EF – CODE FIRST

NUEVA BASE DE DATOS

Nos introduciremos al desarrollo de Code First para una nueva base de datos. Este escenario incluye el ámbito de una base de datos que no existe y que Code First creará o una base de datos a la que code First va a agregar nuevas tablas. Code First le permite definir un modelo usando clases de C# o de VB.Net. La configuración adicional se puede realizar también con los atributos en sus clases y propiedades o usando una API fluida.

REQUISITOS PREVIOS

Necesitará tener instalado NuGet .

1. CREAR LA APLICACIÓN

En este primer ejemplo utilizaremos una aplicación de consola básica que use Code First para el acceso a los datos.

- ✓ Abra Visual Studio
- ✓ **Archivo -> Nuevo -> Proyecto**
- ✓ Seleccione **Windows** en el menú de la izquierda y **aplicación de consola**
- ✓ Escriba **PracticoCF** como nombre
- ✓ Seleccione **Aceptar**
- ✓ Repita el proceso pero con un Proyecto de tipo **Biblioteca** de clases y nombrelo **Dominio**

2. AGREGAR SOPORTE A EF

Ahora vamos a comenzar a usar tipos de Entity Framework de modo que necesitamos agregar el paquete de EntityFramework NuGet.

- ✓ Click derecho sobre **Proyecto (Dominio) → Administrar paquetes de NuGet**. (Nota: si no dispone de la opción Administrar paquetes de NuGet, debe instalarlo:(Herramientas→Administrador de extensiones).
- ✓ Seleccione la pestaña **En línea**
- ✓ Seleccione el paquete **EntityFramework**
- ✓ Haga clic en **Instalar**

3. CREAR EL MODELO

Vamos a definir un modelo muy sencillo con las clases Cliente y Direccion en el proyecto Dominio y vamos a probarlas desde el archivo Program.cs en el poryecto PracticoCF. Bajo el proyecto Dominio cree dos clases Cliente y Direccion. Vamos a utilizar anotaciones para definir las caracteristicas de los campos.

```
public class Cliente {

        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        [Key]
        public virtual int Id { get; set; }

        [Required(ErrorMessage = "La cédula es un dato requerido.")]
        public virtual string Ci { get; set; }

        [Required(ErrorMessage = "El nombre es un dato requerido.")]
        [StringLength(50)]
        public virtual string Nombre { get; set; }
```

```
        [Required(ErrorMessage = "El apellido es un dato requerido.")]
        [StringLength(50)]
        public virtual string Apellido { get; set; }

        public virtual List<Direccion> Direcciones { get; set; }

        [NotMapped]
        public virtual bool Seleccionado { get; set; }

        public Cliente() {
            Direcciones = new List<Direccion>();
        }
    }


public class Direccion{

        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        [Key]
        public virtual int Id { get; set; }

        [Required(ErrorMessage = "El nombre de la calle es un dato requerido.")]
        [StringLength(50)]
        public virtual string Calle { get; set; }

        [Required(ErrorMessage = "El número de puerta es un dato requerido.")]
        public virtual int Numero { get; set; }

        public virtual Cliente ElCliente { get; set; }
    }
```

Vamos a hacer las dos propiedades de navegación (Cliente.Direcciones y Direccion.ElCliente) virtuales. Esto habilita la característica de carga diferida de Entity Framework. La carga diferida significa que el contenido de estas propiedades se cargará automáticamente desde la base de datos al intentar tener acceso. Recuerde incluir las bibliotecas System.ComponentModel.DataAnnotations, System.ComponentModel.DataAnnotations.Schema y System.Data.Entity.

La lista completa de anotaciones que EF admite es:

- KeyAttribute
- StringLengthAttribute
- MaxLengthAttribute
- ConcurrencyCheckAttribute
- RequiredAttribute
- TimestampAttribute
- ComplexTypeAttribute
- ColumnAttribute
- TableAttribute
- InversePropertyAttribute
- ForeignKeyAttribute
- DatabaseGeneratedAttribute
- NotMappedAttribute

4. CREAR UN CONTEXTO

Ahora vamos a definir un contexto derivado de DbContext que representa una sesión con la base de datos, lo que nos permite consultar y guardar los datos. Definimos un contexto que deriva de System.Data.Entity.DbContext y expone un DbSet<TEntity> con tipo para cada clase en nuestro modelo.

Agregue una clase RestauranteContext en el proyecto **PracticoCF** e incluya una instrucción using para System.Data.Entity al principio de RestauranteContext.cs.

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

En la Clase RestauranteContext marque que hereda de DbContext y agregue los siguientes DbSet.

```
public class RestauranteContext  : DbContext
{
    public DbSet<Cliente> Clientes { get; set; }
     public DbSet<Direccion> Direcciones { get; set; }
}
```

Ese es todo el código que necesitamos para empezar a almacenar y recuperar datos.

5. LEER Y ESCRIBIR DATOS

Implemente el método Main en Program.cs como se muestra a continuación. Este código crea una nueva instancia de nuestro contexto y la usa para insertar un nuevo Cliente con su direccion asociada. Usaremos una consulta LINQ para recuperar todos los Clientes de la base de datos ordenados alfabéticamente por Apellido y Nombre.

```
class Program
{
    static void Main(string[] args)
       {
           using (var db = new RestauranteContext())
           {
               string salir=string.Empty;

               while (salir != "0"){
                   // Crear y guardar clientes y direcciones
                   Console.Write("Ingrese el nombre para cliente: ");
                   var nom = Console.ReadLine();

                   Console.Write("Ingrese el apellido del cliente: ");
                   var ape = Console.ReadLine();

                   Console.Write("Ingrese la cedula del cliente: ");
                   var ci = Console.ReadLine();

                   Console.Write("Ingrese la calle del cliente: ");
                   var calle = Console.ReadLine();

                   Console.Write("Ingrese el numero de puerta del cliente: ");
                   var numero = int.Parse(Console.ReadLine());

                   var cliente = new Cliente { Nombre = nom, Apellido = ape, Ci = ci };
                   cliente.Direcciones.Add(new Direccion { Calle = calle, Numero = numero, ElCliente =
cliente });
                   db.Clientes.Add(cliente);
                   db.SaveChanges();

                   Console.Write("Para Salir 0. Cualquier otra tecla para ingresar otro cliente.");
                   salir = Console.ReadLine();
```

```
            }
            // Mostrar todos los clientes de la base de datos
            var query = from b in db.Clientes
                        orderby b.Apellido, b.Nombre
                        select b;

            Console.WriteLine("Todos los Clientes de la base de datos:");
            foreach (var item in query)
            {
                string linea = item.Nombre + " " + item.Apellido;
                if (item.Direcciones.Count > 0)
                        linea += ", " + item.Direcciones[0].Calle + " " +
                                        item.Direcciones[0].Numero;
                Console.WriteLine(linea);
            }

            Console.WriteLine("Presione una tecla para salir...");
            Console.ReadKey();
        }
    }
}
```

Ahora puede ejecutar la aplicación y probarla.

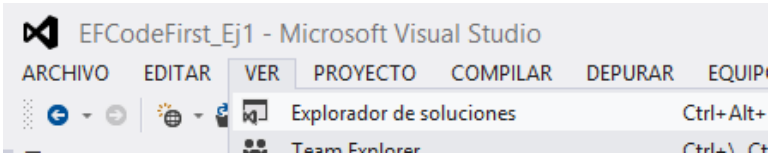
¿DÓNDE ESTÁ LA BASE DE DATOS?

Por convención, DbContext crea una base de datos en alguno de estos lugares.

- ✓ Si existe una instancia local de SQL Express (se instala de forma predeterminada con Visual Studio 2010), Code First ha crea la base de datos en esa instancia
- ✓ Si SQL Express no está instalado, Code First intenta usar LocalDb (se instala de forma predeterminada con Visual Studio 2012)
- ✓ La base de datos creada se va a llamar con el nombre completo del contexto derivado, en nuestro caso **PracticoCF .RestauranteContext**

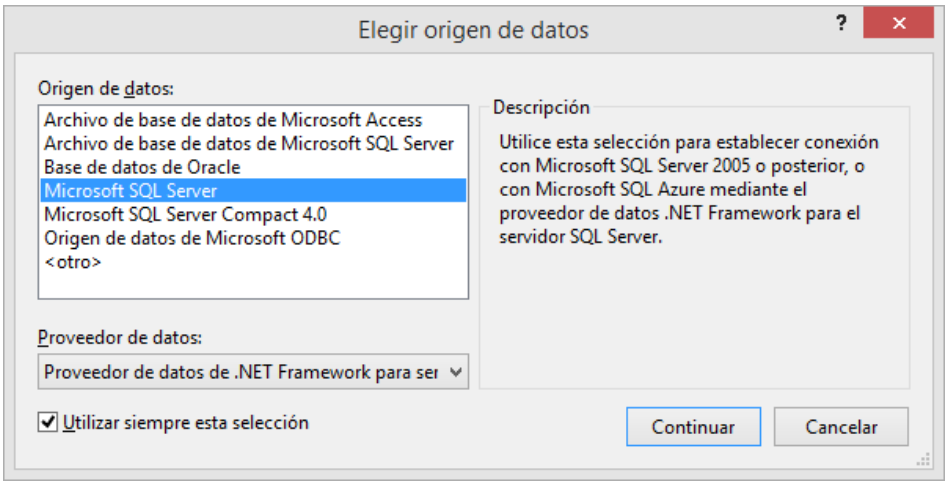
Estas son solo las convenciones predeterminadas y hay varias maneras de cambiar la base de datos que Code First usa. Podemos conectarnos a la base ya sea por el SQL Server Management Studio o con el Explorador de servidores en Visual Studio

- ✓ **Ver -> Explorador de servidores**



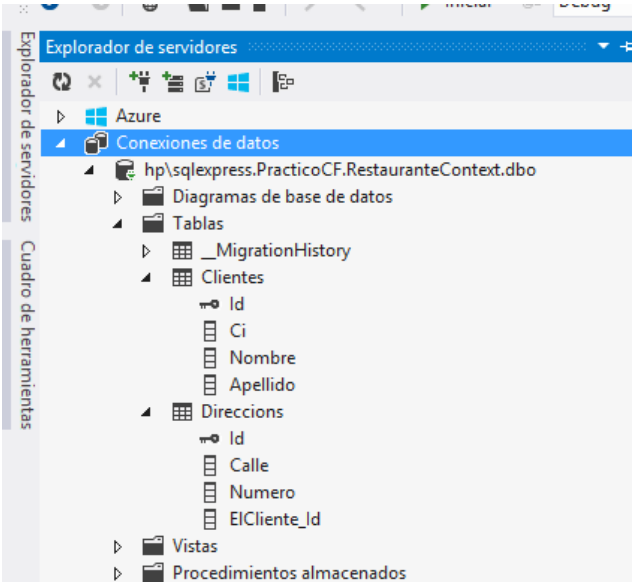
- ✓ Haga clic con el botón secundario en **Conexiones de datos** y seleccione **Agregar conexión**

- ✓ Si no se ha conectado a una base de datos desde el Explorador de servidores antes, tendrá que seleccionar Microsoft SQL Server como origen de datos



- ✓ Conéctese a LocalDb ((localdb)\v11.0) o a SQL Express (.\SQLEXPRESS), según la que haya instalado:

Ahora podemos inspeccionar el esquema que Code First creó.



DbContext determinó las clases que se incluyen en el modelo examinando las propiedades de DbSet que definimos. Después, usa el conjunto predeterminado de convenciones de Code First para determinar los nombres de tabla y columna, los tipos de datos, las claves principales, etc.

5. TRATAR LOS CAMBIOS EN EL MODELO

Ahora vamos a modificar el modelo y eso hara necesario actualizar el esquema de la base de datos. Para esto vamos a usar una característica denominada migraciones de Code First.

Las migraciones nos permiten tener un conjunto ordenado de pasos que describen cómo actualizar (y degradar) nuestro esquema de la base de datos. Cada uno de estos pasos contiene código que describe los cambios que se aplicarán.

El primer paso es habilitar Migraciones de Code First en nuestro RestauranteContext .

- ✓ **Herramientas -> Administrador de paquetes NuGet -> Consola del Administrador de paquetes**
- ✓ Ejecute el comando ***Enable-Migrations*** (Habilitar-migraciones**)** en la Consola del Administrador de paquetes
- ✓ Una carpeta de migraciones se ha agregado a nuestro proyecto y contiene dos elementos:

- ✓ **Configuration.cs**: este archivo contiene las opciones que las migraciones usarán para migrar RestauranteContext . No necesitamos cambiar nada para este tutorial pero aquí es donde puede especificar los datos de inicialización, registrar los proveedores para otras bases de datos, cambiar el espacio de nombres en que las migraciones se generan, etc.
- ✓ **<timestamp>_InitialCreate.cs**: se trata de la primera migración, representa los cambios que ya se han aplicado a la base de datos para llevarla de una base de datos vacía a una que incluye las tablas de Clientes y Direcciones. Aunque permitiéramos que Code First creara automáticamente estas tablas en nuestro lugar, ahora que hemos optado por las migraciones, se han convertido en una migración. Code First también ha registrado en nuestra base de datos local que esta migración ya se ha aplicado. La marca de tiempo en el nombre de archivo se usa para la clasificación.

Mensaje de la consola: Detected database created with a database initializer. Scaffolded migration '201502181935255_InitialCreate' corresponding to existing database. To use an automatic migration instead, delete the Migrations folder and re-run Enable-Migrations specifying the -EnableAutomaticMigrations parameter.

Ahora vamos a realizar un cambio en nuestro modelo, agregando una propiedad Email a la clase Cliente:

```
public class Cliente
{
..
    public virtual  string Email { get; set; }
..
}
```

- ✓ Ejecute el comando **Add-Migration AddEmail** en la Consola del Administrador de paquetes.

  El comando Add-Migration comprueba si hay cambios desde la última migración y aplica scaffold a una nueva migración con los cambios encontrados. Podemos dar un nombre a las migraciones; en este caso, le denominamos "AddEmail".

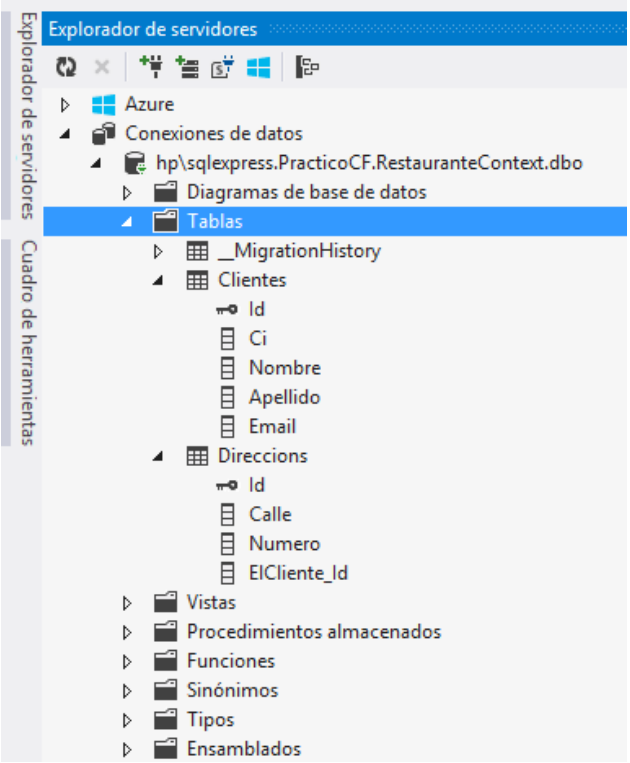
  El código indica que tenemos que agregar una columna Email, que puede contener datos de cadena, a la tabla dbo.Clientes.

```
namespace PracticoCF.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddEmail : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Clientes", "Email", c => c.String());
        }

        public override void Down()
        {
            DropColumn("dbo.Clientes", "Email");
        }
    }
}
```

- ✓ Ejecute el comando **Update-Database** en la Consola del Administrador de paquetes. Este comando aplicará las migraciones pendientes a la base de datos. Nuestra migración InitialCreate ya se ha aplicado, de modo que las migraciones solo se aplicarán a nuestra nueva migración AddEmail.

Sugerencia: puede usar el modificador – Verbose al llamar a Update-Database para ver el código SQL que se ejecuta en la base de datos.

La nueva columna Email se agrega ahora a la tabla de Restaurantes de la base de datos:



6. CONVENCIONES

Si no hubiesemos usado antaciones podriamos dejar que EF detecte el modelo usando sus convenciones predeterminadas, por ejemplo, si definieramos los campos clave de las clases como NombreclaseId, Ej::

```
public class Cliente{
     public virtual int ClienteId  { get; set; }
}
```

7. API FLUIDA

Para los casos en que nuestras clases no siguen las convensiones necesitamos hacer configuraciones adicionales. Las dos formas de hacer esto es con anotaciones (como hicimos en la primera parte del ejercicio) o con la API fluida de Code First.

La mayor parte de la configuración del modelo se puede hacer con anotaciones de datos simples. La API fluida es una forma más avanzada de especificar la configuración del modelo que abarca todo lo que las anotaciones de datos pueden hacer además de otras configuraciones avanzadas que no son posibles con las anotaciones de datos. Las anotaciones de datos y la API fluida se pueden usar juntas.

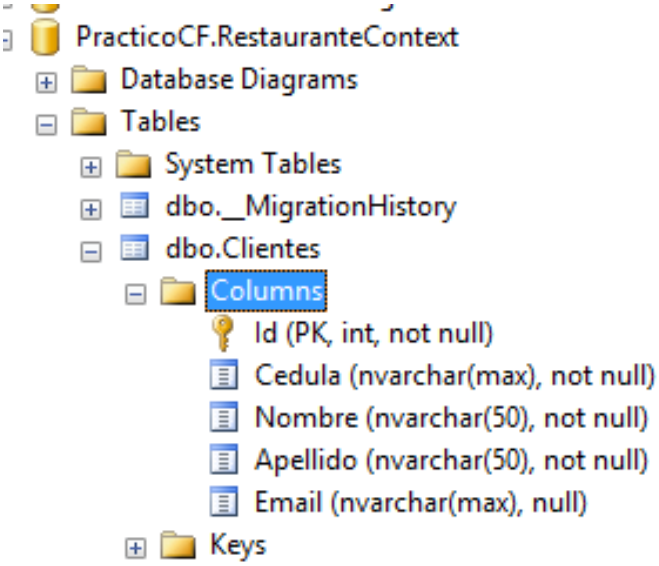
Para tener acceso a la API fluida, sobreescriba el método OnModelCreating de DbContext. Vamos a cambiar el nombre de la columna en que almacenamos Cliente.Ci por Cedula.

- Sobreescriba el método OnModelCreating en RestauranteContext con el siguiente código

```
public class RestauranteContext  : DbContext
{
    public DbSet<Restaurante> Restaurantes { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Cliente>()
            .Property(u => u.Ci)
            .HasColumnName("Cedula");
    }
}
```

- Use el comando **Add-Migration ChangeCedula** para aplicar scaffold a una migración y aplicar estos cambios a la base de datos.
- Ejecute el comando **Update-Database** para aplicar la nueva migración a la base de datos.

El nombre de la columna Ci se cambia por Cedula:



RESUMEN

Examinamos el desarrollo de Code First con una nueva base de datos.

Definimos un modelo con clases y después usamos ese modelo para crear una base de datos y almacenar y recuperar los datos.

Una vez que se creó la base de datos, usamos las Migraciones de Code First para cambiar el esquema.

También vimos cómo configurar un modelo mediante anotaciones de datos y la API fluida.