



TI-Capital Humano

Desarrollador .Net

Manual del Curso



Contenido

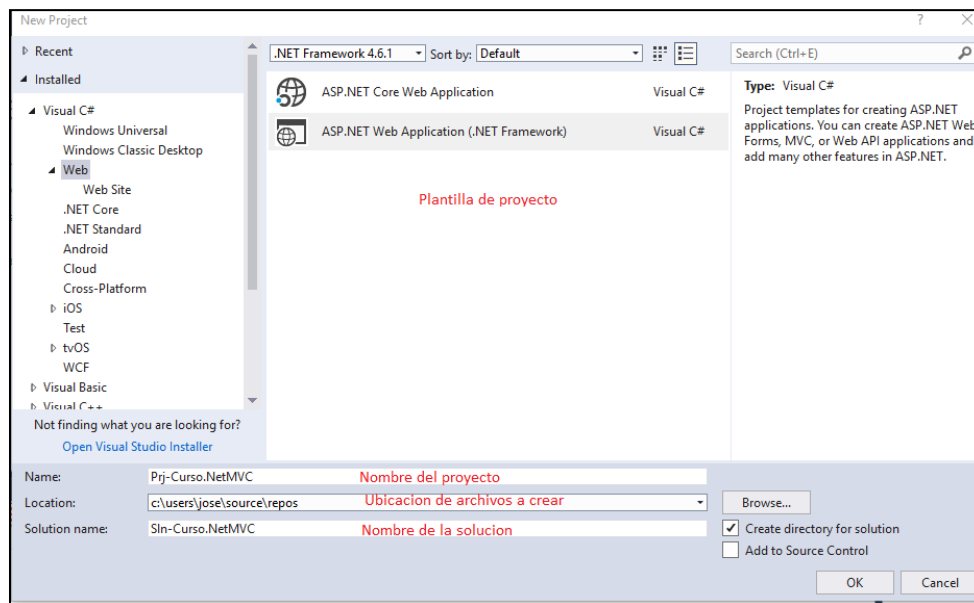
1	ASP .NET – MVC.....	4
1.1	Crear una aplicación Web MVC (.Net Framework 4.6.1)	4
1.2	Anatomía de una aplicación Web MVC (.Net Framework 4.6.1)	5
3.2	Configuración de una aplicación Web MVC en capas.	20
1.2.1	Implementando nuestro proyecto en capas.....	20
1.2.2	Agregar Dependencias entre proyectos	22
1.2.3	Agregar Referencias entre proyectos.	23
1.3	Agregar Modelo de datos con EntityFramework.	25
1.3.1	¿Qué es EntityFramework?	25
3.4.2	Características de EntityFramework.....	25
3.4.3	Enfoques de desarrollo con EntityFramework.....	26
3.4.4	Crear nuestro modelo de datos, enfoque Database First.....	28
3.4.5	Clases de contexto y entidad	31
3.4.6	Separar entidades del modelo de datos.....	32
3.4.7	Actualización del modelo de datos	35
1.4	Implementar patrón Repositorio a nuestras entidades.....	36
1.4.1	Crear interface IRepository.....	36
1.4.2	Crear clase basee implementar interface IRepository.	37
1.4.3	¿Qué son y para que nos sirven las consultas Linq?	41
1.4.4	¿Qué son y para que nos sirven las expresiones Lambda?	44
1.4.5	Implementar BaseRepository y crear métodos para CRUD entidad Usuarios.....	45
1.5	Front-End para entidad Alumnos.	47
1.5.1	Agregar un Controlador	47
1.5.2	Crear modelo para entidad Usuario.....	53
1.5.3	implementandoS en el modelo	54
1.5.4	Tipos de Vistas en MVC	55
1.5.5	Crear vistas para entidad Usuario	56
1.5.6	Guardar en la Base de Datos.	58
1.5.7	Utilizar HelperHtml.DropDownList en Razor	59
1.5.8	ASP.NET MVC – ViewBag	61

1.5.9	Añadir vista sin modelo.....	63
1.5.10	¿Qué son y para qué sirven las propiedades de navegación en EntityFramework?	65
1.5.11	Añadir y utilizar un archivo CSS	66
1.5.12	Añadir y utilizar un archivo JavaScripts	69
1.5.13	Ventana modal con Bootstrap.	71
1.6	WCF como capa servicio.	73
1.6.1	Crear un servicio WCF	77
1.6.2	Probar el servicio.	77
1.6.3	Modificando el servicio.	79

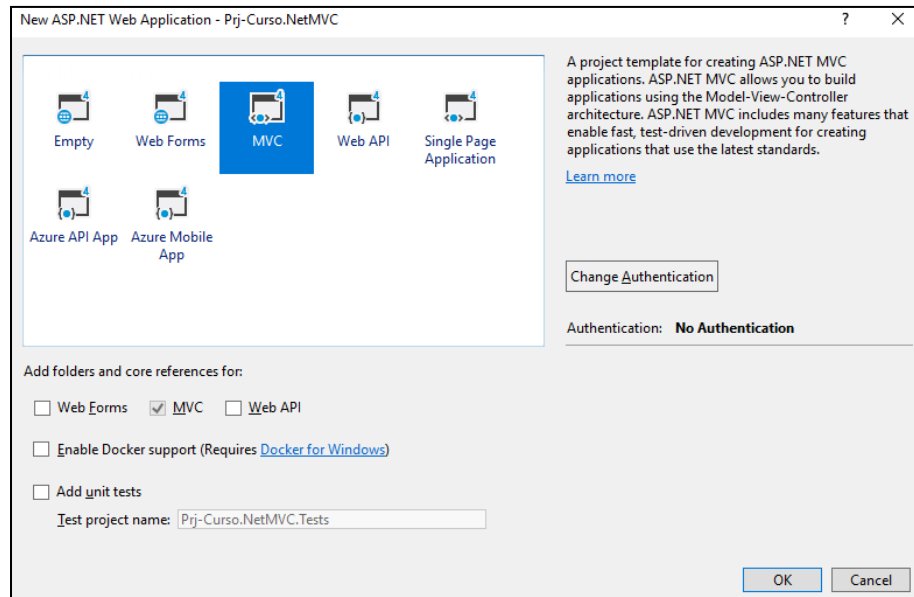
1 ASP .NET – MVC.

1.1 Crear una aplicación Web MVC (.Net Framework 4.6.1)

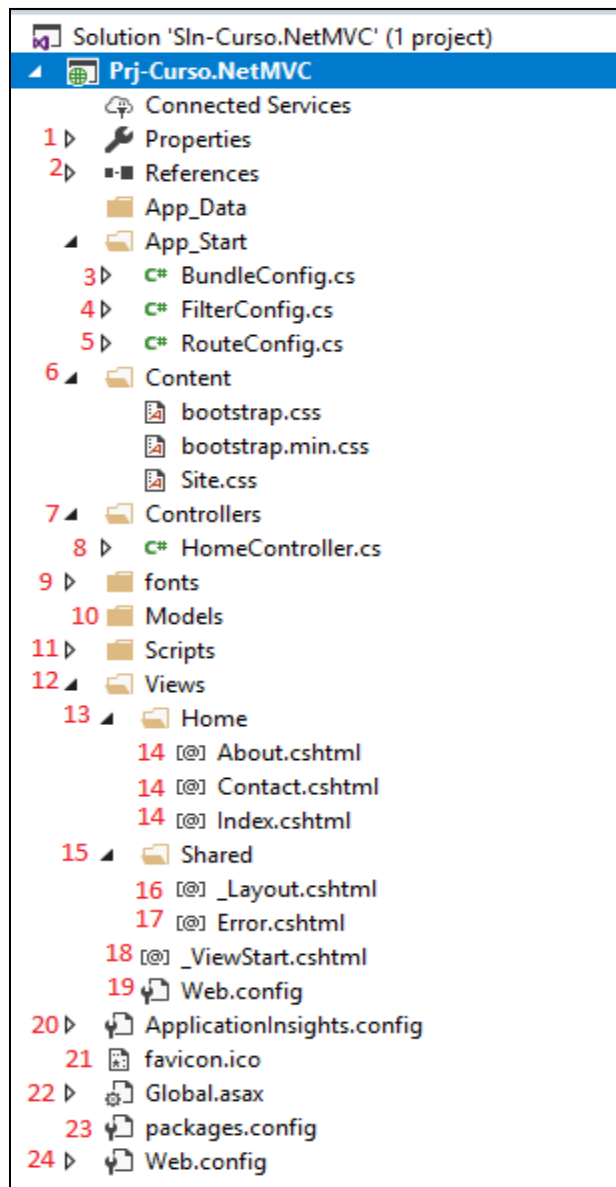
Para crear una aplicación web MVC, abrir Visual Studio e ir a Archivo/Nuevo/Proyecto, posteriormente Web/ASP.NET Application en la versión del .Net Framework en 4.6.1 seguidamente del nombre la solución, nombre del proyecto y la ruta donde se guardarán los archivos del proyecto como se muestra en la imagen.



Posteriormente seleccionar la opción de aplicación MVC, en autenticación elegir sin autenticación y por ultimo OK.



1.2 Anatomía de una aplicación Web MVC (.Net Framework 4.6.1)



1. Properties

Mediante un archivo llamado AssemblyInfo, contiene los atributos como el nombre del producto, la descripción, la marca registrada y los derechos de autor. En general, esta información es de código fijo o se almacena en una base de datos o un archivo plano. El ensamblado .NET proporciona almacenar esta información en el archivo AssemblyInfo y después de la compilación se convierte en parte del ensamblaje. Así que en el tiempo de ejecución uno puede leer esta información.

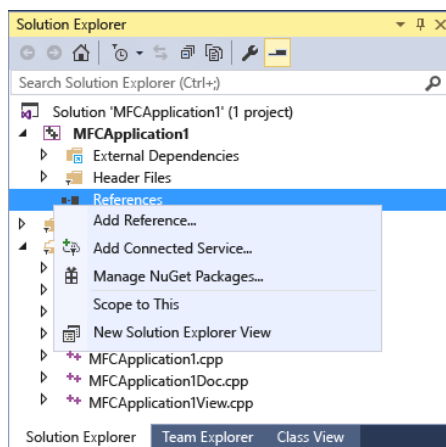
Parte de Información del Ensamblado

- Título de Asamblea: Nombre del título del conjunto.
- AssemblyDescription: Proporciona la descripción detallada del ensamblaje.

- Empresa de montaje: proporciona la información de la empresa desde el montaje.
- Producto de ensamblaje: proporciona la información de producción del ensamblaje.
- AssemblyCopyright: Proporciona los derechos de autor de la asamblea.
- AssemblyTrademark: proporciona la marca registrada del ensamblaje.

2- References.

Antes de escribir código en un componente externo o en un servicio conectado, el proyecto debe contener primero una referencia a él. Una referencia es básicamente una entrada de un archivo de proyecto que contiene la información que Visual Studio necesita para localizar el componente o el servicio. Para agregar una referencia, haga clic con el botón derecho en el nodo **Referencias** o **Dependencias** del **Explorador de soluciones** y elija **Agregar referencia**. También puede hacer clic con el botón derecho en el nodo del proyecto y seleccionar **Agregar > Referencia**. Para obtener más información, vea [Adición o eliminación de referencias](#).



Puede agregar una referencia a los siguientes tipos de componentes y servicios:

- Bibliotecas de clases o ensamblados de .NET Framework
- Aplicaciones para UWP
- componentes COM
- Otros ensamblados o bibliotecas de clases de proyectos de la misma solución
- servicios Web XML

3- BundleConfig.cs

Un Bundle es una agrupación de enlaces a archivos externos desde el HTML. Es algo habitual. Para evitar una tarea tan repetitiva se han creado estos Bundles, que son

accesos directos a agrupaciones de este tipo de archivos. Por ejemplo, este conjunto de llamadas a scripts de javascript:

```
<scriptsrc="~/Scripts/jquery-2.1.1.min.js"></script>
<scriptsrc="~/Scripts/jquery-ui-1.11.1.js"></script>
<scriptsrc="~/Scripts/jquery.validate.min.js"></script>
<scriptsrc="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
```

Puede simplificarse a un Bundle como este:

```
@Scripts.Render("~/bundles/validadoresJquery")
```

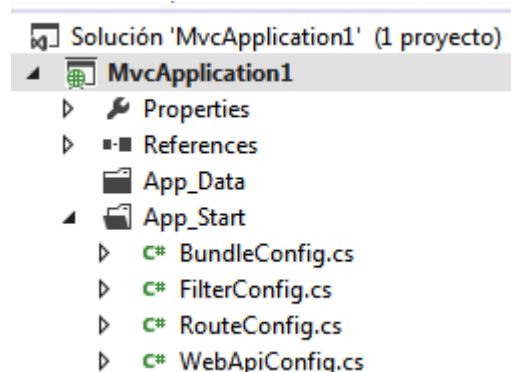
La ventaja simplificando el código es clara. En este caso solo hay cuatro scripts, pero podría haber docenas. Además, no solo se pueden agrupar scripts, sino también Hojas de estilo. La sintaxis Razor aquí es prácticamente igual, en lugar de usar **@Scripts** para agrupar los tags **script**, se usa **@Styles** para los **link**.

```
@Scripts.Render("~/bundles/validadoresJquery")
```

```
@Styles.Render("~/bundles/cssCabeceras")
```

¿Dónde se configuran?

Puedes crear muchos Bundles, conteniendo numerosos scripts u hojas de estilo. Todos ellos se construyen dentro del fichero **BundleConfig.cs** en el **App_Start**.



4- FilterConfig.cs

En ASP.NET MVC, los controladores definen métodos de acción que generalmente tienen una relación de uno a uno con las posibles interacciones del usuario, como hacer clic en un enlace o enviar un formulario. Por ejemplo, cuando el usuario hace clic en un enlace, una solicitud se enruta al controlador designado y se llama al método de acción correspondiente.

A veces desea realizar una lógica antes de llamar a un método de acción o después de que se ejecute un método de acción. Para soportar esto, ASP.NET MVC proporciona filtros. Los filtros son clases personalizadas que proporcionan medios declarativos y programáticos para agregar comportamiento de acción previa y acción posterior a los métodos de acción del controlador.

Tipos de filtro MVC de ASP.NET

ASP.NET MVC admite los siguientes tipos de filtros de acción:

- Filtros de autorización. Estos implementan [IAuthorizationFilter](#) y toman decisiones de seguridad sobre si ejecutar un método de acción, como realizar la autenticación o validar las propiedades de la solicitud. La clase [AuthorizeAttribute](#) y la clase [RequireHttpsAttribute](#) son ejemplos de un filtro de autorización. Los filtros de autorización se ejecutan antes que cualquier otro filtro.
- Filtros de acción. Estos implementan [IActionFilter](#) y envuelven la ejecución del método de acción. La interfaz [IActionFilter](#) declara dos métodos: [OnActionExecuting](#) y [OnActionExecuted](#). [OnActionExecuting](#) se ejecuta antes del método de acción. [OnActionExecuted](#) se ejecuta después del método de acción y puede realizar un procesamiento adicional, como proporcionar datos adicionales al método de acción, inspeccionar el valor de retorno o cancelar la ejecución del método de acción.
- Filtros de resultados. Estos implementan [IResultFilter](#) y envuelven la ejecución del objeto [ActionResult](#). [IResultFilter](#) declara dos métodos: [OnResultExecuting](#) y [OnResultExecuted](#). [OnResultExecuting](#) se ejecuta antes de que se ejecute el objeto [ActionResult](#). [OnResultExecuted](#) se ejecuta después del resultado y puede realizar un procesamiento adicional del resultado, como modificar la respuesta HTTP. La clase [OutputCacheAttribute](#) es un ejemplo de un filtro de resultados.
- Filtros de excepción. Estos implementan [IExceptionFilter](#) y se ejecutan si se produce una excepción no controlada durante la ejecución del canal de ASP.NET MVC. Los filtros de excepción se pueden usar para tareas como registrar o mostrar una página de error. La clase [HandleErrorAttribute](#) es un ejemplo de un filtro de excepción.

La clase [Controller](#) implementa cada una de las interfaces de filtro. Puede implementar cualquiera de los filtros para un controlador específico anulando el método `On <Filter>` del controlador. Por ejemplo, puede anular el método [OnAuthorization](#). El controlador simple incluido en el ejemplo descargable anula cada uno de los filtros y escribe información de diagnóstico cuando se ejecuta cada filtro. Puede implementar los siguientes métodos en `<Filter>` en un controlador:

- [En autorizacion](#)
- [OnException](#)
- [OnActionExecuting](#)
- [OnActionExecuted](#)
- [OnResultExecuting](#)
- [OnResultEjecutado](#)

Filtros provistos en ASP.NET MVC

ASP.NET MVC incluye los siguientes filtros, que se implementan como atributos. Los filtros se pueden aplicar a nivel de método de acción, controlador o aplicación.

- [AuthorizeAttribute](#) . Restringe el acceso mediante autenticación y opcionalmente autorización.
- [HandleErrorAttribute](#) . Especifica cómo manejar una excepción lanzada por un método de acción.

Nota

Este filtro no detecta excepciones a menos que el elemento customErrors esté habilitado en el archivo Web.config.

- [OutputCacheAttribute](#) . Proporciona almacenamiento en caché de salida.
- [RequireHttpsAttribute](#) . Hace que las solicitudes HTTP no seguras se reenvíen a través de HTTPS.

5- RouteConfig.cs

Hablando acerca de las rutas que se generan para los proyectos de MVC, si venimos de un entorno de desarrollo de web forms, notaremos que al acceso a los recursos se realizan de una forma diferente, ya que en web forms, prácticamente la ruta de la página era la dirección física de la página o recurso, esto en MVC cambia, ya que de acuerdo al formato de ruta o routing que tengamos especificado, se mostrara una acción u otra, a continuación, en ejemplo de esto:

En webforms para ver un elemento podríamos tener un URL como la siguiente:

MiPrimeraAplicacionMVC/mostrarelemento.aspx?id=U1

Con MVC podríamos tener una ruta como la siguiente

MiPrimeraAplicacionMVC/elemento/mostrar/U1

La ruta en mvc está obedeciendo el patron siguiente:

{Controlador} / {Acción} / {Id}

Lo que realizará el sistema, será entrar al controlador "elemento", y ejecutará la función "mostrar", y tomará como parámetro id el valor "U1", esta definición de patrones de la URL la encontramos dentro de nuestro proyecto en la carpeta de "App_Start", el archivo se llama "RouteConfig.cs"

Es en este archivo, donde se definen todos los formatos de las rutas, por default en los proyectos nuevos, siempre aparecerá el que te he mostrado, los parámetros de definición del formato de la ruta son los siguientes:

Name: En este parámetro definimos el nombre del formato de la ruta

URL: Aquí especificamos el formato

Default: definimos el objeto que tomara los valores por defecto del formato de la ruta, por ejemplo, en la definición de la captura, de dice que, por default, se va entrar al controlador "Home" y ejecutar la función "Index", y se le indica que el parámetro ID es opcional, por lo tanto no hace falta definirlo

Una vez entendido lo anterior, podríamos crear fácilmente otro formato de ruta, por ejemplo, si requiriéramos que se acceda directamente a una función en el controlador "Home" que se llame "MostrarElemento", con un id como parámetro, pero que no se accediera con la url `MiPrimeraAplicacionMVC/Home/MostrarElemento/u1` si no con la url `MiPrimeraAplicacionMVC/Mostrar/u1`, tendríamos que tener la clase de la siguiente manera:

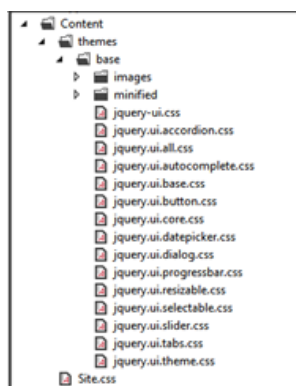
```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Nueva",
            url: "Mostrar/{id}",
            defaults: new { controller = "Home", action = "MostrarElemento" }
        );

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

6- Content.

El directorio Content está pensado para el contenido estático de la aplicación, especialmente útil para archivos css e imágenes asociadas.



ASP.NET MVC nos ofrece por defecto una organización en base a "temas", que nos permita personalizar el aspecto visual de nuestra aplicación de forma fácil y rápida (eso

en teoría... elaborar un tema visual puede requerir de mucho trabajo y la colaboración de un diseñador gráfico, algo que recomiendo siempre que sea posible).

También es el lugar apropiado para los archivos .less en el caso de que utilicemos esta tecnología en nuestra aplicación.

7- Controller

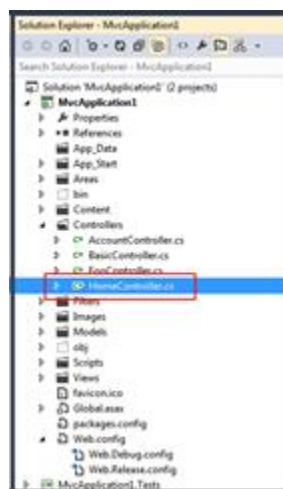
El directorio controllers es el lugar para los controladores, el marco de ASP.NET MVC asigna direcciones URL a las clases a las que se hace referencia como *controladores*. Los controladores procesan solicitudes entrantes, controlan los datos proporcionados por el usuario y las interacciones y ejecutan la lógica de la aplicación adecuada. Una clase de controlador llama normalmente a un componente de vista independiente para generar el marcado HTML para la solicitud.

La clase base para todos los controladores es la clase [ControllerBase](#), que proporciona el control general de MVC. La clase [Controller](#) hereda de ControllerBase y es la implementación predeterminada de un controlador. La clase Controller es responsable de las fases del procesamiento siguientes:

- Localizar el método de acción adecuado para llamar y validar que se le puede llamar.
- Obtener los valores para utilizar como argumentos del método de acción.
- Controlar todos los errores que se puedan producir durante la ejecución del método de acción.
- Proporcionar la clase [WebFormViewEngine](#) predeterminada para representar los tipos de página ASP.NET (vistas).

8- HomeController.

La clase HomeController es el punto de entrada de la aplicación, la página por defecto, cuando creamos un nuevo proyecto ASP.NET MVC se crea también un controlador HomeController situado directamente el folder *Controllers*.



El controlador de ejemplo que se incluye en el proyecto por defecto, incluye tres métodos: *Index*, *About* y *Contact*, que disponen de sus correspondiente vistas en el folder *Home*.

9- Fonts

El directorio fonts está pensado para ubicar los archivos utilizados para las fuentes e iconos que utilizarías en una aplicación, por default la plantilla viene cargada con los fonts de bootstrap.

10-Models

El directorio models es la ubicación que nos propone ASP.NET MVC para las clases que representan el modelo de la aplicación, los datos que gestiona nuestra aplicación. El modelo es la parte de MVC que implementa la lógica del dominio. En términos simples, esta lógica se utiliza para manejar los datos que se pasan entre la base de datos y la interfaz de usuario (UI). El modelo se conoce como objeto de dominio o entidad de dominio. Los objetos de dominio se almacenan en la carpeta Modelos en ASP.NET. El modelo de dominio representa la perspectiva de la aplicación para los datos que se manejarán, mientras que se requiere un modelo de vista para producir el motor que genera la Vista.

11- Scripts

El directorio scripts está pensado para ubicar los archivos de javascript (*.js). El código javascript es ejecutado en el contexto del navegador, es decir, en la parte cliente, y nos permite ejecutar acciones sin necesidad de enviar los datos al servidor.



ASP.NET MVC incluye varias librerías de javascript por defecto:

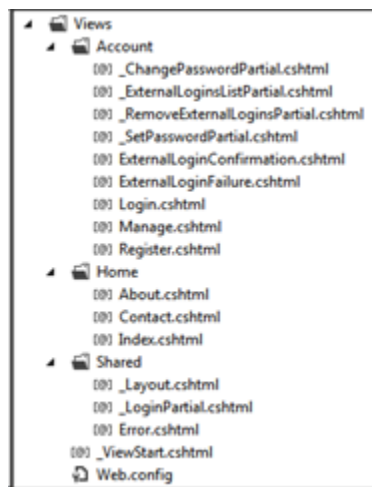
- **jquery.js.** Esta popular y súper útil librería nos va a permitir gestionar fácilmente peticiones [AJAX](#), manipular el DOM en cliente, etc ... Esta librería se ha convertido en un framework muy popular, y es la base de para la creación de [pluggins](#) que nos van a permitir dotar a nuestro sitio web de efecto sorprendentes sin apenas esfuerzo. <http://jquery.com/>

- **jquery.spin.js**. Este plugin nos permite crear animaciones de indicación de carga de contenidos – el circulito que da vueltas para indicarnos que la página esta realizando algún proceso . Es un plugin de jQuery basado en spin.js. <http://fgnass.github.io/spin.js/>
- **jquery.validate.js**. Este plugin nos permite realizar validaciones en el lado cliente fácilmente. Esta vinculado al uso de decoradores en el modelo de nuestra aplicación.
- **knockout.js**. Esta librería nos permite utilizar el patrón de diseño MVVM (Model View ViewModel), que introdujo Microsoft para el desarrollo con WPF y Silverlight en aplicaciones web con javascript. <http://knockoutjs.com/>
- **modernizr.js**. Esta librería nos permite validar fácilmente si el navegador que esta ejecutando la página web es compatible con HTML5, en caso de que no sea así proporcionar un mecanismo alternativo (*polyfill*). <http://modernizr.com/>. Por ejemplo, si nuestro navegador es compatible con HTML5 interpretará sin problema la etiqueta *VIDEO*, pero en el caso de que estemos navegando con un navegador antiguo deberemos ofrecer al usuario un mecanismo alternativo para el video (un flash por ejemplo).

Nota: El nombre de los archivos puede variar dependiendo de la versión instalada, por ejemplo el archivo jQuery.js se llama jquery-1.7.1.js, ya que corresponde con la versión 1.7.1 de la librería.

12- Views

El directorio *Views* contiene los archivos de vista. Como explicamos en la introducción al patrón MVC los controladores devuelven vistas sobre las que inyectamos el modelo de nuestra aplicación. Estas vistas son interpretadas por el motor de renderización – *Razor* en nuestro caso. Son archivos similares a aplicaciones de ASP clásico, donde tenemos código HTML estático y determinadas zonas de código que son ejecutadas en el servidor.



El siguiente ejemplo muestra el clásico “Hola Mundo”.

@{

Layout = "~/Views/Shared/_Layout.cshtml";

```
ViewBag.Title = "HelloWorld";  
}  
<h2>HelloWorld</h2>  
@*El código se servidor se especifica con el caracter @*@
```

Podemos asignar un modelo a la vista a través de la siguiente directiva.

```
@model MvcApplication1.Models.FooModel
```

13-Home

Directorio Home corresponde al controlador HomeController por cada controlador se creara su propia carpeta y a la vez por cada vista en esta carpeta se creara un archivo .cshtml.

14- Vistas creadas por default.

Archivos de vistas creadas por la plantilla del Proyecto elegido.

15- Shared

El directorio Shared contiene vistas que van a ser reutilizadas en otras vistas. Veremos cómo incluir vistas - denominadas parciales – en otra vista cuando veamos Razor, aunque de manera muy breve diremos que se realiza a través del *Helper Html*, de la siguiente forma:

```
@Html.Partial("Error") @*Incluye la vista parcial Error.cshtml, del directorio Shared*@
```

En este caso es muy importante respetar la ubicación de los archivos, ya que cuando desde una vista hagamos la llamada `@Html.Partial("Error")` para incluir la vista de la pantalla de error, el motor buscara en el directorio Shared para encontrar la vista Error.cshtml.

No debemos confundir una vista compartida con los controles .ascx de ASP.NET WebForms. Cuando incluimos vistas compartidas en otra vista, estas son interpretadas por RAZOR sin ejecutarse ningún controlador. Simplemente se renderiza su contenido – no como en los controles ascx donde se ejecuta todo el ciclo de vida del control completo y sus correspondientes eventos.

Es posible utilizar una vista compartida de un modo muy similar al de un control ascx de ASP.NET WebForms, aunque para este debemos forzar la ejecución del controlador a través de los métodos Action y RenderAction del *Helper Html*:

```
@Html.Action("About") @*Ejecuta el método "About" del controlador correspondiente*@
```

16- _Layout.cshtml

El archivo _Layout.cshtml define el layout de la aplicación, que contiene la estructura general de documento, que es reutilizada en el resto de vistas. El archivo _Layout.cshtml se encuentra dentro del directorio Views/Shared. El contenido del archivo layout por defecto se muestra a continuación:

```
<!DOCTYPEhtml>
<html lang="en">
<head>
<metacharset="utf-8" />
<title>@ViewBag.Title - My ASP.NET MVC Application</title>
<link href="/favicon.ico" rel="shortcut icon" type="image/x-icon" />
<metaname="viewport" content="width=device-width" />
@Styles.Render("~/Content/css")
@Scripts.Render("~/bundles/modernizr")

</head>
<body>
<header>
<div class="content-wrapper">
<div class="float-left">
<p class="site-title">@Html.ActionLink("your logo here", "Index", "Home")</p>
</div>
<div class="float-right">
<section id="login">
@Html.Partial("_LoginPartial")
</section>
<nav>
<ul id="menu">
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("About", "About", "Home")</li>
<li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
</nav>
</div>
</div>
</header>
<div id="body">
@RenderSection("featured", required: false)
<section class="content-wrapper main-content clear-fix">
@RenderBody()
</section>
```



```
</div>
<footer>
<div class="content-wrapper">
<div class="float-left">
<p>&copy; @DateTime.Now.Year - My ASP.NET MVC Application</p>
</div>
</div>
</footer>
@Scripts.Render("~/bundles/jquery")
@RenderSection("scripts", required: false)
</body>
</html>
```

Fijemonos en la llamada que hace Razor al método `@RenderBody()`, es ahí donde se procesará la vista que estemos mostrando.

Podemos tener múltiples archivos de layout dentro de nuestro proyecto.

17- Error.cshtml.

Pagina de error por default, layout que la aplicacion web cargara en caso de ocurrir un error al estar ejecutandose la aplicacion.

18- _ViewStart.cshtml

Este archivo establece el layout por defecto de las páginas. El contenido del archivo es sencillo y únicamente especifica el archivo de layout.

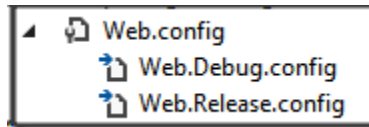
```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

El layout es un archivo con extension `.cshtml` que contiene la estructura general de documento, que es reutilizada en el resto de vistas. De este modo evitamos tener que reescribir el código en todas las vistas, reutilizando el código y permitiendo que este sea mucho más sencillo de mantener.

En cierto modo es similar a las *MasterPages* de ASP.NET WebForms.

19- Web.Config

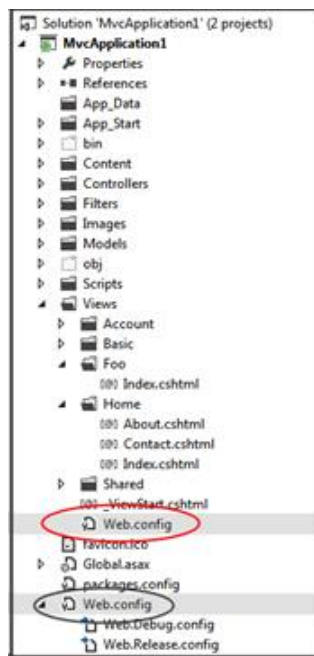
El archivo `web.config` es el archivo principal de configuración de ASP.NET. Se trata de un archivo XML donde se define la configuración de la aplicación. Veremos poco a poco el contenido de este fichero, aunque vamos a ver aquí algunas características generales que es necesario conocer.



Para simplificar el despliegue de las aplicaciones, el archivo de configuración se presenta en diferentes versiones o “sabores”, de forma que podemos modificar el archivo de configuración dependiendo de nuestra configuración de despliegue. Si observamos estos archivos observaremos que contienen transformaciones XML que se aplican sobre el archivo web.config al desplegar la aplicación.

De este modo cuando desplegamos la aplicación en modo *Debug* se aplicarán las transformaciones XML definidas en *Web.Debug.config*.

El archivo de configuración aplica un mecanismo de jerarquía a nivel de los archivos de configuración, en la que un archivo de mayor profundidad dentro de la jerarquía sobrescribe al de menor, en el contexto del recurso solicitado.



Por ejemplo, si dentro de nuestra aplicación tenemos un archivo web.config sobre la raíz del sitio (en gris sobre la imagen), y otro sobre la carpeta Views (en rojo), el archivo de configuración de la carpeta Views prevalece y sobrescribe el valor del archivo de configuración raíz (siempre que el recurso solicitado sea se encuentre el directorio, en este ejemplo *Views*).

De esta forma podemos por ejemplo establecer en nuestro sitio web “areas” públicas y privadas de manera muy sencilla, tan fácil separar los archivos es directorios e como incluir un archivo de configuración con la seguridad activada sobre el directorio privado, y otro accesible a todo el mundo en el directorio privado.

20- ApplicationInsights.config

El SDK de .NET de Application Insights consta de varios paquetes de NuGet. El paquete central proporciona la API para enviar telemetría a Application Insights. Los paquetes adicionales proporcionan módulos de telemetría e inicializadores para rastrear automáticamente la telemetría desde su aplicación y su contexto. Al ajustar el archivo de configuración, puede habilitar o deshabilitar los módulos de telemetría y los inicializadores, y establecer parámetros para algunos de ellos.

El nombre del archivo de configuración ApplicationInsights.config o ApplicationInsights.xml, dependiendo del tipo de su aplicación. Se agrega automáticamente a su proyecto cuando instala la mayoría de las versiones del SDK. Status Monitor también lo agrega a una aplicación web en un servidor IIS, o cuando selecciona la extensión Application Insights para un sitio web de Azure o VM.

21- Favicon.ico

Archivo que representa el icono de la aplicación web, es agregado automáticamente al crear el Proyecto plantilla de Visual Studio.

22- Global.asax

Toda aplicación ASP.NET MVC es una instancia de una clase derivada de System.Web.HttpApplication. Esta clase es el punto de entrada de nuestra aplicación – el *Main* de la aplicación web por decirlo de alguna manera. Como podemos observar, la clase base es la misma que para una aplicación ASP.NET clásica (WebForms) por lo que todos que podemos reutilizar todo lo que ya sabíamos de ASP.NET.

Desde este archivo podemos manejar eventos a nivel de aplicación, sesión, cache, autenticación, etc ...

Este archivo varía mucho desde la versión anterior de ASP.NET MVC, aunque el funcionamiento es el mismo. En ASP.NET MVC 4 se ha incluido el directorio App_Start que nos permite organizar como se inicializa la aplicación.

```
// Por defecto, el namespace corresponde con el nombre del proyecto
namespace MvcApplication1
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
```

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

Como podemos ver, el código se encarga de registrar las áreas definidas en el proyecto, cargar la configuración, filtros, etc..., en definitiva, se encarga de configurar la aplicación para que esta pueda ejecutar de manera correcta. Las clases de configuración se encuentran en el directorio App_Start

23-packages.config

El packages.config archivo se usa en algunos tipos de proyectos para mantener la lista de paquetes referenciados por el proyecto. Esto permite que NuGet restaure fácilmente las dependencias del proyecto cuando el proyecto se transporte a una máquina diferente, como un servidor de compilación, sin todos esos paquetes.

Si se usa, packages.config normalmente se encuentra en una raíz de proyecto. Se crea automáticamente cuando se ejecuta la primera operación NuGet, pero también puede crearse manualmente antes de ejecutar cualquier comando como nuget restore.

Los proyectos que usan PackageReference no usan packages.config.

El esquema es simple: seguir el encabezado XML estándar es un <packages>nodo único que contiene uno o más <package>elementos, uno para cada referencia.

3.2 Configuración de una aplicación Web MVC en capas.

La mayoría de aplicaciones .NET tradicionales se implementan como unidades únicas que se corresponden a un archivo ejecutable o una sola aplicación web que se ejecuta dentro de un único dominio de aplicación de IIS. Este es el modelo de implementación más sencillo y sirve muy bien a muchas aplicaciones internas y públicas más pequeñas. Pero incluso con esta única unidad de implementación, la mayoría de las aplicaciones de negocio importantes se aprovechan de cierta separación lógica en varias capas.

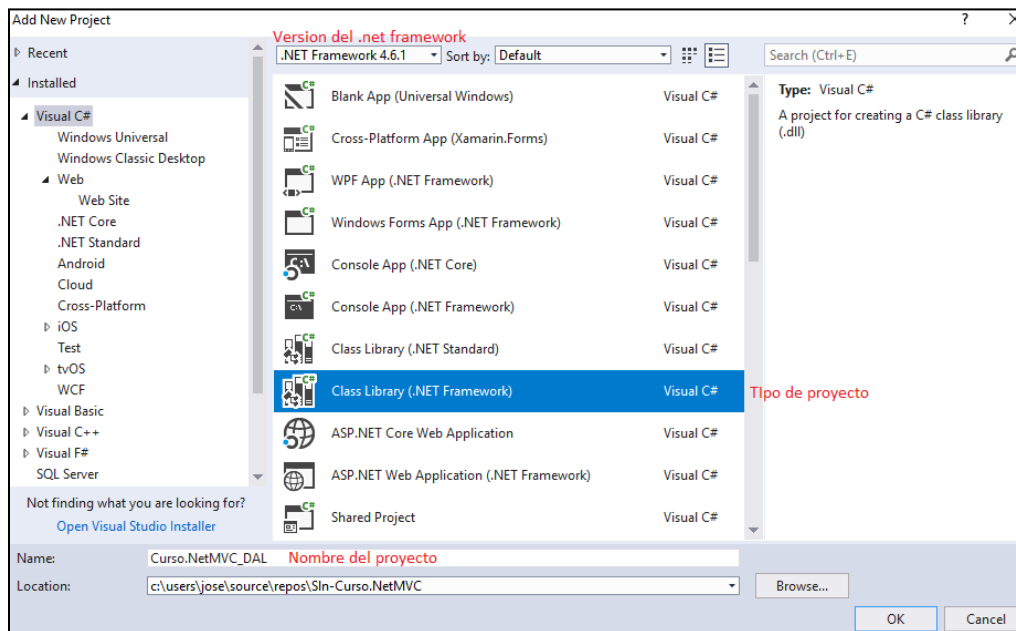
1.2.1 Implementando nuestro proyecto en capas

Para que nuestra aplicación obtenga la estructura en capas, agregaremos 3 nuevos proyectos de tipo librería de clases de .NET Framework y con los siguientes nombres:

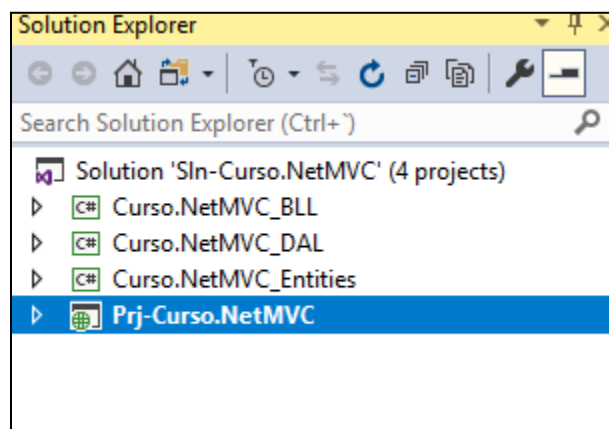
- Curso.NetMVC_DAL
- Curso.NetMVC_BLL
- Curso.NetMVC_Entities

Iniciamos seleccionando nuestra solución >**clic derecho>agregar>nuevo proyecto**.

Es importante tener en cuenta el tipo de proyecto (**Class Library .NET Framework**) la versión del .net Framework sea igual a la versión con la que se está trabajando en la aplicación web y por último la ubicación donde se guardara el proyecto sea la misma donde está la aplicación.



Siguiendo los pasos anteriores repetiremos hasta haber creado los 3 proyectos ya antes mencionados, es importante tomar en cuenta que no hay un orden específico.



Nuestra solución debe de quedar como la imagen anterior, ahora solo falta agregar dependencias y referencias entre los proyectos.

1.2.2 Agregar Dependencias entre proyectos

Cuando se compila una solución que contiene varios proyectos, a veces es necesario compilar primero algunos proyectos, para generar el código que utilizan otros proyectos. Cuando un proyecto utiliza código ejecutable generado por otro proyecto, se hace referencia al proyecto que genera el código como una dependencia de proyecto del proyecto que utiliza el código. Tales relaciones de dependencia pueden definirse en el cuadro de diálogo de Dependencias del proyecto.

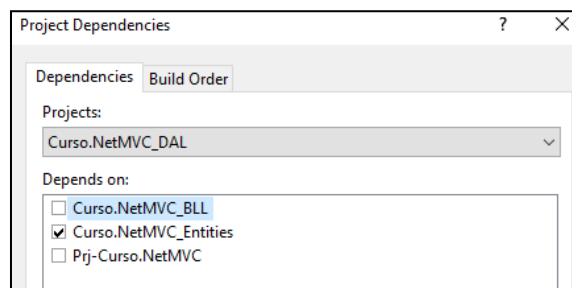
Para asignar dependencias a proyectos

1. En el Explorador de soluciones, seleccione un proyecto.
2. En el menú **Proyecto**, elija **Dependencias del proyecto**.
Se abrirá el cuadro de diálogo **Dependencias del proyecto**.
3. En la ficha **Dependencias**, seleccione un proyecto en el menú desplegable **Proyecto**.
4. En el campo **Depende de**, active la casilla de todos los proyectos que deben compilarse antes que el actual.

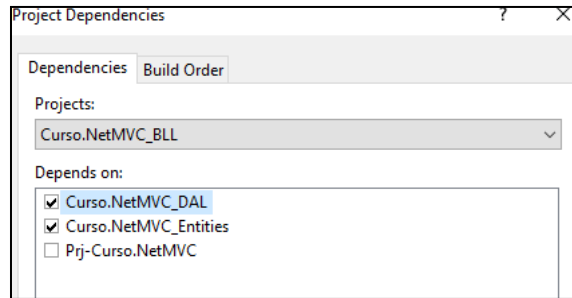
Para poder crear dependencias de proyecto, la solución debe estar compuesta por más de un proyecto.

Siguiendo los pasos anteriores agregaremos las dependencias nuestros proyectos ya creados quedando de la siguiente forma para cada uno.

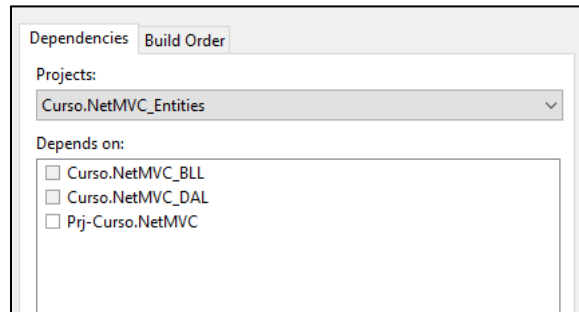
- Curso.NetMVC_DAL



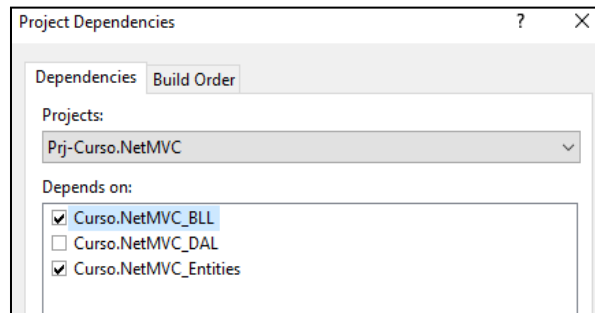
- Curso.NetMVC.BLL



- Curso.NetMVC_Entities



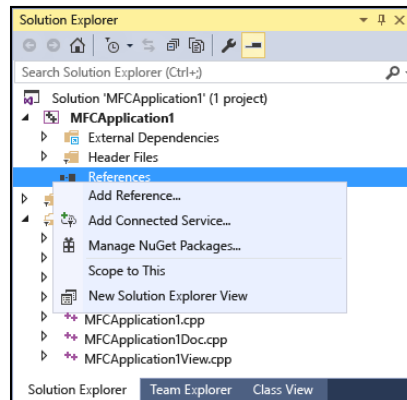
- Prj-Curso.NetMVC



1.2.3 Agregar Referencias entre proyectos.

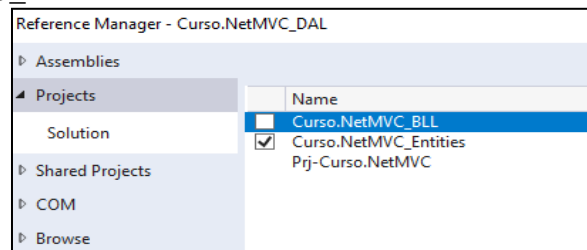
Antes de escribir código en un componente externo o en un servicio conectado, el proyecto debe contener primero una referencia a él. Una referencia es básicamente una entrada de un archivo de proyecto que contiene la información que Visual Studio necesita para localizar el componente o el servicio.

Para agregar una referencia, haga clic con el botón derecho en el nodo Referencias del Explorador de soluciones y elija **Agregar referencia**.

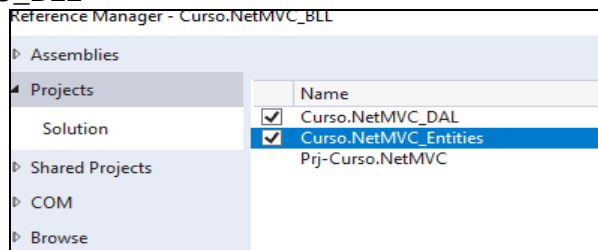


Siguiendo los pasos anteriores agregaremos las referencias entre los proyectos quedando de la siguiente forma:

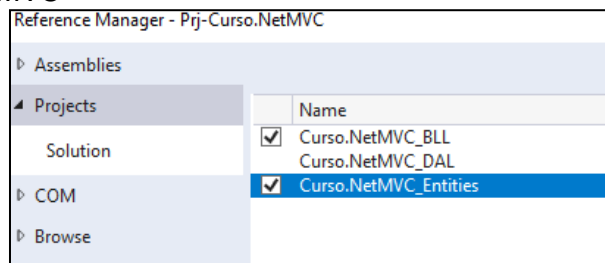
- Curso.NetMVC_DAL



- Curso.NetMVC_BLL



- Curso.NetMVC_Entities (No se hace referencia a ningún proyecto)
- Prj-Curso.NetMVC



Para comprobar que todo lo hicimos bien es necesario hacer clic derecho sobre la solución y compilar, debe ser compilada sin error alguno de lo contrario algo se esta mal.

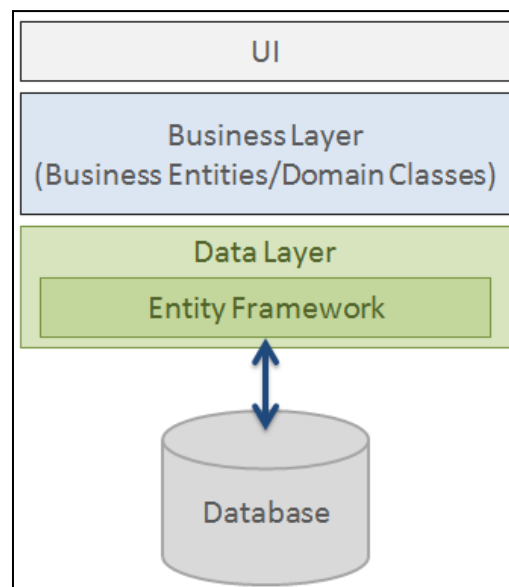
1.3 Agregar Modelo de datos con EntityFramework.

1.3.1 ¿Qué es EntityFramework?

EntityFramework es un marco de ORM de código abierto para aplicaciones .NET admitidas por Microsoft. Permite a los desarrolladores trabajar con datos utilizando objetos de clases específicas del dominio sin centrarse en las tablas y columnas de la base de datos subyacente donde se almacenan estos datos. Con el EntityFramework, los desarrolladores pueden trabajar en un nivel más alto de abstracción cuando tratan con datos, y pueden crear y mantener aplicaciones orientadas a datos con menos código en comparación con las aplicaciones tradicionales.

Definición oficial: "EntityFramework es un asignador relacional de objetos (O / RM) que permite a los desarrolladores .NET trabajar con una base de datos utilizando objetos .NET. Elimina la necesidad de la mayoría del código de acceso a datos que los desarrolladores generalmente necesitan escribir".

La siguiente figura ilustra dónde encaja Entity Framework en una aplicación.



3.4.2 Características de EntityFramework

- **Multiplataforma:** EF Core es un marco multiplataforma que puede ejecutarse en Windows, Linux y Mac.
- **Modelado:** EF (Entity Framework) crea un EDM (Entity Data Model) basado en entidades POCO (Plain Old CLR Object) con propiedades get / set de diferentes tipos de datos. Utiliza este modelo al consultar o guardar datos de la entidad en la base de datos subyacente.
- **Consultas:** EF nos permite usar consultas LINQ (C # / VB.NET) para recuperar datos de la base de datos subyacente. El proveedor de la base de datos traducirá estas consultas LINQ al lenguaje de consulta específico de la base de datos (por ejemplo, SQL para una base de datos relacional). EF también nos permite ejecutar consultas de SQL sin procesar directamente en la base de datos.

- **Seguimiento de cambios:** EF realiza un seguimiento de los cambios ocurridos en instancias de sus entidades (valores de propiedad) que deben enviarse a la base de datos.
- **Guardar:** EF ejecuta los comandos INSERT, UPDATE y DELETE en la base de datos según los cambios que se produjeron en sus entidades cuando llama al SaveChanges() método. EF también proporciona el SaveChangesAsync() método asíncrono.
- **Concurrencia:** EF usa concurrencia optimista de forma predeterminada para proteger los cambios de sobrescritura realizados por otro usuario desde que se obtuvieron los datos de la base de datos.
- **Transacciones:** EF realiza la gestión automática de transacciones al consultar o guardar datos. También proporciona opciones para personalizar la gestión de transacciones.
- **Almacenamiento en caché:** EF incluye el primer nivel de almacenamiento en caché fuera de la caja. Por lo tanto, las consultas repetidas devolverán datos del caché en lugar de golpear la base de datos.
- **Convenciones incorporadas:** EF sigue las convenciones sobre el patrón de programación de la configuración e incluye un conjunto de reglas predeterminadas que configuran automáticamente el modelo EF.
- **Configuraciones:** EF nos permite configurar el modelo de EF mediante el uso de atributos de anotación de datos o API Fluent para anular las convenciones predeterminadas.
- **Migraciones:** EF proporciona un conjunto de comandos de migración que pueden ejecutarse en la consola de NuGet Package Manager o la interfaz de línea de comandos para crear o administrar el esquema de la base de datos subyacente.

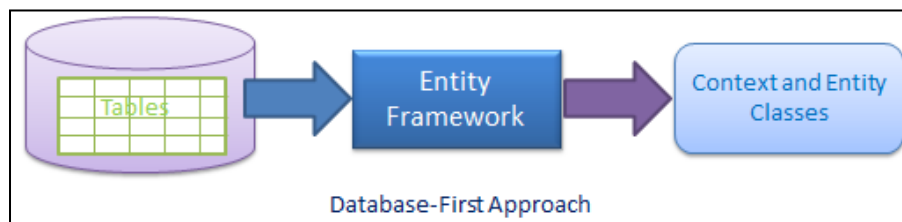
3.4.3 Enfoques de desarrollo con EntityFramework

Hay tres enfoques diferentes que puede utilizar al desarrollar su aplicación utilizando EntityFramework:

1. Base de datos primero
2. Código primero
3. Primer modelo

Primer enfoque de la base de datos

En el primer enfoque de desarrollo de la base de datos, genera el contexto y las entidades para la base de datos existente mediante el asistente EDM integrado en Visual Studio o ejecutando comandos EF.



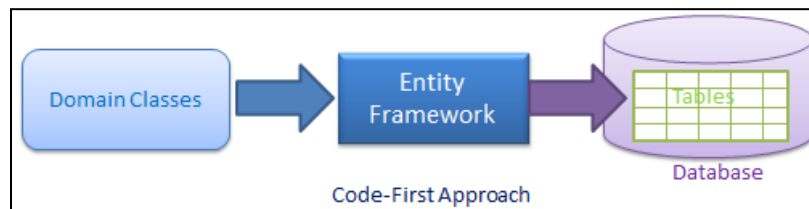
EF 6 apoya ampliamente el enfoque de base de datos. Visite EF 6 DB-Primera sección para aprender sobre el enfoque de base de datos primero utilizando EF 6.

EF Core incluye soporte limitado para este enfoque.

Código de primer acercamiento

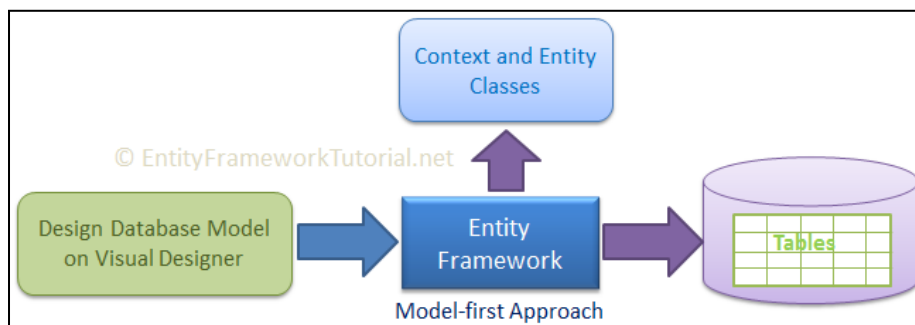
Utilice este enfoque cuando no tenga una base de datos existente para su aplicación. En el enfoque de código primero, comienza a escribir primero las entidades (clases de dominio) y la clase de contexto y luego crea la base de datos a partir de estas clases mediante los comandos de migración.

Los desarrolladores que siguen los principios del Diseño Dirigido por Dominio (DDD), prefieren comenzar con la codificación de sus clases de dominio primero y luego generar la base de datos necesaria para conservar sus datos.



Modelo de primer enfoque

En el enfoque del modelo primero, crea entidades, relaciones y jerarquías de herencia directamente en el diseñador visual integrado en Visual Studio y luego genera entidades, la clase de contexto y el script de base de datos desde su modelo visual.

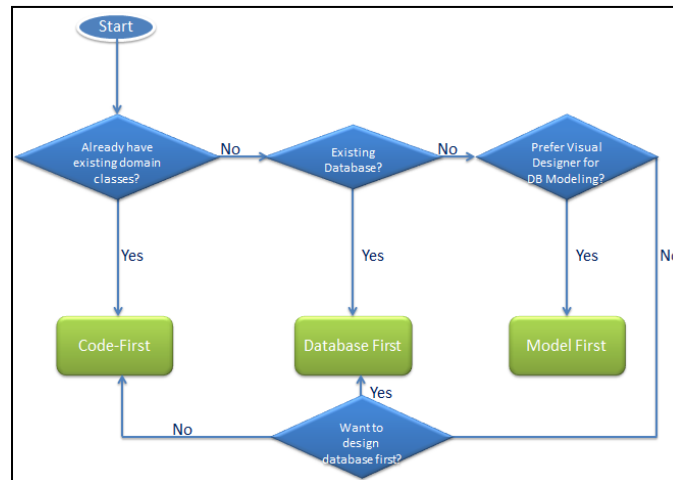


EF 6 incluye soporte limitado para este enfoque.

EF Core no apoya este enfoque.

Como elegir el enfoque de desarrollo para su aplicación

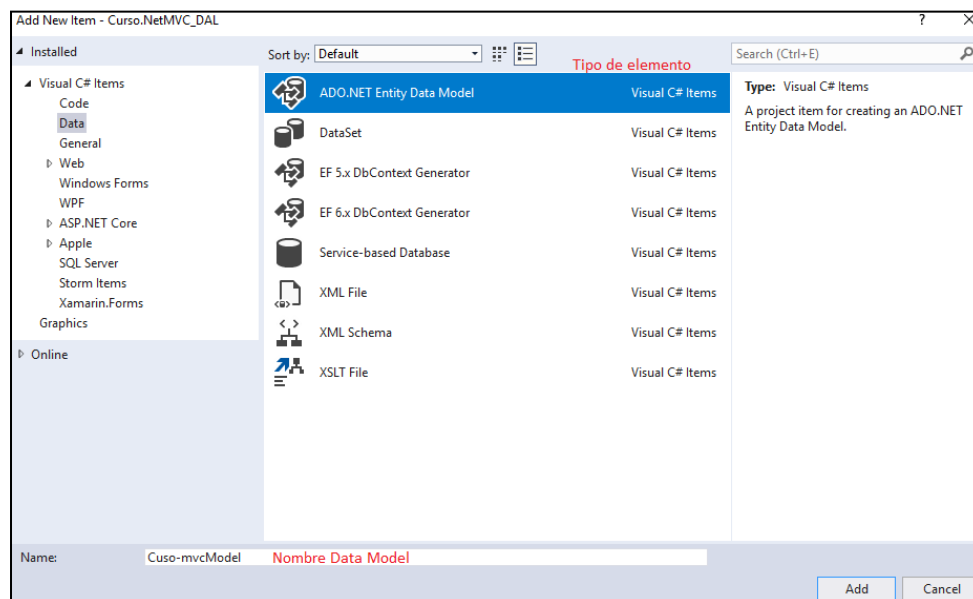
Use el siguiente diagrama de flujo para decidir cuál es el enfoque correcto para desarrollar su aplicación utilizando EntityFramework:



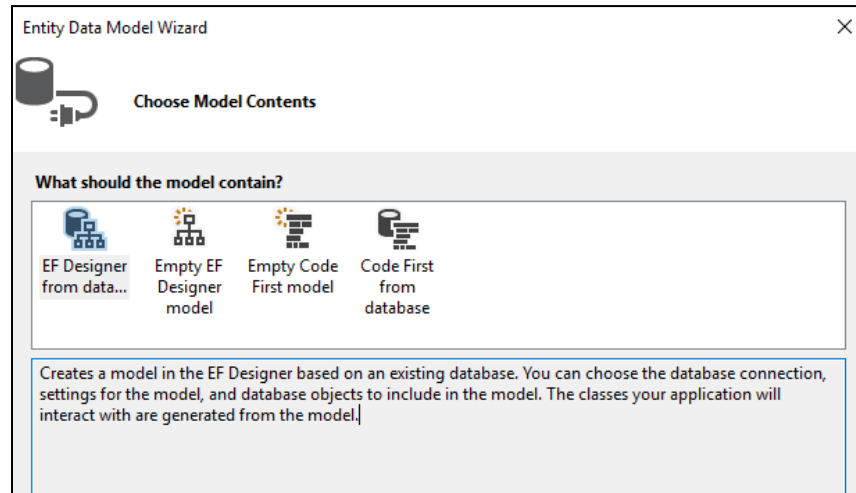
Según la figura anterior, si ya tiene una aplicación con clases de dominio, puede utilizar el enfoque de código primero porque puede crear una base de datos a partir de sus clases existentes. Si tiene una base de datos existente, entonces puede crear un EDM a partir de una base de datos existente en el enfoque de base de datos primero. Si no tiene una base de datos o clases de dominio existentes, y prefiere diseñar su modelo de base de datos en el diseñador visual, busque el enfoque Modelo primero.

3.4.4 Crear nuestro modelo de datos, enfoque Database First.

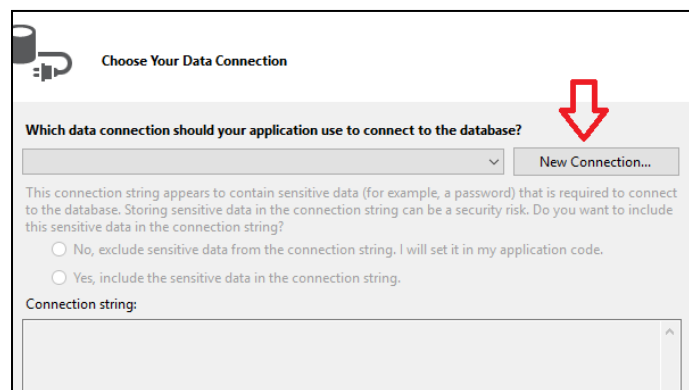
Iniciamos seleccionando nuestro proyecto de acceso a datos>**clíc derecho>agregar>nuevoelemento**, posteriormente en **Data**, el tipo de elemento **ADO.NET Entity Data Model** y por ultimo proporcionamos un nombre al modelo a crear.



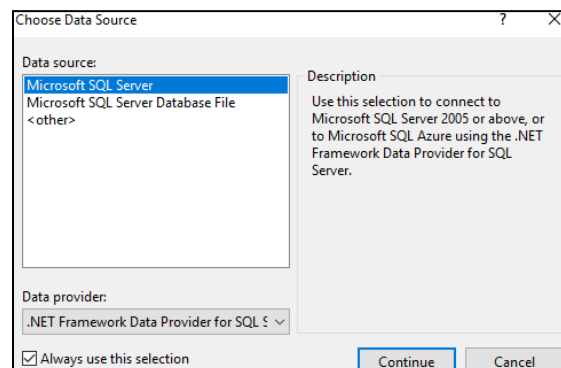
Posteriormente se debe elegir el enfoque con el cual se trabajará en nuestro caso Database firsts.



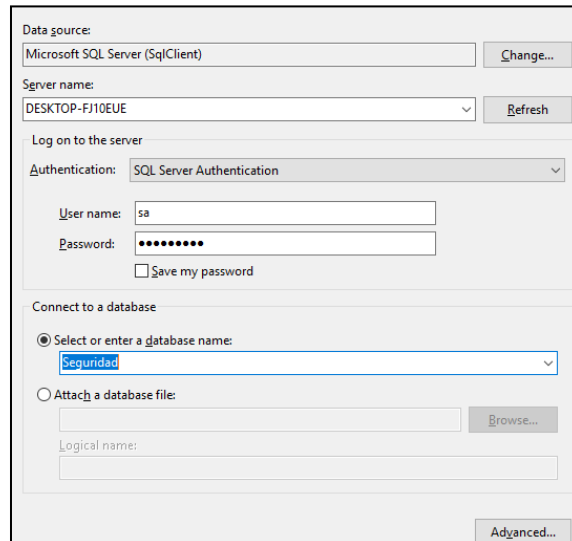
Seguidamente debemos conectar nuestro modelo a la base de datos ya construida con anterioridad.



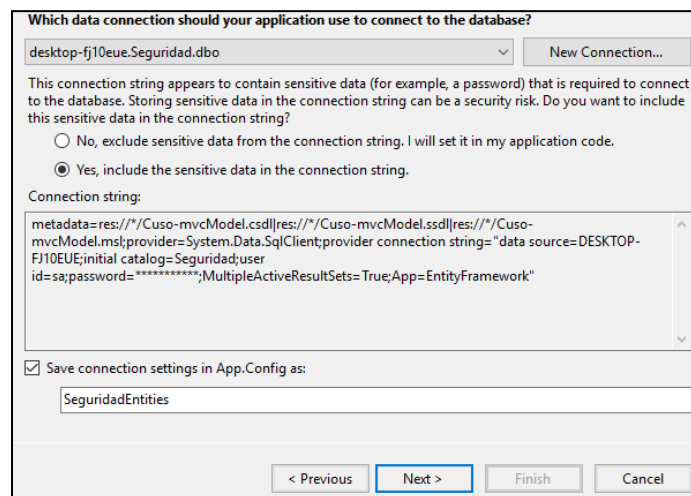
Creamos una nueva conexión a Microsoft SQL Server.



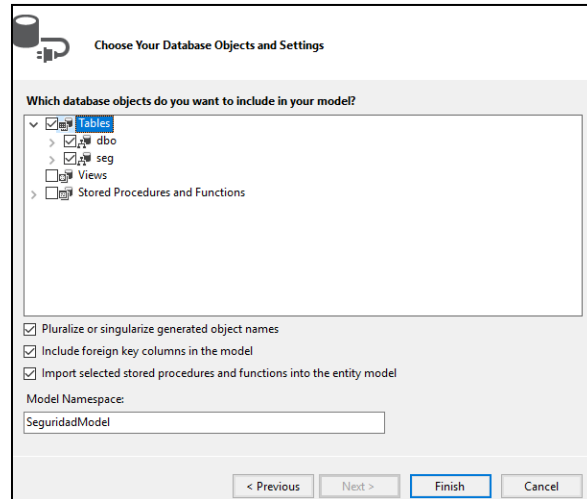
Proporcionamos las credenciales correspondientes.

This is a screenshot of the "Data source" dialog box in SQL Server Enterprise Manager. It is configured for a Microsoft SQL Server (SqlClient) connection. The server name is "DESKTOP-FJ10EUE". The authentication method is "SQL Server Authentication". The user name is "sa" and the password is masked with dots. There is a checkbox for "Save my password" which is unchecked. Under the "Connect to a database" section, the "Select or enter a database name" radio button is selected, and the database name is "Seguridad". There is also an option to "Attach a database file" which is currently unselected. An "Advanced..." button is located at the bottom right.

Seguidamente incluimos la conexión al web.Config

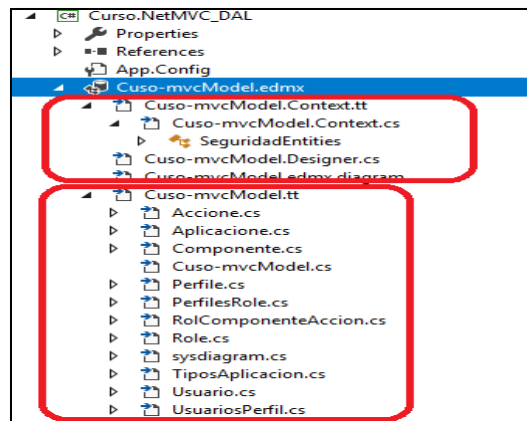
This is a screenshot of a dialog box titled "Which data connection should your application use to connect to the database?". It shows a dropdown menu with "desktop-fj10eue.Seguridad.dbo" selected. Below this, there is a warning message about sensitive data in the connection string. Two radio buttons are present: "No, exclude sensitive data from the connection string. I will set it in my application code." (unchecked) and "Yes, include the sensitive data in the connection string." (checked). The "Connection string:" text box contains the following text: "metadata=res://*/Cuso-mvcModel.csdl|res://*/Cuso-mvcModel.ssdl|res://*/Cuso-mvcModel.msl;provider=System.Data.SqlClient;provider connection string='data source=DESKTOP-FJ10EUE;initial catalog=Seguridad;user id=sa;password=*****;MultipleActiveResultSets=True;App=EntityFramework'". At the bottom, there is a checkbox "Save connection settings in App.Config as:" which is checked, and a text box containing "SeguridadEntities". Navigation buttons at the bottom include "< Previous", "Next >" (highlighted), "Finish", and "Cancel".

Por ultimo seleccionamos las tablas, vistas o stores procedures que necesitemos en nuestra aplicación.



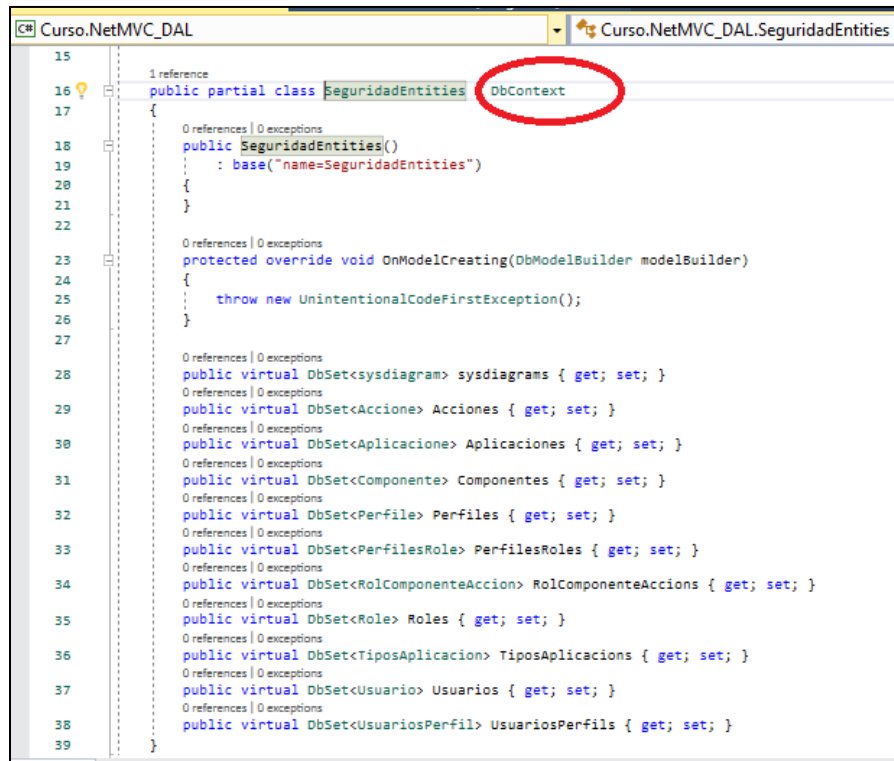
3.4.5 Clases de contexto y entidad

Cada modelo de datos de entidad genera un contexto y una clase de entidad para cada tabla de base de datos. Expanda el `.edmx` archivo en el explorador de soluciones y abra dos archivos importantes `<EDM Name>.Context.tt` y `<EDM Name>.tt`, como se muestra a continuación:



<http://www.entityframeworktutorial.net/images/EF5/create-EDM-fg8.PNG>

Cuso-mvcModel.Context.tt: este archivo de plantilla T4 genera una clase de contexto cada vez que cambia el modelo de datos de entidad (archivo `.edmx`). Puedes ver el archivo de clase de contexto expandiendo `Cuso-mvcModel.Context.tt`. La clase de contexto reside en el `<EDM Name>.context.cs` archivo. El nombre de clase de contexto predeterminado es `<DB Name>Entities`. Por ejemplo, el nombre de la clase de contexto para `Cuso-mvc` es `SeguridadEntities` y se deriva de la `DBContext` clase.



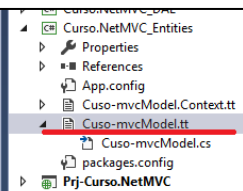
Cuso-mvcModel.tt: Cuso-mvcModel.tt es un archivo de plantilla T4 que genera clases de entidad para cada tabla de base de datos. Las clases de entidad son clases POCO (Plain Old CLR Object).

3.4.6 Separar entidades del modelo de datos.

Si bien construimos nuestra aplicación en capas y agregamos nuestro modelo de datos, hasta el momento no estamos respetando la arquitectura en cuestión, por eso es necesario aplicar una separación de las clases de entidades generadas en nuestro modelo y moverlas a nuestra capa de entidades.

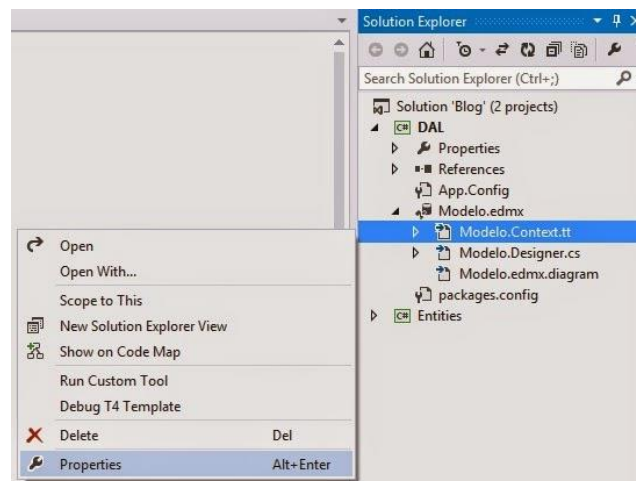
Iniciamos seleccionando nuestro proyecto de Entidades > **clic derecho>agregar>nuevoelemento**, posteriormente en **Data**, el tipo de elemento **EF 6.x DbContext Generator** y por ultimo proporcionamos el nombre, es importante que el nombre sea exactamente igual al que se generó en el modelo de datos en nuestro caso el **Cuso-mvcModel.tt**.


```
const string inputFile = @"$edmxInputFile$";  
var textTransform = DynamicTextTransformation.Create(this);  
var code = new CodeGenerationTools(this);  
var ef = new MetadataTools(this);  
var typeMapper = new TypeMapper(code, ef, textTransform);  
var fileManager = EntityFrameworkTemplateFileManager.Create();  
var itemCollection = new EdmMetadataLoader(textTransform);  
var codeStringGenerator = new CodeStringGenerator(code, typeMapper);  
  
if (!typeMapper.VerifyCaseInsensitiveTypeUniqueness(typeMapper))
```

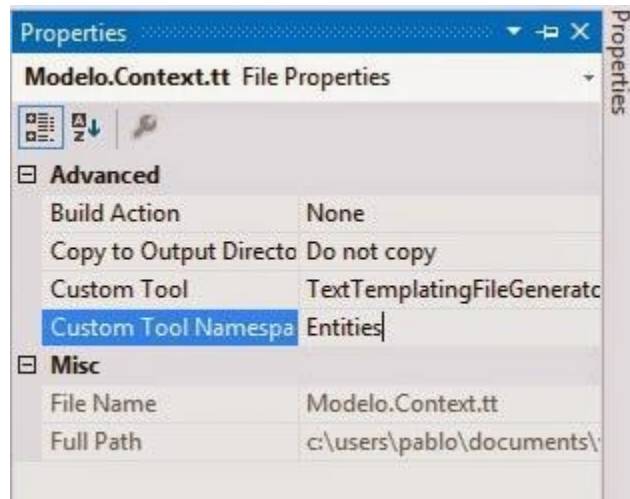


Una vez ubicada la línea señalada debemos modificar el nombre `@"$edmxInputFile$"` por la dirección donde se encuentra nuestro modelo de datos.edmx, en nuestro caso por `@"..\\Curso.NetMVC_DAL\\Curso-mvcModel.edmx"`.

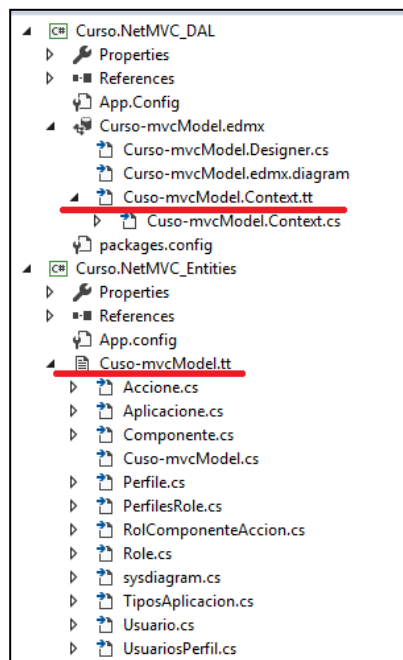
Ahora en el proyecto DAL, dar click derecho en Modelo.Context.tt y seleccionar Propiedades



Especificar el NameSpace donde se encuentra Modelo.tt



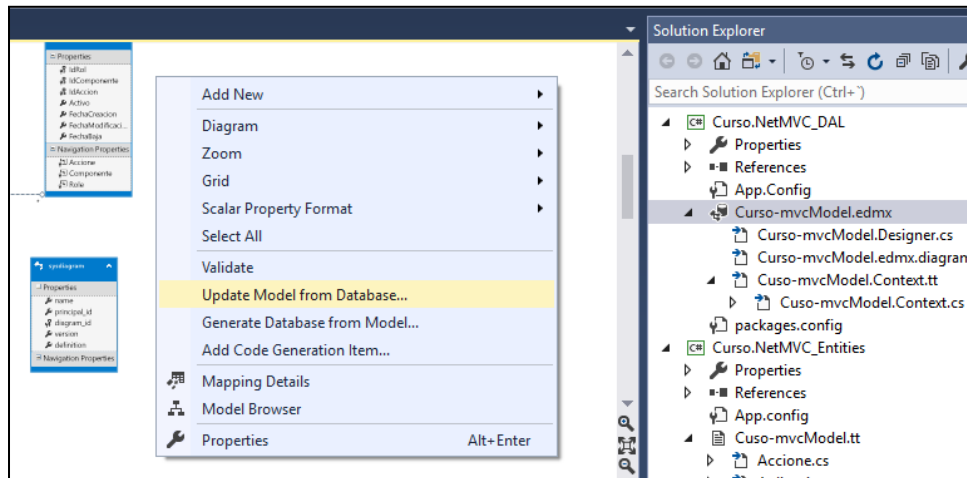
Posteriormente eliminemos en el proyecto Entities el archivo modelo.Context.tt y en el proyecto DAL el archivo con terminación modelo.tt, quedando de la siguiente forma.



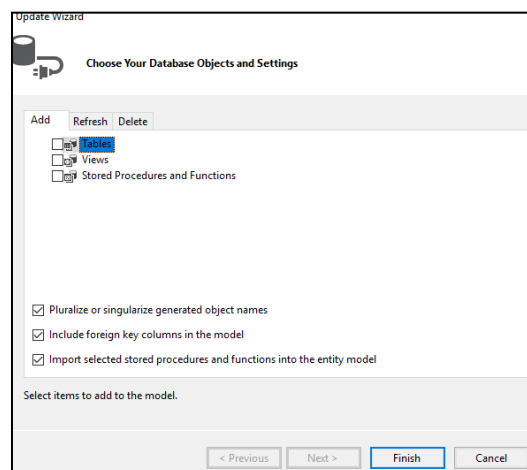
3.4.7 Actualización del modelo de datos

Ahora bien, ya tenemos nuestro modelo de datos conectado a una base de datos que se creó al inicio y tenemos separadas las entidades que representan las tablas, pero que pasa si necesitamos agregar, eliminar o actualizar tablas, vistas, funciones o procedimientos almacenados de nuestra base de datos.

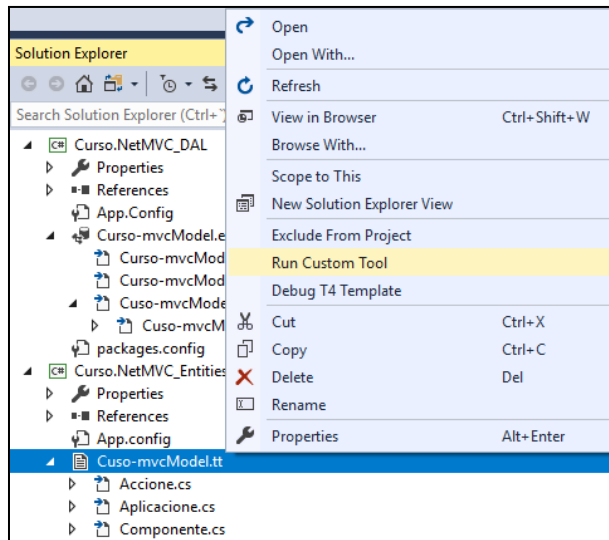
Para ello necesitamos una actualización del modelo de datos que podemos hacer abriendo el modelo de datos en el explorador de soluciones >**clic derecho dentro del modelo>Actualizar modelo desde base de datos**



Posteriormente elegimos la opción que queramos, agregar, actualizar o eliminar ya sea el caso.



Un paso muy importante, lo cual se debe de realizar cada vez que actualicemos el modelo es refrescar nuestras clases de entidades que están en el proyecto Entities, para ello nos situamos sobre el archivo model.tt de nuestro proyecto >**clic derecho>Correr herramienta personalizada**



Con esto terminamos el proceso de actualización de nuestro modelo de datos, el cual podremos realizar cada vez que necesitemos un cambio.

1.4 Implementar patrón Repositorio a nuestras entidades.

El repositorio es una Fachada (Facade) que abstrae el dominio (o capa de lógica de negocio) de la persistencia. Se comporta como una colección (como un `ICollection` o un `IList`) escondiendo los detalles técnicos de la implementación.

Una implementación podrá persistir el estado de los objetos en una base de datos relacional, en un servicio externo, en archivos XML o simplemente en memoria.

DDD (Domain Driven Design) nos dice que la interfaz del Repository vive en el dominio mientras que su implementación pertenece a la capa de acceso a datos (o infraestructura).

Y plantea usar los repositorios sólo para los Aggregate Roots. El ejemplo clásico para explicar qué es un Aggregate Root es el de Factura y LineaFactura, estas dos clases conforman un aggregate donde el primero es el aggregate root, por ello sólo tendré repositorio para la Factura, el cual se encargará de obtener las lineas correspondientes a cada factura (no habrá un repositorio para LineaFactura).

1.4.1 Crear interface IRepository.

El primer paso para implementar el patrón Repositorio es crear una interface en la cual definiremos los métodos que implementaremos en la clase base Repository, para ello seleccionamos nuestro proyecto BLL > clic derecho > agregar clase > con nombre **IRepository**.

Posteriormente modificaremos la declaración de la clase indicando que será una interface de tipo genérica y declarando los métodos que estaremos utilizando tal y cual como se muestra en el siguiente código.

```
interface IRepository<T> where T : class
{
    List<T> GetAll();

    List<T> GetAll(List<Expression<Func<T, object>>> includes);

    T Single(Expression<Func<T, bool>> predicate);

    T Single(Expression<Func<T, bool>> predicate, List<Expression<Func<T, object>>> includes);

    List<T> Filter(Expression<Func<T, bool>> predicate);

    List<T> Filter(Expression<Func<T, bool>> predicate,
        List<Expression<Func<T, object>>> includes);

    void Create(T entity);

    void Update(T entity);

    void Delete(T entity);

    void Delete(Expression<Func<T, bool>> predicate);
}
```

Por ultimo agregar el NameSpace `using System.Linq.Expressions;`

1.4.2 Crear clase basee implementar interface IRepository.

La clase BaseRepository permitirá definir las acciones sobre la persistencia que serán comunes para todas las entidades, para ello agregaremos una nueva clase en nuestro proyecto BLL >**click derecho>agregar clase>con nombre BaseRepository.**

Posteriormente modificaremos la declaración de la clase indicando que será pública abstracta de tipo genérica y que estará implementando la interface **IRepository** quedando de la siguiente forma.

```
public abstract class BaseRepository<T>: IRepository<T> where T : class
{
    SeguridadEntities context;

    public BaseRepository()
    {
        context = new SeguridadEntities();
    }

    public virtual List<T> GetAll()
    {
        return (List<T>)context.Set<T>().ToList();
    }

    public virtual List<T> GetAll(List<Expression<Func<T, object>>> includes)
    {
        List<string> includelist = new List<string>();

        foreach (var item in includes)
        {
            MemberExpression body = item.Body as MemberExpression;

            if (body == null)
            {
                throw new ArgumentException("El cuerpo debe ser miembro de una expresión.");
            }

            includelist.Add(body.Member.Name);
        }

        DbQuery<T> query = context.Set<T>();

        includelist.ForEach(x => query = query.Include(x));

        return (List<T>)query.ToList();
    }

    public virtual T Single(Expression<Func<T, bool>> predicate)
    {
        return context.Set<T>().FirstOrDefault(predicate);
    }
}
```

```
Public virtual T Single(Expression<Func<T, bool>> predicate,
List<Expression<Func<T, object>>> includes)
{
    List<string> includelist = new List<string>();
foreach (var item in includes)
{
    MemberExpression body = item.Body as MemberExpression;
if (body == null)
    thrownew ArgumentException("El cuerpo debe ser miembro de una
expresión.");includelist.Add(body.Member.Name);
}
    DbQuery<T> query = context.Set<T>();
    includelist.ForEach(x => query = query.Include(x));
return query.FirstOrDefault(predicate);
}

Public virtual List<T>Filter(Expression<Func<T, bool>> predicate)
{
return (List<T>)context.Set<T>().Where(predicate).ToList();
}

Public virtual List<T>Filter(Expression<Func<T, bool>> predicate,
List<Expression<Func<T, object>>> includes)
{
    List<string> includelist = new List<string>();
foreach (var item in includes)
{
    MemberExpression body = item.Body as MemberExpression;
if (body == null)
    thrownew ArgumentException("El cuerpo debe ser miembro de una
expresión.

includelist.Add(body.Member.Name);
}
    DbQuery<T> query = context.Set<T>();
    includelist.ForEach(x => query = query.Include(x));
```

```
return (List<T>)query.Where(predicate).ToList();
    }

    Public virtual void Create(T entity)
    {
        context.Set<T>().Add(entity);
        context.SaveChanges();
    }

    public virtual void Update(T entity)
    {
        context.Entry(entity).State = EntityState.Modified;
        context.SaveChanges();
    }

    Public virtual void Delete(T entity)
    {
        context.Entry(entity).State = EntityState.Deleted;
        context.SaveChanges();
    }

    public virtual void Delete(Expression<Func<T, bool>> predicate)
    {
        var entities = context.Set<T>().Where(predicate).ToList();
        entities.ForEach(x => context.Entry(x).State = EntityState.Deleted);
        context.SaveChanges();
    }
}
```

Declaramos una variable de tipo clase de contexto creada en el modelo de datos y en el constructor de la clase BaseRepository la inicializamos para poder ser utilizada en nuestros métodos implementados por la interface IRepository, dichos métodos debemos marcarlos como virtuales para sobre escribirlos en cada clase Repositorio por entidad que crearemos más adelante, es importante no olvidar agregar los NameSpace necesarios. Con esto solo nos falta implementar la clase BaseRepository a las clases repositorio por entidad que necesitemos.

1.4.3 ¿Qué son y para que nos sirven las consultas Linq?

Una consulta es una expresión que recupera datos de un origen de datos. Las consultas se expresan generalmente en un lenguaje de consulta especializado. Con el tiempo, se han desarrollado diferentes idiomas para los diversos tipos de fuentes de datos, por ejemplo, SQL para bases de datos relacionales y XQuery para XML. Por lo tanto, los desarrolladores han tenido que aprender un nuevo lenguaje de consulta para cada tipo de fuente de datos o formato de datos que deben admitir. LINQ simplifica esta situación al ofrecer un modelo consistente para trabajar con datos a través de varios tipos de fuentes de datos y formatos.

En una consulta LINQ, siempre está trabajando con objetos. Utiliza los mismos patrones de codificación básicos para consultar y transformar datos en documentos XML, bases de datos SQL, conjuntos de datos ADO.NET, colecciones .NET y cualquier otro formato para el que esté disponible un proveedor LINQ.

Tres partes de una operación de consulta

Todas las operaciones de consulta LINQ constan de tres acciones distintas:

1. Obtener la fuente de datos.
2. Crear la consulta.
3. Ejecutar la consulta.

El siguiente ejemplo muestra cómo las tres partes de una operación de consulta se expresan en el código fuente. El ejemplo utiliza una matriz de enteros como fuente de datos por conveniencia; sin embargo, los mismos conceptos se aplican a otras fuentes de datos también

```
classIntroToLINQ
{
    staticvoidMain()
    {
        // Las tres partes de una consulta LINQ:
        // 1. Fuente de datos.
        int[] numeros = newint[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Creacion de la consulta.
        // numQuery es un IEnumerable<int>
        var numQuery =
        from num innumeros
        where (num % 2) == 0
        select num;

        // 3. Ejecucion de la consulta.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

```
}
```

1.4.3.1 Operaciones básicas de consulta LINQ (C #)

Filtración

Probablemente la operación de consulta más común es aplicar un filtro en forma de una expresión booleana. El filtro hace que la consulta devuelva solo aquellos elementos para los cuales la expresión es verdadera. El resultado se produce mediante el uso de la where cláusula. El filtro en efecto especifica qué elementos excluir de la secuencia de origen. En el siguiente ejemplo, solo customers se devuelven aquellos que tienen una dirección en Londres.

```
var queryLondonCustomers = from cust in customers
where cust.City == "London"
select cust;
```

Puede utilizar los operadores lógicos AND y de C # familiares OR para aplicar tantas expresiones de filtro como sea necesario en la where cláusula. Por ejemplo, para devolver solo los clientes de "Londres" AND cuyo nombre es "Devon", escribiría el siguiente código:

```
where cust.City=="London"&& cust.Name == "Devon"
```

Para devolver clientes de Londres o París, debe escribir el siguiente código:

```
where cust.City == "London" || cust.City == "Paris"
```

Ordenando

A menudo es conveniente ordenar los datos devueltos. La orderby cláusula hará que los elementos en la secuencia devuelta se ordenen de acuerdo con el comparador predeterminado para el tipo que se ordena. Por ejemplo, la siguiente consulta se puede ampliar para ordenar los resultados según la Name propiedad. Debido a que Name es una cadena, el comparador predeterminado realiza una ordenación alfabética de A a Z.

```
var queryLondonCustomers3 =
from cust in customers
where cust.City == "London"
orderby cust.Name ascending
select cust;
```

Para ordenar los resultados en orden inverso, de Z a A, use la orderby...descending cláusula.

Agrupamiento

La group cláusula le permite agrupar sus resultados según una clave que especifique. Por ejemplo, puede especificar que los resultados se agrupen de City forma que todos los

clientes de Londres o París estén en grupos individuales. En este caso, cust.City es la clave.

```
// queryCustomersByCity is an IEnumerable<IGrouping<string,
Customer>>
var queryCustomersByCity =
from cust in customers
group cust by cust.City;

// customerGroup is an IGrouping<string, Customer>
foreach (var customerGroup in queryCustomersByCity)
{
    Console.WriteLine(customerGroup.Key);
    foreach (Customer customer in customerGroup)
    {
        Console.WriteLine("    {0}", customer.Name);
    }
}
```

Cuando finaliza una consulta con una group cláusula, sus resultados toman la forma de una lista de listas. Cada elemento de la lista es un objeto que tiene un Key miembro y una lista de elementos que se agrupan bajo esa clave. Cuando se itera sobre una consulta que produce una secuencia de grupos, debe usar un foreach bucle anidado. El bucle externo itera sobre cada grupo, y el bucle interno itera sobre los miembros de cada grupo.

Si debe consultar los resultados de una operación de grupo, puede usar la intopalabra clave para crear un identificador que pueda consultarse más. La siguiente consulta devuelve solo aquellos grupos que contienen más de dos clientes:

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
from cust in customers
group cust by cust.City into custGroup
where custGroup.Count() >2
orderby custGroup.Key
select custGroup;
```

Unión

Las operaciones de unión crean asociaciones entre secuencias que no se modelan explícitamente en las fuentes de datos. Por ejemplo, puede realizar una unión para encontrar a todos los clientes y distribuidores que tienen la misma ubicación. En LINQ, la join cláusula siempre funciona contra colecciones de objetos en lugar de tablas de base de datos directamente.

```
var innerJoinQuery =
from cust in customers
join dist in distributors on cust.City equals dist.City
```

```
selectnew { CustomerName = cust.Name, DistributorName = dist.Name
};
```

En LINQ no tiene que usar jointan a menudo como lo hace en SQL porque las claves externas en LINQ se representan en el modelo de objetos como propiedades que contienen una colección de elementos. Por ejemplo, un Customerobjeto contiene una colección de Orderobjetos. En lugar de realizar una unión, accede a los pedidos utilizando la notación de puntos:

```
from order in Customer.Orders...
```

1.4.4 ¿Qué son y para que nos sirven las expresiones Lambda?

Una expresión lambda es una función anónima que puede usar para crear delegados o tipos de árbol de expresiones. Al utilizar expresiones lambda, puede escribir funciones locales que pueden pasarse como argumentos o devolverse como el valor de las llamadas a funciones. Las expresiones Lambda son particularmente útiles para escribir expresiones de consulta LINQ.

Para crear una expresión lambda, especifique los parámetros de entrada (si los hay) en el lado izquierdo del operador lambda `=>`, y coloque la expresión o el bloque de declaración en el otro lado. Por ejemplo, la expresión lambda `x => x * x` especifica un parámetro que tiene nombre `x` y devuelve el valor de `x` cuadrado. Puede asignar esta expresión a un tipo de delegado, como muestra el siguiente ejemplo:

```
delegate int del(int i);

static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

Una expresión lambda con una expresión en el lado derecho del operador `=>` se denomina *expresión lambda*. Las lambdas de expresión se utilizan ampliamente en la construcción de árboles de expresión. Una expresión lambda devuelve el resultado de la expresión y toma la siguiente forma básica:

```
(input-parameters) => expression
```

Los paréntesis son opcionales solo si la lambda tiene un parámetro de entrada; De lo contrario son obligatorios. Dos o más parámetros de entrada están separados por comas entre paréntesis:

```
(x, y) => x == y
```

Declaración de Lambdas

Una declaración lambda se asemeja a una expresión lambda, excepto que la (s) declaración (es) se incluyen entre llaves:

(parámetros de entrada) => {declaración; }

El cuerpo de una declaración lambda puede constar de cualquier número de declaraciones; Sin embargo, en la práctica, normalmente no hay más de dos o tres.

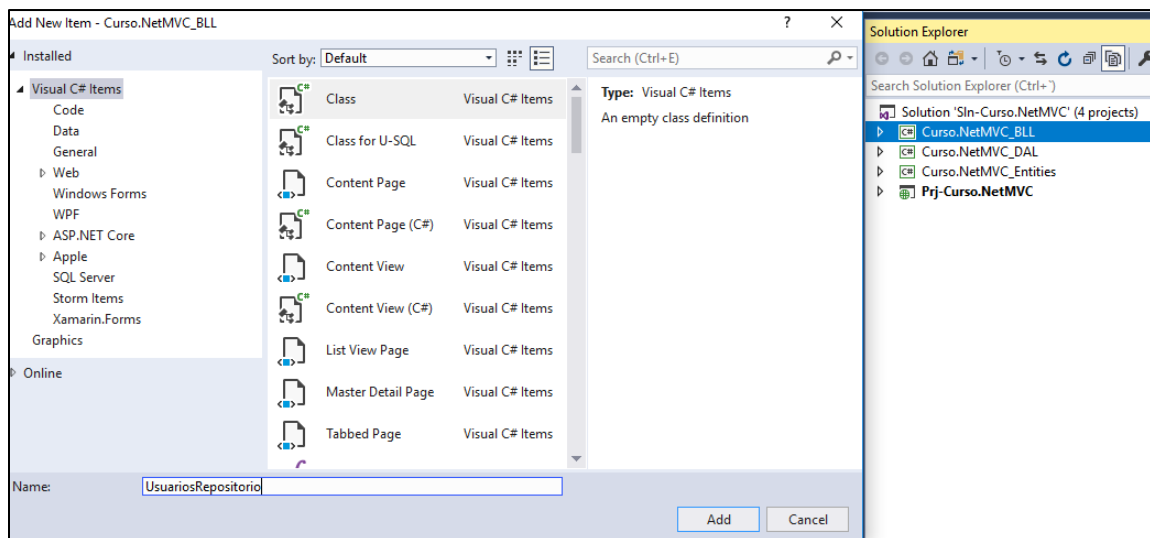
```
delegate void TestDelegate(string s);
```

```
TestDelegate del = n =>{ string s = n + " World";  
    Console.WriteLine(s); };
```

Las lambdas de declaración, al igual que los métodos anónimos, no pueden utilizarse para crear árboles de expresiones.

1.4.5 Implementar BaseRepository y crear métodos para CRUD entidad Usuarios.

Ahora bien, ya tenemos creados nuestros métodos genéricos listos para ser reutilizados en nuestra capa (proyecto) BLL, para ello tomaremos la entidad usuarios como referencia agregando en nuestro proyecto BLL una clase llamada UsuariosRepositorio.cs



Modificaremos la definición de la clase convirtiéndola en pública e implementando nuestra clase pública abstracta llamada BaseRepository pasándole como parámetro nuestra clase de la entidad Usuarios, por herencia tendremos acceso a nuestros métodos genéricos ya

antes definidos de igual forma podremos sobrescribirlos para esta clase en particular y así agregarles validaciones de datos de ser requerido.

Sobrescribiremos el método Create para ello basta solo escribir la palabra reservada override seguidamente de Create, presionamos la tecla **TAB** y por medio del IntelliSense de Visual Studio nos creara la siguiente estructura del método Create..

```
public override void Create(Usuarios entity)
{
    //Validaciones de datos, reglas de negocio o control de excepciones, logs, etc
    base.Create(entity);
}
```

Una de las ventajas que nos provee el trabajar con aplicaciones MVC es el poder tener lógica de negocio compartida en la capa de aplicación y la capa de BLL por medio de los controladores el cual veremos más adelante es decir podríamos poner validación de datos en los controladores y solo mandar a crear en este caso o validar en la capa de negocio ya dependerá de cual sea del agrado.

Haremos lo mismo con las demás entidades de nuestro modelo de datos, al final debemos tener una clase Repositorio por entidad con los métodos siguientes:

```
public override List<Usuarios> GetAll()
{
    //validaciones por realizar
    return base.GetAll();
}
public override void Delete(Usuarios entity)
{
    //validaciones por realizar
    base.Delete(entity);
}
public override void Update(Usuarios entity)
{
    base.Update(entity);
}
public override void Create(Usuarios entity)
{
    base.Create(entity);
}

public override List<Usuarios> Filter(Expression<Func<Usuarios,
bool>> predicate)
{
    return base.Filter(predicate);
}
```

Utilizando expresiones lambda en consultas linq

A modo de ejemplo de como se complementan las expresiones lambda dentro de consultas con linq crearemos un método llamado GetAvanzado el cual tendrá como parámetro un string, en dicho método buscaremos los usuarios que contengan en su

propiedad nombre dicho texto como parámetro, que sean activos y los ordenaremos descendientemente por la fecha de su creación.

```
public List<Usuarios> GetAvanzado(string text)
{
    List<Usuarios> resp;
try
    {
        resp = Filter(c => c.Nombre.Contains(text) && c.Activo == true)
                    .OrderByDescending(c => c.FechaCreacion)
                    .ToList();
    }
catch (Exception ex)
{
    throw;
}
return resp;
}
```

Como podremos observar estamos haciendo uso dos veces del operador lambda => y del lado izquierdo antepone una letra c, de acuerdo a una de las definiciones de nuestro método Filter tenemos Filter(Expression<Func<T, bool>> predicate), recibimos como argumento una expresión la cual contiene una función y dicha función recibe como argumento una clase genérica T. Al implementar la clase BaseRepository a nuestra clase UsuariosRepository le estamos indicando que esa clase genérica será de tipo Usuarios.cs nuestra entidad usuario con todas sus propiedades. Entonces del lado derecho accedemos a la propiedad Name y a través de LINQ le indicamos que contenga el texto que recibimos como argumento y al igual que la propiedad Activo sea TRUE, posteriormente ordenamos el resultado de la primera expresión al igual hacemos uso del operador lambda indicando que sean ordenados por FechaCreacion por último el resultado lo convertimos a una List de tipo Usuarios.cs

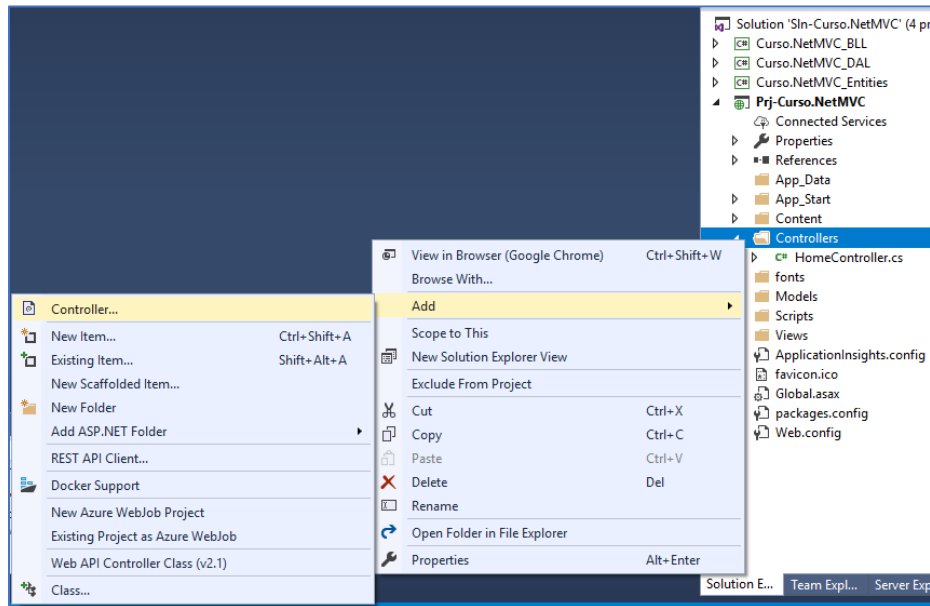
1.5 Front-End para entidad Alumnos.

MVC es el acrónimo *model-view-controller*. MVC es un patrón para desarrollar aplicaciones que están bien diseñadas, pueden probarse y son fáciles de mantener. Las aplicaciones basadas en MVC contienen:

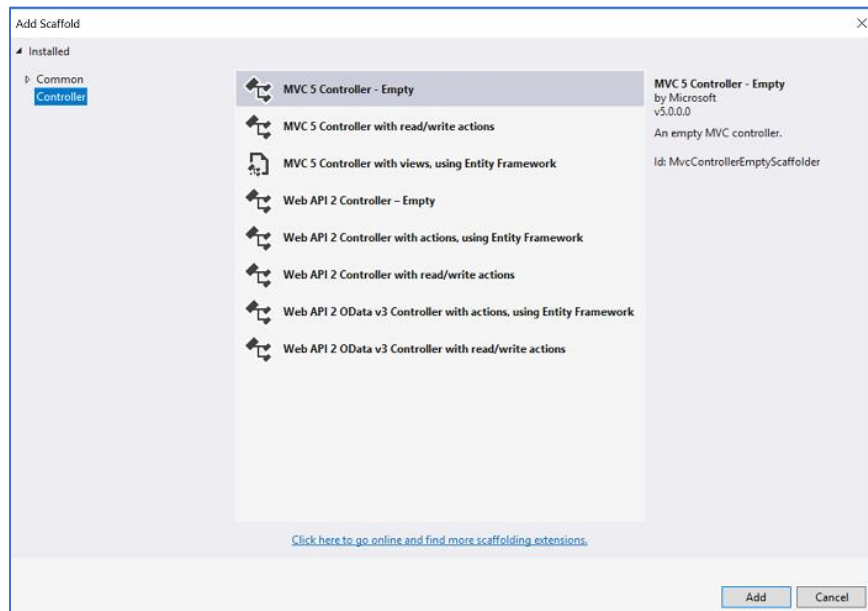
- **M**odelos: clases que representan los datos de la aplicación y que utilizan una lógica de validación para aplicar reglas de negocio para esos datos.
- **V**istas: archivos de plantilla que la aplicación usa para generar respuestas HTML de forma dinámica.
- **C**ontroladores: las clases que controlan las solicitudes entrantes del explorador, recuperan datos del modelo y, especifican las plantillas de vista que devuelven una respuesta al explorador.

1.5.1 Agregar un Controlador

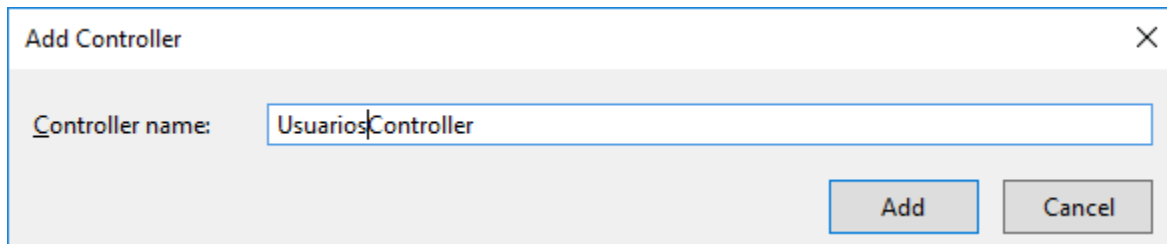
Comencemos para crear una clase de controlador. En el Explorador de soluciones, haga clic en la carpeta **Controllers** y, a continuación, haga clic en agregar, a continuación, controlador.



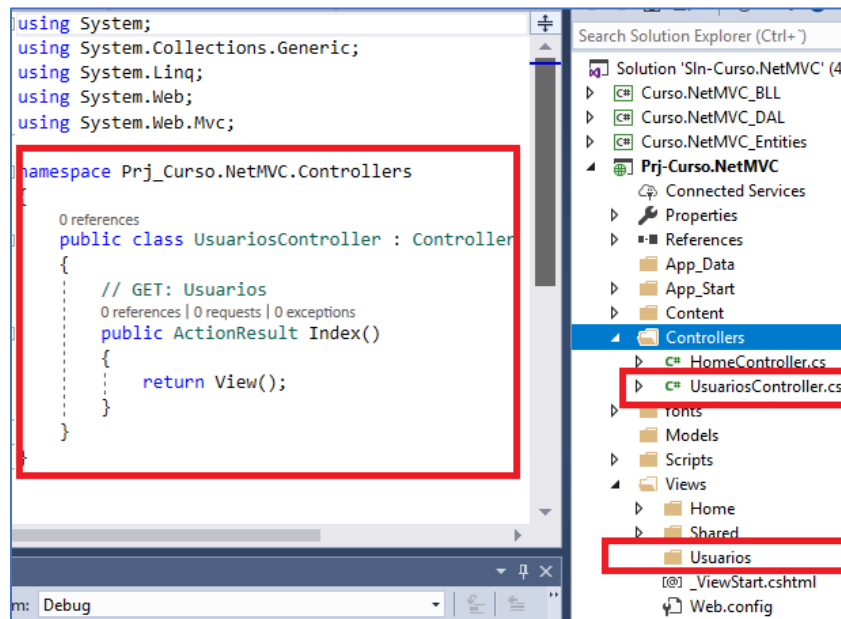
En el **Scaffold agregar** del cuadro de diálogo, haga clic en **controlador MVC 5 – vacío**, a continuación, haga clic en **agregar**.



Asigne un nombre al nuevo controlador "UsuariosController" y haga clic en **agregar**.



Tenga en cuenta en el **Explorador de soluciones** que un nuevo archivo se ha creado con nombre *UsuariosController.cs* y una nueva carpeta *Views\Usuarios*. El controlador está abierto en el IDE.



Reemplace el contenido del archivo con el código siguiente.

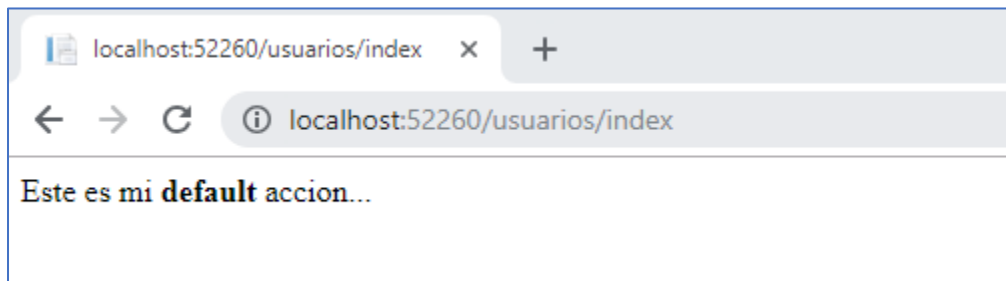
```
using System.Web;
using System.Web.Mvc;
namespace Prj_Curso.NetMVC.Controllers
{
    public class UsuariosController : Controller
    {
        //
        // GET: /HelloWorld/
        public string Index()
        {
            return "Este es mi <b>default</b> accion...";
        }
        //
        // GET: /HelloWorld/Welcome/
        public string Welcome()
        {
            return "Este es el metodo de accion de bienvenida...";
        }
    }
}
```

```

    }
}
}

```

Los métodos del controlador devolverán una cadena HTML como un ejemplo. El controlador se denomina UsuariosController y el primer método se denomina Index. Vamos a invocarlo desde un explorador. Ejecute la aplicación (presione F5 o CTRL+F5). En el explorador, anexe "Usuarios" a la ruta de acceso en la barra de direcciones. (Por ejemplo, en la ilustración siguiente, su <http://localhost:52260/usuarios/index>.) la página en el explorador será similar a la captura de pantalla siguiente. En el método anterior, el código devuelve una cadena directamente.



ASP.NET MVC invoca las clases de controlador diferente (y los métodos de acción diferentes dentro de ellos) según la dirección URL entrante. La lógica de enrutamiento de dirección URL predeterminada usa ASP.NET MVC usa un formato similar al siguiente para determinar qué código debe invocar:

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

Establecer el formato para el enrutamiento en el *aplicación_Start/RouteConfig.cs* archivo.

```

publicstaticvoidRegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

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

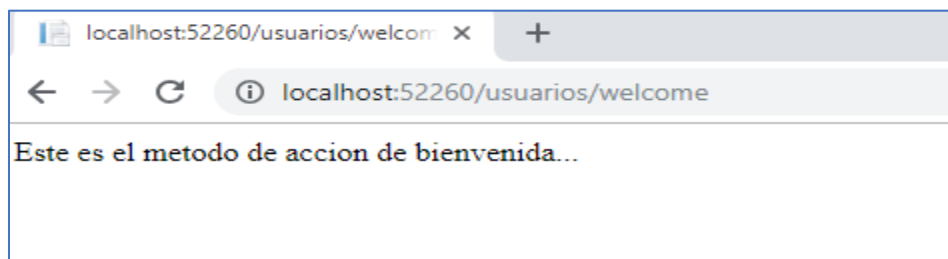
```

Cuando se ejecuta la aplicación y los segmentos de dirección URL no se proporciona, el valor predeterminado es el controlador "Home" y el método de acción "Index" especificados en la sección de valores predeterminados del código anterior.

La primera parte de la dirección URL determina la clase de controlador para ejecutar. Por lo tanto `/Usuarios` se asigna a la `UsuariosController` clase. La segunda parte de la dirección URL determina el método de acción en la clase que se ejecutará. Por lo tanto `/Usuarios/Index` provocaría el `Index` método de la `UsuariosController` clase que se ejecutará. Tenga en cuenta que sólo teníamos que vaya a `/Usuarios` y `Index` usó el método de forma predeterminada.

Esto es porque un método denominado `Index` es el método predeterminado que se llamará en un controlador si no se especifica explícitamente. La tercera parte del segmento de dirección URL (Parameters) es para los datos de ruta.

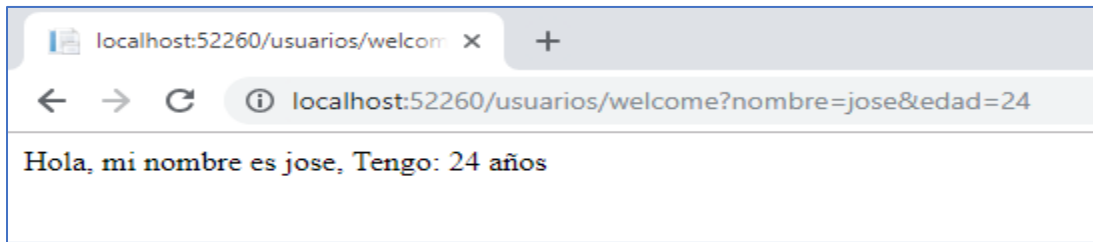
Vaya a `http://localhost:xxxx/Usuarios/Welcome`. El `Welcome` método se ejecuta y devuelve la cadena "este es el método de acción bienvenida... ". La asignación de MVC predeterminada es `/[Controller]/[ActionName]/[Parameters]`. Para esta dirección URL, el controlador es `Usuarios` y `Welcome` es el método de acción.



Vamos a modificar el ejemplo ligeramente para que pueda pasar cierta información del parámetro de la dirección URL al controlador (por ejemplo, `/Usuarios/Welcome? nombre=Jose&edad=4`). Cambiemos el método `Welcome` para incluir dos parámetros, como se muestra a continuación. Tengamos en cuenta que el código usa la característica de parámetro opcional de C# para indicar que edad parámetro, de forma predeterminada en 1 si se pasa ningún valor para ese parámetro.

```
publicstring Welcome(string nombre, int edad = 1)
{
    return HttpUtility.HtmlEncode("Hola, mi nombre es " + nombre
    + ", Tengo: " + edad + "años");
}
```

Ejecute la aplicación y vaya a la dirección URL de ejemplo (`http://localhost:xxxx/usuarios/welcome?nombre=jose&edad=24`). Puede probar valores diferentes para name y numtimes en la dirección URL. El sistema de enlace de modelos ASP.NET MVC asigna automáticamente los parámetros con nombre de la cadena de consulta en la barra de direcciones para los parámetros del método.

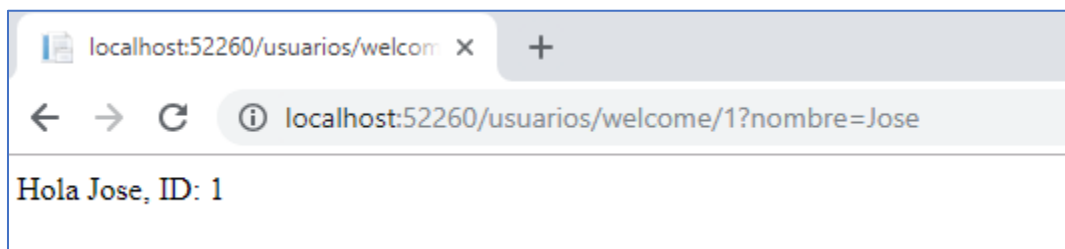


En el ejemplo anterior, el segmento de dirección URL (Parameters) se utiliza el nombre y edad como parámetros que se pasan como cadenas de consulta. El carácter comodín? (signo de interrogación) en la dirección URL anterior es un separador y siguen las cadenas de consulta. El carácter & separa las cadenas de consulta.

Reemplace el método Bienvenido con el código siguiente:

```
publicstringWelcome(string name, int ID = 1)
{
    return HttpUtility.HtmlEncode("Hello " + name + ", ID: " + ID);
}
```

Ejecute la aplicación y escriba la dirección URL siguiente: <http://localhost:xxx/Usuarios/Welcome/1?nombre=Scott>



En este momento el tercer segmento de dirección URL coincide con el parámetro de ruta ID. el Welcome método de acción contiene un parámetro (ID) que coincida con la especificación de dirección URL de la RegisterRoutes método.

En las aplicaciones de ASP.NET MVC, es más habitual para pasar los parámetros como datos de ruta que pasarlos como cadenas de consulta (como hicimos con el identificador anterior). También podría agregar una ruta para pasar tanto el nombre y edad en parámetros como datos de ruta en la dirección URL. En el *aplicación_start\routeconfig*, agregue la ruta de "Hello":

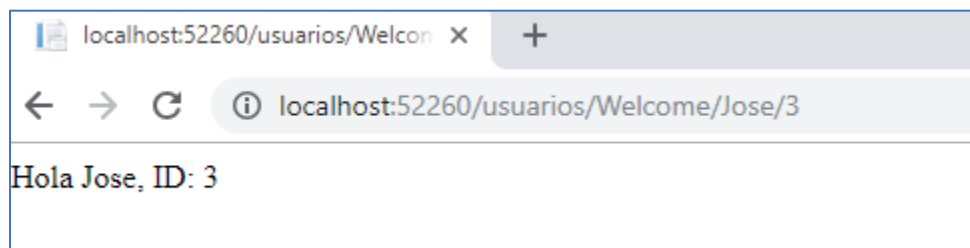
```
publicclassRouteConfig
{
    publicstaticvoidRegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
```

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

```
routes.MapRoute(  
    name: "nombre",  
    url: "{controller}/{action}/{nombre}/{id}"  
);  
}
```

Ejecute la aplicación y vaya a /localhost:XXX/usuarios/Welcome/Jose/3.

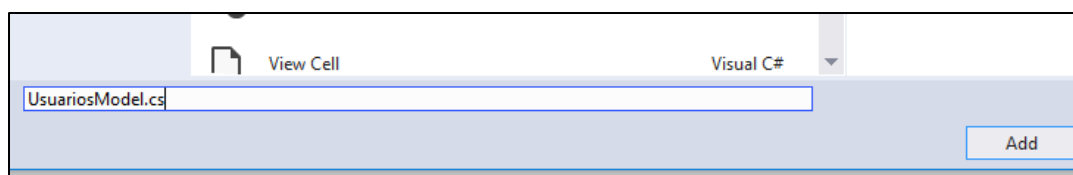


En estos ejemplos el controlador ha realizado la "VC" parte de MVC, es decir, el trabajo de vista y controlador. El controlador devuelve HTML directamente. Normalmente no desea que los controles devuelvan HTML directamente, porque resulta muy complicado de código. En su lugar, vamos a usar normalmente un archivo de plantilla de vista independiente para ayudar a generar la respuesta HTML.

1.5.2 Crear modelo para entidad Usuario.

Comencemos para crear una clase de Modelo. En el Explorador de soluciones, haga clic en la carpeta **Models** y, a continuación, haga clic en agregar, a continuación, clase.

En caso contrario de los controladores que por obligación de MVC debe llevar la terminación *Controller*, el modelo no es necesario, pero por convención y para que el código sea más legible y entendible agregaremos con terminación Model quedando UsuariosModel.cs.



A continuación, debemos agregar las propiedades de nuestra entidad Usuarios, como ya las tenemos recordemos que al utilizar EntityFramework para crear nuestro modelo de datos nos creó todas las entidades con sus respectivas

propiedades, para ello debemos ir al proyecto de *Entities* subimos la entidad usuarios copiamos todas sus propiedades y llevamos a nuestra clase de modelo.

1.5.3 implementando S en el modelo

La anotación de datos en el marco .Net significa agregar un significado adicional a los datos agregando etiquetas de atributos. La ventaja de utilizar la función Anotación de datos es que al aplicar los atributos de datos podemos administrar la definición de datos en un solo lugar y no es necesario volver a escribir las mismas reglas en varios lugares.

La función Anotación de datos se introdujo en .Net 3.5 y el espacio de nombres **System.ComponentModel.DataAnnotations** contiene las clases que se utilizan como atributos de datos.

Los atributos de la anotación de datos se dividen en tres categorías:

- (1) **Atributos de validación:** se utilizan para hacer cumplir las reglas de validación.
- (2) **Atributos de visualización:** se utiliza para especificar cómo se muestran los datos de una clase / miembro en la interfaz de usuario.
- (3) **Atributos de modelado:** se utilizan para especificar el uso previsto del miembro de clase y la relación entre clases.

Entonces lo primero agregamos el espacio de nombre **System.ComponentModel.DataAnnotations** en nuestra clase, posteriormente agregaremos las siguientes data annotation a nuestras propiedades de la entidad Usuario.

```
[Key]
public int IdUsuario { get; set; }
    [Required(ErrorMessage = "{0} es obligatorio")]
    [StringLength(20, MinimumLength = 6, ErrorMessage = "El {0} debe tener minimo
6 maximo 20 caracteres")]
    [DataType(DataType.Text)]
public string Usuario { get; set; }
    [Required(ErrorMessage = "{0} es obligatorio")]
    [StringLength(60, MinimumLength = 3, ErrorMessage = "El {0} debe tener
minimo 3 maximo 5 caracteres")]
    [DataType(DataType.Text)]
public string Iniciales { get; set; }
    [Required(ErrorMessage = "{0} es obligatorio")]
    [StringLength(60, MinimumLength = 3, ErrorMessage = "El {0} debe tener
minimo 3 maximo 60 caracteres")]
    [DataType(DataType.Text)]
public string Nombre { get; set; }
    [Required(ErrorMessage = "{0} es obligatorio")]
    [StringLength(60, MinimumLength = 3, ErrorMessage = "El {0} debe tener
minimo 3 maximo 60 caracteres")]
    [DataType(DataType.Text)]
public string PrimerApellido { get; set; }
    [StringLength(60, MinimumLength = 3, ErrorMessage = "El {0} debe tener
minimo 3 maximo 60 caracteres")]
```

```

        [DataType(DataType.Text)]
publicstring SegundoMaterno { get; set; }
        [Required(ErrorMessage = "{0} es obligatorio")]
        [DataType(DataType.EmailAddress)]
        [EmailAddress]
publicstring Email { get; set; }
        [Required(ErrorMessage = "{0} es obligatorio")]
        [StringLength(15, MinimumLength = 6, ErrorMessage = "El {0} debe tener
minimo 6 maximo 15 caracteres")]
        [DataType(DataType.Text)]
publicstring Password { get; set; }
        [Required(ErrorMessage = "{0} es obligatorio")]
publicint Activo { get; set; }
        [Required(ErrorMessage = "{0} es obligatorio")]
publicint IdPerfil { get; set; }
        [Required(ErrorMessage = "{0} es obligatorio")]
public Nullable<int> IdAplicacion { get; set; }

```

1.5.4 Tipos de Vistas en MVC

ASP.NET MVC nos ofrece tres tipos de vistas:

Master o Layout: son el tipo de vistas que nos permiten compartir una serie de características a través de distintas páginas. Generalmente en esta vista ponemos la base del diseño de la aplicación, el encabezado, los menús, y las referencias comunes a ser usadas (como los archivos css del sitio, librerías JavaScript como jQuery o archivos JavaScript propios). En este tipo de vistas es donde están todas las etiquetas básicas que debe tener una página HTML. Por defecto, un nuevo proyecto de ASP.NET MVC trae el archivo **_Layout.cshtml**. Sin embargo podemos crear todos los Layouts que necesitemos, como por ejemplo uno particular para los accesos realizados mediante dispositivos móviles. En las páginas Layout debemos indicar donde se incluirán las vistas que lo utilicen. Esto se realiza mediante **@RenderBody()**

```

<div class="container">
    <div class="starter-template">
        @RenderBody()
    </div>
</div>

```

Vistas: son las que desarrollamos normalmente en nuestra aplicación, generalmente una por acción que tengamos en nuestros controladores. En las mismas podremos referenciar que Layout utilizar mediante el siguiente código (en caso de que no queramos utilizar ningún Layout, solo debemos asignarle *null*):

```

Layout = "~/Views/Shared/_Layout.cshtml";

```

Vistas parciales: una funcionalidad muy útil y que a veces no la utilizamos mucho. Las vistas parciales son como las vistas normales, pero pueden ser incluidas en otras vistas. En el template de proyecto ASP.NET MVC de Visual Studio se puede observar un ejemplo de esto para la vista _LoginPartial.

```
@{  
    Html.RenderPartial("Index/_DescriptionSection");  
    Html.RenderPartial("Index/_RequirementSection");  
    Html.RenderPartial("Index/_TestCaseSection");  
    Html.RenderPartial("Index/_TraceabilitySection");  
    Html.RenderPartial("Index/_UserSection");  
    Html.RenderPartial("Index/_ProjectSection");  
}
```

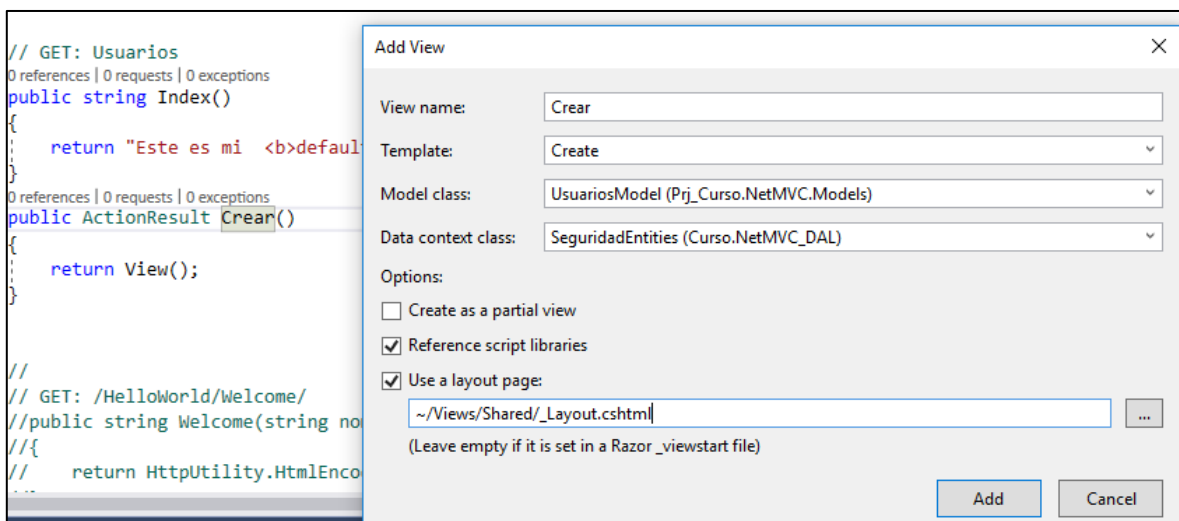
1.5.5 Crear vistas para entidad Usuario

Para crear una vista utilizando el modelo ya creado se necesario posicionarnos en UsuariosController.cs y creamos lo siguiente:

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

Creamos un metodo de accion llamado Crear el cual retorna una vista en su deficion, posteriormente sobre el nombre del metodo hacemos clic derecho, agregar vista, posteriormente el asistente de visual studio nos brinda la posibilidad de crear la vista en base a un modelo y a una clase de context de datos, utilizar un layout base o crear una vista parcial ya antes comentada.

Elegimos la plantilla Crear, el modelo ya antes creado y nuestra clase de context de datos ya creada al crear el modelo de datos y por ultimo utilizaremos el layout por default.

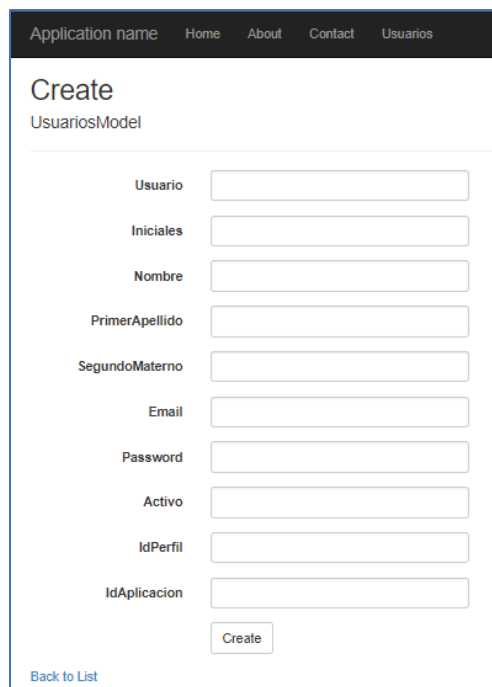


Como podemos ver el asistente de visual studio nos creó un archivo crear.cshtml dentro de una carpeta Usuarios dentro de la carpeta Views,

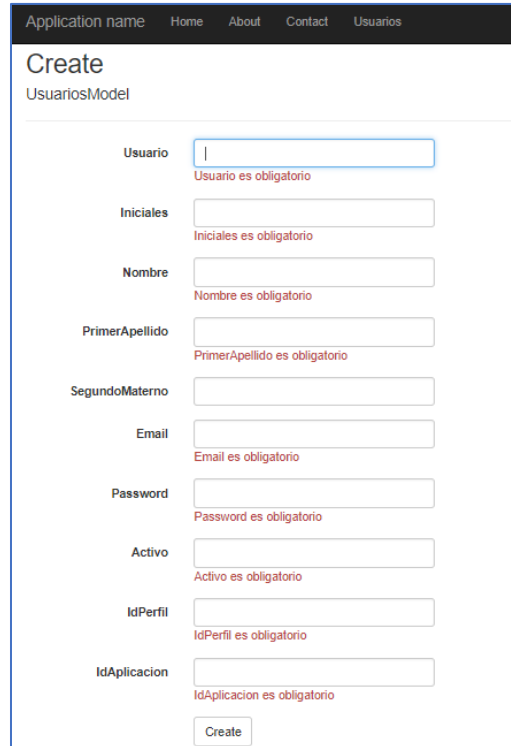
Para ver en realidad la vista y la función de los DataAnnotations agregamos en nuestro layout una opción más al menú como la siguiente línea.

```
<li>@Html.ActionLink("Usuarios", "Create", "Usuario")</li>
```

Una vez ejecutada la aplicación y dirigiendonos a la nueva opción del menú tendremos un formulario con todos los campos definidos en el UsuariosModel.cs. como se representa en la siguiente imagen.

The screenshot shows a web application interface. At the top is a dark navigation bar with links: "Application name", "Home", "About", "Contact", and "Usuarios". Below this is a white content area with the heading "Create" and the sub-heading "UsuariosModel". The form contains several input fields: "Usuario", "Iniciales", "Nombre", "PrimerApellido", "SegundoApellido", "Email", "Password", "Activo", "IdPerfil", and "IdAplicacion". Each field has a corresponding label to its left. At the bottom right of the form is a "Create" button. At the bottom left is a link labeled "Back to List".

Para comprobar la función de los DataAnnotations hacemos clic sobre el botón Crear y posteriormente se muestran las alertas de los campos que marcamos como requeridos y sus validaciones correspondientes quedando como la siguiente imagen.



Application name Home About Contact Usuarios

Create

UsuariosModel

Usuario Usuario es obligatorio

Iniciales Iniciales es obligatorio

Nombre Nombre es obligatorio

PrimerApellido PrimerApellido es obligatorio

SegundoApellido

Email Email es obligatorio

Password Password es obligatorio

Activo Activo es obligatorio

IdPerfil IdPerfil es obligatorio

IdAplicacion IdAplicacion es obligatorio

Create

1.5.6 Guardar en la Base de Datos.

Ya tenemos creado nuestro formulario de la entidad usuario(Front-End) y al igual nuestra clase de acceso a datos (Back-End), ahora conectaremos el front con el back para mandar a guardar información en nuestra base de datos.

Crearemos otro método de acción en nuestro controlador UsuariosController.cs el cual se llamará Crear como el ya creado anteriormente, pero este cambiará ya que recibirá como argumento un objeto de tipo UsuariosModel.cs, al igual algunos filtros o decoradores de métodos de acción quedando de la siguiente forma.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(UsuariosModel usuariosModel){}
```

El decorador HttpPost sirve para indicar que nuestro método será accedido por medio de una llamada Post, el decorador ValidateAntiForgeryToken es un soporte anti-falsificación de MVC escribe un valor único en una cookie solo de HTTP y luego se escribe el mismo valor en el formulario. Cuando se envía la página, se genera un error si el valor de la cookie no coincide con el valor del formulario de esta forma hacemos seguro el envío de la información. Cabe mencionar que dentro de la declaración del formulario en la vista es necesario colocar @Html.AntiForgeryToken() de esta forma se le indica que el formulario se envía de forma segura con clave.

Añadimos un campo a nivel controlador de tipo UsuariosRepositorio.cs donde tenemos nuestros métodos genéricos para nuestra entidad Usuarios.

```
protected UsuariosRepositorio db = new UsuariosRepositorio();
```

Posteriormente dentro de nuestro método tendremos la validación de nuestro modelo para ello se utiliza la validación de ModelState.IsValid la cual nos indica si el modelo enviado cumple con las validaciones ya puestas con los DataAnnotations.

```
        if (ModelState.IsValid)
        {
            Usuarios New = new Usuarios()
            {
                Usuario = usuariosModel.Usuario,
                Iniciales = usuariosModel.Iniciales,
                Nombre = usuariosModel.Nombre,
                ApellidoPrimero = usuariosModel.PrimerApellido,
                ApellidoSegundo = usuariosModel.SegundoMaterno,
                Correo = usuariosModel.Email,
                Contraseña = usuariosModel.Password,
                IdEstatus = usuariosModel.Activo,
                FechaCreacion = DateTime.Now,
                IdAplicacion = usuariosModel.IdAplicacion,
                IdPerfil = usuariosModel.IdPerfil
            };
            db.Create(New);
            return RedirectToAction("Index");
        }
        return View(usuariosModel);
```

Una vez validado el modelo pasamos la informacion a un objeto de tipo Usuarios que tenemos definido en nuestra capa de entidades.

Despues accediendo al metodo Create de nuestra instancia de la clase de UsuariosRepositorio.cs le pasamos nuestro nuevo objeto.

Por ultimo redigimos a la accion index del controlador UsuariosController.cs, con esto ya hemos echo un guardado a la base de datos utilizando la entidad Usuarios.

1.5.7 Utilizar HelperHtml.DropDownList en Razor

Como se podrán dar cuenta en el ejemplo del formulario anterior hay dos campos que dependen de un listado de catálogo el IdPerfil y el IdAplicacion, para ello utilizaremos un helper de tipo DropDownList el cual llenaremos con datos ya antes registrados en la base de datos.

Un **helper** es una herramienta de extensión de ASP.NET MVC que nos permite crear código HTML de forma personalizada y parametrizada. Sus principales objetivos son:

Reutilizar comportamientos comunes que se repiten a lo largo de las vistas.

Facilitar la escritura de nuestras vistas, permitiendo que determinadas porciones de código queden definidas en un solo lugar.

Si bien son similares los puntos planteados anteriormente, con el primero se hace hincapié en cuestiones donde hay una mayor cantidad de lógica asociada a variantes de visualización. Con el segundo, a aquellos aspectos donde siempre escribimos el mismo HTML pero con pequeñas variantes.

Ahora bien, ¿cómo hacemos nuestros helpers? Aquí tenemos dos alternativas disponibles, complementarias entre sí:

- **Html Helpers**
- **Razor Helpers**

Html Helpers

ASP.NET MVC viene con un conjunto de helpers ya disponibles y que solemos usar muy a menudo en nuestras aplicaciones. Son todos aquellos que utilizamos en nuestras vistas con el espacio de nombres "Html". Si vemos nuestra vista Crear.cshtml creada anteriormente podremos ver el uso de estos helpers:

```
<h2>Crear</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>UsuariosModel</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        <div class="col-md-10">
            @Html.LabelFor(model => model.Usuario, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Usuario, new { htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Usuario, "", new { @class = "text-danger" })
            </div>
        </div>

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

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

Razor Helpers

A diferencia de los anteriores, por defecto no vienen helpers de este tipo, por lo que para usarlos deberemos crear los nuestros. Un aspecto clave es que los mismos (como insinúa su nombre) se escriben con la sintaxis de Razor, lo cual hace que sea mucho más práctica la escritura de código HTML, sobre todo para aquellos casos donde tenemos varias líneas del mismo.

Una vez leído la definición de los helpers, primero crearemos nuestras clases Repositorio para la entidad Perfiles y Aplicaciones en nuestro proyecto **BLL**, y sobrescribiremos el método GetAll().

Ejemplo de cómo quedaría la clase AplicacionesRepositorio.cs:

```
public class AplicacionesRepositorio : BaseRepository<Aplicaciones>
{
    public override List<Aplicaciones> GetAll()
    {
        return base.GetAll();
    }
}
```

Posteriormente instanciaremos las dos clases en nuestro controlador UsuariosController.cs quedando de la siguiente forma:

```
protected UsuariosRepositorio db = new UsuariosRepositorio();
protected AplicacionesRepositorio dbAplicaciones = new AplicacionesRepositorio();
protected PerfilesRepositorio dbPerfiles = new PerfilesRepositorio();
```

1.5.8 ASP.NET MVC – ViewBag

Hemos aprendido en la sección anterior que el objeto modelo se utiliza para enviar datos del controlador a una vista y viceversa. Sin embargo, puede haber algún escenario en el que desee enviar una pequeña cantidad de datos temporales a la vista. Por esta razón, el marco MVC incluye ViewBag.

ViewBag puede ser útil cuando desea transferir datos temporales (que no están incluidos en el modelo) desde el controlador a la vista. El ViewBag es una propiedad de tipo dinámico de la clase ControllerBase que es la clase base de todos los controladores.

La siguiente figura ilustra el ViewBag.



Haciendo uso de los ViewBag llenaremos los DropDownList en cuestión, para ello en nuestro método de acción Crear sin argumento crearemos dos ViewBag quedando de la siguiente forma:

```
public ActionResult Crear()
{
    ViewBag.Aplicaciones = dbAplicaciones.GetAll();
    ViewBag.Perfiles = dbPerfiles.GetAll();
    return View();
}
```

Como podemos observar estamos haciendo uso de nuestras instancias antes creadas y al método GetAll() que nos devuelve todos los registros de esas entidades.

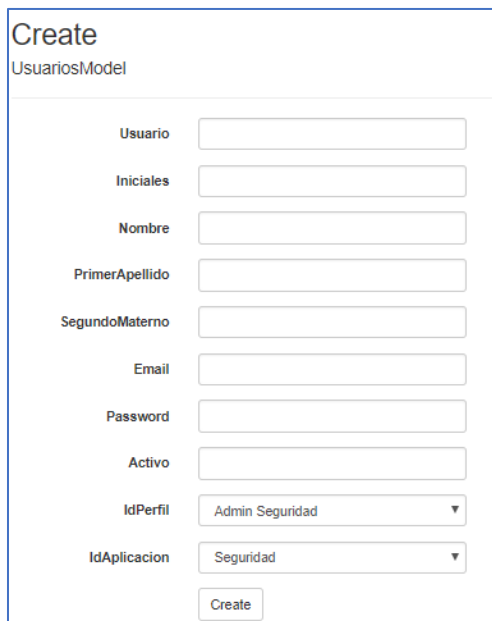
Seguidamente en la vista Crear.cshtml modificaremos los helpers que se crearon por el asistente de visual studio para IdPerfil y IdAplicacion.

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

Quedando de la siguiente forma:

```
<div class="form-group">
    @Html.LabelFor(model => model.IdPerfil, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownList("IdPerfil", new SelectList(ViewBag.Perfiles, "IdPerfil", "Nombre"), htmlAttributes: new { @class = "form-control" })
        @Html.ValidationMessageFor(model => model.IdPerfil, "", new { @class = "text-danger" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(model => model.IdAplicacion, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DropDownList("IdAplicacion", new SelectList(ViewBag.Aplicaciones, "IdAplicacion", "Nombre"), htmlAttributes: new { @class = "form-control" })
        @Html.ValidationMessageFor(model => model.IdAplicacion, "", new { @class = "text-danger" })
    </div>
</div>
```

Como se puede apreciar se está haciendo uso del helper DropDownList que en su primer argumento le indicamos el nombre que recibirá el control ya en html, como segundo argumento le indicamos que recibirá una lista, la cual seleccionará de un ViewBag e indicamos el nombre de la columna que tomará como valor y la columna que tomará como descripción y por ultimo le agregamos un atributo html asignando una clase de css ya definida por bootstrap. Una vez que ejecutemos la aplicación debería quedar como la siguiente imagen.



Create
UsuariosModel

Usuario

Iniciales

Nombre

PrimerApellido

SegundoApellido

Email

Password

Activo

IdPerfil Admin Seguridad ▼

IdAplicacion Seguridad ▼

Create

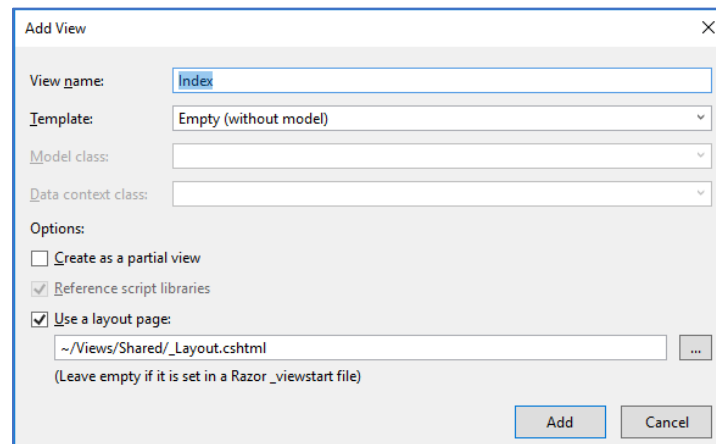
1.5.9 Añadir vista sin modelo.

Ahora bien, ya tenemos datos de usuarios guardados en la base de datos, pero nos hace falta poder administrarlos para ello crearemos una vista donde mostraremos los datos de los usuarios y añadiremos las opciones de modificación y eliminación respectivamente.

Añadimos un nuevo método de acción en Usuarios.Controller.cs llamado Index el cual quedaría de la siguiente forma:

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

Posteriormente hacemos clic derecho sobre Index y **Agregar vista**:



En esta ocasión crearemos la vista vacía o sin usar plantilla, pero si usaremos el layout por defecto, por ultimo tendremos un archivo. cshtml llamado index dentro de Views/Usuarios con el siguiente contenido:



```
1
2 @{
3     ViewBag.Title = "Index";
4     Layout = "~/Views/Shared/_Layout.cshtml";
5 }
6
7 <h2>Index</h2>
8
9
```

A continuación, indicaremos a nuestra vista que recibirá un modelo de tipo lista de Usuarios.cs de igual forma agregaremos el espacio de nombres donde se encuentran las entidades, convirtiendo a nuestra vista fuertemente tipada termino dado a estos tipos de vistas, por ultimo colocaremos un ActionLink como Subtítulo el cual llamará a la vista Crear.

```
@using Curso.NetMVC_Entities
@model List<Curso.NetMVC_Entities.Usuarios>
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Index</h2>
<p>@Html.ActionLink("Crear nuevo", "Crear")</p>
```

Seguidamente crearemos la estructura de una tabla para pintar los registros de usuarios en la vista quedando de la siguiente forma:

```
<table class="table">
    <tr>
        <th>Usuarios</th>
        <th>Nombre</th>
        <th>Primer Apellido</th>
        <th>Segundo Materno</th>
        <th>Email</th>
        <th>Activo</th>
        <th></th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>@Html.DisplayFor(modelItem => item.Usuario)</td>
            <td>@Html.DisplayFor(modelItem => item.Nombre)</td>
            <td>@Html.DisplayFor(modelItem => item.ApellidoPrimero)</td>
            <td>@Html.DisplayFor(modelItem => item.ApellidoSegundo)</td>
            <td>@Html.DisplayFor(modelItem => item.Correo)</td>
            <td>@Html.DisplayFor(modelItem => item.IdEstatus)</td>
            <td>
                @Html.ActionLink("Edita", "Editar", new { id = item.IdUsuario }) |
                @Html.ActionLink("Detalle", "Detalles", new { id = item.IdUsuario }) |
                @Html.ActionLink("Eliminar", "Eliminar", new { id = item.IdUsuario })
            </td>
        </tr>
    }
</table>
```

Una vez creada la estructura de la tabla, modificaremos nuestro método de acción index agregándole lo siguiente:

```
public ActionResult Index()
{
    var mymodel = db.GetAll();
    return View(mymodel);
}
```

Creamos una variable para nuestro modelo y realizamos una consulta a la instancia de UsuariosRepositorio.cs al método GetAll() el cual nos retornara todos los usuarios, dicha variable se pasa como argumento al retornar la vista de esta forma indicamos que este es el modelo de la vista en cuestión, compilando y ejecutando la aplicación en la vista index nos deberá quedar como la siguiente imagen:

Index					
Crear nuevo					
Usuarios	Nombre	Primer Apellido	Segundo Materno	Email	Activo
sa	sa	admin			1 Edit Details Delete
olooppkpkp	kpkpk	pkpk	pkkp	kpkp@oko.com	1 Edit Details Delete
omomopoiijj	0j0j	0j0j0j		kpkp@oko.com	1 Edit Details Delete
© 2019 - My ASP.NET Application					

1.5.10 ¿Qué son y para qué sirven las propiedades de navegación en EntityFramework?

Una propiedad de navegación es una propiedad opcional en un tipo de entidad que permite navegar desde un extremo de una asociación a el otro extremo. A diferencia de otras propiedades, las propiedades de navegación no transportan datos.

Una definición de propiedad de desplazamiento incluye lo siguiente:

Un nombre. (Necesario)

La asociación que navega. (Necesario)

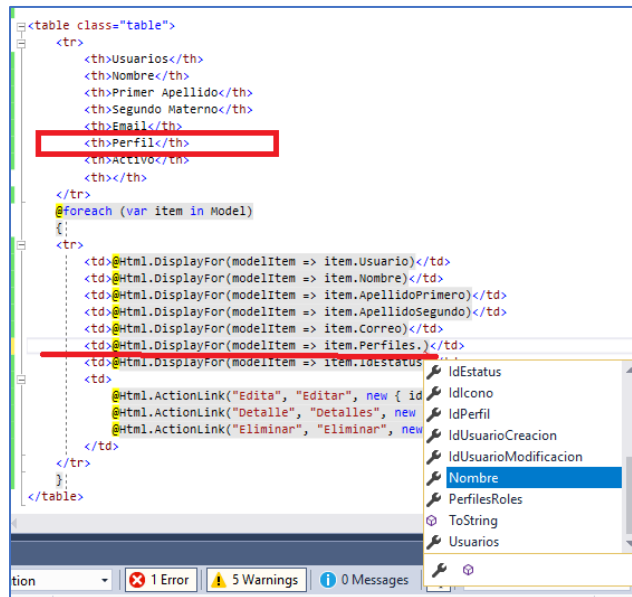
Los extremos de la asociación que navega. (Necesario)

Las propiedades de navegación proporcionan una manera de navegar por una asociación entre dos tipos de entidad. Cada objeto puede tener una propiedad de navegación para cada relación en la que participa. Propiedades de navegación permiten navegar y administrar relaciones en ambas direcciones, devolviendo un objeto de referencia (si la multiplicidad es uno o cero o uno) o una colección (si la multiplicidad es varios). También puede tener una navegación unidireccional, en cuyo caso se define la propiedad de navegación solo en uno de los tipos que participa en la relación y no en ambos.

Al utilizar el enfoque DataBaseFirst para crear nuestro modelo de datos y teniendo nuestra base de datos con sus respectivas relaciones, las propiedades de navegación se crean automáticamente para todas las entidades en las que hay relación con otra entidad.

Haciendo uso de estas propiedades agregaremos el nombre del perfil el cual tiene asignado nuestros Usuarios y lo mostraremos en la vista Index la cual acabamos de crear.

Primero agregaremos un encabezado haciendo relación a el perfil de cada usuario, posteriormente en la definición de las filas agregaremos celda donde por medio de la propiedad de navegación Perfiles de la entidad Usuario accederemos a la propiedad Nombre de la entidad Perfil, quedando como se muestra en la siguiente imagen.



Ejecutando la aplicación estaría quedando de la siguiente forma:

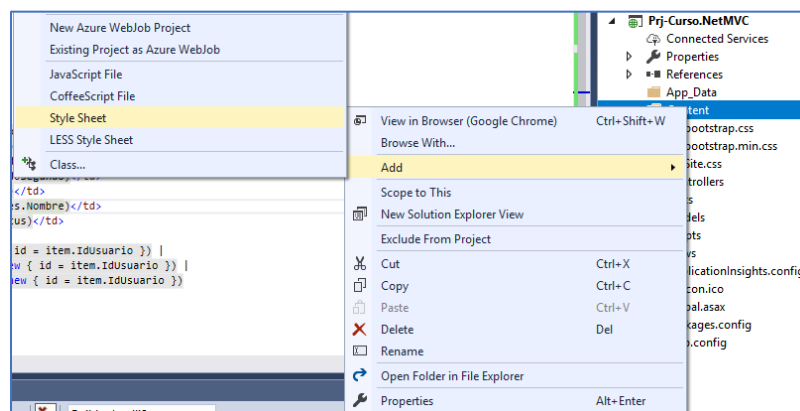
Index							
Crear nuevo							
Usuarios	Nombre	Primer Apellido	Segundo Materno	Email	Perfil	Activo	
sa	sa	admin			Admin Seguridad	1	Edita Detalle Eliminar
olooppkpkp	kpkpk	pkpk	pkpk	kpkp@oko.com	Admin Seguridad	1	Edita Detalle Eliminar
omomopoiijij	OjOj	ojoOj		kpkp@oko.com	Admin Seguridad	1	Edita Detalle Eliminar

© 2019 - My ASP.NET Application

1.5.11 Añadir y utilizar un archivo CSS

Ahora bien, vamos a hacer uso de estilos css para dibujar un círculo de color verde para cuando los usuarios estén activos y rojo para cuando estén inactivos.

Para ello añadiremos una hoja de estilo dentro de la carpeta Content haciendo clic derecho agregar hoja de estilo con nombre Miestilo.css.



Una vez creado el archivo debemos de indicar que será cargado junto con los demás archivos de estilos, para ello ubicamos el archivo de configuración BundleConfig.cs ubicado dentro de la carpeta App_Start, como se explicó al inicio los bundleConfig nos sirven para agrupar distintos archivos externos en el html de esta forma solo se hace un render y así se cargan N archivos.

Una vez ubicado el bundle donde se agrupan los archivos CSS y agregar la referencia a nuestro Myestilo.css quedaría como la siguiente imagen:

```
bundles.Add(new StyleBundle("~/Content/css").Include(
    "~/Content/bootstrap.css",
    "~/Content/site.css",
    "~/Content/Myestilo.css"));
```

Una vez echo lo anterior crearemos dos clases de estilo la cual representaran el estilo de un pequeño círculo en color verde activo y de color rojo inactivo.

```
.ActivoSpan {
    width: 15px;
    height: 15px;
    border-radius: 50%;
    background: #5cb85c;
}

.InactivoSpan {
    width: 15px;
    height: 15px;
    border-radius: 50%;
    background: #ff0000;
}
```

Lo siguiente será implementar estas clases de estilo en la definición de la tabla en la vista Index.cshtml quedando de la siguiente forma:

```
<td>@Html.DisplayFor(modelItem => item.Usuario)</td>
<td>@Html.DisplayFor(modelItem => item.Nombre)</td>
<td>@Html.DisplayFor(modelItem => item.ApellidoPrimero)</td>
<td>@Html.DisplayFor(modelItem => item.ApellidoSegundo)</td>
<td>@Html.DisplayFor(modelItem => item.Correo)</td>
<td>@Html.DisplayFor(modelItem => item.Perfiles.Nombre)</td>
<td>
    @if (item.IdEstatus == 1)
    {
        <div class="ActivoSpan"></div>
    }
    else
    {
        <div class="InactivoSpan"></div>
    }
</td>
```

Como podemos apreciar esta vez estamos haciendo uso de un IF para determinar el estatus del usuario y así determinar la clase a aplicar, quedando como la siguiente imagen:

Index						
Crear nuevo						
Usuarios	Nombre	Primer Apellido	Segundo Materno	Email	Perfil	Activo
sa	sa	admin			Admin Seguridad	● Edita Detalle Eliminar
olooppkpkp	kpkpk	pkpk	pkpk	kpkp@oko.com	Admin Seguridad	● Edita Detalle Eliminar
omomopoiijij	0j0j	ojo0j		kpkp@oko.com	Admin Seguridad	● Edita Detalle Eliminar
U000123	Benito	Lopez	Lopez	BLL@gmail.com	Admin Seguridad	● Edita Detalle Eliminar

A continuación, le daremos estilos a nuestros links de Editar, Detalle y Eliminar para que tomen forma de botones, para ello crearemos una clase de estilo base para botones como la siguiente:

```
.Mybtn {
    text-decoration: none;
    display: inline-block;
    font-weight: 400;
    color: #212529;
    text-align: center;
    vertical-align: middle;
    user-select: none;
    background-color: transparent;
    border: 1px solid transparent;
    padding: .375rem .75rem;
    font-size: 1.3rem;
    line-height: 1.5;
    border-radius: .25rem;
}
```

Posteriormente crearemos una clase para definir qué tipo de botón serian, quedando de la siguiente forma:

```
.Mybtn-Edita {
    color: #ffffff;
    background-color: #17a2b8;
    border-color: #ffc107;
}
.Mybtn-detalle {
    color: #212529;
    background-color: #ffc107;
    border-color: #ffc107;
}
.Mybtn-Elimina {
    color: #ffffff;
    background-color: #dc3545;
    border-color: #ffc107;
}
```

Ahora las asignaremos en la definición de la tabla de la vista Index.cshtml quedando de la siguiente forma:

```
<td>
    @Html.ActionLink("Editar", "Editar", new { id = item.IdUsuario }, htmlAttributes: new { @class = "Mybtn Mybtn-Edita" })
    @Html.ActionLink("Detalle", "Detalles", new { id = item.IdUsuario }, htmlAttributes: new { @class = "Mybtn Mybtn-detalle" })
    @Html.ActionLink("Eliminar", "Eliminar", new { id = item.IdUsuario }, htmlAttributes: new { @class = "Mybtn Mybtn-Elimina" })
</td>
```

Como se puede apreciar estamos agregando un parámetro más al helper ActionLink el cual es un atributo de HTML en nuestro caso una clase de estilo ya creadas por nosotros, por ultimo debemos de tener un resultado como el siguiente:

Index						
Crear nuevo						
Usuarios	Nombre	Primer Apellido	Segundo Materno	Email	Perfil	Activo
sa	sa	admin			Admin Seguridad	● Editar Detalle Eliminar
olooppkpkp	kpkpk	pkpk	pkkp	kpkp@oko.com	Admin Seguridad	● Editar Detalle Eliminar
omomopojijj	0j0j	oj0j0j		kpkp@oko.com	Admin Seguridad	● Editar Detalle Eliminar
U000123	Benito	Lopez	Lopez	BLL@gmail.com	Admin Seguridad	● Editar Detalle Eliminar
© 2019 - My ASPNET Application						

1.5.12 Añadir y utilizar un archivo JavaScripts

Ahora bien, usaremos Ajax para realizar peticiones al servidor en este ejemplo obtendremos una vista de tipo parcial la cual tendrá el detalle de X usuario y la presentará en una ventana modal de bootstrap.

Añadimos un nuevo archivo js dentro de la carpeta de scripts clic derecho añadir archivo de JavaScripts el cual llamaremos MetodosAjax.js

Posteriormente crearemos dos funciones para encapsular las llamadas Ajax de tipo POST y GET, la primera función la llamaremos llamadoAjaxPOSTJson que recibirá como primer parámetro un string representando la dirección url de la llamada, como segundo parámetro un json donde se especifiquen los parámetros y por tercer parámetro el nombre de una función de éxito la cual estaría recibiendo la respuesta de la llamada, una vez dicho esto quedaría de la siguiente forma:

```
function llamadoAjaxPOSTJson(urlAction, parametros, funcExito) {
    $.ajax({
        type: 'POST',
        url: urlAction,
        data: JSON.stringify(parametros),
        contentType: "application/json; charset=utf-8",
        dataType: "json",
        success: funcExito,
        error: errorGenerico
    });
}
```

La segunda función la llamaremos llamadoAjaxGEThtml la cual tendrá como primer parámetro la url de la llamada, por segundo y último parámetro el nombre de la función de éxito, dicha función nos retornará como resultado html.

Por ultimo crearemos otra función llamada errorGenerico para manejar los tipos de errores de las llamadas Ajax dicha función es invocada en la anterior función ya creada, quedando de la siguiente forma:

```
function llamadoAjaxGEThtml(urlAction, funcExito) {  
    $.ajax({  
        type: 'GET',  
        url: urlAction,  
        data: {},  
        contentType: "application/json; charset=utf-8",  
        dataType: "html",  
        success: funcExito,  
        error: errorGenerico  
    });  
}
```

```
function errorGenerico(jqXHR, exception) {  
    var msg = '';  
    if (jqXHR.status === 0) {  
        msg = 'No está conectado, favor de verificar su conexión.';  
    }  
    else if (jqXHR.status === 404) {  
        msg = 'Página no encontrada [404]';  
    }  
    else if (jqXHR.status === 500) {  
        msg = 'Error en el servidor [500]';  
    }  
    else if (jqXHR.status === 'parseerror') {  
        msg = 'El parseo del JSON es erróneo.';  
    }  
    else if (jqXHR.status === 'timeout') {  
        msg = 'Error por tiempo de espera.';  
    }  
    else if (jqXHR.status === 'abort') {  
        msg = 'La petición Ajax fue abortada.';  
    }  
    else {  
        msg = 'Error no conocido. ' + jqXHR.responseText;  
    }  
    alert("Ajax error: " + msg);  
}
```

Una vez creado las tres funciones agregaremos el archivo JS creando un nuevo bundle en el BundleConfig quedando de la siguiente forma:

```
bundles.Add(new ScriptBundle("~/Mybundle/JS").Include("~/Scripts/MetodosAjax.js"));
```

y por ultimo indicamos que se cargue en nuestro _layout.cshtml:

```
@Scripts.Render("~/bundles/jquery")  
@Scripts.Render("~/bundles/bootstrap")  
@Scripts.Render("~/Mybundle/JS")  
  
@RenderSection("scripts", required: false)
```

1.5.13 Ventana modal con Bootstrap.

Ahora bien, crearemos una estructura genérica de una ventana modal utilizando bootstrap la cual estará en nuestro _layout.cshtml quedando de la siguiente forma:

```
@*modal generico*@  
<div class="modal" id="Mymodal" tabindex="-1" role="dialog">  
  <div class="modal-dialog" role="document">  
    <div class="modal-content" id="contentMymodal">  
  
    </div>  
  </div>  
</div>
```

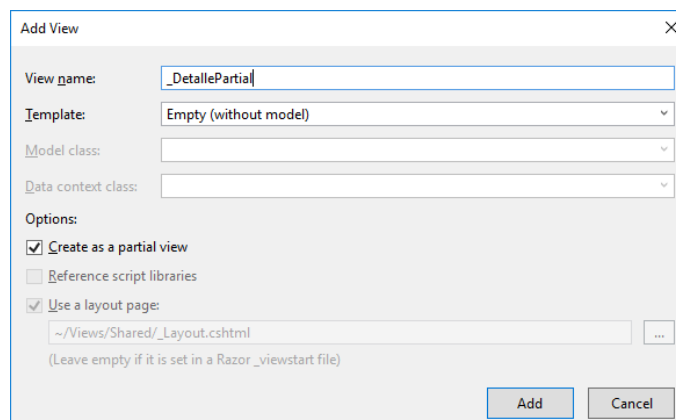
El contenido de nuestro modal lo almacenaremos en una vista parcial, la cual crearemos en nuestro controlador UsuariosController.cs, dicha vista la llamaremos _DetallePartial, primero creamos nuestro método de acción _DetallePartial.

```
0 references | 36 requests | 1 exception  
public ActionResult _DetallePartial(int id)  
{  
    Usuarios Modelusuario = new Usuarios();  
    Modelusuario = new UsuariosRepositorio().Single(c =>  
        c.IdUsuario == id);  
    return PartialView(Modelusuario);  
}
```

Dentro de nuestro método de acción buscamos al usuario en cuestión por medio de su propiedad IdUsuario y su estatus en activo, dicho resultado lo asignamos como el modelo de la vista parcial.

Nota: Al utilizar vistas parciales se recomienda usar la nomenclatura de nombres iniciando con un guion bajo y terminar con la palabra Partial, por último en los métodos de acción se debe hacer referencia que se estará retornando una PartialView().

Agregamos la vista, clic derecho sobre el nombre de la acción y agregar vista, indicamos que será de tipo parcial y agregar.



Una vez creado la vista _DetallePartial.cshtml indicaremos que será una vista fuertemente tipada haciendo referencia que tendrá un objeto modelo de tipo Usuarios.cs al igual que tendrá referencia al proyecto de entidades, posteriormente respetando la estructura de los modales llenaremos con los datos de nuestro modelo.

```
@using Curso.NetMVC_Entities
@model Usuarios

<div class="modal-header">
    <h5 class="modal-title">Detalle de usuario</h5>
    <button type="button" class="close" data-dismiss="modal" aria-label="Close">
        <span aria-hidden="true">&times;</span>
    </button>
</div>
<div class="modal-body">
    @if (Model != null)
    {
        <dl class="dl-horizontal">
            <dt>@Html.DisplayNameFor(model => model.Usuario)</dt>
            <dd>@Html.DisplayFor(model => model.Usuario)</dd>
            <dt>@Html.DisplayNameFor(model => model.Nombre)</dt>
            <dd>@Html.DisplayFor(model => model.Nombre)</dd>
            <dt>@Html.DisplayNameFor(model => model.ApellidoPrimero)</dt>
            <dd>@Html.DisplayFor(model => model.ApellidoPrimero)</dd>
            <dt>@Html.DisplayNameFor(model => model.ApellidoSegundo)</dt>
            <dd>@Html.DisplayFor(model => model.ApellidoSegundo)</dd>
            <dt>@Html.DisplayNameFor(model => model.Correo)</dt>
            <dd>@Html.DisplayFor(model => model.Correo)</dd>
        </dl>
    }
    else
    {
        <p class="text-warning">Información no disponible, favor vuelva a intentar</p>
    }
</div>
<div class="modal-footer">
    <button type="button" class="btn btn-secondary" data-dismiss="modal">Cerrar</button>
</div>
```

Como podemos apreciar se hace una validación al modelo y dependiendo si está cargado o no mostramos la información correcta.

En nuestra vista Index.cshtml indicaremos el uso de código JavaScript donde crearemos tres funciones la primera función se encargará de mostrar un modal indicando el nombre.

```
function MostrarModal(NombreModal) {
    $('#'+ NombreModal).modal('show')
}
```

La segunda función se encargará de hacer la petición de la vista pasando el id de usuario en cuestión y a través de las funciones creadas anteriormente para una llamada GET de AJAX.

```
function Detalle(id) {
    var urlaction = "@Url.Action("_DetallePartial", "Usuarios")" + "/" + id;
    llamadoAjaxGEThtml(urlaction, funcExito);
}
```

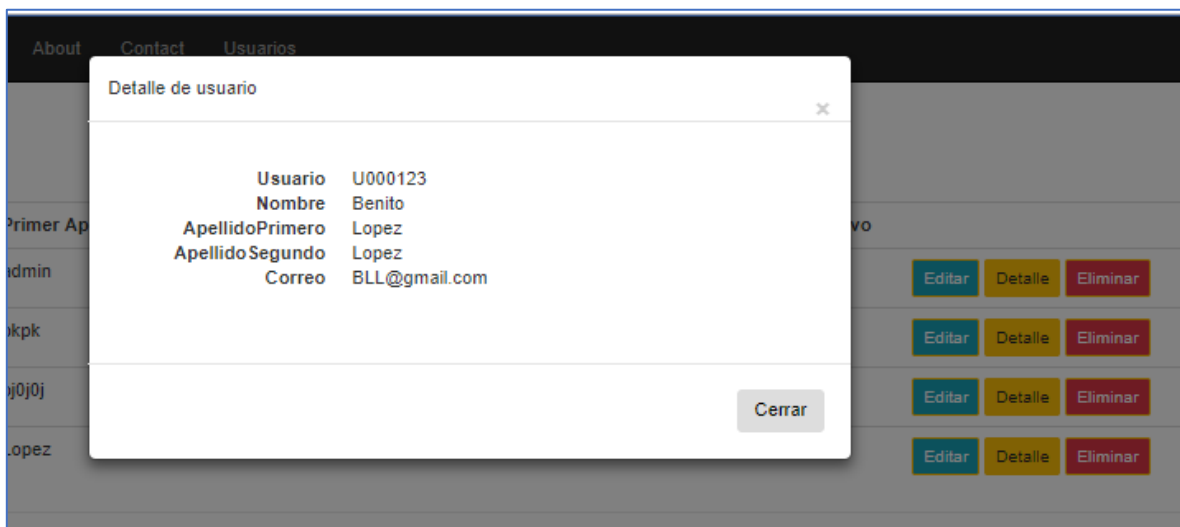

La tercera será la función de éxito enviada en la petición AJAX, la cual recibe la respuesta en este caso HTML el cual insertamos en el contenedor de nuestro modal genérico y por ultimo mostramos el modal con la función MostrarModal.

```
function funcExito(response) {
    $("#contentMymodal").html(response);
    MostrarModal('Mymodal');
}
```

Ahora modificaremos en la definición de la tabla el botón de Detalle reemplazando el helper ActionLink por una etiqueta html que en su evento onClick llamara a la función Detalle y pasara el id del usuario en cuestión, quedando de la siguiente forma:

```
<td>
    @Html.ActionLink("Editar", "Editar", new { id = item.IdUsuario }, htmlAttributes: new { @class = "Mybtn Mybtn-Edita" })
    <a href="#" class="Mybtn Mybtn-detalle" onclick="Detalle(@item.IdUsuario)">Detalle</a>
    @Html.ActionLink("Eliminar", "Eliminar", new { id = item.IdUsuario }, htmlAttributes: new { @class = "Mybtn Mybtn-Elimina" })
</td>
```

Ejecutando la aplicación tendremos el siguiente resultado:



1.6WCFcomo capa servicio.

WCF es un tiempo de ejecución y un conjunto de API para crear sistemas que envían mensajes entre clientes y servicios. La misma infraestructura y API se utilizan para crear aplicaciones que se comuniquen con otras aplicaciones en el mismo sistema del equipo o en un sistema que resida en otra compañía y a la que se obtenga acceso a través de Internet.

Términos de WCF

message

Unidad autónoma de datos que puede constar de varias partes, incluyendo un cuerpo y encabezados.

service

Construcción que expone uno o más extremos, y en la que cada extremo expone una o más operaciones de servicio.

punto de conexión

Construcción en la que se envían o reciben mensajes (o ambos). Consta de una ubicación (una dirección) que define dónde se pueden enviar mensajes, una especificación del mecanismo de comunicación (un enlace) que describe cómo se deben enviar los mensajes, y una definición de un conjunto de mensajes que puede enviar o recibir (o ambos) en el ubicación (un contrato de servicio) que describe qué mensajes se pueden enviar.

dirección

Especifica la ubicación donde se reciben los mensajes. Se especifica como un identificador uniforme de recursos (URI). La parte del esquema URI nombra el mecanismo de transporte que se ha de utilizar para alcanzar la dirección, por ejemplo HTTP y TCP. La parte jerárquica del URI contiene una ubicación única cuyo formato depende del mecanismo de transporte.

La dirección del extremo le permite crear direcciones únicas de extremos para cada extremo de un servicio o, bajo ciertas condiciones, compartir una dirección en los extremos. El siguiente ejemplo muestra una dirección que utiliza el protocolo HTTPS con un puerto no predeterminado:

enlace

Define cómo se comunica un punto de conexión con el mundo. Consta de un conjunto de componentes llamados elementos de enlace que se "apilan" uno sobre el otro para crear la infraestructura de comunicaciones. Como mínimo, un enlace define el transporte (como HTTP o TCP) y la codificación utilizada (por ejemplo, de texto o binaria). Un enlace puede contener elementos de enlace que especifican detalles, por ejemplo los mecanismos de seguridad utilizados para proteger los mensajes o el patrón de mensaje utilizado por un extremo. Para obtener más información, consulte configurar Services.

operación de servicio

Procedimiento definido en el código de un servicio que implementa la funcionalidad de una operación. Esta operación se expone a los clientes como métodos en un cliente de WCF. El método puede devolver un valor y tomar un número opcional de argumentos, o no tomar ningún argumento y no devolver ninguna respuesta. Por ejemplo, una operación que funciona como un simple "Hola" se puede utilizar para notificar acerca de la presencia de un cliente y para comenzar una serie de operaciones.

contrato de servicio

Une varias operaciones relacionadas en una unidad funcional única. El contrato puede definir ajustes del nivel de servicio, como el espacio de nombres del servicio, un contrato de devolución de llamadas correspondiente y otros ajustes de este tipo. En la mayoría de los casos, el contrato se define mediante la creación de una interfaz en el lenguaje de

programación que elija y la aplicación del atributo `ServiceContractAttribute` a la interfaz. El código de servicio real se obtiene al implementar la interfaz.

contrato de operación

Un contrato de operación define los parámetros y el tipo de valor devuelto de una operación. Al crear una interfaz que define el contrato de servicio, se significa un contrato de operación mediante la aplicación del atributo `OperationContractAttribute` a cada definición de método que forma parte del contrato. Las operaciones se pueden modelar para que acepten un único mensaje y devuelvan también un único mensaje, o para que acepten un conjunto de tipos y devuelvan un tipo. En el último caso, el sistema determinará el formato de los mensajes que han de intercambiarse para esa operación.

contrato de mensaje

Describe el formato de un mensaje. Por ejemplo, declara si los elementos del mensaje deberían ir en encabezados frente al cuerpo, qué nivel de seguridad debería aplicarse a qué elementos del mensaje, etc.

contrato de error

Puede estar asociado a una operación de servicio para denotar errores que se pueden devolver al autor de la llamada. Una operación puede tener cero o más errores asociados a ella. Estos errores son errores de SOAP que se modelan como excepciones en el modelo de programación.

contrato de datos

Las descripciones en los metadatos de los tipos de datos que usa un servicio. Esto permite a otros interoperar con el servicio. Los tipos de datos se pueden usar en cualquier parte de un mensaje; por ejemplo, como parámetros o tipos de valores devueltos. Si el servicio sólo utiliza tipos simples, no hay necesidad de utilizar explícitamente contratos de datos.

hospedaje

Un servicio debe hospedarse en algún proceso. Un host es una aplicación que controla la duración del servicio. Los servicios pueden ser autohospedados o estar administrados por un proceso de hospedaje existente.

aplicación cliente

Programa que intercambia mensajes con uno o varios puntos de conexión. La aplicación cliente comienza creando de una instancia de un cliente de WCF y llamando a los métodos del cliente de WCF. Es importante tener en cuenta que una única aplicación pueda ser cliente y servicio.

canal

Implementación concreta de un elemento de enlace. El enlace representa la configuración, y el canal es la implementación asociada a esa configuración. Por consiguiente, hay un canal asociado a cada elemento de enlace. Los canales se apilan uno sobre otro para crear la implementación concreta del enlace: la pila de canales.

cliente de WCF

Construcción de la aplicación cliente que expone las operaciones de servicio como métodos (en el lenguaje de programación .NET Framework de su elección, como Visual Basic o Visual C#). Cualquier aplicación puede hospedar a un cliente de WCF, incluso una aplicación que hospede un servicio. Por tanto, es posible crear un servicio que incluya clientes de WCF de otros servicios.

Un cliente WCF puede generarse automáticamente mediante el uso de la ServiceModel Metadata Utility Tool (Svcutil.exe) y para que apunte a un servicio en ejecución que publique metadatos.

metadatos

En un servicio, describen las características de este que tiene que comprender una entidad externa para comunicarse con él. Los metadatos pueden consumirlos los ServiceModel Metadata Utility Tool (Svcutil.exe) para generar un cliente de WCF y la configuración complementaria que una aplicación cliente puede usar para interactuar con el servicio.

Los metadatos expuestos por el servicio incluyen documentos de esquema XML, que definen el contrato de datos del servicio, y documentos WSDL, que describen los métodos del servicio.

Cuando se habilita, WCFG genera automáticamente los metadatos del servicio mediante la inspección del servicio y sus extremos. Para publicar los metadatos desde un servicio, debe permitir explícitamente al comportamiento de los metadatos.

seguridad

En WCF, incluye confidencialidad (cifrado de mensajes para prevenir su interceptación), integridad (los recursos para la detección de manipulación del mensaje), autenticación (los recursos para la validación de clientes y servidores) y autorización (el control de acceso a recursos). Estas funciones se proporcionan mediante la reutilización de mecanismos de seguridad existentes, como TLS sobre HTTP (también conocido como HTTPS) o la implementación de una o más de las numerosas especificaciones de seguridad WS - *.

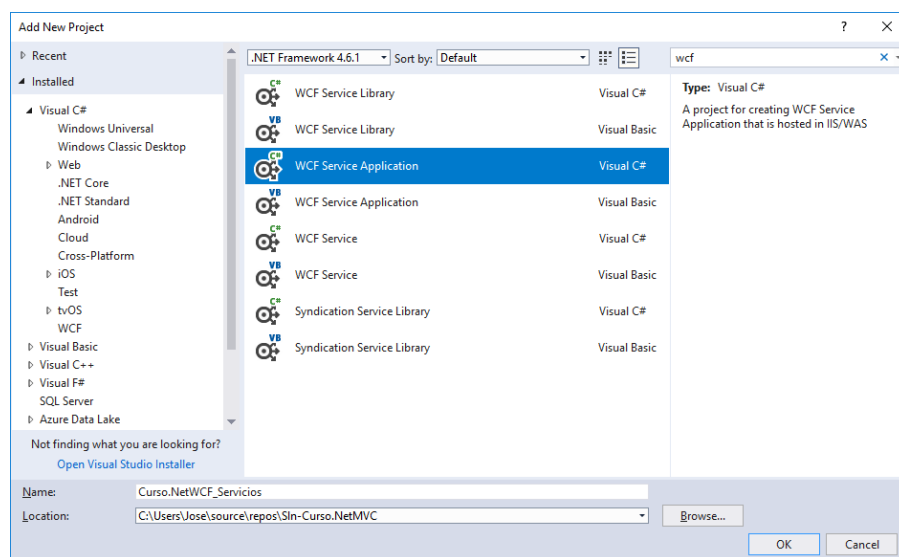
modo de seguridad de transporte

Especifica que los mecanismos del nivel de transporte (como HTTPS) proporcionan confidencialidad, integridad y autenticación. El uso de un transporte como HTTPS en este modo tiene la ventaja de ofrecer un mejor rendimiento. También se entiende bien debido a su predominio en Internet. La desventaja es que este tipo de seguridad se aplica por separado en cada salto en la ruta de comunicación, provocando que la comunicación sea susceptible a un ataque tipo “man in the middle”.

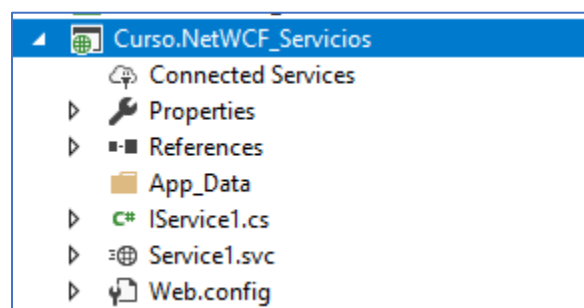
1.6.1 Crear un servicio WCF

Retomando la práctica de administración de usuarios, a continuación, crearemos un servicio tipo WCF donde expondremos dos métodos, el primero lo utilizaremos para realizar la actualización de usuarios consumiendo por referencia web, el segundo realizará la eliminación de usuarios consumido por medio de peticiones Ajax.

Entonces nos posicionamos en la solución de Visual Studio. **Clic derecho, agregar, nuevo proyecto, Servicio de aplicación WCF.**

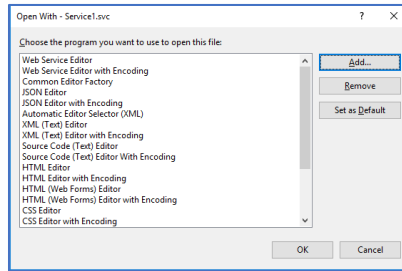


El resultado obtenido debe ser idéntico al siguiente:

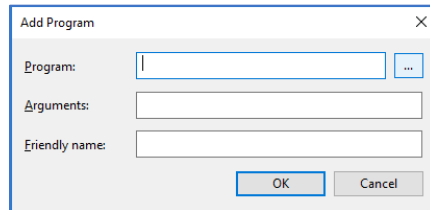


1.6.2 Probar el servicio.

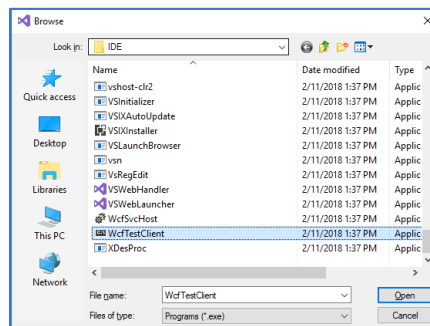
Para realizar una prueba del servicio creado, seleccionamos el archivo **Services1.svc**, **clic derecho, abrir con.**



Seleccionamos agregar, buscar programa.



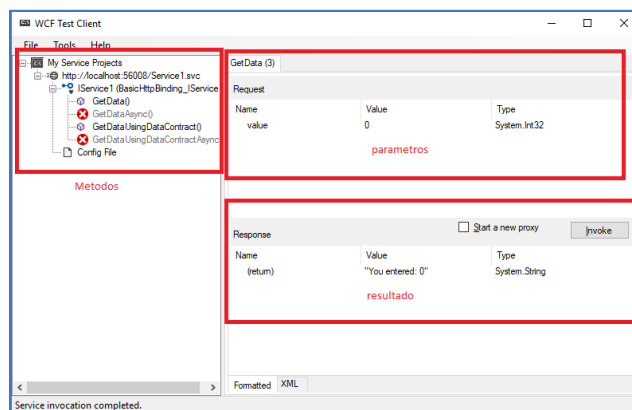
Buscamos la aplicación WcfTestClient, cabe mencionar que esta aplicación se instala en automático al instalar Visual Studio y por ultimo aplicación por default.



Seleccionamos el mismo archivo **Servicio1.svc**, clic derecho y establecer como página de inicio.

Seleccionamos el proyecto **Curso.NetWCF_Servicios**, clic derecho y establecer como proyecto de inicio, y por ultimo ejecutar aplicación.

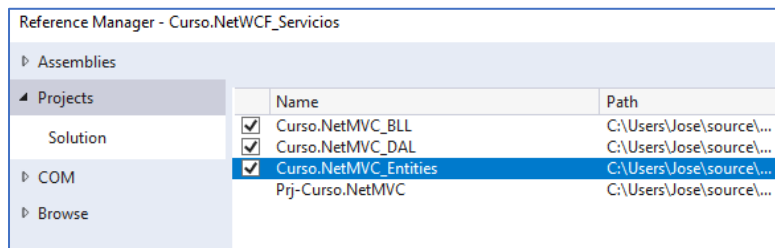
Una vez ejecutada la aplicación tendremos una pantalla como la siguiente:



Como podemos observar del lado izquierdo de la imagen tenemos los métodos disponibles expuestos por el servicio, en la parte superior derecha se encuentran los parámetros de algún método seleccionado del lado izquierdo y en la parte inferior derecha tenemos el botón de invocar al método y su respectiva respuesta. De forma precargada al crear nuestro servicio visual Studio crea dos métodos de pruebas los cuales podemos realizar pruebas de debug, colocando un punto de interrupción en los métodos definidos en el archivo Services1.svc

1.6.3 Modificando el servicio.

Como ya habíamos dicho antes nuestro servicio expone dos métodos para la administración de usuarios. El primer paso será reutilizar los proyectos de entidades, Datos y Negocio, para ello agregaremos las referencias al **proyecto de servicio, agregar, referencias.**



Seguidamente agregaremos una clase al proyecto de Servicio, **clic derecho, agregar, clase**, la llamaremos UsuariosContract.cs el cual será nuestro contrato referente a usuarios que contendrá todas las propiedades que estamos utilizando en la entidad usuarios quedando de la siguiente forma:

```
[DataContract]
public class UsuariosContract
{
    [DataMember]
    public int IdUsuario { get; set; }
    [DataMember]
    public string Usuario { get; set; }
    [DataMember]
    public string Iniciales { get; set; }
    [DataMember]
    public string Nombre { get; set; }
    [DataMember]
    public string ApellidoPrimero { get; set; }
    [DataMember]
    public string ApellidoSegundo { get; set; }
    [DataMember]
    public string Correo { get; set; }
    [DataMember]
    public string Telefono { get; set; }
    [DataMember]
    public string Contraseña { get; set; }
    [DataMember]
    public int IdTipoUsuario { get; set; }
    [DataMember]
    public int IdPerfil { get; set; }
    [DataMember]
    public int IdUsuarioCreacion { get; set; }
}
```

```
}
```

Una vez creado nuestro contrato modificaremos la interface IService1.cs. modificando la declaración de los métodos existentes quedando de la siguiente forma:

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    string Update(UsuariosContract value);
}
```

```
[OperationContract]
string Delete(int value);
}
```

Ahora modificaremos el servicio ya que no está cumpliendo con la implementación de los métodos de acuerdo a la interface quedando por mientras de la siguiente forma:

```
public class Service1 : IService1
{
    public string Delete(int value)
    {
        throw new NotImplementedException();
    }
    public string Update(UsuariosContract values)
    {
        throw new NotImplementedException();
    }
}
```