de la final de la plantilla para hacer la **@item.Rating** valor. A continuación se muestra lo que la plantilla de vista *Index.cshtml* actualizado será similar a:

```

Model IEnumerable <MvcMovie.Models.Movie>

@ {
    ViewBag.Title = "Índice";
}

<H2> Índice </ h2>

<P>
    @ Html.ActionLink ("Crear nuevo", "Crear")
</ P>
<Table>
    <Tr>
        <Th>
            @ Html.DisplayNameFor (model => Model.Title)
        </ Th>
        <Th>
            @ Html.DisplayNameFor (model => model.ReleaseDate)
        </ Th>
        <Th>
            @ Html.DisplayNameFor (model => model.Genre)
        </ Th>
        <Th>
            @ Html.DisplayNameFor (model => model.Price)
        </ Th>
        <Th>
            @ Html.DisplayNameFor (model => model.Rating)
        </ Th>
    <Th> </ th>
    </ Tr>

    foreach (elemento var en el Modelo) {
        <Tr>

```

```

        <Td>
            @ Html.DisplayFor (ModelItem => item.Title)
        </ Td>
        <Td>
            @ Html.DisplayFor (ModelItem => item.ReleaseDate)
        </ Td>
        <Td>
            @ Html.DisplayFor (ModelItem => item.Genre)
        </ Td>
        <Td>
            @ Html.DisplayFor (ModelItem => item.Price)
        </ Td>
        <Td>
            @ Html.DisplayFor (ModelItem => item.Rating)
        </ Td>
        <Td>
            @ Html.ActionLink ("Edit", "Editar", nueva {id = item.ID}) |
            @ Html.ActionLink ("Detalles", "Detalles", nueva {id = item.ID})
        |
            @ Html.ActionLink ("Borrar", "Borrar", nueva {id = item.ID})
        </ Td>
    </ Tr>
}
</ Table>

```

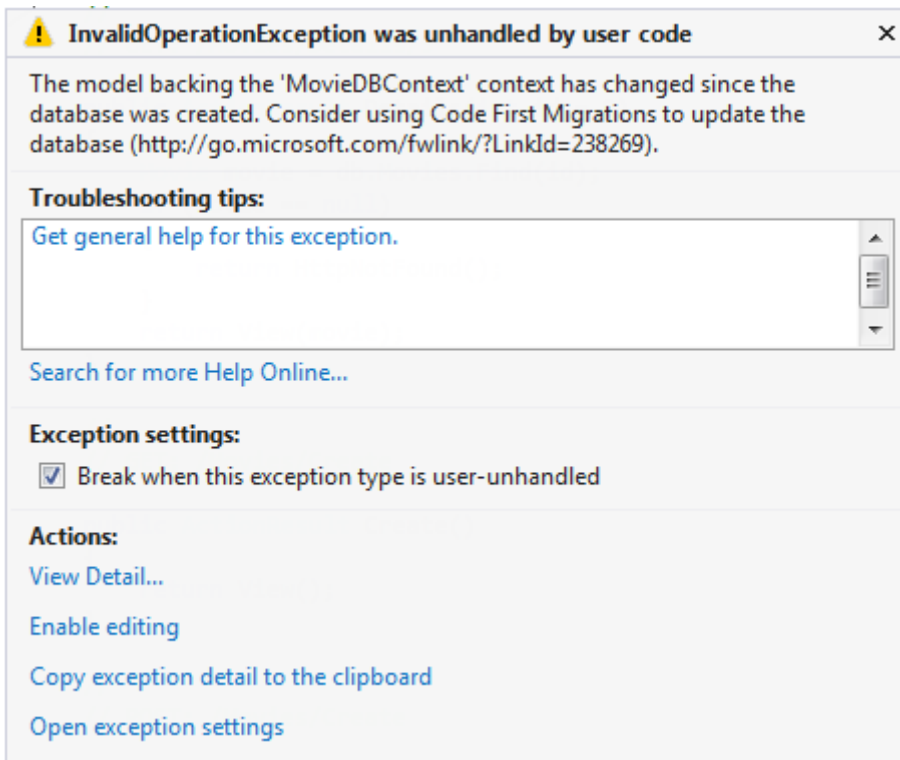
A continuación, abra la carpeta \ *Vistas* \ *Movies* \ archivo *Create.cshtml* y agregue el siguiente marcado cerca del final de la forma. Esto hace que un cuadro de texto para que pueda especificar una clasificación cuando se crea una nueva película.

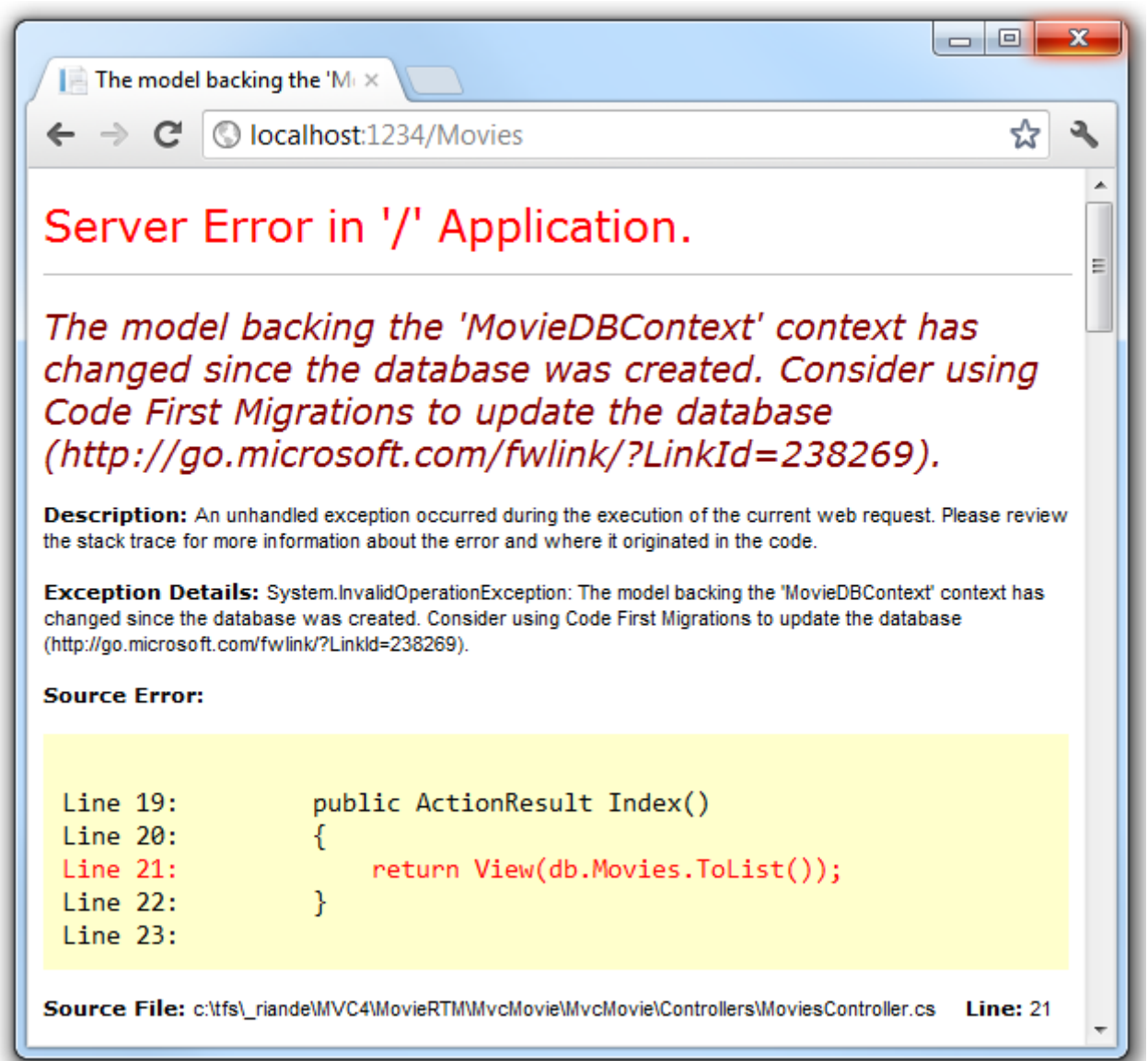
```

<Div class = "editor de etiqueta">
    @ Html.LabelFor (model => model.Rating)
</ Div>
<Div class = "editor-campo">
    @ Html.EditorFor (model => model.Rating)
    @ Html.ValidationMessageFor (model => model.Rating)
</ Div>

```

Ahora usted ha informado el código de la aplicación para apoyar la nueva **Rating** propiedad. Ahora ejecutar la aplicación y vaya a la / *Películas* URL. Al hacer esto, sin embargo, verá uno de los siguientes errores:





Usted está viendo este error porque la actualización **Movie** clase del modelo en la aplicación ahora es diferente que el esquema de la **Movie** de tabla de la base de datos existente. (No hay **Rating** la columna en la tabla de base de datos.)

Hay unos pocos enfoques para resolver el error:

1. Haga que el Entity Framework caer automáticamente y volver a crear la base de datos basado en el nuevo esquema de clase del modelo. Este enfoque es muy conveniente cuando se hace desarrollo activo en una base de datos de prueba; que les permite avanzar rápidamente en el esquema del modelo y base de datos en conjunto. La desventaja, sin embargo, es que se pierde los datos existentes en la base de datos - por lo que *no* quiero utilizar este enfoque en una base de datos de producción! Usando un inicializador para sembrar automáticamente una base de datos con datos de prueba es a menudo una forma productiva para desarrollar una aplicación. Para obtener más información sobre los inicializadores de base de datos de Entity Framework, vea de Tom Dykstra [MVC / Entity Framework ASP.NET tutorial](#) .
2. Explícitamente modificar el esquema de la base de datos existente para que coincida con el modelo de clases. La ventaja de este enfoque es que usted mantenga sus datos. Usted puede hacer este cambio de forma manual o mediante la creación de un script de cambio de base de datos.

3. Usa Código primeras migraciones para actualizar el esquema de base de datos. Para este tutorial, usaremos Código primeras migraciones. Actualizar el método de Semillas de modo que proporciona un valor para la nueva columna. Abrir archivo Migraciones \ Configuration.cs y agregar un campo de Calificación a cada objeto de película.

```
nueva película
{
    Title = "Cuando Harry encontró a Sally",
    ReleaseDate = DateTime.Parse ("11/01/1989"),
    Género = "comedia romántica",
    Rating = "G",
    Precio = 7.99m
},
```

Genere la solución, a continuación, abra la ventana **de la consola del Administrador de paquetes** e introduzca el siguiente comando:

add-migration AddRatingMig

El **add-migration** comando le dice al marco migratorio para examinar el modelo actual de la película con el esquema película DB actual y crear el código necesario para migrar la base de datos para el nuevo modelo. El AddRatingMig es arbitrario y se utiliza para nombrar el archivo de migración. Es útil usar un nombre significativo para el paso de migración.

Cuando este comando acabados, Visual Studio abre el archivo de clase que define la nueva **DbMigration** clase derivada, y en el **Up** método se puede ver el código que crea la nueva columna.

```
pública AddRatingMig clase parcial: DbMigration
{
    public override void Arriba ()
    {
        AddColumn ("dbo.Movies", "Clasificación", c => c.String ());
    }

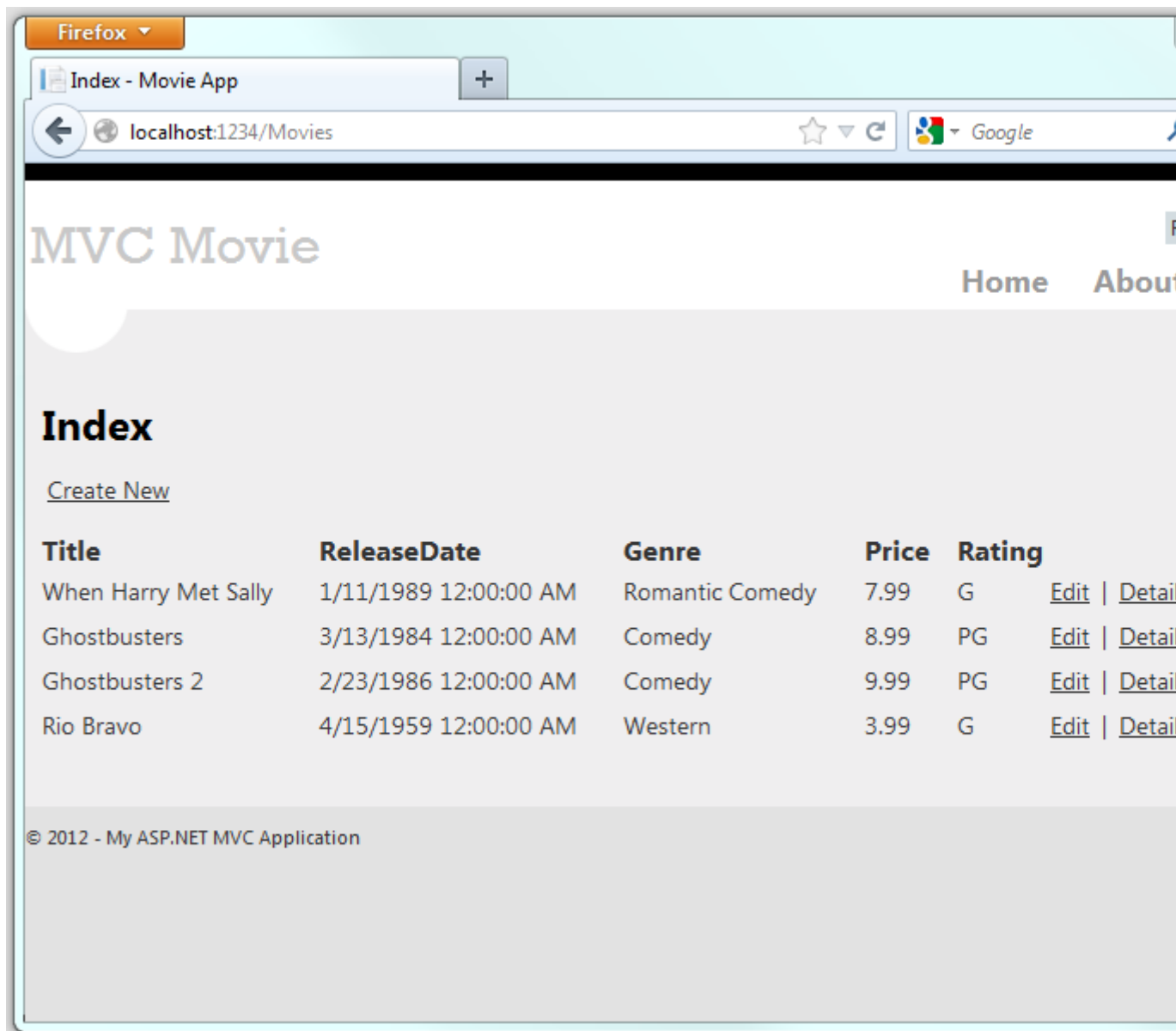
    public override void Abajo ()
    {
        DropColumn ("dbo.Movies", "Clasificación");
    }
}
```

Genere la solución, a continuación, introduzca el comando "update-base de datos" en la ventana **de la consola Administrador de paquetes**.

La imagen siguiente muestra la salida en la ventana **de la consola del Administrador de paquetes** (La fecha del sello preprendiendo AddRatingMig será diferente.)

```
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying code-based migrations: [201207260218501_AddRatingMig].
Applying code-based migration: 201207260218501_AddRatingMig.
Running Seed method.
PM>
```

Vuelva a ejecutar la aplicación y vaya a la / Películas URL. Usted puede ver el nuevo campo Clasificación.



Haga clic en el enlace **crear nuevo** para agregar una nueva película. Tenga en cuenta que usted puede añadir una calificación.

http://localhost: Create - Movie App

MVC Movie

Register Log in

Home About Contact

Create

Title
Rio Bravo II

ReleaseDate
3/15/2012

Genre
Western

Price
9.99

Rating
G

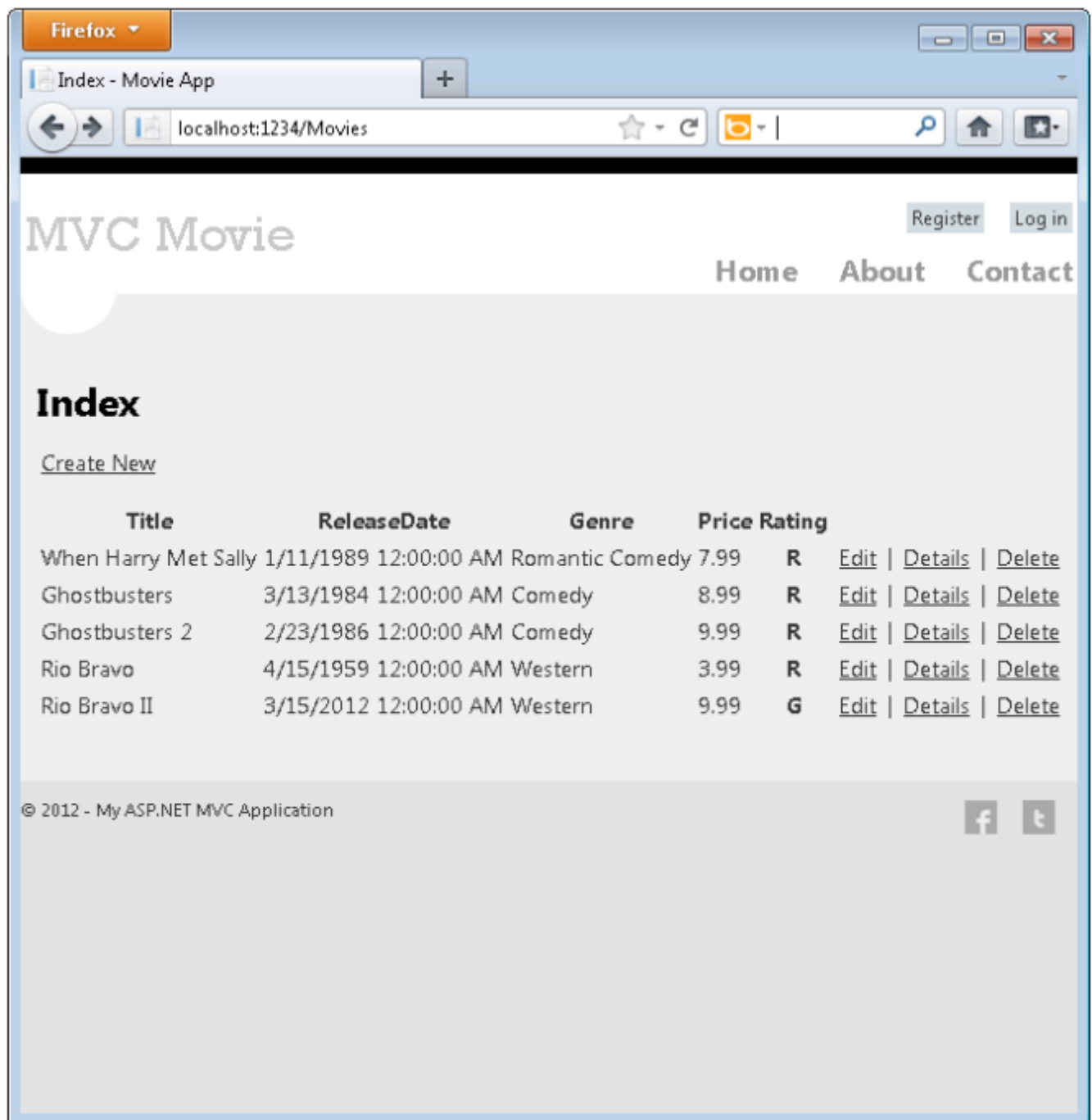
Create

[Back to List](#)

© 2012 - My ASP.NET MVC Application

f t

Haga clic en **Crear**. La nueva película, incluyendo la calificación, ahora aparece en las películas del anuncio:



También debe agregar la **Rating** campo de la edición, Detalles y SearchIndex plantillas de vista. Puede introducir el comando "update-base de datos" en la ventana **de la consola del Administrador de paquetes** de nuevo y no se harían cambios, porque el esquema coincide con el modelo.

En esta sección usted vio cómo se pueden modificar los objetos del modelo y mantener la base de datos en sincronía con los cambios. También ha aprendido una manera de rellenar una base de datos recién creada con datos de muestra para que pueda probar escenarios. A continuación, vamos a ver cómo se pueden agregar más rica lógica de validación a las clases del modelo y habilitar algunas reglas de negocio para ser forzadas.

Adición de validación para el modelo

En esta sección vamos a añadir la lógica de validación para la **Movie** modelo, y se asegurará de que las reglas de validación se aplican cada vez que un usuario intenta crear o editar una película utilizando la aplicación.

Mantener las cosas SECO

Uno de los principios de diseño fundamentales de ASP.NET MVC es DRY ("No te repitas"). ASP.NET MVC le estimula para especificar la funcionalidad o comportamiento sólo una vez, y luego tienen que reflejarse en todas partes en una aplicación. Esto reduce la cantidad de código que necesita para escribir y hace que el código que escribes menos propenso a errores y más fácil de mantener.

El apoyo proporcionado por la validación ASP.NET MVC y Entity Framework Code First es un gran ejemplo del principio DRY en acción. Puede especificar de forma declarativa las reglas de validación en un solo lugar (en la clase del modelo) y las reglas se hacen cumplir por todas partes en la aplicación.

Echemos un vistazo a cómo usted puede tomar ventaja de este soporte de validación en la aplicación de la película.

Adición de reglas de validación para el modelo de la película

Vamos a empezar añadiendo un poco de lógica de validación a la **Movie** de clase.

Abra el archivo *Movie.cs*. Añadir un **using** declaración en la parte superior del archivo que hace referencia a la **System.ComponentModel.DataAnnotations** espacio de nombres:

```
utilizando System.ComponentModel.DataAnnotations.;
```

Observe el espacio de nombres no contiene **System.Web**. **DataAnnotations** proporciona un conjunto integrado de validación de atributos que se pueden aplicar de forma declarativa a cualquier clase o propiedad.

Ahora actualizar el **Movie** clase para tomar ventaja de la incorporada en **Required**, **StringLength** y **Range** atributos de validación. Utilice el código siguiente como un ejemplo de que para aplicar los atributos.

```
public class Movie {
    int ID público {get; establecido; }

    [Obligatorio]
    string Título pública {get; establecido; }

    [Tipo de datos (DataType.Date)]
    pública DateTime ReleaseDate {get; establecido; }

    [Obligatorio]
    public string Género {get; establecido; }

    [Range (1, 100)]
    [Tipo de datos (DataType.Currency)]
    Precio público decimal {get; establecido; }

    [StringLength (5)]
    cadena Clasificación pública {get; establecido; }
```

```
}
```

Ejecutar la aplicación y se le volverá a recibir el siguiente error en tiempo de ejecución:

El modelo de la copia del contexto 'MovieDbContext' ha cambiado desde que se creó la base de datos. Considere el uso de Código primeras migraciones de actualizar la base de datos (<http://go.microsoft.com/fwlink/?LinkId=238269>).

Nosotros nos migraciones para actualizar el schema. Genere la solución, a continuación, abra la ventana **de la consola del Administrador de paquetes** e introduzca los siguientes comandos:

```
add-migration AddDataAnnotationsMig  
update-database
```

Cuando este comando acabados, Visual Studio abre el archivo de clase que define la nueva `DbMigration` clase derivada con el nombre especificado (`AddDataAnnotationsMig`), y en el `Up` método que se puede ver el código que actualiza las restricciones de esquema. Los `Title` y `Genre` campos ya no son anulable (es decir, debe introducir un valor) y la `Rating` campo tiene una longitud máxima de 5.

Los atributos de validación especifican el comportamiento que desea aplicar en las propiedades del modelo que se aplican a. La `Required` atributo indica que una propiedad debe tener un valor; en esta muestra, una película tiene que tener valores para el `Title`, `ReleaseDate`, `Genre` y `Price` propiedades con el fin de ser válida. El `Range` atributo limita a un valor dentro de un rango especificado. El `StringLength` atributo le permite establecer la longitud máxima de una propiedad de cadena y, opcionalmente, su longitud mínima. Tipos intrínsecos (`decimal`, `int`, `float`, `DateTime`) son requeridos por defecto y no es necesario la `Required` atributo.

Primer Código garantiza que las reglas de validación que especifique en una clase del modelo se aplican antes de la aplicación guarda los cambios en la base de datos. Por ejemplo, el código de abajo será una excepción cuando el `SaveChanges` se llama al método, debido a que varios requerida `Movie` valores de las propiedades se han perdido y el precio es cero (que está fuera del rango válido).

```
MovieDbContext db = nuevo MovieDbContext ();  
  
Movie movie = nuevo Película ();  
película. Título = "Lo que el viento se llevó";  
película. Precio = 0.0M;  
  
.. db Películas Añadir (película);  
db . SaveChanges (); // <= Lanzará servidor excepción  
validación del lado
```

Tener reglas de validación forzadas automáticamente por NET Framework ayuda a hacer su aplicación más robusta. También asegura que no se puede olvidar a validar algo y sin querer dejar que los malos datos en la base de datos.

Aquí está una lista completa del código para el archivo `Movie.cs` actualización:

```
using System;  
utilizando System.Data.Entity;  
utilizando System.ComponentModel.DataAnnotations;  
  
MvcMovie.Models de espacio de nombres {  
    public class Movie {  
        int ID público {get; establecido; }  
    }
```

```

[Obligatorio]
string Título pública {get; establecido; }

[Tipo de datos (DataType.Date)]
pública DateTime ReleaseDate {get; establecido; }

[Obligatorio]
public string Género {get; establecido; }

[Range (1, 100)]
[Tipo de datos (DataType.Currency)]
Precio público decimal {get; establecido; }

[StringLength (5)]
cadena Clasificación pública {get; establecido; }
}

MovieDbContext clase pública: DbContext {
    públicas DbSet <Película> Películas {get; establecido; }
}
}

```

Error de validación de la interfaz de usuario en ASP.NET MVC

Vuelva a ejecutar la aplicación y vaya a la / *Películas* URL.

Haga clic en el enlace **crear nuevo** para agregar una nueva película. Rellene el formulario con algunos valores no válidos y, a continuación, haga clic en el botón **Crear**.

Firefox

Create - Movie App

localhost:1234/Movies/Create

Bing

MVC Movie

Register Log in

Home About Contact

Create

Title

ReleaseDate

xyz

The field ReleaseDate must be a date.

Genre

Price

129

Price must be between \$1 and \$100

Rating

Probably G

The field Rating must be a string with a maximum length of 5.

Create

[Back to List](#)

Nota para apoyar la validación jQuery para las configuraciones regionales distintos del inglés que utilizan una coma (",") para el punto decimal, debe incluir *globalize.js* y sus *culturas* específicas / archivo *globalize.cultures.js* (de <https://github.com/jquery/globalize>) y Javascript para usar `Globalize.parseFloat`. El siguiente código muestra las modificaciones de las Vistas \ Movies \ Edit.cshtml archivo para trabajar con la cultura "fr-FR":

```
Scriptssection {
    @ Scripts.Render ("~/ paquetes / jqueryval")
    <Script src = "~/ scripts / globalize.js"> </ script>
```

```

<Script src = "~ / Scripts / globalize.culture.fr-FR.js"> </ script>
<Script>
    $.validator.methods.number = Function (valor, elemento) {
        volver this.optional (elemento) ||
            ! isNaN (Globalize.parseFloat (valor));
    }
    $ (Document) .ready (function () {
        Globalize.culture ("fr-FR");
    });
</ Script>
<Script>
    jQuery.extend (jQuery.validator.methods, {
        rango: function (valor, elemento, param) {
            // Utilice el plugin Globalización para analizar el valor
            var val = $ .global.parseFloat (valor);
            volver this.optional (elemento) || (
                val > = parámetro [0] && val <= parámetro [1]);
        }
    });
</ Script>
}

```

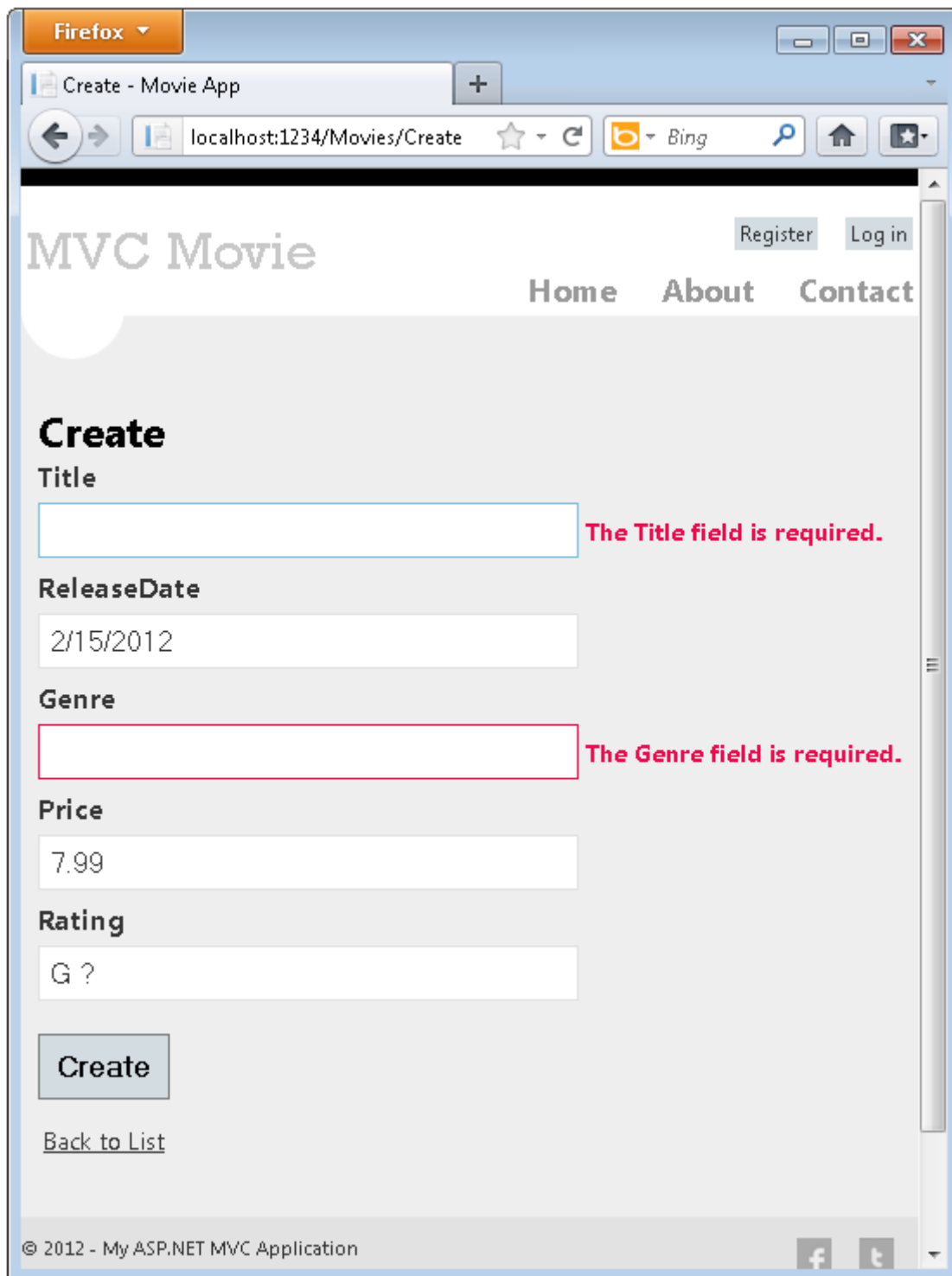
Observe cómo la forma se ha utilizado de forma automática un color de borde rojo para destacar los cuadros de texto que contienen datos no válidos y se ha emitido un mensaje de error de validación apropiado al lado de cada uno. Los errores se aplican tanto del lado del cliente (usando JavaScript y jQuery) y del lado del servidor (en caso de que un usuario tiene JavaScript desactivado).

Un beneficio real es que usted no tiene que cambiar una sola línea de código en el **MoviesController** clase o en la vista *Create.cshtml* a fin de que esta interfaz de usuario de validación. El controlador y las vistas que creó anteriormente en este tutorial recogidos automáticamente las reglas de validación que ha especificado mediante el uso de la validación de atributos en las propiedades de la **Movie** clase del modelo.

Usted puede haber notado las propiedades **Title** y **Genre** , el atributo requerido no se aplica hasta que presente el formulario (pulsar en el botón **Crear**), o escribir texto en el campo de entrada y quitado. Para un campo que está inicialmente vacía (como los campos de la vista *Crear*) y que sólo tiene el atributo necesario y atributos ninguna otra validación, usted puede hacer lo siguiente para activar la validación:

1. Tab en el campo.
2. Escriba un texto.
3. Tab cabo.
4. Tab de nuevo en el campo.
5. Quite el texto.
6. Tab cabo.

La secuencia anterior dará lugar a la validación requerida sin golpear el botón enviar. Simplemente pulsando el botón de enviar, sin entrar en cualquiera de los campos activará la validación del lado del cliente. Los datos del formulario no se envía al servidor hasta que no haya errores de validación del lado del cliente. Usted puede probar esto poniendo un punto de quiebre en el método HTTP POST o el uso de la [herramienta Fiddler](#) o las IE 9 [F12 herramientas para desarrolladores](#) .



¿Cómo se produce la validación en el Crear vista y Crear Método Acción

Usted podría preguntarse cómo se generó la interfaz de usuario de la validación sin cambios en el código del controlador o vistas. El siguiente listado muestra lo que el **Create** métodos en el **MovieController** mirada de clase similar. Son sin cambios desde cómo los creó anteriormente en este tutorial.

```
//
```

```

// GET: / Películas / Crear

ActionResult público Create ()
{
    volver Ver ();
}

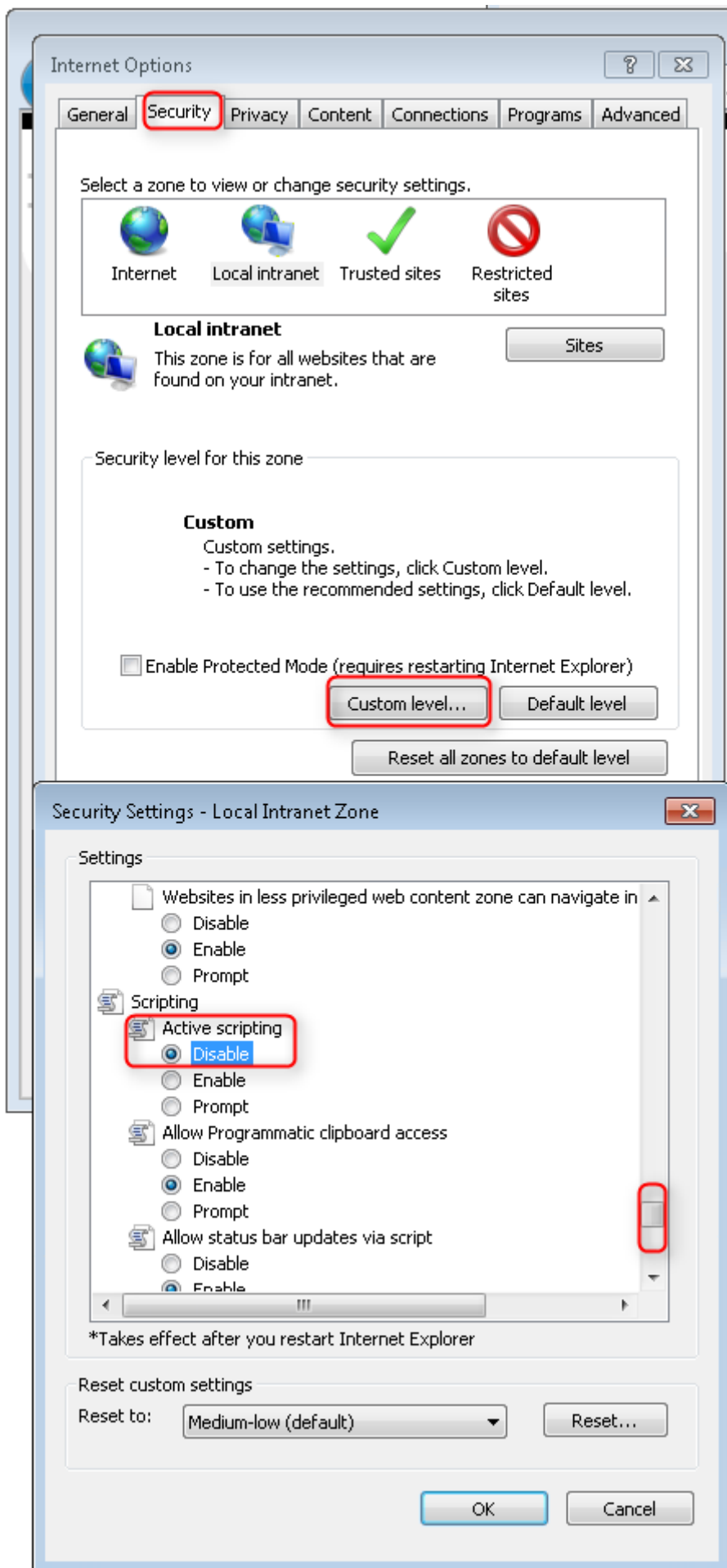
//
// Mensaje: / Películas / Crear

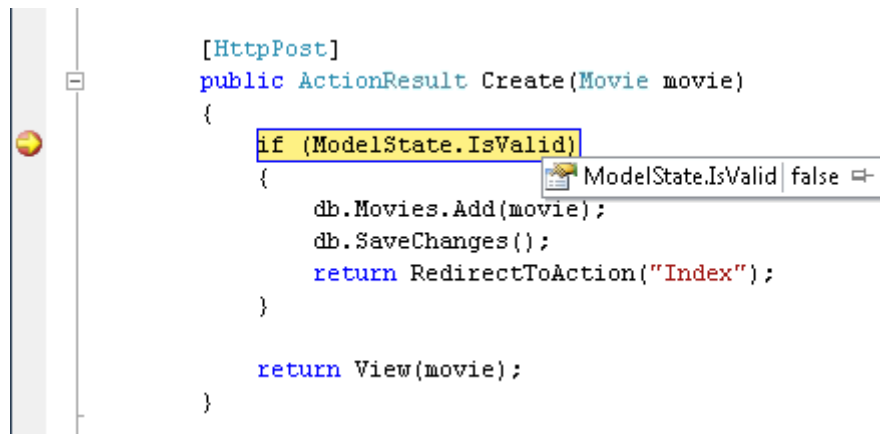
[HttpPost]
ActionResult público Create (película Película)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add (película);
        db.SaveChanges ();
        volver RedirectToAction ("Índice");
    }

    volver Ver (película);
}

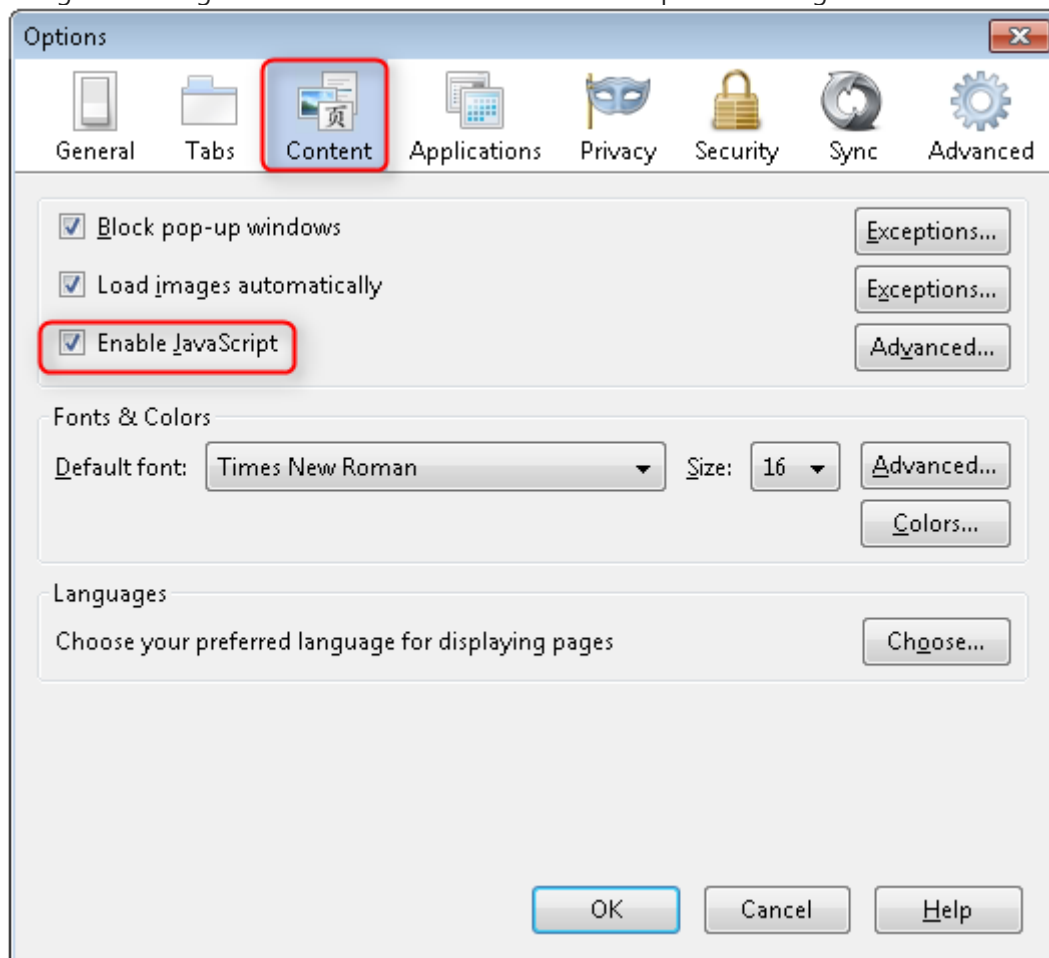
```

El primero (HTTP GET) **Create** método de acción muestra el inicial Crear formulario. La segunda (**[HttpPost]** versión) se encarga de la posterior forma. La segunda **Create** método (El **HttpPost** versión) llama **ModelState.IsValid** para comprobar si la película tiene algún error de validación. Al llamar a este método evalúa los atributos de validación que se han aplicado al objeto. Si el objeto tiene errores de validación, el **Create** método vuelve a mostrar el formulario. Si no hay errores, el método guarda la nueva película en la base de datos. En nuestro ejemplo de la película que estamos utilizando, **el formulario no se ha publicado en el servidor cuando su son errores de validación detectados en el lado del cliente; la segunda Create método nunca es llamado**. Si desactiva JavaScript en su navegador, la validación del cliente está desactivada y el HTTP POST **Create** método llama **ModelState.IsValid** para comprobar si la película tiene algún error de validación. Se puede establecer un punto de quiebre en la **HttpPost Create** método y verificar el método nunca se llama, la validación del lado del cliente no presentará los datos del formulario cuando se detectan errores de validación. Si desactiva JavaScript en su navegador, a continuación, enviar el formulario con errores, el punto de quiebre se verá afectada. Seguirá disfrutando de validación completa sin JavaScript. La siguiente imagen muestra cómo deshabilitar JavaScript en Internet Explorer.

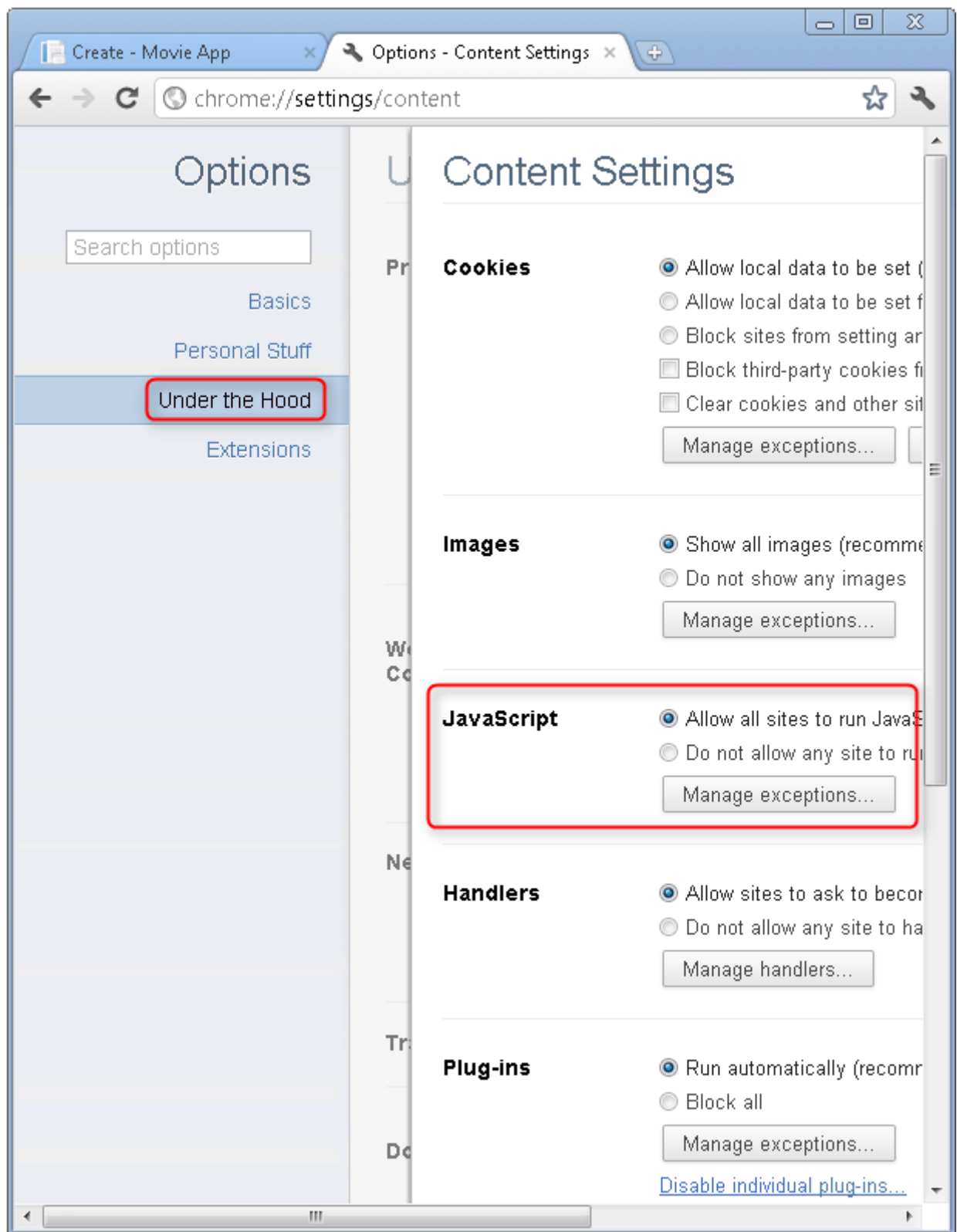




La siguiente imagen muestra cómo deshabilitar JavaScript en el navegador FireFox.



La siguiente imagen muestra cómo deshabilitar JavaScript en el navegador Chrome.



A continuación se muestra la plantilla de vista *Create.cshtml* que scaffolded anteriormente en el tutorial. Es utilizado por los métodos de acción que se muestran arriba tanto para mostrar la forma inicial y para volver a ver en el caso de un error.

```
Model MvcMovie.Models.Movie
```

```
@ {
```

```

    ViewBag.Title = "Crear";
}

<H2> Crear </ h2>

<Script src = "@ Url.Content (" ~ / Scripts / jquery.validate.min.js ")"> </
script>
<Script src = "@ Url.Content (" ~ / Scripts /
jquery.validate.unobtrusive.min.js ")"> </ script>

using (Html.BeginForm ()) {
    @ Html.ValidationSummary (true)

    <Fieldset>
        <Legend> Película </ legend>

        <Div class = "editor de etiqueta">
            @ Html.LabelFor (model => Model.Title)
        </ Div>
        <Div class = "editor-campo">
            @ Html.EditorFor (model => Model.Title)
            @ Html.ValidationMessageFor (model => Model.Title)
        </ Div>

        <Div class = "editor de etiqueta">
            @ Html.LabelFor (model => model.ReleaseDate)
        </ Div>
        <Div class = "editor-campo">
            @ Html.EditorFor (model => model.ReleaseDate)
            @ Html.ValidationMessageFor (model => model.ReleaseDate)
        </ Div>

        <Div class = "editor de etiqueta">
            @ Html.LabelFor (model => model.Genre)
        </ Div>
        <Div class = "editor-campo">
            @ Html.EditorFor (model => model.Genre)
            @ Html.ValidationMessageFor (model => model.Genre)
        </ Div>

        <Div class = "editor de etiqueta">
            @ Html.LabelFor (model => model.Price)
        </ Div>
        <Div class = "editor-campo">
            @ Html.EditorFor (model => model.Price)
            @ Html.ValidationMessageFor (model => model.Price)
        </ Div>
        <Div class = "editor de etiqueta">
            @ Html.LabelFor (model => model.Rating)
        </ Div>
        <Div class = "editor-campo">

```

```

        @ Html.EditorFor (model => model.Rating)
        @ Html.ValidationMessageFor (model => model.Rating)
    </ Div>

    <P>
        <Input type = "submit" value = "Crear" />
    </ P>
</ Fieldset>
}

<Div>
    @ Html.ActionLink ("Volver a la lista", "Índice")
</ Div>

```

Observe cómo el código utiliza un `Html.EditorFor` ayudante para emitir la `<input>` elemento para cada `Movie` propiedad. Junto a esta ayuda es una llamada a la `Html.ValidationMessageFor` método de ayuda. Estos dos métodos auxiliares trabajan con el modelo de objetos que se pasa por el controlador a la vista (en este caso, una `Movie` objeto). Se ven de forma automática para los atributos de validación especificadas en los mensajes de error de modelo y de visualización, según proceda.

Lo que es realmente bueno de este enfoque es que ni el controlador ni la plantilla de vista `Crear` sabe nada acerca de las reglas de validación reales están aplicando o sobre los mensajes de error específicos que se muestran. Las reglas de validación y las cadenas de error se especifican sólo en la `Movie` de clase. Estas mismas reglas de validación se aplican automáticamente a la vista de edición y cualquier otro visitas plantillas que se van a crear que edite su modelo. Si desea cambiar la lógica de validación posterior, puede hacerlo en exactamente un lugar mediante la adición de validación atributos al modelo (en este ejemplo, la `movie` clase). Usted no tendrá que preocuparse acerca de las diferentes partes de la aplicación es incompatible con el modo en que se hacen cumplir las reglas - toda la lógica de validación se define en un solo lugar y se utiliza en todas partes. Esto mantiene el código muy limpio, y hace que sea fácil de mantener y evolucionar. Y esto significa que que podrás honrar plenamente el principio DRY.

Agregar formato a la Modelo Movie

Abra el archivo `Movie.cs` y examinar

la `Movie` clase. El `System.ComponentModel.DataAnnotations` espacio de nombres proporciona atributos de formato, además de la incorporada en el conjunto de atributos de validación. Ya hemos aplicado un `DataType` valor de enumeración a la fecha de lanzamiento y de los campos de precio. El código siguiente muestra los `ReleaseDate` y `Price` propiedades con el apropiado `DisplayFormat` atributo.

```

[Tipo de datos (tipo de datos. Fecha)]
pública    DateTime    ReleaseDate    {    conseguir;
establecido;    }

[Tipo de datos (tipo de datos. Moneda)]
pública    decimal    Precio    {    conseguir;    establecido;
}

```

Los `DataType` atributos no son atributos de validación, se utilizan para indicar al motor de vista de cómo hacer que el código HTML. En el ejemplo anterior, el `DataType.Date` atributo muestra las fechas de películas como únicas fechas, sin tiempo. Por ejemplo, las siguientes `DataType` atributos no validan el formato de los datos:

```

[Tipo de datos (DataType.EmailAddress)]
[Tipo de datos (DataType.PhoneNumber)]

```

[Tipo de datos (DataType.Url)]

Los atributos enumerados anteriormente sólo proporcionan consejos para el motor de vista para dar formato a los datos (y los atributos de suministro tales como <a> de URL y para el correo electrónico. Puede utilizar el [RegularExpression](#) atributo para validar el formato de los datos.

Un enfoque alternativo para el uso de los **DataType** atributos, usted podría establecer explícitamente un **DataFormatString** valor. El código siguiente muestra la propiedad fecha de lanzamiento con una cadena de formato de fecha (es decir, "d"). Usted tendría que utilizar esta opción para especificar que no desea en cuando como parte de la fecha de lanzamiento.

```
[FormatSalida (DataFormatString = "{0: d}")]
pública DateTime ReleaseDate { conseguir;
establecido; }
```

El siguiente código da formato al **Price** de propiedad como de divisas.

```
[FormatSalida (DataFormatString = "{0: c}")]
pública decimal Precio { conseguir; establecido; }
```

El total **Movie** de clase se muestra a continuación.

```
public class Movie {
    int ID público {get; establecido; }

    [Obligatorio]
    string Título pública {get; establecido; }

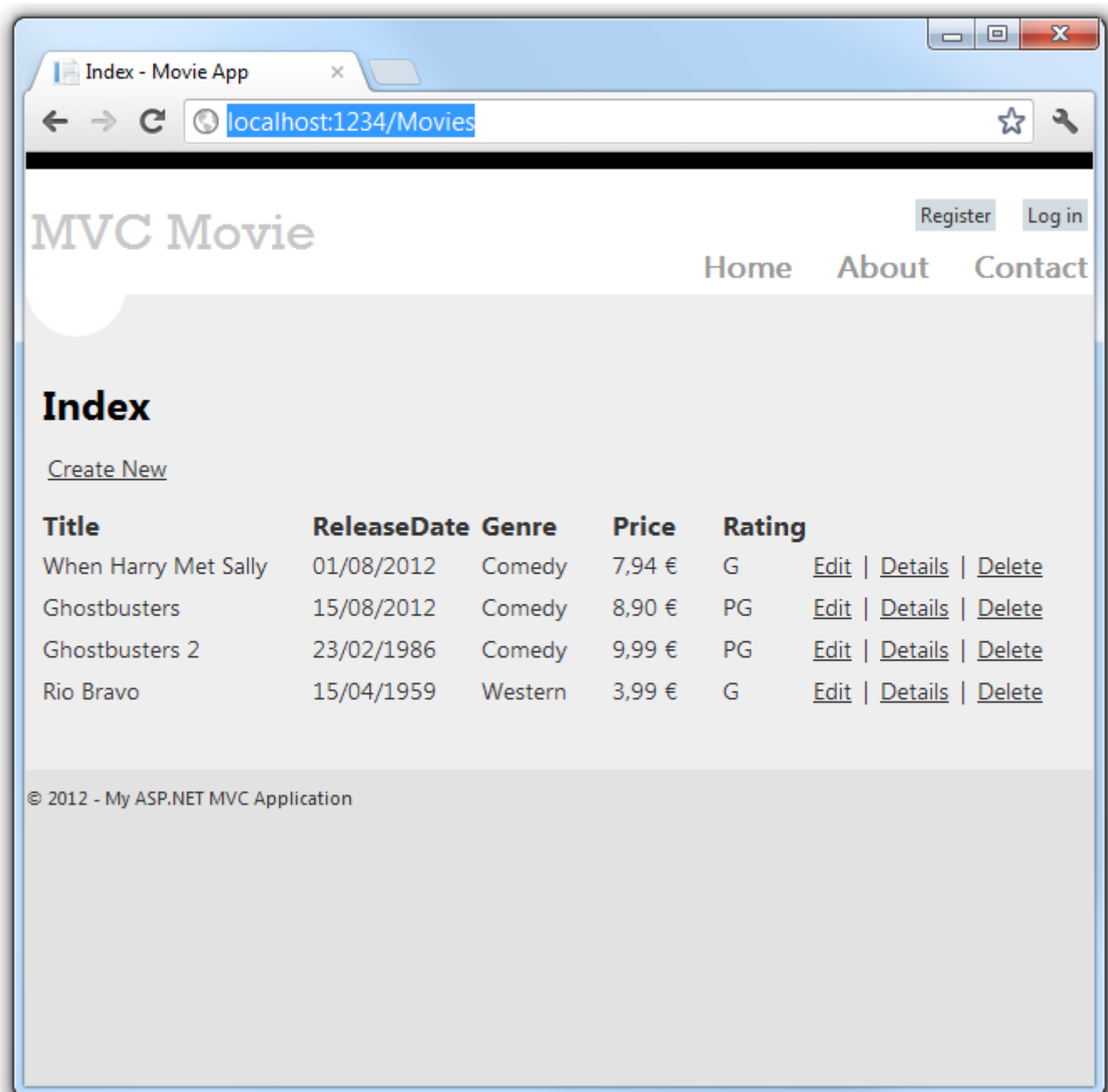
    [Tipo de datos (DataType.Date)]
    pública DateTime ReleaseDate {get; establecido; }

    [Obligatorio]
    public string Género {get; establecido; }

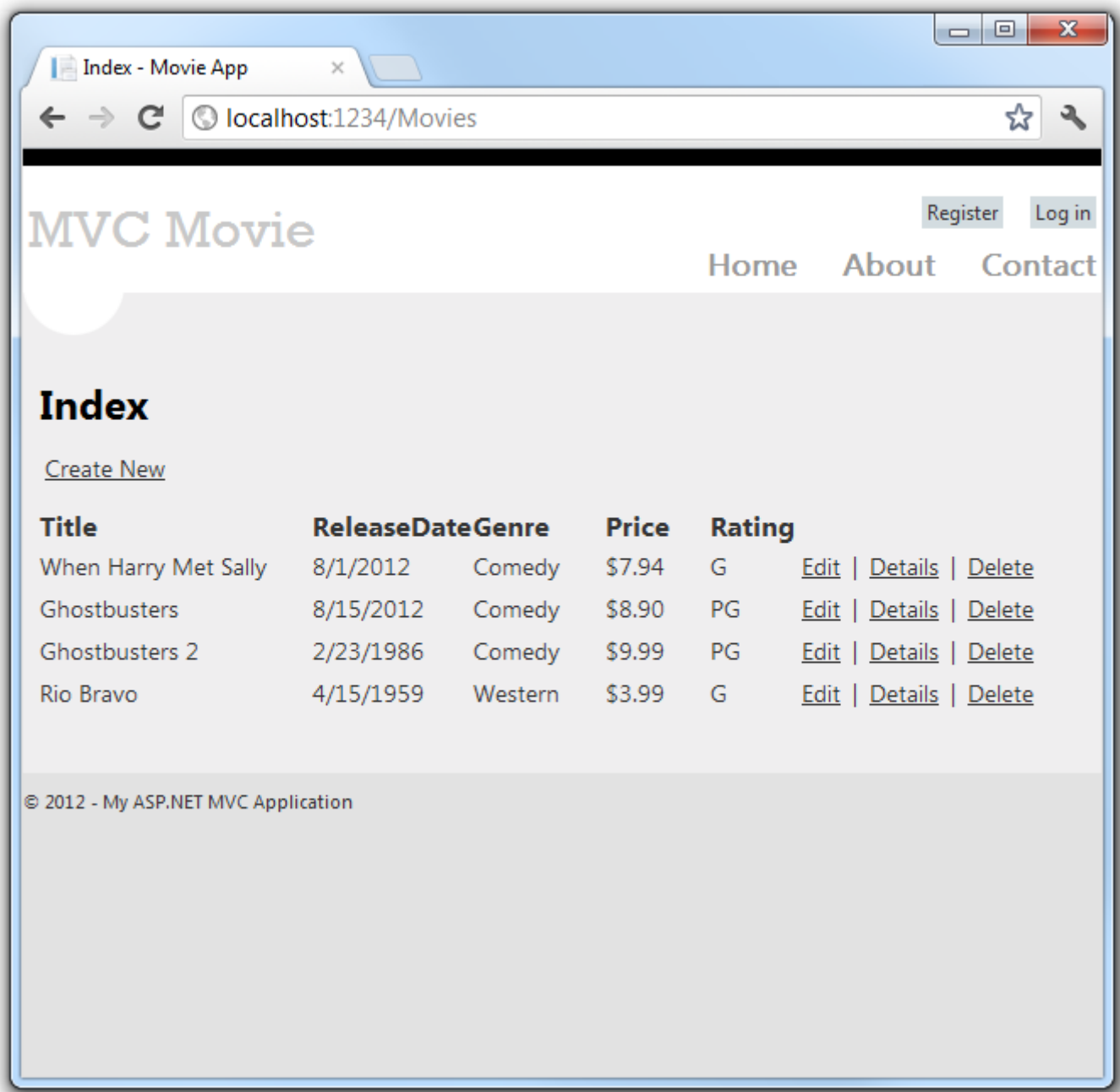
    [Range (1, 100)]
    [Tipo de datos (DataType.Currency)]
    Precio público decimal {get; establecido; }

    [StringLength (5)]
    cadena Clasificación pública {get; establecido; }
}
```

Ejecutar la aplicación y busque el **Movies** controlador. La fecha de lanzamiento y precio están bien formateados. La imagen de abajo muestra la fecha de lanzamiento y precio usando "fr-FR" como la cultura.



La imagen de abajo muestra los mismos datos que se muestran con la cultura por defecto (Inglés de EE.UU.).



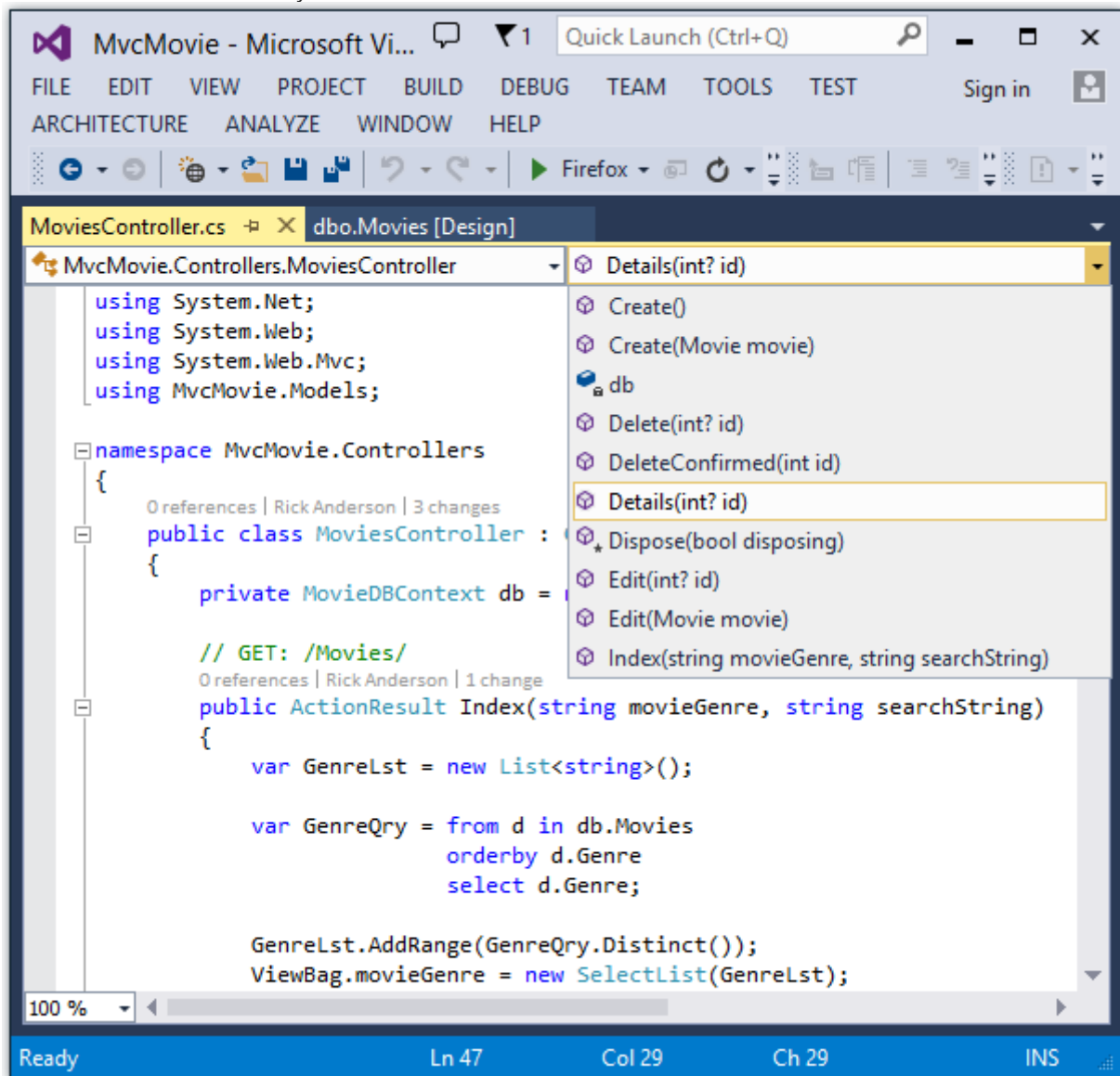
En la siguiente parte de la serie, vamos a revisar la aplicación y hacer algunas mejoras a la generada automáticamente **Details** y **Delete** métodos.

Examinar los detalles y eliminar los métodos

En esta parte del tutorial, podrás examinar la generada automáticamente **Details** y **Delete** métodos.

Examinar los detalles y eliminar los métodos

Abra el **Movie** controlador y examinar la **Details** método.



```
pública ActionResult detalles (int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult (HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find (id);
    if (película == null)
    {
        volver HttpNotFound ();
    }
    volver Ver (película);
}
```


El motor de andamios MVC que creó este método de acción agrega un comentario que muestra una petición HTTP que invoca el método. En este caso se trata de un **GET** solicitud con tres segmentos de URL, el **Movies** controlador, el **Details** método y un **ID** de valor. Primer Código hace que sea fácil de buscar datos utilizando la **Find** método. Una característica importante de seguridad incorporado en el método es que el código verifica que el **Find** método ha encontrado una película antes de que el código intenta hacer algo con ella. Por ejemplo, un hacker podría introducir errores en el sitio cambiando la URL creado por los enlaces de *http://localhost:xxxx/Cine/Details/1* a algo como *http://localhost:xxxx/Cine/Details/12345* (o algún otro valor que no representa una película real). Si no marcó para una película nula, una película nula daría lugar a un error de base de datos. Examine los **Delete** y **DeleteConfirmed** métodos.

```
// GET: / Películas / Borrar / 5
ActionResult público Delete (int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult (HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find (id);
    if (película == null)
    {
        volver HttpNotFound ();
    }
    volver Ver (película);
}

// Mensaje: / Películas / Borrar / 5
[HttpPost,  ActionName ("Eliminar") ]
[ValidateAntiForgeryToken]
pública ActionResult DeleteConfirmed (int id)
{
    Movie movie = db.Movies.Find (id);
    db.Movies.Remove (película);
    db.SaveChanges ();
    volver RedirectToAction ("Índice");
}
```

Tenga en cuenta que el **HTTP Get Delete** método no elimina la película especificado, devuelve una vista de la película donde se puede presentar (**HttpPost**) la eliminación .. Realización de una operación de eliminación en respuesta a una solicitud GET (o para el caso, la realización de una operación de edición, cree el funcionamiento, o cualquier otra operación que cambie los datos) abre un agujero de seguridad. Para obtener más información sobre esto, ver de Stephen Walther entrada de blog [MVC ASP.NET Tip # 46 - No utilice Eliminar Vínculos porque crean agujeros de seguridad](#) .

El **HttpPost** método que elimina los datos se denomina **DeleteConfirmed** para dar el método HTTP POST una firma o nombre único. Las dos firmas de los métodos se muestran a continuación:

```
// GET: / Películas / Borrar / 5
pública ActionResult Eliminar (int? Id)

//
```

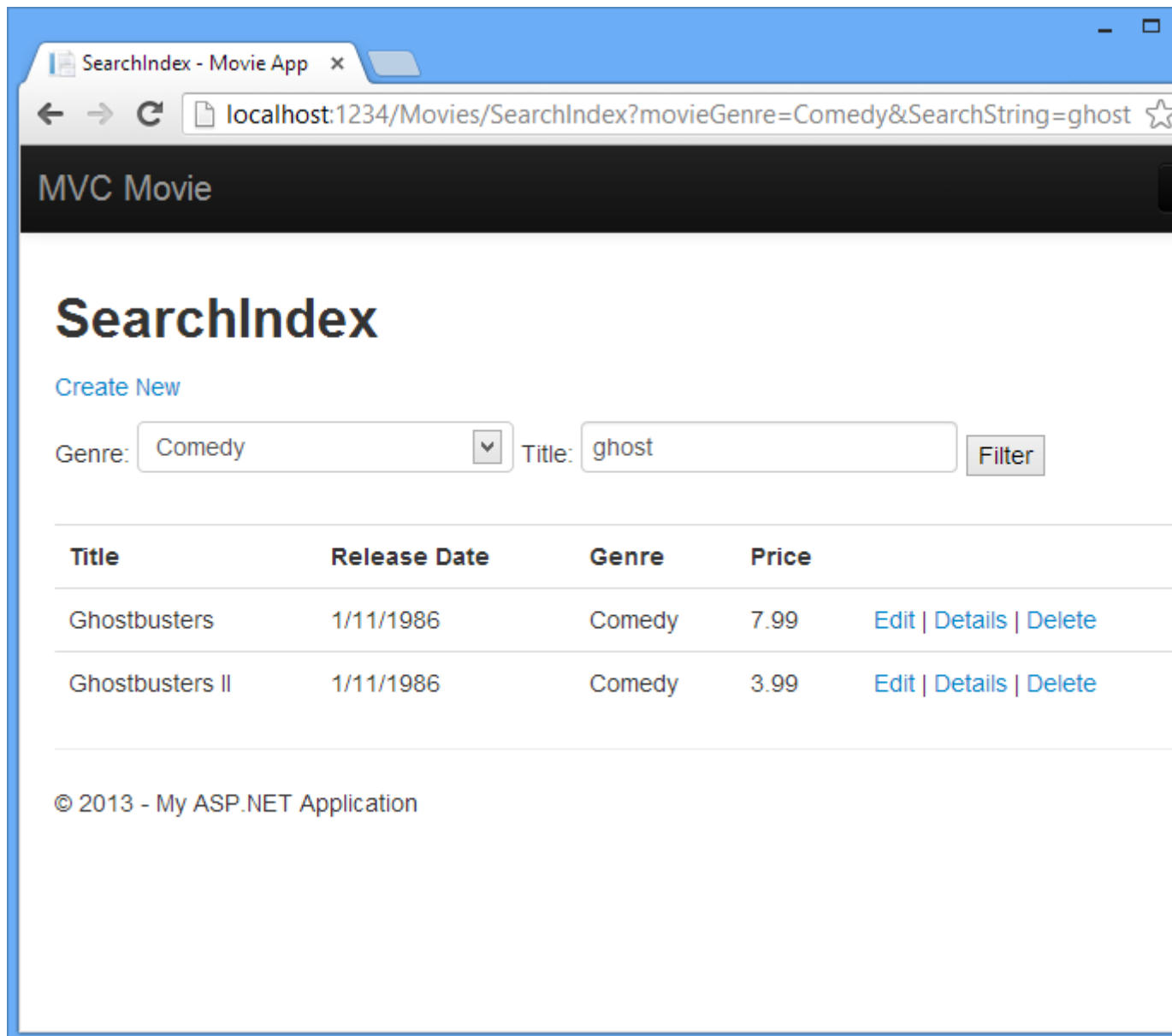
```
// Mensaje: / Películas / Borrar / 5
[HttpPost,      ActionName ("Eliminar")]
pública      ActionResult      DeleteConfirmed (int id)
```

El Common Language Runtime (CLR) requiere métodos sobrecargados tener una firma única de parámetros (mismo nombre de método pero diferente lista de parámetros). Sin embargo, aquí se necesitan dos métodos Delete - uno para GET y otro para POSTE - que ambos tienen la misma firma parámetro. (Ambos tienen que aceptar un único entero como parámetro.) Para solucionar esto, se puede hacer un par de cosas. Uno de ellos es dar a los métodos diferentes nombres. Eso es lo que el mecanismo de andamios hizo en el ejemplo anterior. Sin embargo, esto introduce un pequeño problema: los mapas ASP.NET segmentos de una URL a los métodos de acción por su nombre, y si cambia el nombre de un método, el enrutamiento normalmente no sería capaz de encontrar ese método. La solución es lo que se ve en el ejemplo, que consiste en añadir el `ActionName("Delete")` atributo al `DeleteConfirmed` método. Esto realiza efectivamente asignación para el sistema de enrutamiento de manera que una dirección URL que incluye `/ Borrar /` para una solicitud POST se encuentra el `DeleteConfirmed` método. Otra forma común para evitar un problema con métodos que tienen nombres y firmas idénticas es cambiar artificialmente la firma del método POST para incluir un parámetro no utilizado. Por ejemplo, algunos desarrolladores agregar un tipo de parámetro `FormCollection` que se pasa al método POST, y luego simplemente no usan el parámetro:

```
pública      ActionResult      Eliminar (FormCollection fcNotUsed,      int id =
0)
{
    Movie movie = db Películas Encontrar (id)..;
    si      (Película ==      null)
    {
        retorno      HttpNotFound ();
    }
    .. db Películas Quitar (película);
    db SaveChanges ().;
    retorno      RedirectToAction ("Índice");
}
```

Resumen

Ahora tiene una completa aplicación ASP.NET MVC que almacena datos en una base de datos local DB. Puede crear, leer, actualizar, eliminar y buscar películas.



Pasos siguientes

Después de haber construido y probado una aplicación web, el siguiente paso es ponerlo a disposición de otras personas a utilizar a través de Internet. Para hacer eso, usted tiene que implementarlo en un proveedor de alojamiento web.

Fin