	<p align="center">UNIVERSIDAD DON BOSCO FACULTAD DE INGENIERIA ESCUELA DE COMPUTACIÓN</p>
<p align="center">Ciclo II</p>	<p>PRACTICA DE LABORATORIO No. 2</p> <p>Nombre de la practica: "Introducción a ASP.NET MVC 5 Parte 2"</p> <p>Lugar de ejecución Centro de cómputo</p> <p>Tiempo estimado: 2 horas</p> <p>Materia Desarrollo de software empresarial</p>

Objetivo General

Adquirir conocimientos, habilidades y destrezas en la creación de aplicaciones web básicas utilizando ASP.NET MVC 5 con Visual Studio 2017 y utilizando la base de datos localDB de SQLServer proporcionada por el IDE.

Objetivos Específicos

- Que el estudiante aprenda a crear una cadena de conexión a localDB para un proyecto ASP.NET MVC.
- Que el estudiante aprenda a acceder a los datos de una base de datos localDB y pueda mostrar estos datos en una vista.
- Que el estudiante aprenda a realizar métodos de búsqueda para atributos específicos de un modelo y mostrar los datos filtrados en una vista.

Descripción

El Laboratorio está dividido en seis partes. La primera parte veremos cómo crear la cadena de conexión con la base de datos localDB SQL que utilizaremos en nuestra aplicación, en la segunda parte aprenderemos a crear controladores que manejen los datos del modelo (CRUD), en la tercera parte veremos cómo trabajar con SQL Server localDB y poder acceder a los datos de la base de datos desde Visual Studio, en la cuarta parte examinaremos los métodos del controlador de películas, más concretamente los métodos de edición así como la vista de edición de películas, en la quinta parte examinaremos como se procesan las solicitudes post que utilizan los métodos que gestionan los objetos de película y en la sexta parte aprenderemos a crear métodos de búsqueda y cómo implementarlos desde la vista Index. Durante el laboratorio se harán una serie de ejercicios para desarrollar las habilidades y destrezas del alumno en la construcción de páginas web en servidor usando la tecnología ASP.NET MVC, empleando el lenguaje de programación C#.

Materiales y Recursos Didácticos

- Guía de Laboratorio.
- PC con internet y Navegador Web.
- PC con IDE Visual Studio 2017.
- Haber completado la guía de laboratorio anterior.

I. Crear una cadena de conexión y trabajar con LocalDB de SQL Server.

La clase PeliculaDBContext que creó en la guía anterior maneja la tarea de conectarse a la base de datos y asignar objetos Pelicula a los registros de la base de datos. Sin embargo, una pregunta que puede hacerse es cómo especificar a qué base de datos se conectará. En realidad, no tiene que especificar qué base de datos usar, Entity Framework usará LocalDB de manera predeterminada. En esta sección, agregaremos explícitamente una cadena de conexión en el archivo Web.config de la aplicación.

SQL Server Express LocalDB

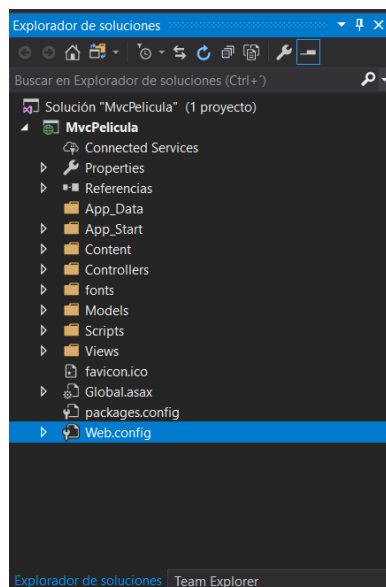
LocalDB es una versión ligera del Motor de base de datos SQL Server Express, que se inicia bajo demanda y se ejecuta en modo de usuario. LocalDB se ejecuta en un modo de ejecución especial de SQL Server Express, que le permite trabajar con bases de datos como archivos .mdf. Por lo general, los archivos de la base de datos LocalDB se guardan en la carpeta App_Data de un proyecto web.

No se recomienda el uso de SQL Server Express en aplicaciones web de producción. LocalDB en particular no debe usarse para producción con una aplicación web, porque, no está diseñado para funcionar con IIS. Sin embargo, una base de datos LocalDB se puede migrar fácilmente a SQL Server o SQL Azure.

Nota: En Visual Studio 2017, LocalDB se instala de manera predeterminada con Visual Studio.

De manera predeterminada, Entity Framework busca una cadena de conexión llamada igual que la clase de contexto de objeto (PeliculaDBContext para este proyecto).

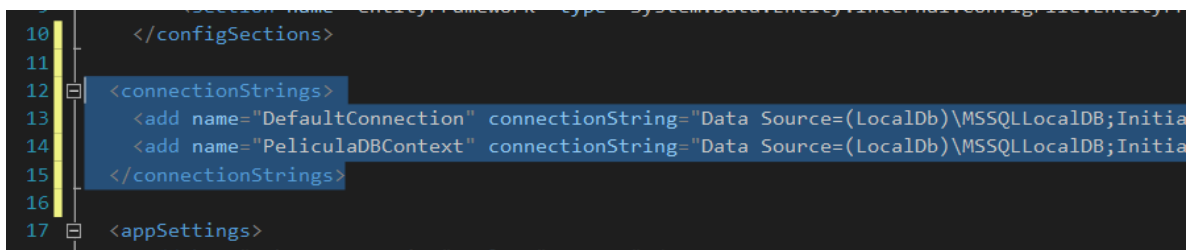
Abra el archivo Web.config que se encuentra en la raíz de la aplicación (Proyecto), el archivo se muestra a continuación. (No es el archivo Web.config en la carpeta Vistas).



Dentro del archivo encuentra el elemento <configSections> y luego de que el elemento cierre </configSections> coloque el código que agregara la cadena de Conexión.

```
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data
Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=aspnet-MvcPelícula-fefdc1f0-bd81-
4ce9-b712-93a062e01031;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\aspnet-MvcPelícula-fefdc1f0-bd81-
4ce9-b712-93a062e01031.mdf" providerName="System.Data.SqlClient" />
  <add name="PelículaDbContext" connectionString="Data
Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=aspnet-MvcPelícula;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\Películas.mdf"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

El resultado debe quedarle de la siguiente manera:



Las dos cadenas de conexión son muy similares. Se nombra la primera cadena de conexión DefaultConnection y se usa para la base de datos de membresía, para controlar quién puede acceder a la aplicación. La segunda cadena de conexión que ha agregado especifica una base de datos LocalDB llamada Película.mdf ubicada en la carpeta App_Data. No usaremos la base de datos de membresía para nuestra aplicación, usaremos la base de datos Película.mdf.(ejecute el script o cree su propia base de datos para este ejemplo)

```
create database pelicula;
create table peliculas (
  ID INT PRIMARY KEY IDENTITY (1, 1),
  titulo VARCHAR(60),
  fechaLanzamiento DATETIME,
  genero VARCHAR(1),
  precio DECIMAL);
```

Nota: El nombre de la cadena de conexión debe coincidir con el nombre de la clase DbContext.

```

13     }
14     0 referencias
15     public class PeliculaDBContext : DbContext
16     {
17         0 referencias
18         public DbSet<Pelicula> Peliculas { get; set; }
19     }

```

En realidad, no se necesita agregar la cadena de conexión PeliculaDBContext. Si no especifica una cadena de conexión, Entity Framework creará una base de datos LocalDB en el directorio de usuarios con el nombre completo de la clase DbContext (en este caso MvcPelicula.Models.PeliculaDBContext). Puede nombrar la base de datos como desee, siempre que tenga el sufijo .MDF. Por ejemplo, podríamos nombrar la base de datos MisFilmes.mdf.

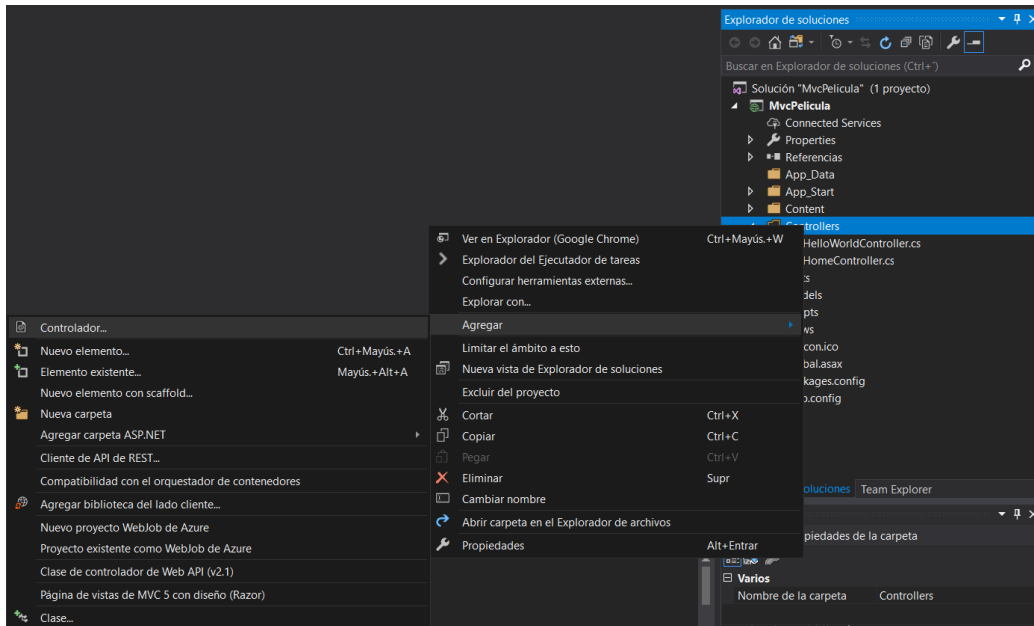
A continuación, agregaremos el controlador PeliculasController y lo usaremos para mostrar los datos de la película y permitir a los usuarios crear nuevas listas de películas.

II. Acceso a los datos de su modelo desde un controlador (CRUD).

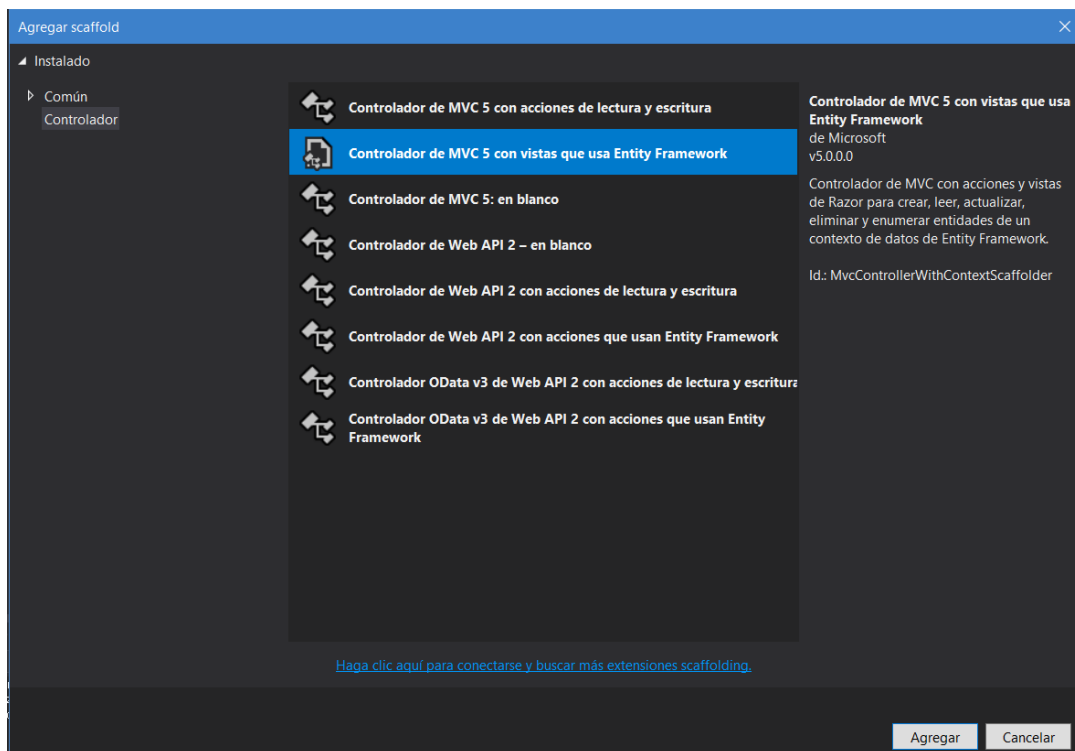
En esta sección, creará una nueva clase PeliculasController y escribiremos el código que recupera los datos de la película para mostrarlos en el navegador usando una plantilla de vista.

Compile la aplicación antes de pasar al siguiente paso. Si no compila la aplicación, recibirá un error al agregar un controlador.

En el Explorador de soluciones, haga clic con el botón derecho en la carpeta Controllers y luego haga clic en Agregar, luego en Controlador.



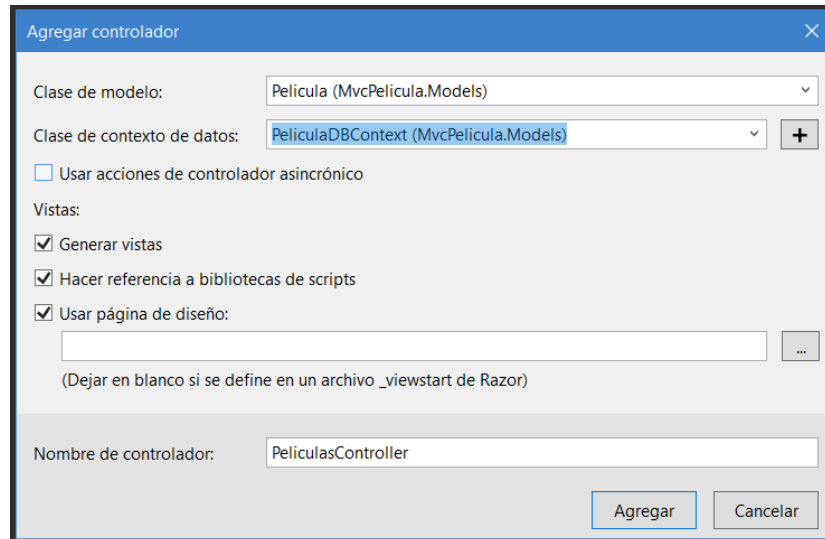
En el cuadro de diálogo Agregar scaffold, haga clic en Controlador MVC 5 con vistas que usa Entity Framework, y luego haga clic en Agregar.



- Seleccione Película (MvcPelícula.Models) para la Clase Modelo.

- Seleccione PeliculaDbContext (MvcPelicula.Models) para la Clase de Contexto de Datos.
- Para el nombre del controlador, ingrese PeliculasController.

La siguiente imagen muestra el cuadro de diálogo completado.

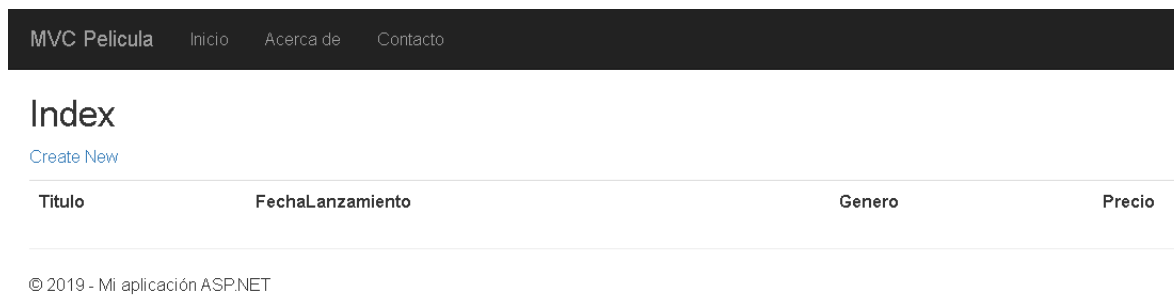


Haga clic en Agregar. (Si obtiene un error, probablemente no compiló la aplicación antes de comenzar a agregar el controlador). Visual Studio crea los siguientes archivos y carpetas:

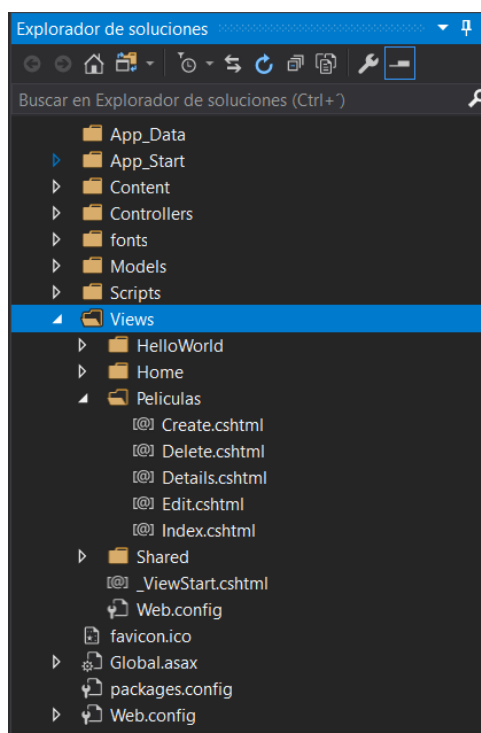
- Un archivo PeliculasController.cs en la carpeta Controllers.
- Una carpeta Views\Películas.
- Create.cshtml, Delete.cshtml, Details.cshtml, Edit.cshtml e Index.cshtml en la nueva carpeta Views\Películas.

Visual Studio creó automáticamente los métodos y vistas de acción CRUD (crear, leer, actualizar y eliminar) para usted (**la creación automática de métodos y vistas de acción CRUD se conoce como scaffolding**). Ahora tiene una aplicación web totalmente funcional que le permite crear, enumerar, editar y eliminar entradas de películas.

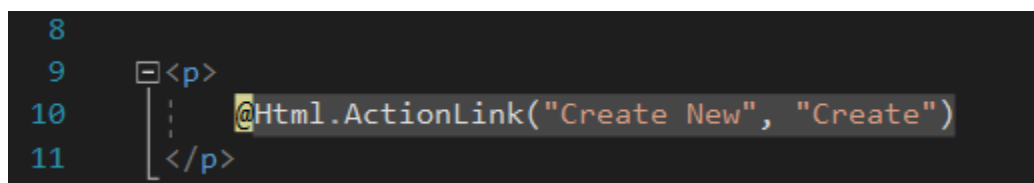
Ejecute la aplicación y haga clic en el enlace MVC Pelicula (o busque el controlador Peliculas agregando /Películas a la URL en la barra de direcciones de su navegador). Debido a que la aplicación se basa en el enrutamiento predeterminado (definido en el archivo App_Start\RouteConfig.cs), la solicitud del navegador <http://localhost:xxxxx/Peliculas> se enruta al método de acción Index predeterminado del controlador Peliculas. En otras palabras, la solicitud del navegador <http://localhost:xxxxx/Peliculas> es efectivamente la misma que la solicitud del navegador <http://localhost:xxxxx/Peliculas/Index>. El resultado es una lista vacía de películas, porque todavía no ha agregado ninguna.



Antes de crear una película vamos a traducir las vistas generadas al crear el controlador, para eso tendremos que abrir los archivos de Vista creados anteriormente.



Comenzaremos por la vista Index.cshtml y traduciremos los diálogos mostrados.



Traducimos solamente el primer atributo correspondiente al dialogo mostrado en la vista y queda de la siguiente manera.

```

9      <p>
10         @Html.ActionLink("Crear Nuevo", "Create")
11     </p>

```

Luego pasaremos a traducir los botones de acción mostrados en la misma vista.

```

43     <td>
44         @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
45         @Html.ActionLink("Details", "Details", new { id=item.ID }) |
46         @Html.ActionLink("Delete", "Delete", new { id=item.ID })
47     </td>

```

Asegurándonos de traducir solamente el primer atributo del método el cual corresponde al dialogo mostrado en la vista.

```

43     <td>
44         @Html.ActionLink("Editar", "Edit", new { id=item.ID }) |
45         @Html.ActionLink("Consultar", "Details", new { id=item.ID }) |
46         @Html.ActionLink("Eliminar", "Delete", new { id=item.ID })
47     </td>

```

Una vez modificado el archivo guardamos ejecutamos nuestra aplicación y en la pantalla de Inicio pulsamos el apartado MVC Pelicula el cual nos dirigirá al Index de PeliculasController, el resultado será el siguiente.

MVC Pelicula Inicio Acerca de Contacto

Index

[Crear Nuevo](#)

Titulo	FechaLanzamiento	Genero	Precio	
Votos de amor	6/2/2012 00:00:00	Romance	6.99	Editar Consultar Eliminar

© 2019 - Mi aplicación ASP.NET

Pasemos a traducir la vista de Create.cshtml la cual nos tiene que quedar de la siguiente manera.

```

7      <h2>Crear</h2>
8

```

Para traducir el texto mostrado en el botón de Create hay que modificar el value mostrado en la etiqueta <input> de la siguiente manera.


```

50     <div class="form-group">
51         <div class="col-md-offset-2 col-md-10">
52             <input type="submit" value="Crear" class="btn btn-default" />
53         </div>
54     </div>

```

Ahora solo nos queda en link para regresar a la lista de películas, el cual deberá quedarnos de la siguiente manera.

```

58 <div>
59     @Html.ActionLink("Regresar a la lista", "Index")
60 </div>

```

El resultado nos quedara de la siguiente manera.

MVC Pelicula Inicio Acerca de Contacto

Crear Película

Titulo

FechaLanzamiento

Genero

Precio

[Regresar a la lista](#)

© 2019 - Mi aplicación ASP.NET

Nota: Queda como tarea del estudiante traducir las tres vistas restantes correspondientes al CRUD de Películas.

Crear una película

Seleccione el enlace Crear Nuevo. Ingrese algunos detalles sobre una película y luego haga clic en el botón Crear.

MVC Pelicula

Inicio

Acerca de

Contacto

Crear

Pelicula

Titulo

Votos de Amor

FechaLanzamiento

6/2/2012

Genero

Romance

Precio

6.99

Crear

[Regresar a la lista](#)

© 2019 - Mi aplicación ASP.NET

Al hacer clic en el botón Crear, el formulario se publica en el servidor, donde la información de la película se guarda en la base de datos. Luego lo redirige a la URL de /Peliculas, donde puede ver la película recién creada en la lista.

MVC Pelicula

Inicio

Acerca de

Contacto

Index

[Crear Nuevo](#)

Titulo	FechaLanzamiento	Genero	Precio	
Votos de Amor	6/2/2012 00:00:00	Romance	6.99	Editar Consultar Eliminar

© 2019 - Mi aplicación ASP.NET

Cree un par de entradas de películas más. Además, pruebe los enlaces Editar, Detalles y Eliminar, que son todos funcionales.

Examinando el Código Generado

Abra el archivo Controllers\PeliculasController.cs y examine el método Index generado. A continuación, se muestra una parte del controlador de película con el método Index.

```
public class PeliculasController : Controller
{
    private PeliculaDbContext db = new PeliculaDbContext();
```

```
// GET: Peliculas
public ActionResult Index()
{
    return View(db.Peliculas.ToList());
}
```

Una solicitud al controlador Peliculas devuelve todas las entradas en la tabla Peliculas y luego pasa los resultados a la vista Index. La siguiente línea de la clase PeliculasController crea una instancia de un contexto de base de datos de películas, como se describió anteriormente. Puede usar el contexto de la base de datos de películas para consultar, editar y eliminar películas.

```
private PeliculaDBContext db = new PeliculaDBContext();
```

Modelos fuertemente tipados y la palabra clave @model.

En la guía anterior, vimos cómo un controlador puede pasar datos u objetos a una plantilla de vista utilizando el objeto ViewBag. El ViewBag es un objeto dinámico que proporciona una forma conveniente de pasar información a una vista.

MVC también proporciona la capacidad de pasar objetos fuertemente tipados a una plantilla de vista. Este enfoque fuertemente tipado permite una mejor verificación en tiempo de compilación de su código e IntelliSense (IntelliSense es el término general que se usa para describir varias características: Lista de miembros, Información de parámetros, Información rápida y Palabra completa. Estas características le permiten obtener más información acerca del código que utiliza, a realizar un seguimiento de los parámetros que escribe y a agregar llamadas a propiedades y a métodos con tan solo presionar unas teclas.) más rico en el editor de Visual Studio. El mecanismo de scaffolding en Visual Studio, usó este enfoque (es decir, pasar un modelo fuertemente tipado) con las plantillas de clase y vista PeliculasController cuando creó los métodos y las vistas.

En el archivo Controllers\PeliculasController.cs, examine el método Details generado. El método Details se muestra a continuación.

```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Pelicula pelicula = db.Peliculas.Find(id);
    if (pelicula == null)
    {
        return HttpNotFound();
    }
    return View(pelicula);
}
```

El parámetro id generalmente se pasa como datos de ruta, por ejemplo <http://localhost:1234/peliculas/details/1>, establecerá el controlador en el controlador de la película, la acción a details y el id 1. También puede pasar la identificación con una cadena de consulta de la siguiente manera:

<http://localhost:xxxxx/peliculas/details?id=1>

Si se encuentra una película, se pasa una instancia del modelo de película a la vista Details:

```
return View(pelicula);
```

Examine el contenido del archivo Views\Peliculas\Details.cshtml:

```
@model MvcPelicula.Models.Pelicula

@{
    ViewBag.Title = "Details";
}

<h2>Detalles</h2>

<div>
    <h4>Pelicula</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Titulo)
        </dt>
        @ *Codigo omitido para mayor claridad. * @
    </dl>
</div>
<p>
    @Html.ActionLink("Editar", "Edit", new { id = Model.ID }) |
    @Html.ActionLink("Regresar a la lista", "Index")
</p>
```

Al incluir una declaración @model en la parte superior del archivo de plantilla de vista, puede especificar el tipo de objeto que la vista espera. Cuando creó el controlador de película, Visual Studio incluyó automáticamente la siguiente declaración @model en la parte superior del archivo Details.cshtml:

```
@model MvcPelicula.Models.Pelicula
```

Esta directiva @model le permite acceder a la película que el controlador pasó a la vista utilizando un objeto Model fuertemente tipado. Por ejemplo, en la plantilla Details.cshtml, el código pasa cada campo de película a los Helpers HTML DisplayNameFor y DisplayFor con el objeto Model fuertemente tipado. Los métodos Crear y editar y las plantillas de vista también pasan un objeto modelo de película.

Examine la plantilla de vista Index.cshtml y el método Index en el archivo PeliculasController.cs. Observe cómo el código crea un objeto List cuando llama al método auxiliar View en el método de acción Index. El código luego pasa esta lista Peliculas del método de acción Index a la vista:

```
public ActionResult Index()
{
    return View(db.Peliculas.ToList());
}
```

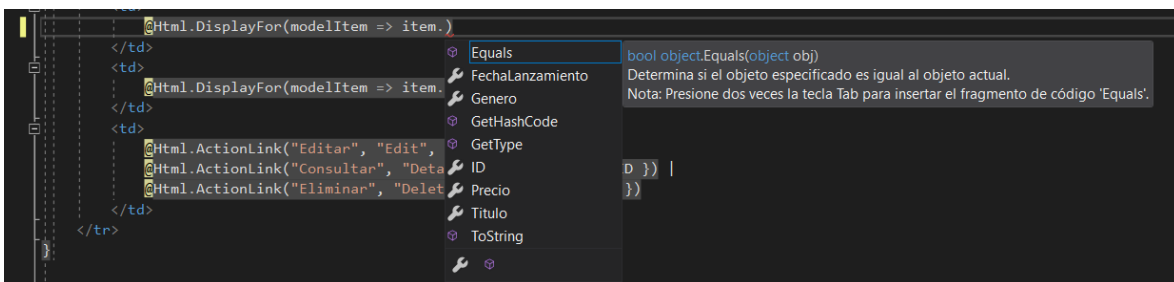
Cuando creó el controlador de película, Visual Studio incluyó automáticamente la siguiente declaración @model en la parte superior del archivo Index.cshtml:

```
@model IEnumerable<MvcPelicula.Models.Pelicula>
```

Esta directiva @model le permite acceder a la lista de películas que el controlador pasó a la vista mediante el uso de un objeto Model fuertemente tipado. Por ejemplo, en la plantilla Index.cshtml, el código recorre las películas haciendo una declaración foreach sobre el objeto Model fuertemente tipado:

```
@foreach (var item in Model) {  
    <tr>  
        <td>  
            @Html.DisplayFor(modelItem => item.Titulo)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.FechaLanzamiento)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.Genero)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.Precio)  
        </td>  
        <td>  
            @Html.ActionLink("Editar", "Edit", new { id=item.ID }) |  
            @Html.ActionLink("Consultar", "Details", new { id=item.ID }) |  
            @Html.ActionLink("Eliminar", "Delete", new { id=item.ID })  
        </td>  
    </tr>  
}
```

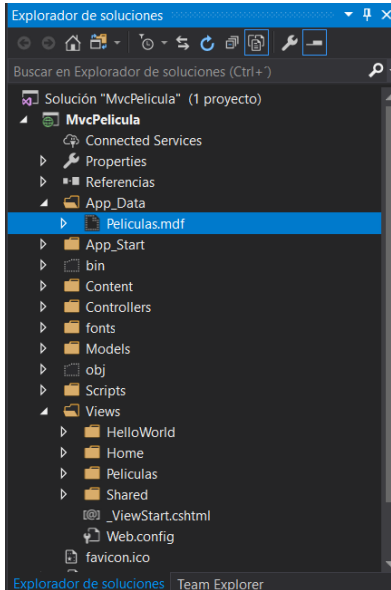
Debido a que el objeto Model está fuertemente tipado (como un objeto IEnumerable<Pelicula>), cada objeto item en el ciclo se tipea como Pelicula. Entre otros beneficios, esto significa que obtiene una verificación en tiempo de compilación del código y soporte completo de IntelliSense en el editor de código:



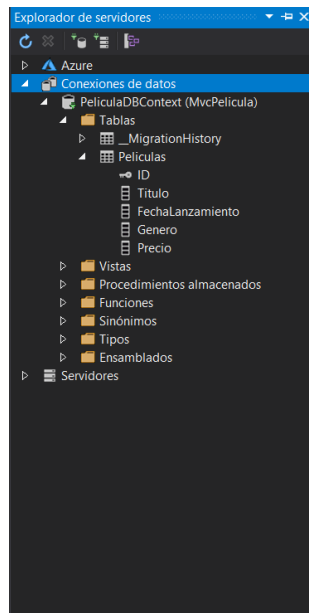
III. Trabajando con SQL Server LocalDB.

Entity Framework Code First detectó que la cadena de conexión de la base de datos que se proporcionó apuntaba a una base de datos Peliculas que aún no existía, por lo que Code First creó la base de datos automáticamente. Puede verificar que se haya creado buscando en la carpeta App_Data. Si no ve el archivo Peliculas.mdf, haga clic en el botón Mostrar todos los archivos en la

barra de herramientas del Explorador de soluciones, haga clic en el botón Actualizar y luego expanda la carpeta App_Data.

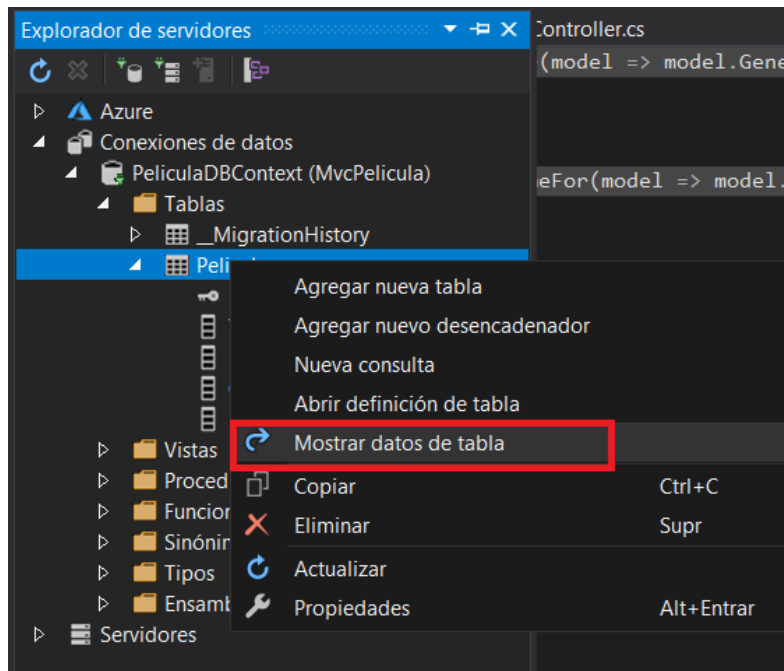


Haga doble clic en Películas.mdf para abrir SERVER EXPLORER, luego expanda la carpeta Tablas para ver la tabla Películas. Tenga en cuenta el icono de la llave junto a ID. Por defecto, EF hará que una propiedad llamada ID sea la clave principal.



Nota: Si da error al intentar abrir la base de datos Pelicula cierre Visual Studio, vuelva a ejecutarlo y abra de nuevo la base de datos.

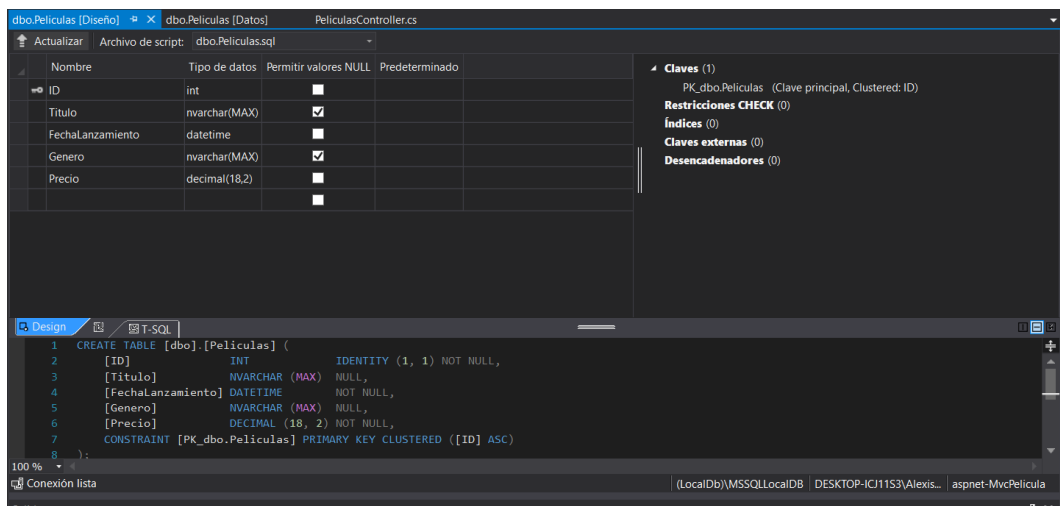
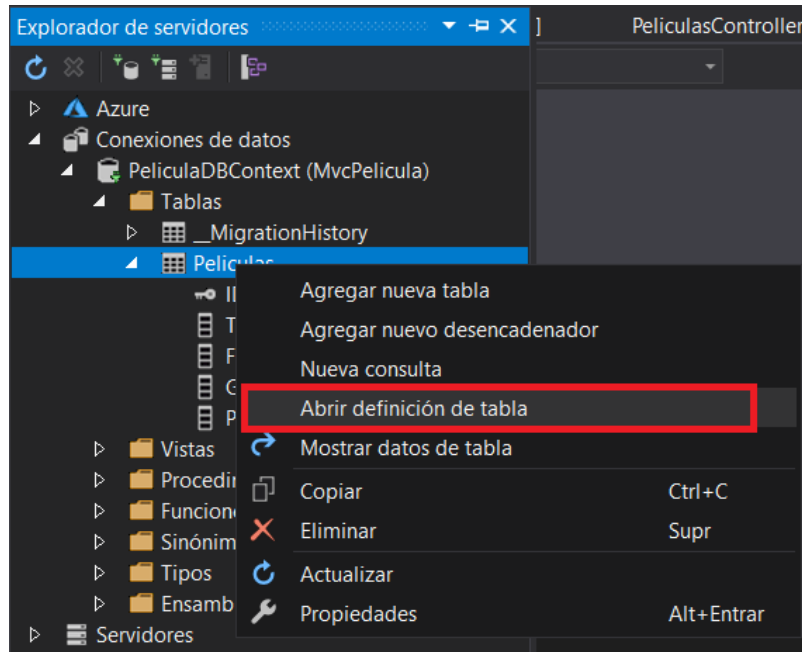
Haga clic con el botón derecho en la tabla Películas y seleccione Mostrar datos de tabla para ver los datos que creó.



The screenshot shows the 'dbo.Películas [Datos]' data table in SQL Server. The table has columns: ID, Titulo, FechaLanza..., Genero, and Precio. The first row shows ID 2, Titulo 'Votos de A...', FechaLanza... '6/2/2012 00...', Genero 'Romance', and Precio '6.99'. The second row shows NULL values.

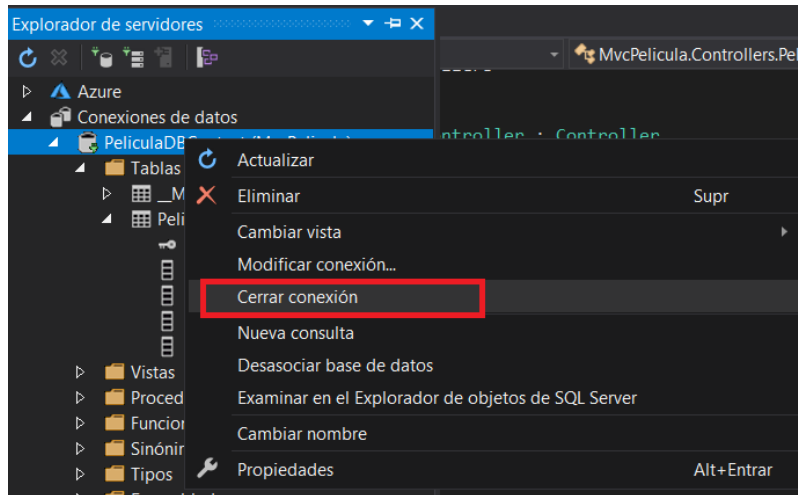
	ID	Titulo	FechaLanza...	Genero	Precio
▶	2	Votos de A...	6/2/2012 00...	Romance	6.99
⚙	NULL	NULL	NULL	NULL	NULL

Haga clic con el botón derecho en la tabla Películas y seleccione Abrir definición de tabla para ver la estructura de la tabla que Entity Framework Code First creó para usted.



Observe cómo el esquema de la tabla Películas se asigna a la clase Película que creó anteriormente. Entity Framework Code First creó automáticamente este esquema para usted en función de su clase Película.

Cuando haya terminado, cierre la conexión haciendo clic derecho en PeliculaDBContext y seleccionando Cerrar conexión. (Si no cierra la conexión, puede recibir un error la próxima vez que ejecute el proyecto).



Ahora tiene una base de datos y páginas para consultar, editar, actualizar y eliminar datos. Más adelante en esta guía, examinaremos el resto del código scaffolded y agregaremos un método SearchIndex y una vista SearchIndex que le permite buscar películas en esta base de datos.

IV. Examinar los métodos de edición y la vista de edición.

En esta sección, examinaremos los métodos de acción generados Edit y las vistas para el controlador de película. Pero primero tomaremos un pequeño desvío para que la fecha de lanzamiento se vea mejor. Abra el archivo Models\Pelicula.cs y agregue las líneas resaltadas que se muestran a continuación:

```
using System;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;

namespace MvcPelícula.Models
{
    public class Película
    {
        public int ID { get; set; }
        public string Titulo { get; set; }
        [Display(Name = "Fecha de Lanzamiento")]
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode
= true)]
        public DateTime FechaLanzamiento { get; set; }
        public string Genero { get; set; }
        public decimal Precio { get; set; }
    }
    public class PelículaDbContext : DbContext
    {
        public DbSet<Película> Películas { get; set; }
    }
}
```

Vamos a cubrir DataAnnotations mas adelante en esta guía. El atributo Display especifica qué mostrar para el nombre de un campo (en este caso, "Fecha de lanzamiento" en lugar de "FechaLanzamiento"). El atributo DataType especifica el tipo de datos, en este caso es una fecha, por lo que no se muestra la información de tiempo almacenada en el campo. El atributo DisplayFormat es necesario para un error en el navegador Chrome que representa los formatos de fecha incorrectamente.

Ejecute la aplicación y busque el controlador Peliculas. Mantenga el puntero del mouse sobre un enlace Editar para ver la URL a la que se vincula.

MVC Pelicula

Inicio

Acerca de

Contacto

Index

[Crear Nuevo](#)

Título	Fecha de Lanzamiento	Genero	Precio	
Votos de Amor	2012-02-06	Romance	6.99	Editar Consultar Eliminar

© 2019 - Mi aplicación ASP.NET

localhost:51983/Peliculas/Edit/2

El enlace Editar fue generado por el método `Html.ActionLink` en la vista `Views\Películas\Index.cshtml`:

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

```
<td>
    @Html.ActionLink("Editar", "Edit", new { id=item.ID }) |
    @Html.ActionLink("Consultar", "Details", new { id=item.ID }) |
    @Html.ActionLink("Eliminar", "Delete", new { id=item.ID })
</td>
```

El objeto `Html` es un helper que se expone utilizando una propiedad en la clase base `System.Web.Mvc.WebViewPage`. El método `ActionLink` del helper facilita la generación dinámica de hipervínculos HTML que enlazan con métodos de acción en los controladores. El primer argumento para el método `ActionLink` es el texto del enlace a representar (por ejemplo, `<a>Editar`). El segundo argumento es el nombre del método de acción a invocar (en este caso, el metodo `Edit`). El argumento final es un objeto anónimo que genera los datos de la ruta (en este caso, el ID de 2).

El enlace generado que se muestra en la imagen anterior es `http://localhost:51983/Peliculas/Edit/2`. La ruta predeterminada (establecida en `App_Start\RouteConfig.cs`) toma el patrón de URL `{controller}/{action}/{id}`. Por lo tanto, ASP.NET se traduce `http://localhost:51983/Peliculas/Edit/2`. En una solicitud al método de acción `Edit` del controlador `Peliculas` con el parámetro `ID` igual a 2. Examine el siguiente código del archivo `App_Start\RouteConfig.cs`. El método `MapRoute` se utiliza para enrutar las solicitudes HTTP al controlador y método de acción correctos, y proporcionar el

parámetro de ID opcional. El método MapRoute también lo utilizan los HtmlHelpers, como ActionLink para generar URLs dados en el controlador, el método de acción y cualquier dato de ruta.

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id =
        UrlParameter.Optional }
        );
        routes.MapRoute(
            name: "Hola",
            url: "{controller}/{action}/{nombre}/{id}"
        );
    }
}
```

También puede pasar parámetros del método de acción utilizando una cadena de consulta. Por ejemplo, la URL <http://localhost:51983/Peliculas/Edit?ID=3> también pasa el parámetro ID de 3 al método de acción Edit del controlador Peliculas.

Abramos el controlador Peliculas. Los dos métodos de acción Edit se muestran a continuación.

```
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Pelicula pelicula = db.Peliculas.Find(id);
    if (pelicula == null)
    {
        return HttpNotFound();
    }
    return View(pelicula);
}

// POST: Peliculas/Edit/5
// Para protegerse de ataques de publicación excesiva, habilite las
propiedades específicas a las que desea enlazarse. Para obtener
// más información vea https://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include =
"ID,Titulo,FechaLanzamiento,Genero,Precio")] Pelicula pelicula)
{
    if (ModelState.IsValid)
    {
        db.Entry(pelicula).State = EntityState.Modified;
```

```

        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(pelicula);
}

```

Observe que el segundo método de acción Edit está precedido por el atributo `HttpPost`. Este atributo especifica que la sobrecarga del método Edit solo se puede invocar para solicitudes POST. Puede aplicar el atributo `HttpGet` al primer método de edición, pero eso no es necesario porque es el predeterminado. (Nos referiremos a los métodos de acción que están asignados implícitamente al atributo `HttpGet` como métodos `HttpGet`). El atributo `Bind` es otro mecanismo de seguridad importante que evita que los hackers publiquen datos en exceso en su modelo. Solo debe incluir propiedades en el atributo de enlace que desea cambiar. En el modelo simple utilizado en esta guía vincularemos todos los datos del modelo. El atributo `ValidateAntiForgeryToken` se usa para evitar la falsificación de una solicitud y se combina con `@Html.AntiForgeryToken()`, el archivo de vista de edición (`Views\Peliculas\Edit.cshtml`), a continuación, se muestra una parte:

```

@model MvcPelicula.Models.Pelicula

@{
    ViewBag.Title = "Edit";
}

<h2>Editar</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Pelicula</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        @Html.HiddenFor(model => model.ID)

        <div class="form-group">
            @Html.LabelFor(model => model.Titulo, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Titulo, new { htmlAttributes = new
{ @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Titulo, "", new { @class
= "text-danger" })
            </div>
        </div>
    </div>
}

```

`@Html.AntiForgeryToken()` genera un token anti-falsificación de forma oculta que debe coincidir con el método Edit del controlador Peliculas.

El método `HttpGet` Edit toma el parámetro ID de la película, busca la película con el método Entity Framework Find y devuelve la película seleccionada a la vista Edit. Si no se puede encontrar una película, se devuelve `HttpNotFound`. Cuando el sistema de scaffolding creó la vista Edit, examinó la clase Pelicula y creó el código para representar elementos `<label>` y `<input>` para cada propiedad

de la clase. El siguiente ejemplo muestra la vista Editar que generó el sistema de scaffolding de Visual Studio:

```
@model MvcPelicula.Models.Pelicula

@{
    ViewBag.Title = "Edit";
}

<h2>Editar</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Pelicula</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        @Html.HiddenFor(model => model.ID)

        <div class="form-group">
            @Html.LabelFor(model => model.Titulo, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Titulo, new { htmlAttributes = new
{ @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Titulo, "", new { @class
= "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.FechaLanzamiento, htmlAttributes: new
{ @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.FechaLanzamiento, new
{ htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.FechaLanzamiento, "",
new { @class = "text-danger" })
            </div>
        </div>

        @ * Código omitido para mayor claridad. * @

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
```

```

        @Html.ActionLink("Regresar a la lista", "Index")
    </div>

    @section Scripts {
        @Scripts.Render("~/bundles/jqueryval")
    }

```

Observe cómo la plantilla de vista tiene una declaración `@model MvcPelicula.Models.Pelicula` en la parte superior del archivo; esto especifica que la vista espera que el modelo para la plantilla de vista sea de tipo `Pelicula`.

El código scaffolded utiliza varios métodos auxiliares para simplificar el marcado HTML. El asistente `Html.LabelFor` muestra el nombre del campo ("Título", "Fecha de lanzamiento", "Género" o "Precio"). El helper `Html.EditorFor` representa un elemento HTML `<input>`. El asistente `Html.ValidationMessageFor` muestra los mensajes de validación asociados con esa propiedad.

Ejecute la aplicación y navegue hasta la URL de `/Películas`. Haz clic en un enlace `Editar`. En el navegador, vea la fuente de la página. El HTML para el elemento de formulario se muestra a continuación.

```

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Pelicula</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        @Html.HiddenFor(model => model.ID)

        <div class="form-group">
            @Html.LabelFor(model => model.Titulo, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Titulo, new { htmlAttributes = new
{ @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Titulo, "", new { @class
= "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.FechaLanzamiento, htmlAttributes: new
{ @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.FechaLanzamiento, new
{ htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.FechaLanzamiento, "",
new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">

```

```

        @Html.LabelFor(model => model.Genero, htmlAttributes: new { @class =
"control-label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.Genero, new { htmlAttributes = new
{ @class = "form-control" } })
            @Html.ValidationMessageFor(model => model.Genero, "", new { @class
= "text-danger" })
        </div>
    </div>

    <div class="form-group">
        @Html.LabelFor(model => model.Precio, htmlAttributes: new { @class =
"control-label col-md-2" })
        <div class="col-md-10">
            @Html.EditorFor(model => model.Precio, new { htmlAttributes = new
{ @class = "form-control" } })
            @Html.ValidationMessageFor(model => model.Precio, "", new { @class
= "text-danger" })
        </div>
    </div>

    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Guardar" class="btn btn-default" />
        </div>
    </div>
</div>
}

```

V. Procesando la solicitud POST.

El siguiente código muestra la versión HttpPost del método de acción Edit.

```

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include =
"ID,Titulo,FechaLanzamiento,Genero,Precio")] Pelicula pelicula)
{
    if (ModelState.IsValid)
    {
        db.Entry(pelicula).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(pelicula);
}

```

El atributo ValidateAntiForgeryToken valida el token XSRF generado por la llamada en la vista @Html.AntiForgeryToken().

El modelo ASP.NET MVC model binder toma los valores de introducidos en el formulario y crea un objeto Pelicula que se pasa como parámetro pelicula. El ModelState.IsValid verifica que los datos

presentados en el formulario se pueden utilizar para modificar (editar o actualizar) un objeto Pelicula. Si los datos son válidos, los datos de la película se guardan en la lista Peliculas de la db (instancia PeliculaDBContext). Los nuevos datos de la película se guardan en la base de datos llamando al método SaveChanges de PeliculaDBContext. Después de guardar los datos, el código redirige al usuario al método de acción Index de la clase PeliculasController, que muestra la lista de películas, incluidos los cambios que se acaban de realizar.

Tan pronto como la validación del lado del cliente determina que el valor de un campo no es válido, se muestra un mensaje de error. Si JavaScript está deshabilitado, la validación del lado del cliente está deshabilitada. Sin embargo, el servidor detecta que los valores introducidos no son válidos y los valores del formulario se vuelven a mostrar con mensajes de error.

La validación se examina con más detalle en la siguiente guía.

Los helpers Html.ValidationMessageFor en la plantilla de vista Edit.cshtml se encargan de mostrar los mensajes de error apropiados.

MVC Pelicula Inicio Acerca de Contacto

Editar

Pelicula

Titulo

Fecha de Lanzamiento
El campo Fecha de Lanzamiento es obligatorio.

Genero

Precio
El campo Precio es obligatorio.

[Regresar a la lista](#)

© 2019 - Mi aplicación ASP.NET

Todos los métodos HttpGet siguen un patrón similar. Obtienen un objeto de película (o una lista de objetos, en el caso de Index) y pasan el modelo a la vista. El método Create pasa un objeto de película vacío a la vista Crear. Todos los métodos que crean, editan, eliminan o modifican datos lo hacen en la sobrecarga del método HttpPost. La modificación de datos en un método HTTP GET es un riesgo de seguridad. La modificación de datos en un método GET también viola las mejores prácticas de HTTP y el patrón arquitectónico REST, que especifica que las solicitudes GET no deberían cambiar el estado de su aplicación. En otras palabras, realizar una operación GET debería ser una operación segura que no tenga efectos secundarios y no modifique sus datos persistentes.

VI. Método de Búsqueda.

Agregar un método de búsqueda y una vista de búsqueda.

En esta sección, agregaremos la capacidad de búsqueda al método de acción Index que le permite buscar películas por género o nombre.

Prerrequisitos

Para hacer coincidir las capturas de pantalla de esta sección, debe ejecutar la aplicación (F5) y agregar las siguientes películas a la base de datos.

Título	Fecha de lanzamiento	Género	Precio
John Wick	24/10/2014	Acción	6,99
John Wick II	26/10/2015	Acción	6,99
John Wick III	17/05/2019	Acción	6.99

Actualización del formulario de índice

Comencemos actualizando el método Index de acción a la clase PeliculasController existente. Aquí está el código:

```
public ActionResult Index(string buscarString)
{
    var peliculas = from p in db.Peliculas
                    select p;

    if (!String.IsNullOrEmpty(buscarString))
    {
        peliculas = peliculas.Where(s => s.Titulo.Contains(buscarString));
    }
    return View(peliculas);
}
```

La primera línea del método Index crea la siguiente consulta LINQ (Language-Integrated Query (LINQ) es el nombre de un conjunto de tecnologías basadas en la integración de capacidades de consulta directamente en el lenguaje C #.) para seleccionar las películas:

```
var peliculas = from p in db.Peliculas
                select p;
```

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

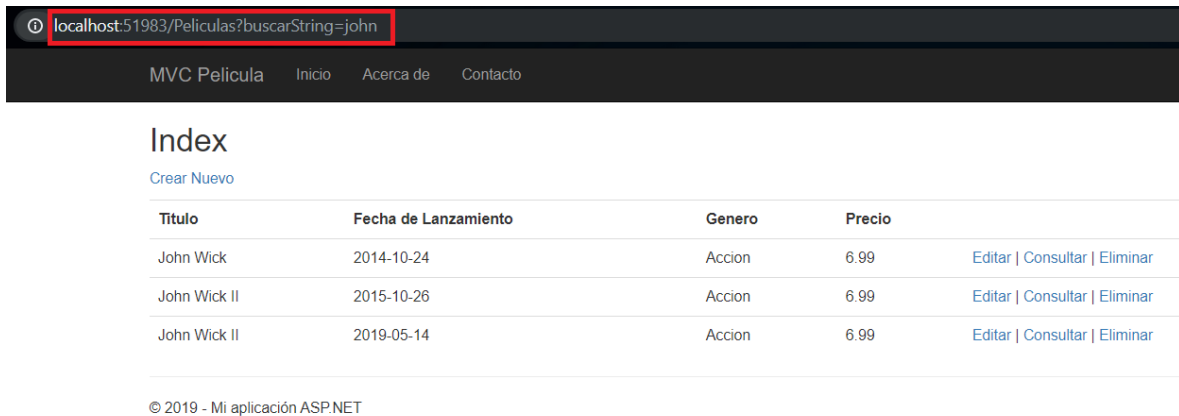
Si el parámetro buscarString contiene una cadena, la consulta de películas se modifica para filtrar el valor de la cadena de búsqueda, utilizando el siguiente código:

```
if (!String.IsNullOrEmpty(buscarString))
{
    peliculas = peliculas.Where(s => s.Titulo.Contains(buscarString));
}
```

El código `s => s.Title` anterior es una expresión de Lambda (Una lambda de expresión que tiene una expresión como cuerpo: `(input-parameters) => expression`, Una lambda de instrucción que tiene un bloque de instrucciones como cuerpo: `(input-parameters) => { <sequence-of-statements> }`). Las lambdas se utilizan en consultas LINQ basadas en métodos como argumentos para métodos de operador de consulta estándar, como el método `Where` utilizado en el código anterior. Las consultas LINQ no se ejecutan cuando se definen o cuando se modifican llamando a un método como `Where` u `OrderBy`. En cambio, la ejecución de la consulta se difiere, lo que significa que la evaluación de una expresión se retrasa hasta que su valor realizado se repite o se llama al método `ToList`. En el ejemplo `Search`, la consulta se ejecuta en la vista `Index.cshtml`.

Ahora puede actualizar la vista `Index` que mostrará el formulario al usuario.

Ejecute la aplicación y navegue hasta `/Películas/Index`. Agregue una cadena de consulta como `?buscarString=john` a la URL. Continuación se muestran las películas filtradas.



Titulo	Fecha de Lanzamiento	Genero	Precio	
John Wick	2014-10-24	Accion	6.99	Editar Consultar Eliminar
John Wick II	2015-10-26	Accion	6.99	Editar Consultar Eliminar
John Wick II	2019-05-14	Accion	6.99	Editar Consultar Eliminar

© 2019 - Mi aplicación ASP.NET

Si cambia la firma del método `Index` para que tenga un parámetro llamado `id`, el parámetro `id` coincidirá con el marcador de posición `{id}` para las rutas predeterminadas establecidas en el archivo `App_Start\RouteConfig.cs`.

`{controller}/{action}/{id}`

El método `Index` original se ve así:

```
public ActionResult Index(string buscarString)
{
    var peliculas = from p in db.Peliculas
                    select p;

    if (!String.IsNullOrEmpty(buscarString))
    {
        peliculas = peliculas.Where(s => s.Titulo.Contains(buscarString));
    }
    return View(peliculas);
}
```

El método `Index` modificado se vería de la siguiente manera:

```

public ActionResult Index(string id)
{
    string buscarString = id;
    var peliculas = from p in db.Peliculas
                    select p;

    if (!String.IsNullOrEmpty(buscarString))
    {
        peliculas = peliculas.Where(s => s.Titulo.Contains(buscarString));
    }
    return View(peliculas);
}

```

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

localhost:51983/Peliculas/index/john

MVC Pelicula Inicio Acerca de Contacto

Index

[Crear Nuevo](#)

Titulo	Fecha de Lanzamiento	Genero	Precio	
John Wick	2014-10-24	Accion	6.99	Editar Consultar Eliminar
John Wick II	2015-10-26	Accion	6.99	Editar Consultar Eliminar
John Wick II	2019-05-14	Accion	6.99	Editar Consultar Eliminar

© 2019 - Mi aplicación ASP.NET

Sin embargo, no puede esperar que los usuarios modifiquen la URL cada vez que quieran buscar una película. Así que ahora agregará la interfaz de usuario para ayudarlos a filtrar películas. Si cambió la firma del método Index para probar cómo pasar el parámetro ID vinculado a la ruta, cámbielo de nuevo para que su método Index tome un parámetro de cadena llamado buscarString:

```

public ActionResult Index(string buscarString)
{
    var peliculas = from p in db.Peliculas
                    select p;

    if (!String.IsNullOrEmpty(buscarString))
    {
        peliculas = peliculas.Where(s => s.Titulo.Contains(buscarString));
    }
    return View(peliculas);
}

```

Abra el archivo Views\Peliculas\Index.cshtml, y justo después de @ Html.ActionLink ("Crear nuevo", "Create"), agregue el marcado de formulario resaltado a continuación:

```
@model IEnumerable<MvcPelicula.Models.Pelicula>
```

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

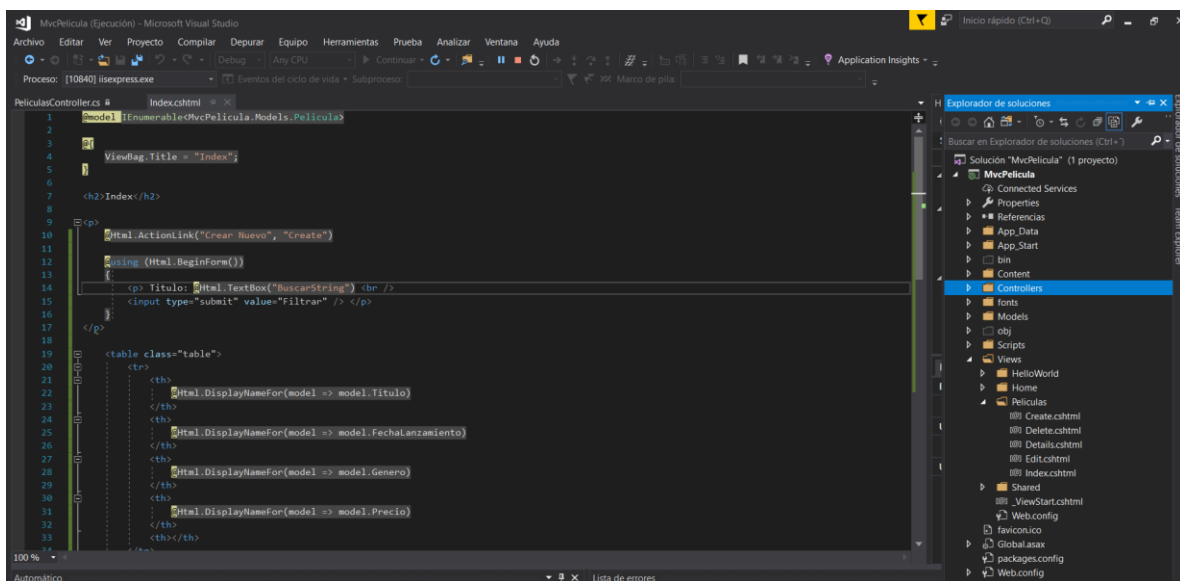
<h2>Index</h2>

<p>
    @Html.ActionLink("Crear Nuevo", "Create")

    @using (Html.BeginForm())
    {
        <p> Title: @Html.TextBox("BuscarString") <br />
        <input type="submit" value="Filter" /> </p>
    }
</p>
```

El helper `Html.BeginForm` crea una etiqueta de apertura `<form>`. El helper `Html.BeginForm` hace que el formulario se publique en sí mismo cuando el usuario envía el formulario haciendo clic en el botón Filtro.

Visual Studio 2017 tiene una buena mejora al mostrar y editar archivos de visualización. Cuando ejecuta la aplicación con un archivo de vista abierto, Visual Studio 2017 invoca el método de acción del controlador correcto para mostrar la vista.



Con la vista `Index` abierta en Visual Studio (como se muestra en la imagen de arriba), toque `Ctrl F5` o `F5` para ejecutar la aplicación y luego intente buscar una película.

Index

[Crear Nuevo](#)Titulo:

Titulo	Fecha de Lanzamiento	Genero	Precio	
John Wick	2014-10-24	Accion	6.99	Editar Consultar Eliminar
John Wick II	2015-10-26	Accion	6.99	Editar Consultar Eliminar
John Wick II	2019-05-14	Accion	6.99	Editar Consultar Eliminar

© 2019 - Mi aplicación ASP.NET

No hay sobrecarga HttpPost del método Index. No lo necesita, porque el método no cambia el estado de la aplicación, solo filtra los datos.

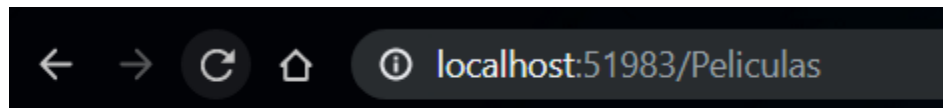
Puede agregar el siguiente método HttpPost Index al controlador PeliculasController.

```
public ActionResult Index(string buscarString)
{
    var peliculas = from p in db.Peliculas
                    select p;

    if (!String.IsNullOrEmpty(buscarString))
    {
        peliculas = peliculas.Where(s => s.Titulo.Contains(buscarString));
    }
    return View(peliculas);
}

[HttpPost]
public string Index(FormCollection fc, string buscarString)
{
    return "<h3> From [HttpPost]Index: " + buscarString + "</h3>";
}
```

En ese caso, el invocador de acción coincidiría con el método HttpPost Index, y el método HttpPost Index se ejecutaría como se muestra en la imagen a continuación.



From [HttpPost]Index: John

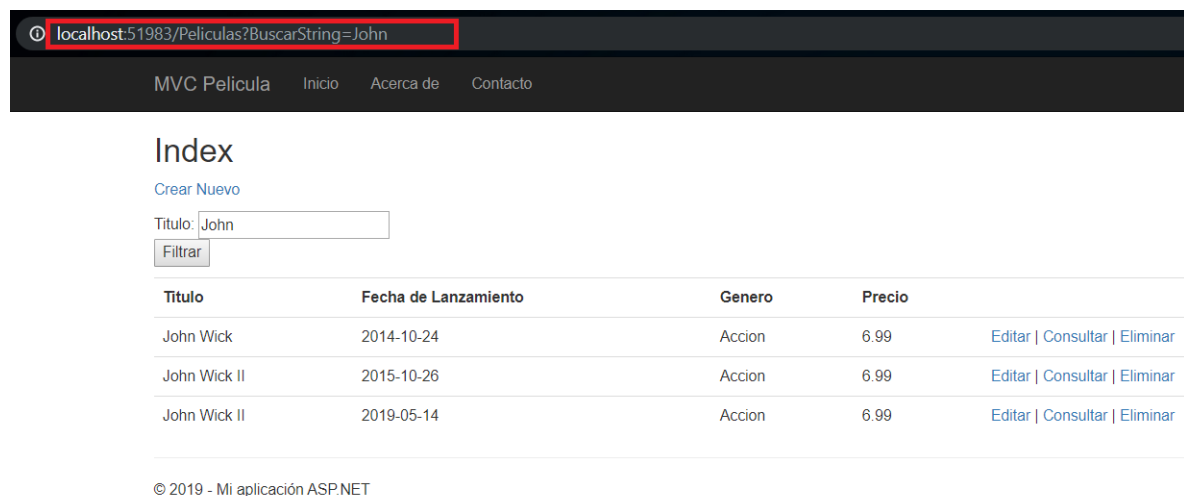
Sin embargo, incluso si agrega esta versión HttpPost del método Index, hay una limitación en cómo se ha implementado todo esto. Imagine que desea marcar una búsqueda en particular o desea enviar un enlace a amigos en el que puedan hacer clic para ver la misma lista filtrada de películas. Tenga en cuenta que la URL de la solicitud HTTP POST es la misma que la URL de la solicitud GET (localhost:xxxxx/Peliculas/Index): no hay información de búsqueda en la URL en sí. 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 capturar esa información de búsqueda para marcar o enviar a amigos en una URL.

La solución es utilizar una sobrecarga BeginForm que especifique que la solicitud POST debe agregar la información de búsqueda a la URL y que debe enrutarse a la versión HttpGet del método Index. Reemplace el método BeginForm de la vista Index.cshtml sin parámetros existente con el siguiente código marcado:

```
@using (Html.BeginForm("Index", "Peliculas", FormMethod.Get))
```

```
9  <p>
10  @Html.ActionLink("Crear Nuevo", "Create")
11
12  @using (Html.BeginForm("Index", "Peliculas", FormMethod.Get))
13  {
14      <p> Titulo: @Html.TextBox("BuscarString") <br />
15      <input type="submit" value="Filtrar" /> </p>
16  }
17  </p>
18
```

Ahora, cuando envía una búsqueda, la URL contiene una cadena de consulta de búsqueda. La búsqueda también irá al método de acción HttpGet Index, incluso si tiene un método HttpPost Index.



localhost:51983/Peliculas?BuscarString=John

MVC Pelicula Inicio Acerca de Contacto

Index

[Crear Nuevo](#)

Titulo: John

Filtrar

Titulo	Fecha de Lanzamiento	Genero	Precio	
John Wick	2014-10-24	Accion	6.99	Editar Consultar Eliminar
John Wick II	2015-10-26	Accion	6.99	Editar Consultar Eliminar
John Wick II	2019-05-14	Accion	6.99	Editar Consultar Eliminar

© 2019 - Mi aplicación ASP.NET

Agregar búsqueda por género

Si agregó la versión HttpPost del método Index, elimínelo ahora.

A continuación, agregará una función para permitir a los usuarios buscar películas por género. Reemplace el método Index con el siguiente código:

```
public ActionResult Index(string buscarString, string generoPelicula)
{
    var GeneroLst = new List<string>();

    var GeneroQry = from d in db.Peliculas
                    orderby d.Genero
                    select d.Genero;

    GeneroLst.AddRange(GeneroQry.Distinct());
    ViewBag.generoPelicula = new SelectList(GeneroLst);

    var peliculas = from p in db.Peliculas
                    select p;

    if (!String.IsNullOrEmpty(buscarString))
    {
        peliculas = peliculas.Where(s => s.Titulo.Contains(buscarString));
    }

    if (!string.IsNullOrEmpty(generoPelicula))
    {
        peliculas = peliculas.Where(x => x.Genero == generoPelicula);
    }

    return View(peliculas);
}
```

Esta versión del método Index toma un parámetro adicional, llamado generoPelicula. Las primeras líneas de código crean un objeto List para contener géneros de películas de la base de datos.

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

```
var GeneroQry = from d in db.Peliculas
                orderby d.Genero
                select d.Genero;
```

El código usa el método AddRange de la colección genérica List para agregar todos los géneros distintos a la lista. (Sin el modificador Distinct, se agregarían géneros duplicados; por ejemplo, la acción se agregaría tres veces en nuestra muestra). El código luego almacena la lista de géneros en el objeto ViewBag.generoPelicula. Almacenar datos de categoría (como géneros de películas) como un objeto SelectList en un ViewBag, luego acceder a los datos de categoría en un cuadro de lista desplegable el cual es un enfoque típico para las aplicaciones MVC.

El siguiente código muestra cómo verificar el parámetro generoPelicula. Si no está vacío, el código restringe aún más la consulta de películas para limitar las películas seleccionadas al género especificado.

```
if (!string.IsNullOrEmpty(generoPelicula))
{
```

```

        películas = películas.Where(x => x.Genero == generoPelícula);
    }

```

Como se indicó anteriormente, la consulta no se ejecuta en la base de datos hasta que la lista de películas se repite (lo que ocurre en la Vista, después de que el método de acción Index regrese).

Agregar código a la vista Index para admitir la búsqueda por género.

Agregue un asistente Html.DropDownList al archivo Views\Películas\Index.cshtml, justo antes del asistente TextBox. El código completo se muestra a continuación:

```

@model IEnumerable<MvcPelícula.Models.Película>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Crear Nuevo", "Create")

    @using (Html.BeginForm("Index", "Películas", FormMethod.Get))
    {
        <p>
            Genre: @Html.DropDownList("generoPelícula", "All")
            Título: @Html.TextBox("BuscarString") <br />
            <input type="submit" value="Filtrar" />
        </p>
    }

```

En el siguiente código:

```

@Html.DropDownList("generoPelícula", "All")

```

El parámetro "generoPelícula" proporciona la clave para que el helper DropDownList encuentre un IEnumerable<SelectListItem> en el ViewBag. El ViewBag se llenó en el método de acción:

```

public ActionResult Index(string buscarString, string generoPelícula)
{
    var GeneroLst = new List<string>();

    var GeneroQry = from d in db.Películas
                    orderby d.Genero
                    select d.Genero;

    GeneroLst.AddRange(GeneroQry.Distinct());
    ViewBag.generoPelícula = new SelectList(GeneroLst);

    var películas = from p in db.Películas
                    select p;

    if (!String.IsNullOrEmpty(buscarString))

```



```

        {
            peliculas = peliculas.Where(s => s.Titulo.Contains(buscarString));
        }

        if (!string.IsNullOrEmpty(generoPelicula))
        {
            peliculas = peliculas.Where(x => x.Genero == generoPelicula);
        }

        return View(peliculas);
    }

```

El parámetro "Todos" proporciona una etiqueta de opción. Si inspecciona esa opción en su navegador, verá que su atributo "valor" está vacío. Dado que nuestro controlador solo filtra "if" la cadena no está null o vacía, al enviar un valor vacío para generoPelicula muestra todos los géneros.

También puede establecer una opción para que se seleccione de forma predeterminada. Si quisieras "Acción" como tu opción predeterminada, cambiarías el código en el Controlador de la siguiente manera:

```

public ActionResult Index(string buscarString, string generoPelicula)
{
    var GeneroLst = new List<string>();

    var GeneroQry = from d in db.Peliculas
                    orderby d.Genero
                    select d.Genero;

    GeneroLst.AddRange(GeneroQry.Distinct());
    ViewBag.generoPelicula = new SelectList(GeneroLst, "accion");

    var peliculas = from p in db.Peliculas
                    select p;

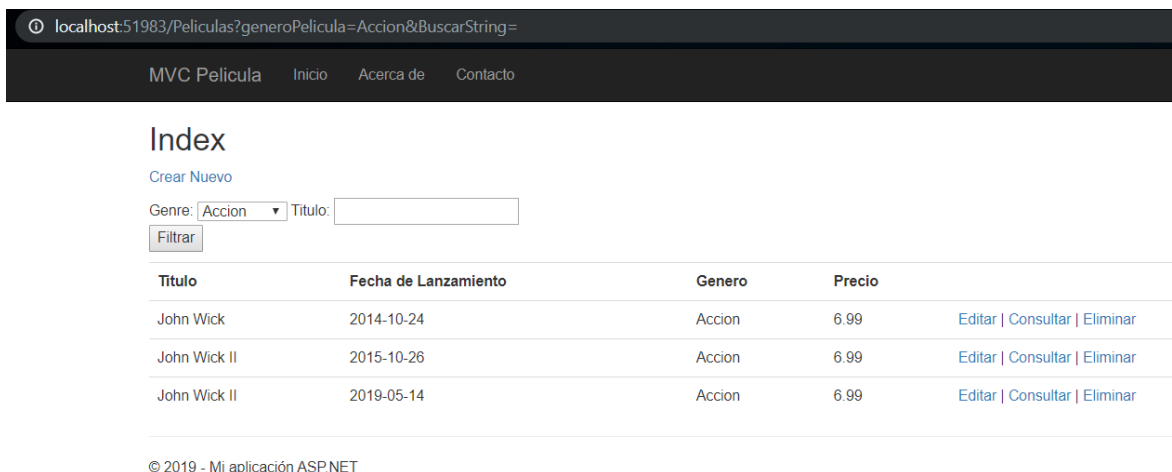
    if (!String.IsNullOrEmpty(buscarString))
    {
        peliculas = peliculas.Where(s => s.Titulo.Contains(buscarString));
    }

    if (!string.IsNullOrEmpty(generoPelicula))
    {
        peliculas = peliculas.Where(x => x.Genero == generoPelicula);
    }

    return View(peliculas);
}

```

Ejecute la aplicación y busque /Peliculas/Index. Intente una búsqueda por género, por nombre de película y por ambos criterios.



En esta sección, creamos un método de acción de búsqueda y una vista que permite a los usuarios buscar por título y género de película. En la siguiente sección, veremos cómo agregar una propiedad al modelo Pelicula y cómo agregar un inicializador que creará automáticamente una base de datos de prueba.

VII. Ejercicios

1. Agregue una nueva condición de búsqueda para el proyecto de Pelicula que permita buscar las películas en base a su precio.
2. En base al ejercicio 2 de la guía de laboratorio anterior, cree la cadena de conexión del proyecto Persona, cree el controlador PersonasController el cual será el encargado de gestionar el modelo de Persona y agregue los métodos de búsqueda por atributo DUI y Nombre, tenga en cuenta que el desarrollo de este ejercicio será necesario para realizar los ejercicios de guía posterior.

VIII. Referencias

- Rick-Anderson, R. A. (2018, 4 octubre). Getting Started with ASP.NET MVC 5. Recuperado 2 noviembre, 2019, de <https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/introduction/getting-started>
- Archiveddocs, A. R. (2015, 10 junio). Utilizar IntelliSense. Recuperado 2 noviembre, 2019, de [https://docs.microsoft.com/es-es/previous-versions/visualstudio/visual-studio-2013/hcw1s69b\(v=vs.120\)?redirectedfrom=MSDN](https://docs.microsoft.com/es-es/previous-versions/visualstudio/visual-studio-2013/hcw1s69b(v=vs.120)?redirectedfrom=MSDN)