Code First un ejemplo:

1. Crear un nuevo proyecto y llamarlo Tienda.
2. Agregar un nuevo proyecto de Biblioteca de clases a la solución y llamarlo Models.
3. Agregar una referencia a Models desde Tienda.
4. Agregar al proyecto Models el paquete Entity Framework desde
5. Agregar el siguiente código al proyecto Models, con el que estamos definiendo nuestro modelo sólo a través de código y clases POCO

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Data.Entity;    namespace Models  {      public class TiendaContext : DbContext      {          public DbSet<Cliente> Clientes { get; set; }          public DbSet<Producto> Productos { get; set; }          public DbSet<Pedido> Pedidos { get; set; }          public DbSet<LineaPedido> LineasPedido { get; set; }      }        public class Cliente      {       public Cliente()          {              Pedidos = new HashSet<Pedido>();          }          public  int ClienteId { get; set; }          public  string Nombre { get; set; }          public virtual ICollection<Pedido> Pedidos { get; set; }      }        public class Producto      {          // Comentar si ProxyCreationEnabled = true          public Producto()          {              Lineas = new HashSet<LineaPedido>();          }          public  int ProductoId { get; set; }          public  string Descripcion { get; set; }          public  decimal Precio { get; set; }          public virtual ICollection<LineaPedido> Lineas { get; set; }      }        public class Pedido      {          // Comentar si ProxyCreationEnabled = true          public Pedido()          {              Lineas = new HashSet<LineaPedido>();          }          public  int PedidoId { get; set; }          public  DateTime FechaCreacion { get; set; }          public  int ClienteId { get; set; }          public virtual Cliente Cliente { get; set; }          public virtual ICollection<LineaPedido> Lineas { get; set; }      }        public class LineaPedido      {          public  int LineaPedidoId { get; set; }          public  int PedidoId { get; set; }          public  int ProductoId { get; set; }          public  int Unidades { get; set; }          public virtual Pedido Pedido { get; set; }          public virtual Producto Producto { get; set; }      }  } |

A grandes rasgos, para habilitar la creación automática de proxys de **carga diferida** debemos especificar virtual en todas las propiedades de navegación y establecer a true las propiedades *LazyLoadingEnabled* y *ProxyCreationEnabled* (por defecto están activadas).

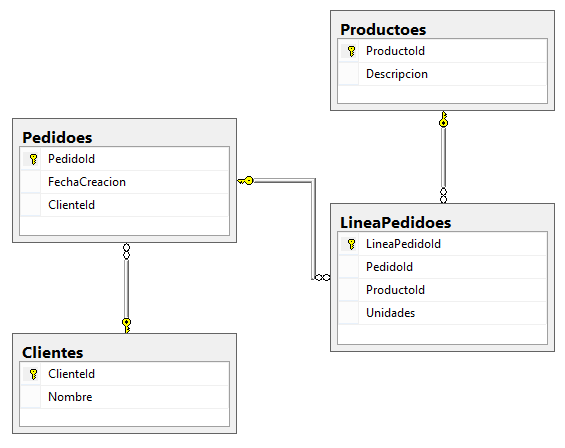
Para los proxys de **seguimiento de cambios**, todas las propiedades de nuestra clase deben ser virtual y las propiedades de navegación de colección tienen que ser *ICollection* (las cuales serán convertidas después a *EntityCollection<TEntity>*). Además tiene que valer *true* la propiedad *ProxyCreationEnabled* (fíjate que aquí no hay una propiedad específica para habilitar o deshabilitar la creación de estos proxys como sí la hay para los proxys de carga diferida).

En lo relativo a inicializar las propiedades de navegación de colección en el constructor de la clase, sirve para cuando no generamos ningún tipo de proxy y queremos inicializar la colección. Si este código está presente y está activa la generación de proxys de seguimiento de cambios se producirá una excepción (no así con los proxys de carga diferida).

En cuanto a si es o no conveniente la creación automática de proxys, hay todo un debate abierto con gente a favor y gente en contra. deshabilitaremos la creación de cualquier tipo de proxy (con *ProxyCreationEnabled* igual a false). Estos proxys serán del tipo *System.Data.Entity.DynamicProxies.EntityType\_…*, si por algún motivo se quisiera saber el tipo base en tiempo de ejecución (el tipo original de la entidad) podemos obtenerlo con *ObjectContext.GetObjectType(entity.GetType())*.

1. Por último, simplemente utilizar el modelo a través de la clase de contexto.

Que ha generado automáticamente la base de datos Models.TiendaContext en nuestra instalación de SQL Express con las siguientes tablas y relaciones:

[](http://lh5.ggpht.com/-gRYMgO0qApM/UWv_7TFcTGI/AAAAAAAABgQ/ONs3uyc72X4/s1600-h/clip_image001%5b4%5d%5b2%5d.png)

A partir de aquí, surgen muchas preguntas:

* ¿Por qué se creó la base de datos en SQL Express y por qué con ese nombre?
* ¿En qué momento se creó la base de datos?
* ¿Qué pasa si el modelo cambia?
* ¿Qué pasa si la estructura de base de datos no me convence, puedo cambiar algo?

Lo primero es entender cómo funciona Code First.

El primer paso que lleva a cabo Code First es leer las clases accesibles en la clase de contexto y crear un modelo en memoria. A continuación infiere un esquema de base de datos válido para persistir el modelo.

Code First se basa en convenciones, por ejemplo:

1. si nuestra clase tiene una propiedad *<NombreClase>Id* o simplemente ***Id***, esta propiedad será elegida para ser la clave primaria en la base de datos. Si además es numérica, creará el campo clave como autonumérico, etc.

¿Qué cuáles son exactamente las convenciones de Code First para inferir el esquema de base de datos? Pues son muchas, todas ellas en el espacio de nombres ***System.Data.Entity.ModelConfiguration.Conventions***, pero a grandes rasgos las más relevantes que se han utilizado para crear nuestro modelo han sido las siguientes:

* El nombre de la base de datos ha sido el nombre del contexto plenamente cualificado
* La base de datos se ha creado en la instancia de SQL Express.
* El nombre de la tabla será el nombre de la clase en plural. Como el servicio de pluralización sólo está disponible en inglés, tenemos nombres tan divertidos y ocurrentes como “Pedidoes”. Lógicamente esto no es muy acertado y después veremos cómo solucionarlo.
* La clave primaria de cada tabla ha sido <NombreClase>Id.
* Las claves primarias son todas autonuméricas.
* A partir de las propiedades de navegación y propiedades de clave externa que encontró en el modelo, creó en base de datos las relaciones oportunas. Code First reconoció las relaciones porque los nombres de las mismas siguieron las convenciones predeterminadas. Por ejemplo:
  + Clientes.Pedidos es una propiedad de navegación de colección.
  + Pedidos.Cliente es una propiedad de navegación de referencia
  + Pedidos.ClienteId es una propiedad de clave externa que será utilizada para llevar a cabo la relación en base de datos entre Pedidos y Clientes.

Como podemos ver, la mayoría de las convenciones de Code First parecen muy razonables y pienso que merece la pena cumplirlas para trabajar lo menos posible. Sin embargo, puede haber situaciones en las que alguna convención no nos satisfaga (como por ejemplo el nombre en plural de las tablas) o bien no podamos cumplirla por algún motivo (imagina por ejemplo que el campo Pedidos.ClienteId tuviera que ser creado en código con el nombre Pedidos.ClienteAsociadoId).

En estos casos, lo que necesitamos es poder invalidar las convenciones predeterminadas de Code First e instruirle de forma precisa sobre cómo queremos que se realicen ciertos mapeos. Es decir, estamos tomando el control y decidiendo por nosotros mismos como inferir el esquema. Además de poder eliminar convenciones, lo más habitual es realizar ajustes a través de alguna de estas vías:

* DataAnnotations.
* Fluent API.
* Cuando elegir una u otra es un tema de profundo debate, pero a grandes rasgos:
* DataAnnotations es más sencillo de utilizar que Fluent API.
* Con DataAnnotations hay ciertos escenarios que no podemos conseguir.
* Con DataAnnotations “ensuciamos” en cierto modo nuestras clases POCO, tanto a la vista del desarrollador como a efectos de interoperabilidad.
* Con Fluent API tenemos más control
* Al respecto de los ajustes, la precedencia que aplica Code First es primero Fluent API, después DataAnnotations y por último las convenciones predeterminadas.

Dicho esto, para ajustar nuestro modelo veremos ambas opciones y así después podremos decidir.

Los ajustes que realizaremos a modo de ejemplo sobre la entidad Pedido serán:

* Elegir el nombre “Pedidos” para la tabla.
* Crear el campo PedidoId como no autonumérico (queremos controlar el número de pedido).
* La propiedad de clave externa con Clientes tiene que llamarse ClienteAsociadoId en nuestro modelo pero tiene que continuar llamándose ClienteId en la base de datos (está claro que este requerimiento es un poco absurdo, pero nos servirá para el ejemplo).

Para ajustar el modelo con DataAnnotations tendremos que agregar una referencia al ensamblado *System.ComponentModel.DataAnnotations* en nuestro proyecto Models y decorar la clase Pedido con los siguientes atributos:

|  |
| --- |
| [Table("Pedidos")]      public class Pedido      {          [DatabaseGenerated(DatabaseGeneratedOption.None)]          public int PedidoId { get; set; }          public DateTime FechaCreacion { get; set; }          [ForeignKey("Cliente")]          [Column("ClienteId")]          public int ClienteAsociadoId { get; set; }          public Cliente Cliente { get; set; }          public virtual List<LineaPedido> Lineas { get; set; }      } |

Si esto mismo lo hiciéramos con Fluent API, tendríamos que agregar una referencia en Models para el ensamblado *System.Data.Entity* y escribir los ajustes en el método*OnModelCreating* de nuestra clase de contexto:

|  |
| --- |
| protected override void OnModelCreating(DbModelBuilder modelBuilder)  {      // Eliminar convención de pluralización de nombres de tablas      modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();        modelBuilder.Entity<Pedido>().ToTable("Pedidos");      modelBuilder.Entity<Pedido>()                  .Property(p => p.PedidoId)                  .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);      modelBuilder.Entity<Pedido>()                  .Property(p => p.ClienteAsociadoId)                  .HasColumnName("ClienteId");      modelBuilder.Entity<Pedido>()                  .HasRequired(p => p.Cliente)                  .WithMany(p => p.Pedidos)                  .HasForeignKey(p => p.ClienteAsociadoId);  } |

A priori Fluent API parece mucho código para tan poco requerimiento, además parece algo complicado pero, lo cierto es que, después de entender cómo funciona y con algo de práctica, nos daremos cuenta que no se llama API Fluida por nada, realmente fluirá y nos resultará sencillo realizar ajuste por esta vía con poco esfuerzo.

En este punto, algo interesante es saber cómo podríamos gestionar estos ajustes para no ensuciar las clases (si hablamos de DataAnnotations) o no tener un métodoOnModelCreating de 1000 líneas (si hablamos de Fluent API).

Para el caso de DataAnnotations podríamos utilizar las famosas clases buddy (clase colega) para separar la clase POCO de los atributos.

|  |
| --- |
| [MetadataType(typeof(PedidoMetadata))]      public class Pedido      {          public int PedidoId { get; set; }          public DateTime FechaCreacion { get; set; }          public int ClienteAsociadoId { get; set; }          public Cliente Cliente { get; set; }          public virtual List<LineaPedido> Lineas { get; set; }      }        [Table("Pedidos")]      public class PedidoMetadata      {          [DatabaseGenerated(DatabaseGeneratedOption.None)]          public int PedidoId { get; set; }          [ForeignKey("Cliente")]          [Column("ClienteId")]          public int ClienteAsociadoId { get; set; }      } |

En este ejemplo podemos ver:

         Se ha indicado la clase buddy con el atributo *MetadataType*, en la clase Pedido.

         En la clase PedidoMetadata (llamada así por convención pero podría haberse llamado de cualquier otra forma) hemos indicado los *DataAnnotations* sólo para las propiedades que queremos ajustar.

Con esto hemos conseguido separar la definición de nuestra clase de las*DataAnnotations* para Code First.

Si queremos lograr esta misma separación de conceptos a través de Fluent API, tendremos que crear una configuración para nuestra clase Pedido en una nueva clase que herede de *EntityTypeConfiguration* y registrarla durante el evento *OnModelCreating*.

Primero la clase de configuración:

|  |
| --- |
| public class PedidoConfiguration : EntityTypeConfiguration<Pedido>  {      public PedidoConfiguration()      {          ToTable("Pedidos");          Property(p => p.PedidoId)                      .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);          Property(p => p.ClienteAsociadoId)                      .HasColumnName("ClienteId");          HasRequired(p => p.Cliente)                          .WithMany(p => p.Pedidos)                          .HasForeignKey(p => p.ClienteAsociadoId);      }  }  Después registrarla en el evento de creación del modelo:  protected override void OnModelCreating(DbModelBuilder modelBuilder)  {      modelBuilder.Configurations.Add(new PedidoConfiguration());  } |

Llegados a este punto, ya sabemos cómo se infiere el modelo y cómo podemos ajustarlo, así que ha llegado el momento de saber cuándo, cómo y dónde se crea la base de datos.

Por defecto, Code First intenta crear la base de datos en el primer acceso a la misma (en nuestro ejemplo anterior ocurrió cuando intentamos agregar un cliente a TiendaContext.Clientes). Además la intenta crear en la instancia de SQL Express (.\SQLEXPRESS) o en su defecto en [LocalDb](http://henry416.wordpress.com/2013/12/31/undocumented-sql-server-2012-express-localdb/) ((localdb)\v11.0), motor de base de datos que viene incluido a partir de VS 2012. Respecto al nombre de la base de datos elegirá el nombre del contexto plenamente cualificado (Models.TiendaContext). Cómo lógicamente casi nunca nos servirá esta localización de base de datos, comencemos por ver cómo elegir donde crear la base de datos.

La forma más sencilla de elegir donde crear la base de datos es crear una cadena de conexión en nuestro fichero web.config/app.config con el nombre del contexto. Si hacemos esto también tendremos que agregar una referencia al paquete EF en el proyecto para evitar el error “No Entity Framework provider found for the ADO.NET provider with invariant name 'System.Data.SqlClient'

|  |
| --- |
| <connectionStrings>      <add name="TiendaContext"           providerName="System.Data.SqlClient"           connectionString="Data Source=(local)\sqlexpress;Initial Catalog=Tienda;Integrated Security=SSPI;MultipleActiveResultSets=True;Application Name=Tienda" />    </connectionStrings> |

De la cadena de conexión podemos comentar lo siguiente:

* El atributo “*name*” puede ser el nombre de la clase de contexto o el nombre plenamente cualificado de la clase de contexto, es decir, valdría tanto TiendaContext como Models.TiendaContext.
* No olvidar agregar el parámetro *MultipleActieResultSets=True* (necesario para el correcto funcionamiento de Entity Framework).
* Agregar un nombre de aplicación si no quieres después sorpresas con*TransationScope*.
* Otra forma de especificar donde conectará Code First es a través del constructor de*DbContext*.

|  |
| --- |
| public class TiendaContext : DbContext      {          public TiendaContext()          {          }          public TiendaContext(string nameOrConnectionString)              : base(nameOrConnectionString)          {          }      } |

Ahora serían válidas todas estas combinaciones:

|  |
| --- |
| // Por defecto, cadena de conexión con el nombre TiendaContext  var context = new TiendaContext();    // Cadena de conexión con el nombre Tienda  // Sino existe creará una base de datos con el nombre Tienda  var context2 = new TiendaContext("Tienda");    // Cadena de conexión con el nombre Tienda  // Sino existe fallará  var context2 = new TiendaContext("Name=Tienda");    // Cadena de conexión explícita  var connectionString =      @"Data Source=(local)\sqlexpress;"+      "Initial Catalog=Tienda;"+      "Integrated Security=SSPI;"+      "MultipleActiveResultSets=True;"+      "Application Name=Tienda";  var context3 = new TiendaContext(connectionString); |

Ahora que ya sabemos especificar la localización exacta de la base de datos, todavía nos queda pendiente controlar el momento de la creación de la misma. Por defecto, Code First intentará crearla la primera vez que la necesite según la API de *DbContext*. En cualquier caso siempre podemos explícitamente decidir cuándo crearla con el método *Initialize*:

using (var context = new TiendaContext())

{

    context.Database.Initialize(false);

}

El objeto *Database*, además de *Initialize* nos suministra algunos otros métodos muy útiles:

         CreateIfNotExists

         CompatibleWithModel

         Delete

         Create

         ExecuteSqlCommand

Especialmente interesante es el método CompatibleWithModel. Este método nos informa si el modelo actual es o no compatible con el esquema de la base de datos. Que devuelva true o false depende del valor asignado al parámetro throwIfNoMetada y de la existencia o no de la tabla \_\_MigrationHistory. Esta tabla es creada por Code First cuando se crea automáticamente la base de datos o cuando se habilita Code First Migrations CompatibleWithModel tiene las siguientes combinaciones:

|  |  |  |
| --- | --- | --- |
| throwIfNoMetada | Existe \_\_MigrationHistory | Resultado |
| false | No | true, no hay forma de comparar el modelo con el esquema y entonces se asume que es compatible. |
| false | Sí | true o false en función de si el modelo es o no compatible con el esquema. |
| true | No | Se lanzará una excepción porque no se han podido comparar modelo y esquema. |
| true | Sí | true o false en función de si el modelo es o no compatible con el esquema. |

Sea como fuere, la pregunta que surge a continuación es ¿Qué pasa si el modelo cambia? Es decir, si el esquema de base de datos no es compatible con el modelo ¿Qué ocurrirá? Pues la respuesta a estas preguntas son las distintas estrategias de inicialización de Code First.

Por defecto, Code First incorpora las siguientes estrategias de inicialización:

* *CreateDatabaseIfNotExists*
* *DropCreateDatabaseIfModelChanges*
* *DropCreateDatabaseAlways*

Con ***CreateDatabaseIfNotExists*,** si la base de datos no existe se crea, y si existe y el modelo no es compatible se lanza un error (aunque puedes no tener \_\_*MigrationHistory*y entonces devolverá siempre que la base de datos y el modelo son compatibles).

Con ***DropCreateDatabaseIfModelChanges*** la diferencia está en que si el modelo no es compatible con la base de datos, la misma se eliminara y se volverá a crear de nuevo (aquí es obligado tener \_\_*MigrationHistory* porque si no lanzará una excepción).

Por último, con DropCreateDatabaseAlways siempre se elimina y se crea la base de datos, sin importar si es o no compatible con el modelo (lógicamente aquí tampoco importa si tienes o no \_\_MigrationHistory, simplemente no se consulta).

Por defecto, la estrategia activa es CreateDatabaseIfNotExists (¡menos mal!). Esta estrategia es la única que nos garantiza que nuestra aplicación en producción no eliminará vilmente nuestra base de datos en cada ejecución o si el modelo cambia. Por el contrario, si el modelo no es compatible la inicialización fallará y nuestra aplicación quedará inaccesible. Como resolver este escenario lo veremos más adelante.

Respecto a la estrategia DropCreateDatabaseIfModelChanges, es normalmente utilizada en el entorno de desarrollo donde los datos son “prescindibles” y queremos agilidad a la hora de programar nuestra aplicación sin importar si ha habido o no cambios en el modelo.

Por último, con DropCreateDatabaseAlways estamos yendo un paso más allá y podría resultar útil si realizamos pruebas unitarias que tengan acceso a la base de datos y queremos asegurarnos de disponer siempre de una base de datos vacía en cada ejecución.

El cómo establecer un inicializador es muy sencillo:

Database.SetInitializer(new CreateDatabaseIfNotExists<TiendaContext>());

using (var context = new TiendaContext())

{

    // hacer algo...

}

Llegado el caso también podemos no establecer ningún inicializador. Esto puede resultarnos útil si estamos trabajando contra una base de datos existente y no queremos que Code First lleve a cabo ninguna estrategia de inicialización.

Database.SetInitializer<TiendaContext>(null);

Como era de suponer, podemos crear nuestra propia estrategia de inicialización personalizada para Code First. Para nuestro ejemplo, crearemos un inicializador con las siguientes características:

* Si el modelo cambia, se recreará la base de datos.
* También se recreará la base de datos si se encuentra un setting DropIsRequired con el valor true en nuestro app.config.
* Por otro lado, también debería ser posible ejecutar sentencias sql personalizadas después de haber creado la base de datos.

El código de inicializador, a continuación:

    public class DropCreateIfModelChangesOrDropIsRequired<TContext>

        : IDatabaseInitializer<TContext>

where TContext : DbContext

    {

public IEnumerable<string> Sentences { get; set; }

        public void InitializeDatabase(TContext context)

        {

            var created = false;

            if (!context.Database.Exists())

            {

                context.Database.Create();

                created = true;

            }

            else

            {

                var dropIsRequired = false;

                if (ConfigurationManager.AppSettings["DropIsRequired"] != null)

                {

                    var result = false;

                    if (bool.TryParse(ConfigurationManager.AppSettings["DropIsRequired"], outresult))

                    {

                        dropIsRequired = result;

                    }

                }

                if (dropIsRequired || !context.Database.CompatibleWithModel(false))

                {

                    context.Database.Delete();

                    context.Database.Create();

                    created = true;

                }

            }

            if (created && Sentences != null)

            {

                foreach (var sentence in Sentences)

                {

                    context.Database.ExecuteSqlCommand(sentence);

                }

            }

        }

    }

Y el código necesario para la inicialización:

var initializer = new DropCreateIfModelChangesOrDropIsRequired<TiendaContext>();

initializer.Sentences = new List<string>()

    {

        "ALTER DATABASE CURRENT SET RECOVERY SIMPLE"

    };

Database.SetInitializer(initializer);

Otro punto interesante es saber cómo podemos agregar datos iniciales a nuestra base de datos. En los inicializadores que trae Code First de serie, podemos sobreescribir el método virtual *Seed* para centralizar la inicialización de datos después de que la estrategia de inicialización haya concluido. Por ejemplo, para *CreateDatabaseIfNotExists*

    public class CreateDatabaseIfNotExistsWithSeedData :CreateDatabaseIfNotExists<TiendaContext>

    {

        protected override void Seed(TiendaContext context)

        {

            context.Clientes.Add(new Cliente() { Nombre = "Sergio" });

        }

    }

Si hemos optado por crear un inicializador personalizado, no podremos sobreescribir este método (básicamente porque no existe) pero nada impide que lo implementemos y lo utilicemos igualmente. Por ejemplo, a nuestra anterior clase*DropCreateIfModelChangesOrDropIsRequired* le agregaremos el método *Seed* y una llamada a *context.SaveChanges* si la base de datos fue creada correctamente:

public class DropCreateIfModelChangesOrDropIsRequired<TContext>

    : IDatabaseInitializer<TContext>

    where TContext : DbContext

{

    public IEnumerable<string> Sentences { get; set; }

    public void InitializeDatabase(TContext context)

    {

        var created = false;

        if (!context.Database.Exists())

        {

            context.Database.Create();

            created = true;

        }

        else

        {

            var dropIsRequired = false;

            if (ConfigurationManager.AppSettings["DropIsRequired"] != null)

            {

                var result = false;

                if (bool.TryParse(ConfigurationManager.AppSettings["DropIsRequired"], outresult))

                {

                    dropIsRequired = result;

                }

            }

            if (dropIsRequired || !context.Database.CompatibleWithModel(true))

            {

                context.Database.Delete();

                context.Database.Create();

                created = true;

            }

        }

        if (created)

        {

            if (Sentences != null)

            {

                foreach (var sentence in Sentences)

                {

                    context.Database.ExecuteSqlCommand(sentence);

                }

            }

            Seed(context);

            context.SaveChanges();

        }

    }

**protected virtual void Seed(TContext context)**

**{**

**}**

}

Ahora crearemos una nueva clase que herede de*DropCreateIfModelChangesOrDropIsRequired* y que fijará el parámetro genérico al tipo TiendaContext. Además también aprovecharemos para incluir en el constructor por defecto nuestras sentencias personalizadas SQL:

    public class DropCreateIfModelChangesOrDropIsRequiredTiendaContext :DropCreateIfModelChangesOrDropIsRequired<TiendaContext>

    {

        protected override void Seed(TiendaContext context)

        {

            context.Clientes.Add(new Cliente() { Nombre = "Cliente por defecto" });

        }

        public DropCreateIfModelChangesOrDropIsRequiredTiendaContext()

        {

            Sentences = new List<string>() {

                        "ALTER DATABASE CURRENT SET RECOVERY SIMPLE" };

        }

    }

Ahora nuestra inicialización pasaría a ser la siguiente:

var initializer = new DropCreateIfModelChangesOrDropIsRequiredTiendaContext();

Database.SetInitializer(initializer);

Con esto habríamos logrado crear una nueva estrategia de inicialización personalizada de Code First y además implementar la inicialización de datos y sentencias personalizadas de SQL..

Cómo podía cambiar de estrategia de inicialización según la configuración activa (algo parecido a lo que hicimos con las cadenas de conexión según estábamos en Debug o Release). Es decir, imagina que en Debug queremos el inicializador *DropCreateIfModelChangesOrDropIsRequiredTiendaContext*y en Release queremos CreateDatabaseIfNotExists . Existe una solución que viene de serie con Code First y es la de elegir la estrategia de inicialización a través del fichero de configuración (de nuevo nuestro recurrente fichero app.config).

Para seleccionar la estrategia adecuada a través del App.config tendremos que agregar un setting con la clave *DatabaseInitializerForType*. Esto y junto a la transformación de ficheros de configuración, nos permitirán cambiar de estrategia sin tocar ni una sola línea de código.