



INTRODUCCIÓN A ASP.NET MVC 4

Instituto Tecnológico de Zacatecas



22 DE JUNIO DE 2013
ISC RICARDO VASQUEZ SIERRA

Contacto	3
Software Necesario	3
Instalación de Visual Studio 2012	3
Estructura de Archivos y Directorios Dentro de una Aplicación ASP.Net MVC 4.0 (Visual Studio 2012)	5
Controladores.....	6
Vistas	7
Vistas Parciales.....	8
Modelos y algo de Bases de Datos.....	10
Scaffolding de Datos.....	11
Sincronizacion Automatica de Base de Datos	13
Pre llenado de la Base de Datos.....	13
Respondiendo a Mensajes POST	15
Forms y Html Helpers.....	17
Forms.....	17
Html Helpers	19
Html.BeginForm	19
Html.ValidationSummary	20
Html.TextBox y Html.TextArea.....	20
Html.Label	20
Html.DropDownList y Html.ListBox.....	20
Html.ValidationMessage	21
Html.Hidden	22
Html.Password	22
Helpers Inflexibles de Tipo (Strongly Typed helpers).....	22
Ayudantes con Plantilla (Templated Helpers).....	23
Anotaciones de Datos Y Validación (Data Annotations and Validation).....	23
Required	25
Longitud de Cadena (StringLength).....	26
Expresiones Regulares.....	26
Display	26
Formato de Datos.....	27
Rangos	27

Validaciones Personalizadas	29
SEGURIDAD Y SESIONES	31
AJAX.....	44
JQUERY (Escribe menos, has más)	44
Características de jQuery	44
jQuery Function.....	44
jQuery Selectors	46
Ejemplo de Ajax haciendo uso de jQuery.....	46
Subiendo Archivos al Servidor.....	56
Envío de E-mail a través de smtp.live.com.....	59

Contacto

El material que tienes a tu disposición fue elaborado por el ISC Ricardo Vásquez Sierra del Instituto Tecnológico de Zacatecas. Puedes contactarlo en la siguiente dirección para dudas, sugerencias y corrección de errores que pudieras hallar en el mismo.

ricardopanchitometalujus@gmail.com

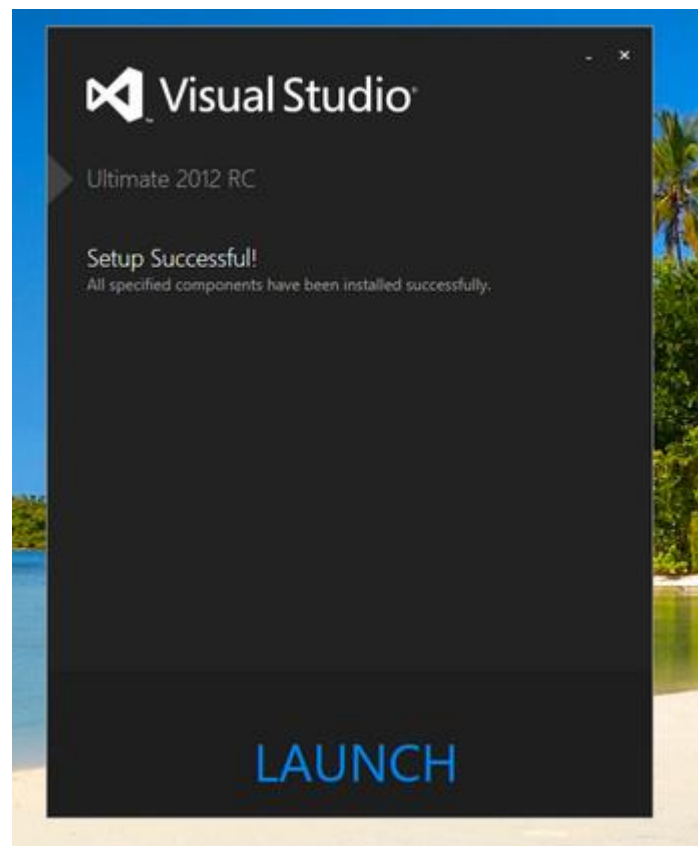
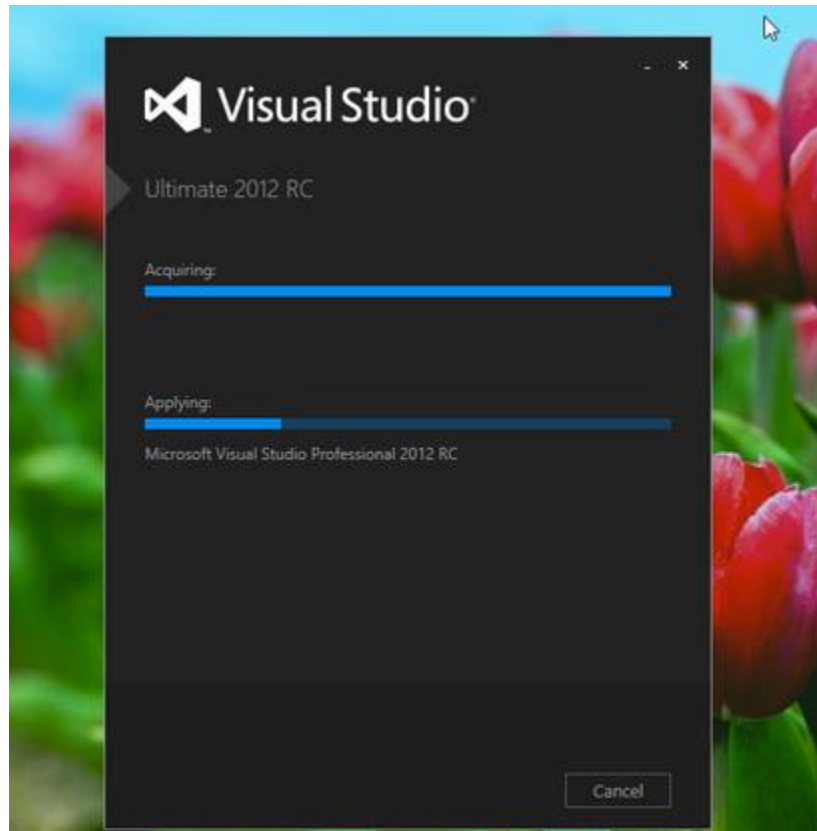
Software Necesario

Todos los ejemplos fueron realizados con la versión 2012 de Visual Studio en una máquina con Windows 8 Pro (x64), pero puede que utilices también Windows 7 y Visual Studio 2010 o 2012.

Instalación de Visual Studio 2012



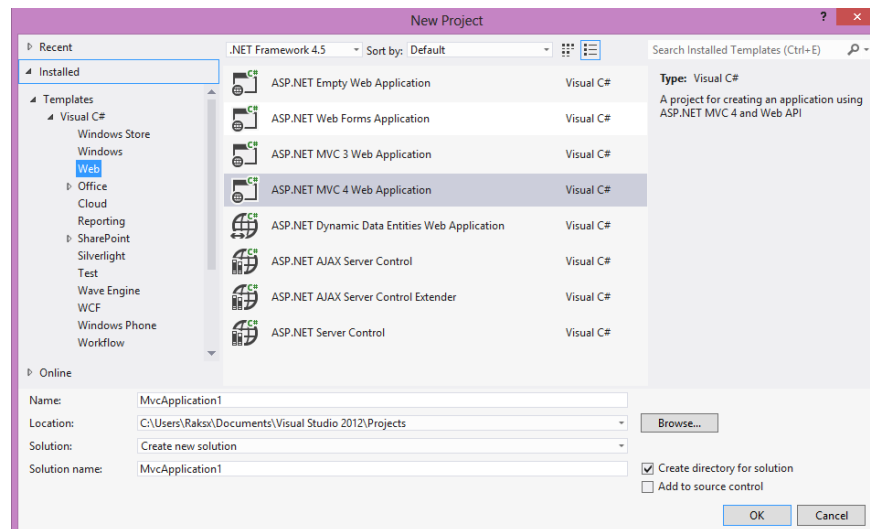
Una vez que tengamos montado el ISO en nuestra máquina ya sea la versión de 32 bits o de 64 bits de Visual Studio, lo único que debemos hacer será lanzar el autorun del mismo, y dar siguiente a todos los pasos hasta finalizar la instalación.



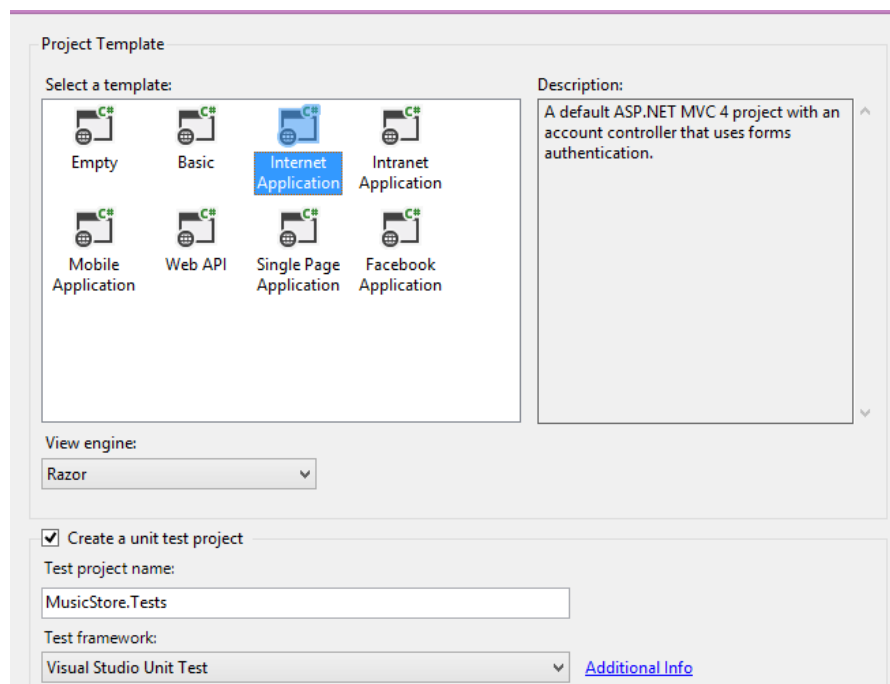
Estructura de Archivos y Directorios Dentro de una Aplicación ASP.Net MVC 4.0 (Visual Studio 2012)

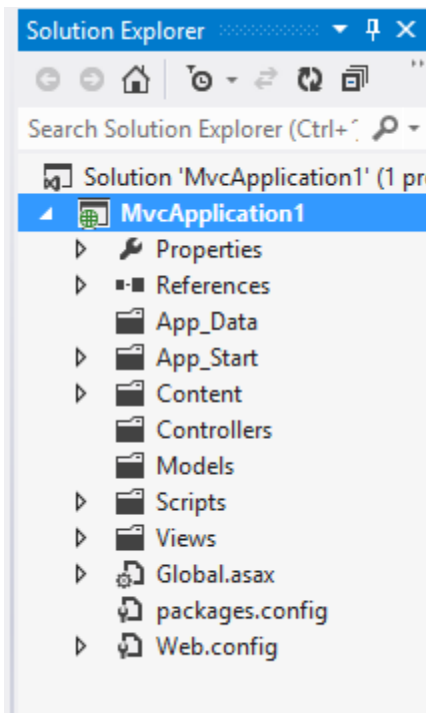
Una vez que hemos instalado todo el software necesario y habiendo creado la base de datos, estamos listos para crear nuestra primera aplicación.

Una vez dentro de Visual Studio crearemos una nueva aplicación con la plantilla de MVC 4.0



Después de haber asignado un nombre y una ubicación para nuestro proyecto veremos desplegada la siguiente pantalla:





Dejaremos seleccionada la plantilla Internet Application y proseguiremos. Una vez hecho esto se creará la plantilla de nuestro proyecto. En el panel del lado derecho podemos ver una serie de carpetas que Visual Studio ha creado automáticamente para nosotros.

Carpeta	Propósito
/Controllers	Los controladores responden a las requests del usuario, deciden qué hacer con dicha request y responden a las mismas.
/Views	Las vistas son el GUI de la aplicación
/Models	Los modelos guardan y manipulan datos
/Content	Aquí se almacena todo el contenido de la aplicación como imágenes, hojas de estilo, etc.
/Scripts	Esta carpeta guarda scripts de JavaScript

Controladores

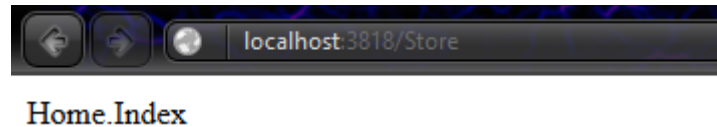
Botón derecho sobre carpeta Controllers, y clic en Add—>New Controller. Llamaremos a este controlador **StoreController**, una vez hecho esto modificaremos el método Index a lo siguiente:

```
public string Index()
{
    return "Home.Index";
}
```

Hecho esto compilaremos el proyecto y agregaremos a la barra de navegación lo siguiente:

```
*/Store
```

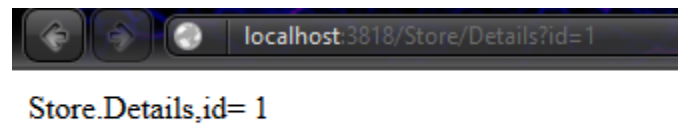
Automáticamente el framework mapeará la dirección hacia el controlador de Store, a la acción Index



Al controlador anterior agregaremos otro método:

```
public string Details(int id)
{
    return HttpUtility.HtmlEncode("Store.Details,id= " + id);
}
```

Compilamos nuevamente y agregamos a la barra de direcciones /Store/Details?id=1



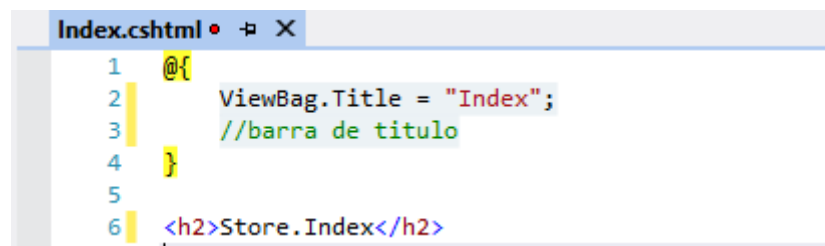
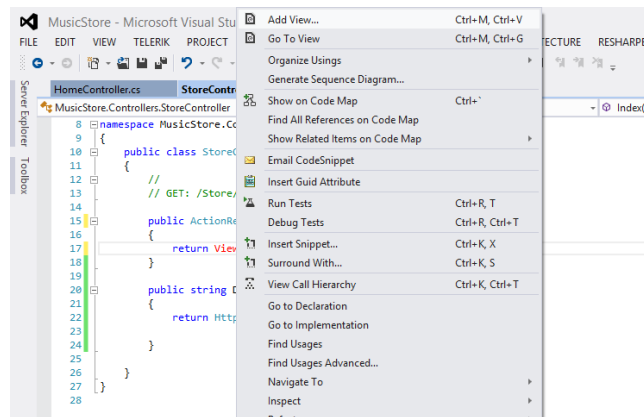
HttpUtility.HtmlEncode se utiliza para protección contra la inyección SQL

Vistas

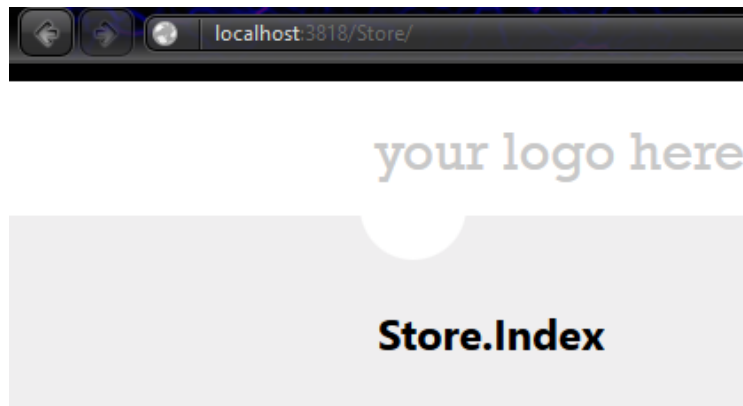
El ejemplo anterior son solo controladores a los que estamos llamando manualmente, pero esta no es la manera en la que están hechas las aplicaciones en el mundo real. Para desplegar datos verdaderos al usuario lo hacemos a través de GUIs, o en este caso a través de **Views**.

Podemos agregar Views de manera igual a como agregamos los controladores en el capítulo anterior, solo haciéndolo en la carpeta de Views.

Otra manera de hacerlo es dando clic derecho en algún método de un controlador de tipo **ActionResult** y agregándolo con **AddView**.



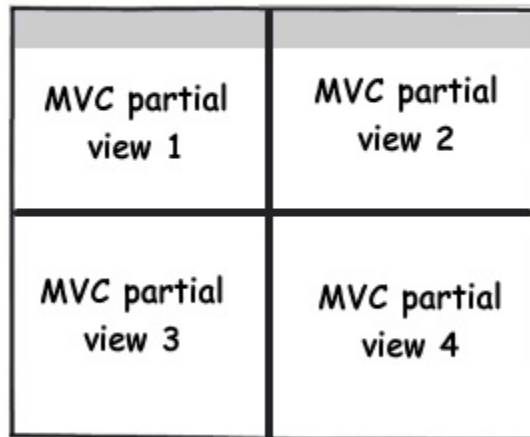
Al compilarlo:



Vistas Parciales

Las vistas parciales son similares a los controles de usuario en ASP.Net Web Forms, es decir, código que puede ser reutilizado dentro de las Vistas.

MVC View



En el controlador de Store agregaremos lo siguiente:

```
public ActionResult Message()
{
    return PartialView();
}
```

Agregaremos la vista de la acción del controlador especificando esta vez que será una **vista parcial**:

A diferencia de las otras vistas, esta aparecerá completamente vacía. Ahora proseguiremos a agregar lo siguiente:

```
<div>Vista Parcial @ViewBag.Message</div>
```

Por ultimo, agregaremos lo siguiente en Index de la Vista Home:

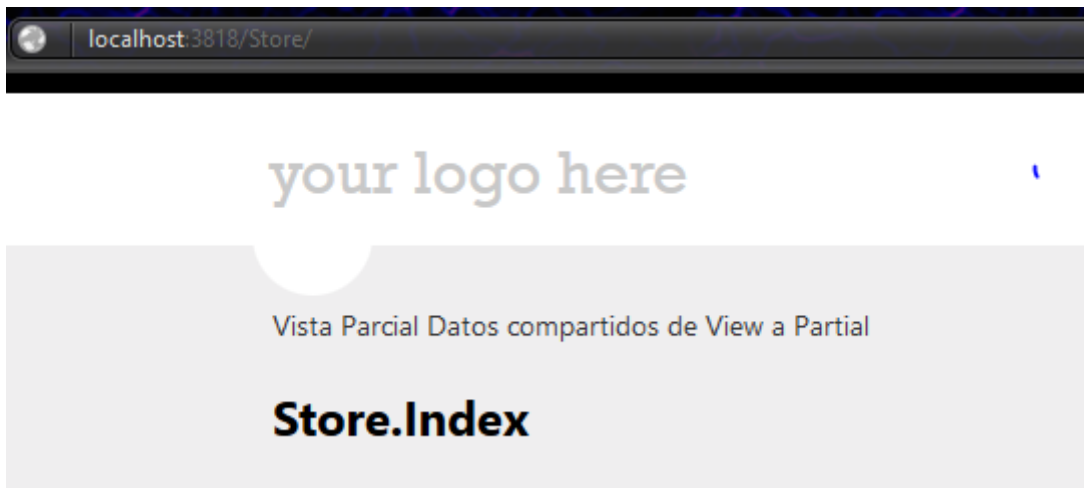
```

1  @{|
2      ViewBag.Title = "Index";
3      ViewBag.Message = "Datos compartidos de View a Partial";
4
5  }|
6  <div>@Html.Partial("Message")</div>
7  <h2>Store.Index</h2>

```

Lo anterior hará lo siguiente: Al mandar llamar /Store lo que se hará, será mandar llamar la vista parcial , pasándole como parámetro "Datos comprar...". Una vez que la Vista Parcial recibe este string lo concatena con el contenido de la misma ("Vista Parcial"), y al final se muestra dentro de la Vista que hizo la llamada.

Al compilarlo veremos lo siguiente:



Modelos y algo de Bases de Datos

Aquí nos referiremos a los modelos que podemos usar para enviar y obtener información de la base de datos, además de usarlos como contenido directo para nuestras vistas.

A nuestro proyecto existente agregaremos 3 clases (extensión .cs) a la carpeta de modelos:

```

public class Album
{
    public virtual int    AlbumId    { get; set; }
    public virtual int    GenreId    { get; set; }
    public virtual int    ArtistId   { get; set; }
    public virtual string Title      { get; set; }
    public virtual decimal Price     { get; set; }
    public virtual string AlbumArtUrl { get; set; }
    public virtual Genre  Genre      { get; set; }
    public virtual Artist Artist     { get; set; }
}

public class Artist
{
    public virtual int    ArtistId { get; set; }
    public virtual string Name     { get; set; }
}

public class Genre
{
    public virtual int    GenreId    { get; set; }
    public virtual string Name       { get; set; }
    public virtual string Description { get; set; }
    public virtual List<Album> Albums { get; set; }
}

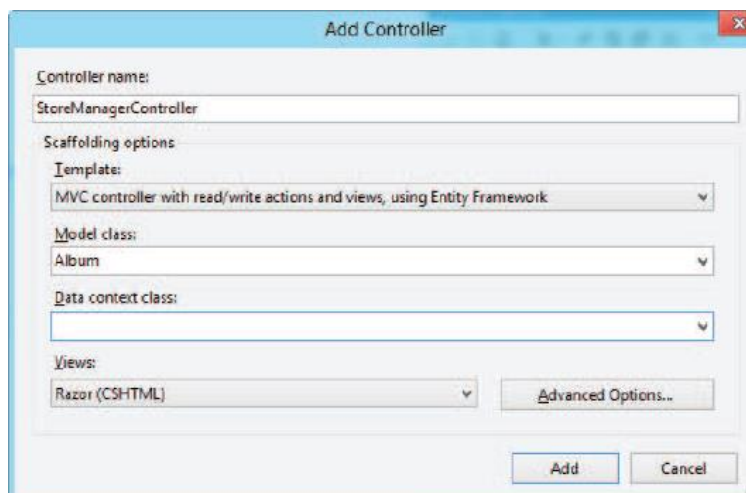
```

La manera como es tana la información aquí es orientada objetos, por lo mismo no nos preocupa ntanto términos como llaves foráneas, propios del modelo relacional.

Scaffolding de Datos

Una vez que se tienen los modelos, generaremos los métodos necesarios para leer, escribir, modificar y eliminar elementos a través del Scaffolding. Este mecanismo es capaz de generar también los controladores y vistas asociadas a los modelos en cuestión.

Lo siguiente seria agregar un controlador que nos permita editar la información de los modelos ya mencionados, asi que agregaremos un controlador llamado StoreManagerController a partir de un modelo.



En DataContext class seleccionaremos new context, y en la caja de dialogo escribiremos MusicStoreDBContext. Esto generara todo el código necesario de la clase Album y sus derivados.

El código generado por lo anterior seria el siguiente:

```
public MusicStoreDBContext() : base("name=MusicStoreDBContext")
{
}

public DbSet<Album> Albums { get; set; }

public DbSet<Genre> Genres { get; set; }

public DbSet<Artist> Artists { get; set; }
```

Esto anterior es nuestra base de datos, que solo necesita ser instanciada para hacer uso de la misma.

EL código que se genero en el **StoreManager (Solo una parte)**:

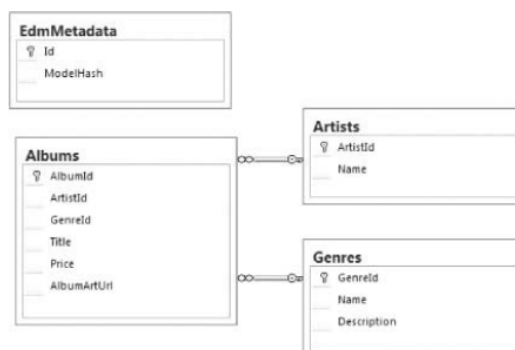
```
namespace MusicStore.Controllers
{
    public class StoreManagerController : Controller
    {
        private MusicStoreDBContext db = new MusicStoreDBContext();

        //
        // GET: /StoreManager/

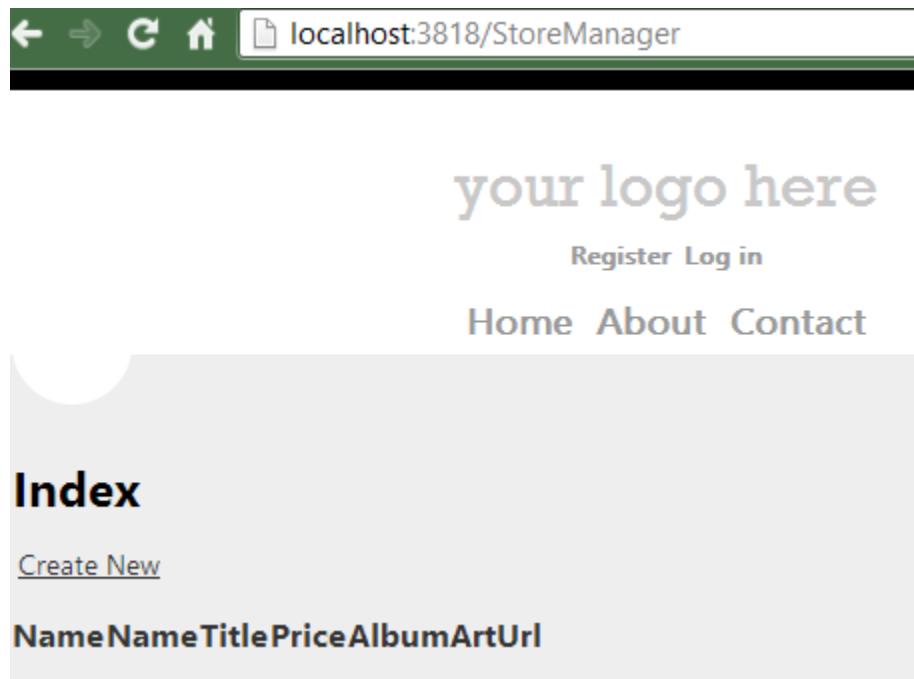
        public ActionResult Index()
        {
            var albums = db.Albums.Include(a => a.Genre).Include(a => a
            return View(albums.ToList());
        }

        //
        // GET: /StoreManager/Details/5
```

EL código que el Framework ha generado para la Base de Datos seria el siguiente:



Al compilar la solución tendrías lo siguiente:



Sincronizacion Automatica de Base de Datos

Para que el framework se encargue de mantener actualizada la DB al menor cambio tenemos que hacer unos cambios al archivo **global.asax**.

```
protected void Application_Start()
{
    Database.SetInitializer(new DropCreateDatabaseAlways<MusicStore.Models.MusicStoreDbContext>());
    AreaRegistration.RegisterAllAreas();

    WebApiConfig.Register(GlobalConfiguration.Configuration);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
    AuthConfig.RegisterAuth();
}
```

Pre llenado de la Base de Datos

Como su nombre lo indica, también podemos realizar un llenado de la DB desde su creación, es decir, desde el inicio de la aplicación. Para esto agregaremos una clase a Models llamada **MusicStoreDbInitializer**

```

namespace MusicStore.Models
{
    public class MusicStoreDbInitializer
        : DropCreateDatabaseAlways<MusicStoreDBContext>
    {
        protected override void Seed(MusicStoreDBContext context)
        {
            context.Artists.Add(new Artist {Name = "Al Di Meola"});
            context.Genres.Add(new Genre {Name = "Jazz"});
            context.Albums.Add(new Album
            {
                Artist = new Artist {Name = "Rush"},
                Genre = new Genre {Name = "Rock"},
                Price = 9.99m,
                Title = "Caravan"
            });
            base.Seed(context);
        }
    }
}

```

Como se mencionó, se ejecutara desde el inicio de la aplicación, por tanto debemos llamarlo desde **global.asax**.

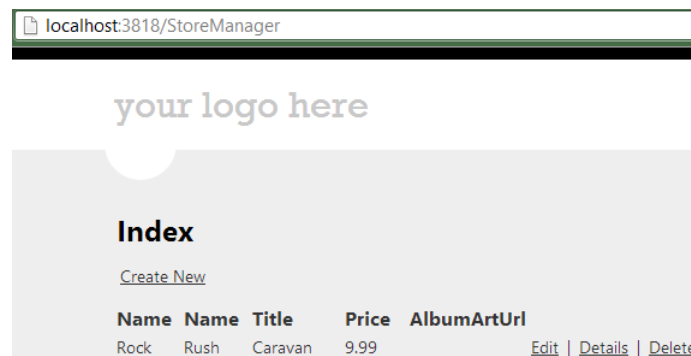
```

protected void Application_Start()
{
    Database.SetInitializer(new DropCreateDatabaseAlways<MusicStore.Models.MusicStoreDBContext>());
    Database.SetInitializer(new MusicStoreDbInitializer());
    AreaRegistration.RegisterAllAreas();

    WebApiConfig.Register(GlobalConfiguration.Configuration);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
    AuthConfig.RegisterAuth();
}

```

Al compilar nuevamente nuestra solución:



Respondiendo a Mensajes POST

Hasta el momento solo hemos trabajado con mensajes de tipo GET (si es que no te habías puesto a pensarlo), es decir, solo hemos estado haciendo consultas a través del navegador al escribir el nombre de las vistas que nos interesan. En el ejemplo anterior puedes darte cuenta que a vista de **StoreManager** nos da la opción de editar, ver detalles y eliminar algún registro de nuestra Base de datos.

Por lógica entonces, te daras cuenta de que POST es el contrario a GET, y los escenarios mas comunes para estos mensajes es en los formularios Web, ya que una vez llenados todos los campos y validados por el framework, son enviados de vuelta al controlador correspondiente para verificarse y ser tratados.

En el ejemplo anterior el controlador **StoreManager.Edit** es el encargado de esto.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(Album album)
{
    //Verificar Si cada propiedad del objeto referente a la vista ha sido llenado y
    //tiene el tipo de valor especificado en el modelo
    if (ModelState.IsValid)
    {
        //Verifica si el objeto actual del View ha cambiado
        //Framework busca por si mismo el valor en la DB y lo modifica
        db.Entry(album).State = EntityState.Modified;
        db.SaveChanges();

        return RedirectToAction("Index");
    }

    //Estos datos se enviaran de vuelta a la vista, para decirle al usuario que campos debe corregir
    //o mejor dicho, porque no ha posido llevarse con exito la modificación
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name", album.ArtistId);
    return View(album);
}
```

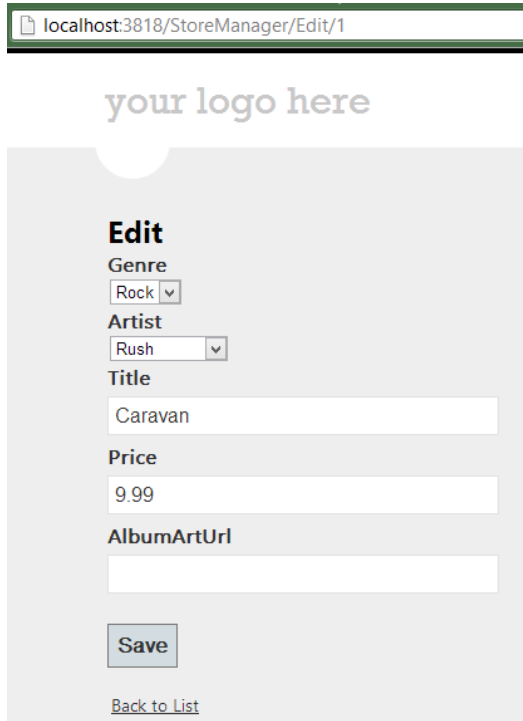
Podemos observar el atributo **[HttpPost]** sobre el método. Si intentas llamar este método como lo hemos visto anteriormente desde el navegador te daría error por lo que ya se mencionó.

Lo que hace específicamente este método es verificar si el **ModelState** es válido antes de procesar los cambios. Que quiere decir?, pues que todos los campos necesarios fueron llenados y validados correctamente. Solo si se cumple dicha condición se procederá a guardar los cambios.

Arriba de este método puedes darte cuenta que se encuentra otro método **Edit** pero en su caso, recibe como parámetro un entero.


```
// GET: /StoreManager/Edit/5

public ActionResult Edit(int id = 0)
{
    Album album = db.Albums.Find(id);
    if (album == null)
    {
        return HttpNotFound();
    }
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name", album.ArtistId);
    return View(album);
}
```



localhost:3818/StoreManager/Edit/1

your logo here

Edit

Genre
Rock

Artist
Rush

Title
Caravan

Price
9.99

AlbumArtUrl

Save

[Back to List](#)

Este método si puede ser accesado desde el navegador: /StoreManager/Edit/1.

Al ejecutar lo anterior podemos ver que nos muestra el álbum con ID=1 almacenado en nuestra base de datos.

Si modificamos algún valor y guardamos los cambios, se dispararía el mensaje POST del cual ya hablamos arriba, en este caso, sería manejado por la primera acción Edit de la que se habló para posteriormente regresarnos a la vista **StoreManager.Index**.

Forms y Html Helpers

Forms

Las formas son la parte más pesada de toda aplicación, y estas mismas es donde es necesario un uso extensivo de los Html helpers. Muchos podrán decir que no son necesarias, pero sin su utilización, la mayoría de las páginas web del mundo serian simples hojas con texto plano.

Las formas con contenedores para cajas de texto, botones , etc etc. Y son estos los que hacen posible que un usuario pueda hacer Login en cualquier sitio, o dicho de otro modo, enviar información hacia el servidor, pero ¿a cual servidor?, y ¿Cómo es que se envía esa información?.

Acción y Método

El atributo **Action** le dice al explorador adonde mandar la info.

```
<form action=http://www.google.com/search>  
    </input name="ValorIntro" type="text" method="get"/>  
    <input type="submit" value="Buscar"/>  
</form>
```

En este ejemplo se enviaria un mensaje de tipo Http Get (lectura de informacion) a la página de busque da de google: [http://www.google.com/search?ValorIntro= google](http://www.google.com/search?ValorIntro=google).

Nota. Los mensajes de tipo GET se utilizan para leer datos solamente, mientras que los mensajes de tipo POST se utilizan cuando se desea realizar algún cambio en la información del lado del servidor como en nuestra vista de Edit.

```

6  @section featured {
7      <section class="featured">
8          <div class="content-wrapper">
9              <form action="/Home/Search" method="get">
10                 <input type="text" name="q" />
11                 <input type="submit" value="Search" />
12             </form>
13
14             <hgroup class="title">
15                 <h1>@ViewBag.Title.</h1>
16                 <h2>@ViewBag.Message</h2>
17             </hgroup>
18

```

Añadiendo Form de Búsqueda

En una tienda musical es necesario que el usuario pueda música directamente desde el Home de la aplicación. Justo como el ejemplo de búsqueda anterior vamos a necesitar una forma con

una Acción y un Método. Modificaremos la **vista** del controlador **HomeController**.

Igualmente necesitaremos agregar el Método search en el **HomeController**:

```

34  public ActionResult Search(string q)
35  {
36      MusicStoreDBContext db = new MusicStoreDBContext();
37      var albums = db.Albums.Where (artista => artista.Artist.Name.Contains(q));
38      return View(albums);

```

Aquí podrás darte cuenta que el método es de tipo GET, pues esta consultando cierta info de la DB. También puedes apreciar que el parámetro que recibe (q), es el mismo que la Acción del Form en el View.

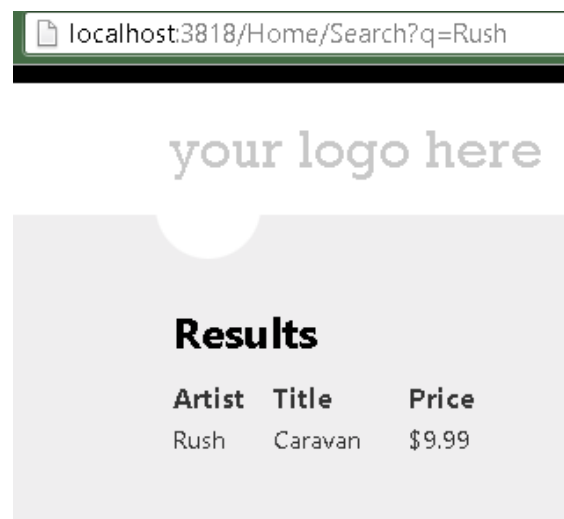
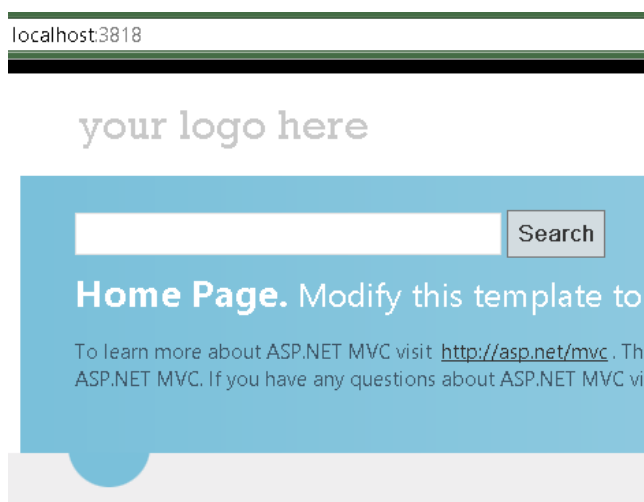
Seguido de esto, ahora tenemos que agregar una Vista para el controlador de Search, que se encargara de mostrarnos los resultados.

```

1  @model IEnumerable<MusicStore.Models.Album>
2  @{ ViewBag.Title = "Search"; }
3  <h2>Results</h2>
4  <table>
5      <tr>
6          <th>Artist</th>
7          <th>Title</th>
8          <th>Price</th>
9      </tr>
10     @foreach (var item in Model)
11     {
12         <tr>
13             <td>@item.Artist.Name</td>
14             <td>@item.Title</td>
15             <td>@String.Format("{0:c}", item.Price)</td>
16         </tr>
17     }
18 </table>

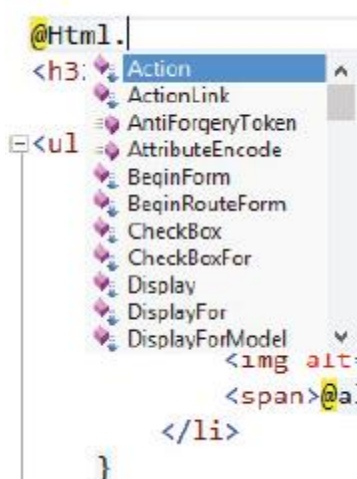
```

Finalmente al compilarlo tendríamos lo siguiente:



Html Helpers

Son métodos que pueden ser invocados desde el Html de las vistas, contenidos en el namespace System.Web.Mvc.Html.



Html.BeginForm

Envía un mensaje de tipo POST a la acción. Ejemplo:

```
<form action="/StoreManager/Edit/52" method="post">
```

Html.ValidationSummary

Despliega una lista desordenada de todos los errores de validación en el diccionario de **ModelState**. Comúnmente estos se agregan en los controladores. Ejemplo:

```
ModelState.AddModelError("Price", "Formato equivocado");
```

Explicado esto, podemos cambiar el código de la vista de Edit:

```

9  @using (Html.BeginForm()) {
10     @Html.AntiForgeryToken()
11     @Html.ValidationSummary(true)
12
13     <fieldset>
14         <legend>Album</legend>
15
16         <p>
17             @Html.Label("GenreId")
18             @Html.DropDownList("GenreId", ViewBag.Genres as SelectList)
19         </p>
20         Html Helpers C:
21         <p>
22             @Html.Label("Title")
23             @Html.TextBox("Title", Model.Title)
24             @Html.ValidationMessage("Title")
25         </p>
26         <input type="submit" value="Guardar" />
27     }

```

Html.TextBox y Html.TextArea

El helper de la caja de texto, renderiza una etiqueta de tipo input y atributos de tipo texto.

```
@Html.TextBox("Artist", Model.Artist)
```

TextArea en cambio se utiliza para renderizar una caja de texto multilinea:

```
@Html.TextArea("Texto", "hola mundo y <br/> todos quienes lo habitan")
```

Html.Label

Una simple etiqueta que renderiza alguna propiedad enviada a la vista.

```
@Html.LabelFor(artista=>artista.Nombre)
```

Html.DropDownList y Html.ListBox

Ambos contienen un elemento `<select />`. DropDownLists permiten selección simple de algún elemento, mientras que un ListBox permite selección simultanea de varios elementos de quererse así.

Ambas se usan generalmente para mostrar una colección de objetos.

Para mandar una colección desde el controlador se hace de la siguiente manera:

```
ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
```

Los parámetros del constructor son:

- *Colección de Origen
- *Nombre de la propiedad a usar como valor
- *Nombre de la propiedad a usar como texto
- *Valor del elemento seleccionado

Html.ValidationMessage

Cuando hay un error en algún campo en el diccionario de ModelState, puedes usar un helper ValidationMessage para mostrar un error.

Por ejemplo, en un controlador podríamos tener lo siguiente:

```
[HttpPost]
public ActionResult Edit(int id,)
{
    var album = storeDB.Albums.Find(id);
    ModelState.AddModelError("Price", "Valor no aceptado");
    return View(album);
}
```

Y en la vista podemos cazar el error por medio de:

```
@Html.ValidationMessage("Price")
```

Otra manera de hacerlo, sería llamando desde la vista la validación:

```
@Html.ValidationMessage("Price","Solo se aceptan números")
```

Price

Solo se aceptan números

Html.Hidden

Renderiza un campo oculto. Como por ejemplo los Id de los modelos

```
@Html.HiddenFor(model=>model.Id)
```

Html.Password

Una caja de texto para colocar alguna contraseña. La sintaxis sería la siguiente:

```
@Html.PasswordFor(model=>model.Pwd)
```

Helpers Inflexibles de Tipo (Strongly Typed helpers)

Anteriormente los valores se estuvieron pasando ya fuera a través de ViewBag.Propiedad, y utilizándolos escribiendo su nombre en la Vista. Si no te parece un enfoque muy práctico también existen los ayudantes inflexibles de tipo, los cuales usan expresiones lambda:

```
@using (Html.BeginForm())
{
    @Html.ValidationSummary(excludePropertyErrors: true)
    <fieldset>
        <legend>Edit Album</legend>
        <p>
            @Html.LabelFor(m => m.GenreId)
            @Html.DropDownListFor(m => m.GenreId, ViewBag.Genres as SelectList)
        </p>
        <p>
            @Html.TextBoxFor(m => m.Title)
            @Html.ValidationMessageFor(m => m.Title)
        </p>
        <input type="submit" value="Save" />
    </fieldset>
}
```

El código anterior será válido siempre y cuando, nuestra vista reciba algún modelo en específico:

```
1 @model MusicStore.Models.Album
2
```

A diferencia de los ayudantes tradicionales, estos ofrecen beneficios como revisión paso por paso del código, fácil refactorización, así como el uso de Intellisense.

Ayudantes con Plantilla (Templated Helpers)

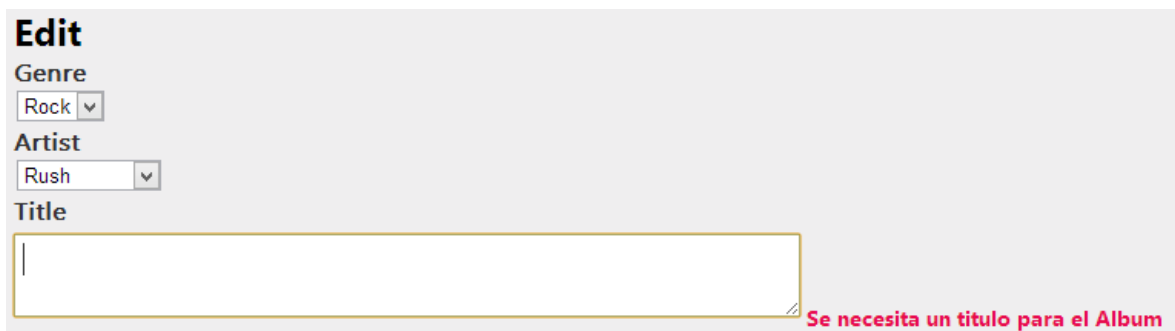
Construyen HTML utilizando una plantilla. En los helpers anteriores hemos visto que para cada tipo de dato existe un Helper, por ejemplo si queremos una caja de texto (TextBoxFor) o una etiqueta que muestre alguna cadena (LabelFor). En este caso se puede usar algo “genérico”, y dejar que el compilador asigne los tipos a los helpers de tipo `@Html.Editor`, acorde a los valores que provengan de un determinado valor o modelo. Si analizaste el código que genero el framework al crear la base de datos, recordaras que la vista de Edit contenía algo como lo siguiente:

```
@Html.EditorFor(m => m.Title)
```

Este Helper va a renderizar el mismo contenido que un `TextBoxFor`. Cuando se utilizan este tipo de ayudantes, lo que le estamos pidiendo al ambiente es que genere lo que el editor crea que pueda encajar. Agregaremos una Anotación a la apropiada propiedad `Title` de la clase `Album`:

```
[Required(ErrorMessage = "Se necesita un titulo para el Album")]
[StringLength(160)]
[DataType(DataType.MultilineText)]
public virtual string Title { get; set; }
```

En este caso lo que sucedería es que el `EditorFor` determinaría que el mejor control para renderizar el contenido de `Album.Title` sería un `TextArea`.



The screenshot shows a web form titled "Edit". It contains three fields: "Genre" with a dropdown menu showing "Rock", "Artist" with a dropdown menu showing "Rush", and "Title" which is a large text area. Below the text area, there is a red error message that reads "Se necesita un titulo para el Album".

Anotaciones de Datos Y Validación (Data Annotations and Validation)

En el último ejemplo agregamos unos atributos a las propiedades de nuestro modelo. En esta parte ahondaremos en dichas propiedades, y veremos como pueden facilitarnos tareas arduas y tediosas como lo es la validación.

Anteriormente también se vio que la validación se puede llevar a cabo a través del helper `@Html.ValidationMessage`. En esta parte veremos como manejar dichas validaciones de manera declarativa.

Agregaremos un Modelo llamado orden, el cual contendrá lo siguiente:

```
public class Order
{
    [ReadOnly(true)]
    public int OrderId { get; set; }
    public DateTime OrderDate { get; set; }
    public string Username { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
    public string Phone { get; set; }
    public string Email { get; set; }
    public decimal Total { get; set; }
}
```

Agregaremos un Controlador llamado Order, al cual no agregaremos nada. Así mismo agregaremos una vista fuertemente tipeada del modelo de orden:

The screenshot shows the 'Add View' dialog box with the following configuration:

- View name: Order
- View engine: Razor (CSHTML)
- ☒ Create a strongly-typed view
- Model class: Order (MusicStore.Models)
- Scaffold template: Empty
- ☒ Reference script libraries
- ☐ Create as a partial view
- ☒ Use a layout or master page:
- ContentPlaceholder ID: MainContent

The 'Add' button is highlighted at the bottom right.

La vista contendrá lo siguiente:

```

1  @model MusicStore.Models.Order
2
3  @{
4      ViewBag.Title = "Index";
5  }
6  @using (Html.BeginForm())
7  {
8      @Html.AntiForgeryToken()
9      @Html.ValidationSummary(true)
10     <fieldset>
11         <legend>Shipping Information</legend>
12         @Html.EditorForModel()
13     <p>
14         <input type="submit" value="Guardar" />
15     </p>
16 </fieldset>
17 }
18 @section Scripts {
19     @Scripts.Render("~/bundles/jqueryval")
20 }
21

```

Finalmente lo compilaremos y el resultado será el siguiente:

En lo anterior podras observar, que generamos una vista completa a partir de un modelo usando el helper `Html.EditorForModel`.

Required

Con esto hacemos que determinados campos sean obligatorios

```
[Required]
public string FirstName { get; set; }

[Required]
public string LastName { get; set; }
```

Longitud de Cadena (StringLength)

Con este atributo podemos establecer los tamaños mínimos y máximos de los campos.

```
[Required]
[StringLength(160)]
public string FirstName { get; set; }

[Required]
[StringLength(160)]
public string LastName { get; set; }
```

Expresiones Regulares

Algunas propiedades requieren mas que una simple revisión de la longitud de una cadena. Como por ejemplo un campo de Email.

```
[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]
public string Email { get; set; }
```

Display

Este atributo coloca el texto a mostrar para cada propiedad. También nos permite enumerar el orden en que se agrupara cada propiedad del modelo (Valor default es 1000).

```
[Required]
[StringLength(160)]
[Display(Name="Last Name", Order=15001)]
[MaxWords(10, ErrorMessage="There are too many words in {0}")]
public string LastName { get; set; }

[Required]
[StringLength(160, MinimumLength=3)]
[Display(Name="First Name", Order=15000)]
public string FirstName { get; set; }
```

Formato de Datos

Por medio de esta anotación podemos darle cierto formato a las propiedades a mostrar en base al tipo de datos.

```
[DisplayFormat(ApplyFormatInEditMode = true, DataFormatString = "{0:d2}")]
public decimal Price { get; set; }
```

Nota. d es para decimales, c para dinero

Tipo de Datos

Este atributo te permite establecer el tipo específico que va a cargar la propiedad en la que se ha usado.

```
[Required]
[DataType(DataType.EmailAddress)]
[Display(Name="Correo Electronico")]
public string Email { get; set; }
```

Correo Electronico

Please enter a valid email address.

Rangos

Los rangos establecen los valores máximos y mínimos para valores numéricos.

```
[Range(35, 44)]
public int Age { get; set; }
```

```
[Range(typeof(decimal), "0.00", "49.99")]
public decimal Price { get; set; }
```

Así como las anteriores hay muchísimas validaciones, de las cuales solo mencionamos algunas. También podemos tomar valores desde ítems de tipo .resx, donde podemos almacenar valores para reutilizarlos desde nuestros modelos. Esto es especialmente útil en casos de tener varios idiomas.

Considerando que en el proyecto has agregado ErrorMessage.resx:

ErrorMessage.resx		
Strings	Add Resource	Remove Resource
		Access Modifier: Internal
Name	Value	
LastNameRequired	Campo obligatorio	
LastNameTooLong	El apellido es demasiado largo	

En el modelo de order, vamos a agregar unos atributos al campo LastName:

```
[Required(ErrorMessageResourceType = typeof(ErrorMessage), ErrorMessageResourceName = "LastNameRequired")]
[StringLength(160, ErrorMessageResourceType = typeof(ErrorMessage), ErrorMessageResourceName = "LastNameTooLong")]
public string LastName { get; set; }
```

LastName

Campo obligatorio

LastName

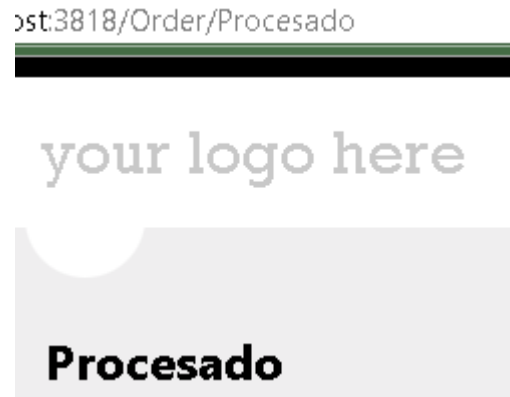
El apellido es demasiado largo

Complementando el ejemplo anterior en el controlador de Order agregaremos lo siguiente:

```
[HttpPost]
public ActionResult Index(Order order)
{
    return RedirectToAction("Procesado");
}

public ActionResult Procesado()
{
    return View();
}
```

Lo que haré esto si recuerdas secciones pasadas, será contestar al `HttpPost` cuando demos clic en el botón submit en la acción `Index` de tipo `Post`. Al recibir el objeto de tipo `Order` desde la vista lo que haré será regresarnos otra vista llamada `Procesado`, que nos mostrara el siguiente mensaje si todo ha ido bien:



Validaciones Personalizadas

Todas las anotaciones de datos derivan de la clase `ValidationAttribute` y vive en `System.ComponentModel.DataAnnotations`. Lo que haremos a continuación será sobrescribir los métodos que la misma contiene para construir variantes personalizadas de los mismos.

Nota. Agrega al proyecto una clase llamada `AtributoMaxPalabras.css` a la carpeta de `models`.

```
public class AtributoMaxPalabras:ValidationAttribute
{
    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
    {
        return ValidationResult.Success;
    }
}
```

El parámetro `ValidationContext` nos proveerá de mas información que podemos usar dentro del método `IsValid`, dándonos acceso directo al modelo, y para mostrar la propiedad que actualmente se intenta validar.

El parámetro `value`, como su nombre lo indica, es el valor de la propiedad que se está tratando de validar y si dicho valor es válido retornaremos un `Success`, para lo que necesitamos primero enseñarle al método como determinar cuantas palabras se consideran como muchas palabras.

```

private readonly int _maxWords;
public AtributoMaxPalabras(int maxWords)

{
    _maxWords = maxWords;
}

```

Lo siguiente sería Implementar la lógica de la validación para atrapar los errores que pudieran surgir de la misma.

```

protected override ValidationResult IsValid(object value, ValidationContext validationContext)
{
    if (value != null)
    {
        var valueAsString = value.ToString();
        if (valueAsString.Split(' ').Length > _maxWords)
        {
            var errorMessage = FormatErrorMessage(validationContext.DisplayName);
            return new ValidationResult(errorMessage);
        }
    }
    return ValidationResult.Success;
}

```

lo que hace el código anterior después de haber contado las palabras es, que de ser mayor de N palabras regresara al controlador un error contenido en DisplayName (si es que existe en una anotación sobre el campo) ocasionando que tengamos que verificar en el controlador que todo fue validado correctamente (OrderController.Index)

```

[HttpPost]
public ActionResult Index(Order order)
{
    if (ModelState.IsValid )
        return RedirectToAction("Procesado");
    return View();
}

```

Si ModelState.IsValid devuelve falso, redireccionara al usuario a la Vista Order.Index mostrándole los errores que ocasionaron que no pudiese llevarse con éxito el proceso de los datos. De otra manera se mostrara la vista de procesado.

Finalmente para poder probar esta validación lo que tendríamos que hacer seria mandarla llamar directamente desde una anotación en nuestro modelo de order.

```

[Required(ErrorMessage="Campo requerido")]
[AtributoMaxPalabras(5,ErrorMessage="No mas de 5 palabras")]
public string Username { get; set; }

```

st:2265/Order

your logo here

Username

1 2 3 4 5 6

No mas de 5 palabras

SEGURIDAD Y SESIONES

Este es uno de los temas de más interés y más extensos cuando se habla de ASP.NET MVC. En la plantilla que hemos venido usando a lo largo de todo este material automáticamente se creó todo lo necesario para gestionar seguridad de manera automática cuando creamos el proyecto. Tan es así que para gestionar la seguridad lo único que tendríamos que hacer sería colocar el atributo **[Authorize]** en las acciones o controladores enteros donde quisiéramos tener identificado al usuario que estuviese en determinado momento accedendo a ella. Por ejemplo si agregamos en el StoreManager dicho atributo veríamos lo siguiente al querer acceder a el:

```
[Authorize]
public class StoreManagerController : Controller
```



your logo here

Log in.

Use a local account to log in.

User name

Password

☐ Remember me?

Log in

[Register](#) if you don't have an account.

Lo anterior es muy útil y ahorra muchísimo tiempo al estar haciendo una aplicación, pero a mi consideración y espero que a la tuya también, no nos sirve de nada si no sabemos cómo funciona, por lo cual a continuación nos daremos a la tarea de desarrollar la lógica de Logins y Logouts desde cero.

Para poder desarrollar esto desde cero, tendremos que hacer un nuevo proyecto en blanco, pero en vez de seleccionar **Internet Application**, seleccionaremos **Basic Application** la cual no contiene todo el proceso de Logueo por defecto.

Una vez que ya tengas tu proyecto listo, comenzaremos agregando el controlador con el que probaremos la seguridad (HomeController), así mismo agregaremos una vista correspondiente a la acción Index que aparece por defecto en Home.

Después de esto agregaremos otro controlador llamado AccountController, el cual, será el que el framework busque de manera automática al hacer uso de [Authorize] en nuestros controladores.

Lo siguiente será agregar un Modelo llamado AccountModel, el cual contendrá las propiedades que usaremos para las vistas en 3 diferentes clases, todas contenidas dentro del mismo modelo.

```
public class ChangePasswordModel
{
    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Current password")]
    public string OldPassword { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "New password")]
    public string NewPassword { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm new password")]
    [System.ComponentModel.DataAnnotations.Compare("NewPassword", ErrorMessage = "The new password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}

public class LogOnModel
{
    [Required]
    [Display(Name = "User name")]

```

```

        public string UserName { get; set; }

        [Required]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string Password { get; set; }

        [Display(Name = "Remember me?")]
        public bool RememberMe { get; set; }
    }

    public class RegisterModel
    {
        [Required]
        [Display(Name = "User name")]
        public string UserName { get; set; }

        [Required]
        [DataType(DataType.EmailAddress)]
        [Display(Name = "Email address")]
        public string Email { get; set; }

        [Required]
        [StringLength(100, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirm password")]
        [System.Web.Mvc.Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
        public string ConfirmPassword { get; set; }
    }

```

Como te darás cuenta, cada clase representa una acción distinta, las cuales son Registrar, Iniciar Sesión y Cambiar contraseña.

Primeramente al controlador de Account debemos agregar lo siguiente para los códigos de error que pudieran surgir al momento de llevar a cabo todas las acciones ya mencionadas.

```

#region Status Codes
private static string ErrorCodeToString(MembershipCreateStatus createStatus)
{
    // See http://go.microsoft.com/fwlink/?LinkID=177550 for
    // a full list of status codes.
    switch (createStatus)
    {
        case MembershipCreateStatus.DuplicateUserName:

```

```

        return "User name already exists. Please enter a different u
ser name.";

        case MembershipCreateStatus.DuplicateEmail:
            return "A user name for that e-
mail address already exists. Please enter a different e-mail address.";

        case MembershipCreateStatus.InvalidPassword:
            return "The password provided is invalid. Please enter a val
id password value.";

        case MembershipCreateStatus.InvalidEmail:
            return "The e-
mail address provided is invalid. Please check the value and try again.";

        case MembershipCreateStatus.InvalidAnswer:
            return "The password retrieval answer provided is invalid. P
lease check the value and try again.";

        case MembershipCreateStatus.InvalidQuestion:
            return "The password retrieval question provided is invalid.
Please check the value and try again.";

        case MembershipCreateStatus.InvalidUserName:
            return "The user name provided is invalid. Please check the
value and try again.";

        case MembershipCreateStatus.ProviderError:
            return "The authentication provider returned an error. Pleas
e verify your entry and try again. If the problem persists, please contact your
system administrator.";

        case MembershipCreateStatus.UserRejected:
            return "The user creation request has been canceled. Please
verify your entry and try again. If the problem persists, please contact your sy
stem administrator.";

        default:
            return "An unknown error occurred. Please verify your entry
and try again. If the problem persists, please contact your system administrator
.";
    }
}
#endregion

```

Lo siguiente que haremos será crear las acciones en el controlador Account. Comenzando por Login, la cual como ya te has de imaginar, tendrá una acción Get y una Post, después de haber agregado el atributo [Authorize] al controlador.

```

//
// GET: /Account/LogOn

public ActionResult Login()
{

```

```

        return View();
    }

    //
    // POST: /Account/LogOn

    [HttpPost]
    public ActionResult Login(LogOnModel model, string returnUrl)
    {
        if (ModelState.IsValid)
        {
            if (Membership.ValidateUser(model.UserName, model.Password))
            {
                FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);

                if (Url.IsLocalUrl(returnUrl) && returnUrl.Length > 1 && returnUrl.StartsWith("/")
                    && !returnUrl.StartsWith("//") && !returnUrl.StartsWith("/\\"))
                {
                    return Redirect(returnUrl);
                }
                else
                {
                    return RedirectToAction("Index", "Home");
                }
            }
            else
            {
                ModelState.AddModelError("", "The user name or password provided is incorrect.");
            }
        }

        // If we got this far, something failed, redisplay form
        return View(model);
    }

```

El código de La vista:

```
@model MusicStore.Models.LogOnModel
```

```
@{
    ViewBag.Title = "Log On";
}
```

```
<h2>Log On</h2>
```

```
<p>
```

```
Please enter your user name and password. @Html.ActionLink("Register", "Register") if you don't have an account.
```

</p>

```
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javasc
ript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type=
"text/javascript"></script>
```

```
@Html.ValidationSummary(true, "Login was unsuccessful. Please correct the errors
and try again.")
```

```
@using (Html.BeginForm()) {
    <div>
        <fieldset>
            <legend>Account Information</legend>

            <div class="editor-label">
                @Html.LabelFor(m => m.UserName)
            </div>
            <div class="editor-field">
                @Html.TextBoxFor(m => m.UserName)
                @Html.ValidationMessageFor(m => m.UserName)
            </div>

            <div class="editor-label">
                @Html.LabelFor(m => m.Password)
            </div>
            <div class="editor-field">
                @Html.PasswordFor(m => m.Password)
                @Html.ValidationMessageFor(m => m.Password)
            </div>

            <div class="editor-label">
                @Html.CheckBoxFor(m => m.RememberMe)
                @Html.LabelFor(m => m.RememberMe)
            </div>

            <p>
                <input type="submit" value="Log On" />
            </p>
        </fieldset>
    </div>
}
```

Log On

Please enter your user name and password. [Register](#) if you don't have an account.

Account Information

User name

Password

☐ Remember me?

La acción Login lo único que hará será verificar la existencia de un determinado usuario en la Base de datos de usuarios. EL problema aquí es que dicha base de datos aun no existe!!, por lo cual pasaremos a la siguiente acción de Registrar.

En el controlador al igual que el Login, agregaras el siguiente código:

```
//
// GET: /Account/Register

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

//
// POST: /Account/Register

[HttpPost]
public ActionResult Register(RegisterModel model)
{
    if (ModelState.IsValid)
    {
        // Attempt to register the user
        MembershipCreateStatus createStatus;
        Membership.CreateUser(model.UserName, model.Password, model.Email, "question", "answer", true, null, out createStatus);

        if (createStatus == MembershipCreateStatus.Success)
        {
            FormsAuthentication.SetAuthCookie(model.UserName, false /* createPersistentCookie */);
            return RedirectToAction("Index", "Home");
        }
        else
        {

```

```

        ModelState.AddModelError("", ErrorCodeToString(createStatus)
    );
    }
}

// If we got this far, something failed, redisplay form
return View(model);
}

```

Y en la Vista:

```
@model MusicStore.Models.RegisterModel
```

```
@{
}

```

```
    ViewBag.Title = "Register";
```

```
<h2>Create a New Account</h2>
```

```
<p>
```

```
    Use the form below to create a new account.
```

```
</p>
```

```
<p>
```

```
    Passwords are required to be a minimum of @Membership.MinRequiredPasswordLength
    characters in length.
```

```
</p>
```

```
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
```

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

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

```
    @Html.ValidationSummary(true, "Account creation was unsuccessful. Please correct the errors and try again.")
```

```
    <div>
```

```
        <fieldset>
```

```
            <legend>Account Information</legend>
```

```
            <div class="editor-label">
```

```
                @Html.LabelFor(m => m.UserName)
```

```
            </div>
```

```
            <div class="editor-field">
```

```
                @Html.TextBoxFor(m => m.UserName)
```

```
                @Html.ValidationMessageFor(m => m.UserName)
```

```
            </div>
```

```
            <div class="editor-label">
```

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

```
            </div>
```

```
            <div class="editor-field">
```

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

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

```
            </div>
```

```

        <div class="editor-label">
            @Html.LabelFor(m => m.Password)
        </div>
        <div class="editor-field">
            @Html.PasswordFor(m => m.Password)
            @Html.ValidationMessageFor(m => m.Password)
        </div>

        <div class="editor-label">
            @Html.LabelFor(m => m.ConfirmPassword)
        </div>
        <div class="editor-field">
            @Html.PasswordFor(m => m.ConfirmPassword)
            @Html.ValidationMessageFor(m => m.ConfirmPassword)
        </div>

        <p>
            <input type="submit" value="Register" />
        </p>
    </fieldset>
</div>
}

```

De manera breve el controlador de registro es donde creamos a los usuarios, pero a diferencia de aplicaciones normales, aquí el framework nos ayuda de manera excepcional, pues al crear, modificar o eliminar usuarios este lo hace prácticamente todo. Si no hay una base de datos creada, el framework se encarga de hacerlo basándose en el modelo que le dimos.

El código anterior aun no es funcional, pues necesitamos agregar un Hipervínculo que nos dirccione a Register, por lo cual en el _layout dentro de Views/Shared colocaremos lo siguiente:

```

<section id="login">
    @Html.Partial("_LoginPartial")
</section>

```

Dentro de ese mismo directorio crearemos una nueva vista llamada _LoginPartial:

```

@if (Request.IsAuthenticated) {
    <text>
        Hello, @Html.ActionLink(User.Identity.Name, "ChangePassword", "Account", routeValues: null,
        htmlAttributes: new { @class = "username", title = "Manage" })!
    </text>
} else {
    <ul>
        <li>@Html.ActionLink("Register", "Register", "Account", routeValues: null,
        htmlAttributes: new { id = "registerLink" })</li>
    </ul>
}

```

El primer if se hace con motivo de que si ya se detecto que hay una sesión activa en la aplicación se mostrara una etiqueta saludando al usuario, y en caso contrario solo se mostrara el hipervínculo a Register. Recordemos que _layout.cshtml se mostrara siempre en nuestra aplicación.

Lo anterior nos mostraría lo siguiente:

Log On

Please enter your user name and p

Account Information

User name

Password

☐ Remember me?

Log On

- [Register](#)

Create a New Account

Use the form below to create a new

Passwords are required to be a mini

Account Information

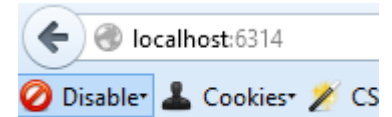
User name

Email address

Password

Confirm password

Register



Index

Hello, [usuario](#)!

Finalmente pero no menos importante, restaría el cambio de contraseña. Esta vista se activara cuando demos clic sobre el nombre del usuario en el mensaje de bienvenida. Si revisas el código de la vista parcial te daras cuenta de que ese mensaje de bienvenida hace referencia a una acción llamada ChangePassword, la cual agregaremos al controlador a continuación:

```
//
// GET: /Account/ChangePassword

[Authorize]
public ActionResult ChangePassword()
{
    return View();
}

//
// POST: /Account/ChangePassword

[Authorize]
[HttpPost]
public ActionResult ChangePassword(ChangePasswordModel model)
{
    if (ModelState.IsValid)
    {

```

```

        // ChangePassword will throw an exception rather
        // than return false in certain failure scenarios.
        bool changePasswordSucceeded;
        try
        {
            MembershipUser currentUser = Membership.GetUser(User.Identity.Name, true /* userIsOnline */);
            changePasswordSucceeded = currentUser.ChangePassword(model.OldPassword, model.NewPassword);
        }
        catch (Exception)
        {
            changePasswordSucceeded = false;
        }

        if (changePasswordSucceeded)
        {
            return RedirectToAction("ChangePasswordSuccess");
        }
        else
        {
            ModelState.AddModelError("", "The current password is incorrect or the new password is invalid.");
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}

//
// GET: /Account/ChangePasswordSuccess

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

```

Aquí esta ultima vista lo único que hara será indicarnos que se ha cambiado con éxito la contraseña.

El código para la vista ChangePassword y ChangePasswordSuccess respectivamente es el siguiente:

ChangePassword.cshtml

```

@model MusicStore.Models.ChangePasswordModel

@{
    ViewBag.Title = "Change Password";
}

<h2>Change Password</h2>
<p>

```

Use the form below to change your password.

</p>

<p>

New passwords are required to be a minimum of @Membership.MinRequiredPasswordLength characters in length.

</p>

```
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
```

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

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

```
    @Html.ValidationSummary(true, "Password change was unsuccessful. Please correct the errors and try again.")
```

```
    <div>
```

```
        <fieldset>
```

```
            <legend>Account Information</legend>
```

```
            <div class="editor-label">
```

```
                @Html.LabelFor(m => m.OldPassword)
```

```
            </div>
```

```
            <div class="editor-field">
```

```
                @Html.PasswordFor(m => m.OldPassword)
```

```
                @Html.ValidationMessageFor(m => m.OldPassword)
```

```
            </div>
```

```
            <div class="editor-label">
```

```
                @Html.LabelFor(m => m.NewPassword)
```

```
            </div>
```

```
            <div class="editor-field">
```

```
                @Html.PasswordFor(m => m.NewPassword)
```

```
                @Html.ValidationMessageFor(m => m.NewPassword)
```

```
            </div>
```

```
            <div class="editor-label">
```

```
                @Html.LabelFor(m => m.ConfirmPassword)
```

```
            </div>
```

```
            <div class="editor-field">
```

```
                @Html.PasswordFor(m => m.ConfirmPassword)
```

```
                @Html.ValidationMessageFor(m => m.ConfirmPassword)
```

```
            </div>
```

```
        <p>
```

```
            <input type="submit" value="Change Password" />
```

```
        </p>
```

```
    </fieldset>
```

```
</div>
```

```
}
```

ChangePasswordSuccess

```
@{
```

```
    ViewBag.Title = "Change Password";
```

}

`<h2>Change Password</h2>``<p>``Your password has been changed successfully.``</p>`

Ahora ya hecho todo esto, al momento de querer cambiar una contraseña y, habiéndolo hecho con éxito deberíamos ver lo siguiente:

Change Password

Use the form below to change your password.

New passwords are required to be a minimum of

Account Information

Current password

New password

Confirm new password

Change Password

Change Password

Your password has been changed successfully.

Hello, [usuario!](#)

Al final de haber hecho todo esto podrás preguntar porque no se hizo tanto énfasis en cada parte del código como en parte anteriores de este material, y la respuesta es sencilla. Todo el código que viste anteriormente es un fragmento muy pequeño de una plantilla para manejo de sesiones que puede ser instalada fácilmente en menos de 2 minutos a través de NuGet. Obviamente la plantilla completa tiene muchísimas cosas más, en otras palabras, muchísimas más líneas de código que las mostradas arriba, pero considero a pesar de eso que es un muy buen ejemplo de una implementación bastante sencilla sobre el uso de sesiones, y sobre el cual confió en que habrás entendido lo sencillo que es implementarlo en ASP.NET MVC.

Algo que es importante resaltar, es que no es la única manera de manejo de sesiones, pero si la más sencilla. Si te interesa buscar más revisa sobre el tema en internet.

AJAX

Hoy en día es muy raro construir una aplicación web que no utilice Ajax en alguna manera. Ajax significa Javascript Asíncrono y Xml. Ajax proporciona las mejores técnicas para construir aplicaciones web altamente responsivas, de manera conjunta con una gran experiencia de usuario.

JQUERY (Escribe menos, has más)

Es una librería de JavaScript, soportada en todos los exploradores modernos. Cuando se utiliza jQuery el trabajo a hacer se acorta en cantidades inmensurables.

jQuery es una librería OpenSource, por lo cual puedes hallar todo lo relacionado al mismo en jquery.com. Microsoft soporta jQuery, por lo que cada plantilla para MVC va a colocar los scripts de jQuery en la carpeta Scripts como se mencionó al principio.

Características de jQuery

jQuery Function

Este objeto es el que nos hace posible acceder a las características de jQuery. Una de las partes más confusas cuando se está comenzando con jQuery es el uso de \$, y más aun como es que se pasa cualquier tipo de argumento a través de la función \$, quien deducirá que es lo que se está intentando almacenar.

```
$(function () {
    $("#album-list img").mouseover(function () {
        $(this).animate({ height: '+=25', width: '+=25' })
        .animate({ height: '-=25', width: '-=25' });
    });
});
```

La primera línea de código está invocando la función(\$) y pasando una función anónima como parámetro.

Cuando se pasa una función como primer parámetro, jQuery asume que le estamos proveyendo una función para ejecutarla tan pronto como el explorador que estemos usando haya terminado de construir un documento HTML proporcionado por el servidor. Aquí es cuando podemos ejecutar de manera segura script.

```
$(function () {
    $("#album-list img").mouseover(function () {
        $(this).animate({ height: '+=25', width: '+=25' })
        .animate({ height: '-=25', width: '-=25' });
    });
});
```

jQuery interpreta esta cadena como un **Selector**, el cual le dice a jQuery que elementos estamos buscando en el DOM (document object model). Estos elementos pueden ser buscados a través de tus atributos, nombres de clases, posición, entre muchas otras. El selector de la segunda línea le dice a jQuery que encuentre todas las imágenes dentro del elemento con un valor **id** de `album-list`.

Cuando el selector se ejecuta, regresa un conjunto de cero o más elementos. Cualquier método adicional de jQuery que se invoque va a operar en contra de los elementos del conjunto. Por ejemplo, el método **mouseover** liga un manejador de evento al evento **onmouseover** para cada imagen que haya coincidido el selector.

jQuery explota las capacidades de programación funcional de JavaScript. Muchas veces te encontraras a ti mismo creando y pasando funciones como parámetros en métodos de jQuery. El método **mouseover** por ejemplo, sabe cómo ligar un manejador de evento para **onmouseover** sin importar el explorador en el que se esté ejecutando, pero no sabe que es lo que se quiere que se haga cuando dichos eventos sean disparados. Para expresar que es lo que se quiere que ocurra cuando esos eventos sean disparados, debes pasar una función que contenga el código disparado por el evento.

```
$(function () {
    $("#album-list img").mouseover(function () {
        $(this).animate({ height: '+=25', width: '+=25' })
        .animate({ height: '-=25', width: '-=25' });
    });
});
```

En el código anterior, el código anima un elemento durante el evento **mouseover**. El elemento que el código anima es referenciado por la palabra **this** (**apunta al elemento que disparo el evento**). Nótese como es que el código pasa el elemento a jQuery (`$(this)`). jQuery ve el argumento como una referencia hacia un elemento y regresa un empaquetado con el elemento dentro.

Una vez que se tiene el elemento dentro de jQuery, puedes invocar métodos de jQuery como **anímate** para manipular al elemento. EL código en el ejemplo hace que la imagen crezca por 25 pixeles, y que se encoja 25 pixeles.

jQuery Selectors

Como ya se mencionó, los selectores son las cadenas que se pasan a la función jQuery para seleccionar elementos en el DOM. En el ejemplo anterior usamos #album-list img como selector para encontrar etiquetas de imágenes. Si piensas que la cadena luce como algo que podrías usar en una hoja de estilo (CSS), podrías estar en lo correcto. Los selectores de jQuery derivan directamente de los selectores de CSS 3.0 con algunas adiciones.

Ejemplo	Significado
\$('#header')	Encontrar elemento con un id de "header"
\$('.editor-label')	Encontrar todos los elementos con un nombre de clase de ".editor-label"
\$('div')	Encontrar todos los elementos <div>
\$('#header div')	Encontrar todos los elementos <div> descendientes de un element con un id de "header".
\$('#header > div')	Encontrar todos los elementos que son hijos del elemento con un id de "header"
\$('a:even')	Encontrar etiquetas de ancho enumeradas

Ejemplo de Ajax haciendo uso de jQuery

Para dar un ejemplo lo suficientemente claro, y que sea complementario con lo que hemos venido haciendo hasta el momento, haremos un carrito de compras con actualizaciones AJAX.

Agregaremos a modelos las siguientes clases:

```
public class Cart
{
    [Key]
    public int      RecordId      { get; set; }
    public string   CartId        { get; set; }
    public int      AlbumId       { get; set; }
    public int      Count         { get; set; }
    public System.DateTime DateCreated { get; set; }
    public virtual Album Album    { get; set; }
}
```

```
public partial class Order
{
    public int      OrderId      { get; set; }
    public string   Username     { get; set; }
    public string   FirstName    { get; set; }
    public string   LastName     { get; set; }
    public string   Address       { get; set; }
    public string   City          { get; set; }
    public string   State         { get; set; }
    public string   PostalCode    { get; set; }
```

```

        public string Country      { get; set; }
        public string Phone        { get; set; }
        public string Email        { get; set; }
        public decimal Total       { get; set; }
        public System.DateTime OrderDate { get; set; }
        public List<OrderDetail> OrderDetails { get; set; }
    }

    public class MusicStoreEntities : DbContext
    {
        public DbSet<Album> Albums { get; set; }
        public DbSet<Genre> Genres { get; set; }
        public DbSet<Artist> Artists {
get; set; }
        public DbSet<Cart>
Carts { get; set; }
        public DbSet<Order> Orders
{ get; set; }
        public DbSet<OrderDetail>
OrderDetails { get; set; }
    }

    public partial class ShoppingCart
    {
        MusicStoreDbContext storeDB = new MusicStoreDbContext ();
        string ShoppingCartId { get; set; }
        public const string CartSessionKey = "CartId";
        public static ShoppingCart GetCart(HttpContextBase context)
        {
            var cart = new ShoppingCart();
            cart.ShoppingCartId = cart.GetCartId(context);
            return cart;
        }
        // Helper method to simplify shopping cart calls
        public static ShoppingCart GetCart(Controller controller)
        {
            return GetCart(controller.HttpContext);
        }
        public void AddToCart(Album album)
        {
            // Get the matching cart and album instances
            var cartItem = storeDB.Carts.SingleOrDefault(
                c => c.CartId == ShoppingCartId
                && c.AlbumId == album.AlbumId);

            if (cartItem == null)
            {
                // Create a new cart item if no cart item exists
                cartItem = new Cart
                {
                    AlbumId = album.AlbumId,
                    CartId = ShoppingCartId,
                    Count = 1,
                    DateCreated = DateTime.Now
                };
            }
        }
    }

```



```

        storeDB.Carts.Add(cartItem);
    }
    else
    {
        // If the item does exist in the cart,
        // then add one to the quantity
        cartItem.Count++;
    }
    // Save changes
    storeDB.SaveChanges();
}
public int RemoveFromCart(int id)
{
    // Get the cart
    var cartItem = storeDB.Carts.Single(
        cart => cart.CartId == ShoppingCartId
        && cart.RecordId == id);

    int itemCount = 0;

    if (cartItem != null)
    {
        if (cartItem.Count > 1)
        {
            cartItem.Count--;
            itemCount = cartItem.Count;
        }
        else
        {
            storeDB.Carts.Remove(cartItem);
        }
        // Save changes
        storeDB.SaveChanges();
    }
    return itemCount;
}
public void EmptyCart()
{
    var cartItems = storeDB.Carts.Where(
        cart => cart.CartId == ShoppingCartId);

    foreach (var cartItem in cartItems)
    {
        storeDB.Carts.Remove(cartItem);
    }
    // Save changes
    storeDB.SaveChanges();
}
public List<Cart> GetCartItems()
{
    return storeDB.Carts.Where(
        cart => cart.CartId == ShoppingCartId).ToList();
}
public int GetCount()
{
    // Get the count of each item in the cart and sum them up
    int? count = (from cartItems in storeDB.Carts

```

```

        where cartItems.CartId == ShoppingCartId
        select (int?)cartItems.Count).Sum();
    // Return 0 if all entries are null
    return count ?? 0;
}
public decimal GetTotal()
{
    // Multiply album price by count of that album to get
    // the current price for each of those albums in the cart
    // sum all album price totals to get the cart total
    decimal? total = (from cartItems in storeDB.Carts
        where cartItems.CartId == ShoppingCartId
        select (int?)cartItems.Count *
            cartItems.Album.Price).Sum();

    return total ?? decimal.Zero;
}
public int CreateOrder(Order order)
{
    decimal orderTotal = 0;

    var cartItems = GetCartItems();
    // Iterate over the items in the cart,
    // adding the order details for each
    foreach (var item in cartItems)
    {
        var orderDetail = new OrderDetail
        {
            AlbumId = item.AlbumId,
            OrderId = order.OrderId,
            UnitPrice = item.Album.Price,
            Quantity = item.Count
        };
        // Set the order total of the shopping cart
        orderTotal += (item.Count * item.Album.Price);

        storeDB.OrderDetails.Add(orderDetail);
    }
    // Set the order's total to the orderTotal count
    order.Total = orderTotal;

    // Save the order
    storeDB.SaveChanges();
    // Empty the shopping cart
    EmptyCart();
    // Return the OrderId as the confirmation number
    return order.OrderId;
}
// We're using HttpContextBase to allow access to cookies.
public string GetCartId(HttpContextBase context)
{
    if (context.Session[CartSessionKey] == null)
    {
        if
(!string.IsNullOrEmpty(context.User.Identity.Name))
        {

```

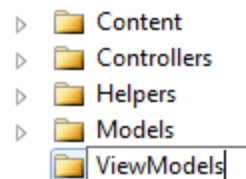
```

        context.Session[CartSessionKey] =
            context.User.Identity.Name;
    }
    else
    {
        // Generate a new random GUID using System.Guid class
        Guid tempCartId = Guid.NewGuid();
        // Send tempCartId back to client as a cookie
        context.Session[CartSessionKey] =
tempCartId.ToString();
    }
    }
    return context.Session[CartSessionKey].ToString();
}
// When a user has logged in, migrate their shopping cart to
// be associated with their username
public void MigrateCart(string userName)
{
    var shoppingCart = storeDB.Carts.Where(
        c => c.CartId == ShoppingCartId);

    foreach (Cart item in shoppingCart)
    {
        item.CartId = userName;
    }
    storeDB.SaveChanges();
}
}

```

A continuación lo que haremos será crear una carpeta llamada ViewModels, que contendrá StronglyTypes views para manejar la logica del negocio a través del uso directo de los modelos, haciendo todas las tareas del carrito de compra más sencillas.



Una vez creada agregaremos una clase llamada **ShoppingCartViewModel** dentro de la misma y contendrá lo siguiente:

```

public class ShoppingCartViewModel
{
    public List<Cart> CartItems { get; set; }
    public decimal CartTotal { get; set; }
}

```

Igualmente agregaremos otra clase llamada **ShoppingCartRemoveViewModel** con sus respectivas propiedades:

```
public string Message { get; set; }
public decimal CartTotal { get; set; }
public int CartCount { get; set; }
public int ItemCount { get; set; }
public int DeleteId { get; set; }
```

Después de lo anterior, pasaremos a los controladores que tendrán la tarea de añadir, editar y eliminar ítems del carrito. Agregaremos un nuevo controlador entonces en el cual al igual que se ha estado haciendo desde el principio de este material, haremos uso del **MusicStoreEntities** para acceder a la base de datos.

El código que contendrá el controlador para ejecutar las tareas que se mencionaron sería el siguiente:

```
public class ShoppingCartController : Controller
{
    MusicStoreDbContext storeDB = new MusicStoreDbContext ();
    //
    // GET: /ShoppingCart/
    public ActionResult Index()
```

```

{
    var cart = ShoppingCart.GetCart(this.HttpContext);

    // Set up our ViewModel
    var viewModel = new ShoppingCartViewModel
    {
        CartItems = cart.GetCartItems(),
        CartTotal = cart.GetTotal()
    };
    // Return the view
    return View(viewModel);
}
//
// GET: /Store/AddToCart/5
public ActionResult AddToCart(int id)
{
    // Retrieve the album from the database
    var addedAlbum = storeDB.Albums
        .Single(album => album.AlbumId == id);

    // Add it to the shopping cart
    var cart = ShoppingCart.GetCart(this.HttpContext);

    cart.AddToCart(addedAlbum);

    // Go back to the main store page for more shopping
    return RedirectToAction("Index");
}
//
// AJAX: /ShoppingCart/RemoveFromCart/5
[HttpPost]
public ActionResult RemoveFromCart(int id)
{
    // Remove the item from the cart
    var cart = ShoppingCart.GetCart(this.HttpContext);

    // Get the name of the album to display confirmation
    string albumName = storeDB.Carts
        .Single(item => item.RecordId == id).Album.Title;

    // Remove from cart
    int itemCount = cart.RemoveFromCart(id);

    // Display the confirmation message
    var results = new ShoppingCartRemoveViewModel
    {
        Message = Server.HtmlEncode(albumName) +
            " has been removed from your shopping cart.",
        CartTotal = cart.GetTotal(),
        CartCount = cart.GetCount(),
        ItemCount = itemCount,
        DeleteId = id
    };
    return Json(results);
}
//
// GET: /ShoppingCart/CartSummary

```

```

[ChildActionOnly]
public ActionResult CartSummary()
{
    var cart = ShoppingCart.GetCart(this.HttpContext);

    ViewData["CartCount"] = cart.GetCount();
    return PartialView("CartSummary");
}
}

```

Ahora pasaremos a la parte de las actualizaciones Ajax con jQuery. Para la acción anterior de Index crearemos una vista, pero esta estará fuertemente tipeada al modelo **ShoppingCartViewModel**, solo que en esta ocasión usaremos la plantilla de lista.

En la vista lo que haremos con jQuery será ligar el evento click para todos los elementos que encontremos en la lista el cual regresará una llamada al método en nuestro controlador. Dicho evento nos regresará un objeto serializado en formato JSON.

Código de vista:

```

@model MusicStore.ViewModels.ShoppingCartViewModel
@{
    ViewBag.Title = "Shopping Cart";
}

```

```

}
<script src="/Scripts/jquery-1.4.4.min.js"
type="text/javascript"></script>
<script type="text/javascript">
    $(function () {
        // Document.ready -> link up remove event handler
        $(".RemoveLink").click(function () {
            // Get the id from the link
            var recordToDelete = $(this).attr("data-id");
            if (recordToDelete != '') {
                // Perform the ajax post
                $.post("/ShoppingCart/RemoveFromCart", {"id":
recordToDelete },
                    function (data) {
                        // Successful requests get here
                        // Update the page elements
                        if (data.ItemCount == 0) {
                            $('#row-' + data.DeleteId).fadeOut('slow');
                        } else {
                            $('#item-count-' +
data.DeleteId).text(data.ItemCount);
                        }
                        $('#cart-total').text(data.CartTotal);
                        $('#update-message').text(data.Message);
                        $('#cart-status').text('Cart (' + data.CartCount
+ ')');
                    });
            }
        });
    });
</script>
<h3>
    <em>Review</em> your cart:
</h3>
<p class="button">
    @Html.ActionLink("Checkout
>>", "AddressAndPayment", "Checkout")
</p>
<div id="update-message">
</div>
<table>
    <tr>
        <th>
            Album Name
        </th>
        <th>
            Price (each)
        </th>
        <th>
            Quantity
        </th>
    </tr>
    @foreach (var item in
Model.CartItems)
    {
        <tr id="row-@item.RecordId">

```

```

        <td>
            @Html.ActionLink(item.Album.Title,
"Details", "Store", new { id = item.AlbumId }, null)
        </td>
        <td>
            @item.Album.Price
        </td>
        <td id="item-count-@item.RecordId">
            @item.Count
        </td>
        <td>
            <a href="#" class="RemoveLink"
data-id="@item.RecordId">Remove
from cart</a>
        </td>
    </tr>
}
<tr>
    <td>
        Total
    </td>
    <td>
    </td>
    <td>
    </td>
    <td id="cart-total">
        @Model.CartTotal
    </td>
</tr>
</table>

```

Y finalmente para que podamos probar lo anterior, necesitamos poder agregar ítems al carrito de compra, para lo cual alteraremos un poco la vista **StoreDetails** para incluir un botón que diga Add.

```

@model MusicStore.Models.Album
@{
    ViewBag.Title = "Album - " + Model.Title;
}
<h2>@Model.Title</h2>
<p>
    
</p>
<div id="album-details">
    <p>
        <em>Genre:</em>
        @Model.Genre.Name
    </p>
    <p>
        <em>Artist:</em>
        @Model.Artist.Name
    </p>
    <p>
        <em>Price:</em>
        @String.Format("{0:F}",
Model.Price)

```



```

    </p>
    <p class="button">
        @Html.ActionLink("Add to
cart", "AddToCart",
        "ShoppingCart", new { id = Model.AlbumId }, "")
    </p>
</div>

```

localhost:2265/ShoppingCart/AddToCart/1

your logo here

Review your cart:

[Checkout>>](#)

Album WellKnownAttributes.NamePrice (each)Quantity

Caravan	9.99	1	Remove from cart
Total			9.99

Subiendo Archivos al Servidor

Esta tarea es especialmente sencilla desde que Html ya nos proporciona elementos con los cuales podemos auxiliarnos para desplegar cajas de dialogo para selección de archivos. Esta tarea es muy sencilla la haremos a continuación basándonos en dos restricciones.

- 1.- El archivo a subir solo puede ser de extensión .jpeg
- 2.- El archivo a subir no puede pesar más de 1 MB

Comenzaremos agregando un controlador llamado Archivos, al cual de momento no agregaremos nada. Seguido a esto lo que haremos será agregar la vista de nuestro controlador la cual contendrá lo siguiente:

```

{
    ViewBag.Title = "Archivos";
}

```

```

@using (Html.BeginForm("Index", "Archivos", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
    <h3>@ViewBag.Message</h3>
    <h2>Subir archivos</h2>
    <label for="file">Archivo: </label>
    <input type="file" name="file" id="file"/>
    @Html.ValidationMessage("Tipo")
    <input type="submit" value="Subir" />
}

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

```

Aquí solo destacaremos el ValidationMessage, el cual desplegará los errores posibles a las 2 restricciones que mencionamos al comienzo, las cuales serán enviadas al modelo desde el controlador una vez que trabajemos el flujo del archivo.



En este punto marcaría error al subir un archivo, pues aún no tenemos hecho el método que tratará el Flujo del archivo seleccionado. A continuación agregaremos el correspondiente código al controlador de ArchivosController.

```

//
// GET: /Archivos/

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

[HttpPost]
public ActionResult Index(HttpPostedFileBase file)
{

```

```

char[] chars={};
try//Se comprueba que haya un archivo
{
    StreamReader stream = new StreamReader(file.InputStream);
    chars = new Char[file.InputStream.Length];

}
catch { ModelState.AddModelError("Tipo", "Seleccione primero un arch
ivo");return View();}

if (chars.Count() < 1000024) //Archivos menores a 1 MB
{
    if (System.IO.Path.GetExtension(file.FileName) == ".jpeg" )
    {
        var fileName = Path.GetFileName(file.FileName);
        var path = "~/App_Data/"+file.FileName;
        file.SaveAs(Server.MapPath(path));
        ViewBag.Message = "Archivo subido con exito!!";
    }
    else{ ModelState.AddModelError("Tipo", "Solo se admiten archivo
s tipo JPEG");}

}
else { ModelState.AddModelError("Tipo", "Solo se admiten archivos me
nores a 1 MB"); }

//Al final regresamos a la Vista de Archivos
return View();
}

```

Aquí claramente lo destacable es el Index de tipo POST, el cual recibe el flujo y de primera mano verifica si realmente el usuario selecciono un archivo antes de dar clic al botón subir, de no haber archivo pasamos al modelo un error con identificador Tipo.

Subir archivos

Archivo:

No se eligió archivo

Seleccione primero un achivo

Lo segundo que se hace después de ver que hay un archivo es si el archivo es menor a 1 MB, de no ser así muestra su error correspondiente.

Subir archivos

Archivo:

No se eligió archivo

Solo se admiten archivos menores a 1 MB

Finalmente después de verificar que el archivo en efecto pesa menos de 1 MB, verificamos si tiene la extensión permitida para la subida al servidor, de no cumplirse muestra su error correspondiente.

Subir archivos

Archivo:

No se eligió archivo

Solo se admiten archivos tipo JPEG

Y en el caso contrario, es decir, que todo lo anterior se cumpla al 100% veríamos un mensaje diciéndonos que todo ha ido bien.

Archivo subido con éxito!!

Subir archivos

Archivo:

No se eligió archivo

Envío de E-mail a través de smtp.live.com

Este tema es especialmente interesante, pues a pesar de lo que podrías estar pensando, esta tarea es más sencilla de lo que te imaginas. El protocolo a usar en este ejemplo será el smtp de Windows Live.

En los protocolos SMTP lo único que necesitamos son credenciales validas que nos permitan acceder a los servicios de mensajería del mismo. La tarea de envío de correos electrónicos se simplifica mucho mas, ya que .Net nos proporciona una librería a través de la cual podemos hacer uso de estos servicios de una manera muy sencilla (**System.IO.Mail**).

Para el siguiente ejemplo primero crearemos una clase dentro de la carpeta de Models llamada **SendMailRequest**.

```
public class SendMailRequest
{
    [Required]
    public string From { get; set; }

    [Required]
    public string To { get; set; }

    [Required]
    public string Subject { get; set; }

    [DataType(DataType.MultilineText), Required]
    public string Body { get; set; }
}
```

Este modelo será usado para pasar valores de nuestro Controlador a la Vista y viceversa. Una vez que tenemos el modelo listo, agregaremos el controlador en la carpeta de Controllers y lo llamaremos **MailController**.

```
//
// GET: /Mail/

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

[HttpPost]
public ActionResult Index(SendMailRequest mail)
{
    if (ModelState.IsValid)
    {
        //Definimos credenciales a usar
        string email = "cuenta@hotmail.com";
        string password = "contraseña de cuenta";
        var loginInfo = new NetworkCredential(email, password);
    }
}
```

```

var mail = new MailMessage();
//Servicio que enviara el correo
var SmtServer = new SmtpClient("smtp.live.com");
SmtServer.Port = 587;

/*Aquí llenamos el correo que va a
 * ser enviado con los valores contenidos
 * en el objeto SendMailRequest enviado desde la vista*/
mail.From = new MailAddress(maill.From );
mail.To.Add(maill.To );
mail.Subject =maill.Subject ;
mail.Body = maill.Body ;

mail.IsBodyHtml = true;
SmtServer.DeliveryMethod = SmtpDeliveryMethod.Network;
SmtServer.EnableSsl = true;
SmtServer.UseDefaultCredentials = false;
SmtServer.Credentials = loginInfo;
//Enviamos correo
SmtServer.Send(mail);

}

return RedirectToAction("Index");
}

```

Y listo!!, así de fácil es habilitar el envío de correos electrónicos a través de un SMTP.