EBook Gratis

APRENDIZAJE Entity Framework

Free unaffiliated eBook created from
Stack Overflow contributors.

#entity-framework

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: entity-framework

It is an unofficial and free Entity Framework ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Entity Framework.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Entity Framework

Observaciones

Entity Framework (EF) es un asignador relacional de objetos (ORM) que permite a los desarrolladores de .NET trabajar con datos relacionales utilizando objetos específicos del dominio. Elimina la necesidad de la mayoría del código de acceso a datos que los desarrolladores generalmente necesitan escribir.

Entity Framework le permite crear un modelo escribiendo código o utilizando cuadros y líneas en el Diseñador EF. Ambos enfoques se pueden utilizar para apuntar a una base de datos existente o crear una nueva base de datos.

Entity Framework es el ORM principal que Microsoft proporciona para .NET Framework y la tecnología de acceso a datos recomendada por Microsoft.

Versiones

Versión	Fecha de lanzamiento
1.0	2008-08-11
4.0	2010-04-12
4.1	2011-04-12
4.1 Actualización 1	2011-07-25
4.3.1	2012-02-29
5.0	2012-08-11
6.0	2013-10-17
6.1	2014-03-17
Core 1.0	2016-06-27

Notas de la versión: https://msdn.microsoft.com/en-ca/data/jj574253.aspx

Examples

Usando Entity Framework desde C # (Primero el Código)

El código primero le permite crear sus entidades (clases) sin usar un diseñador de GUI o un

archivo .edmx. Primero se llama *Código* , porque puede crear sus modelos *primero* y *Entity framework* creará la base de datos según sus asignaciones automáticamente. O también puede usar este enfoque con la base de datos existente, que se llama *código primero con la base de datos existente.* Por ejemplo, si desea que una tabla contenga una lista de planetas:

```
public class Planet
{
    public string Name { get; set; }
    public decimal AverageDistanceFromSun { get; set; }
}
```

Ahora cree su contexto, que es el puente entre sus clases de entidad y la base de datos. Dale una o más propiedades `DbSet<>` :

```
using System.Data.Entity;

public class PlanetContext : DbContext
{
    public DbSet<Planet> Planets { get; set; }
}
```

Podemos usar esto haciendo lo siguiente:

```
using(var context = new PlanetContext())
{
    var jupiter = new Planet
    {
        Name = "Jupiter",
        AverageDistanceFromSun = 778.5
    };

    context.Planets.Add(jupiter);
    context.SaveChanges();
}
```

En este ejemplo, creamos un nuevo `Planet` con la propiedad `Name` con el valor de `"Jupiter"` y la propiedad `AverageDistanceFromSun` con el valor de `778.5`

Luego podemos agregar este `Planet` al contexto usando el `DbSet Add() DbSet` y confirmar nuestros cambios a la base de datos usando el método `SaveChanges()` .

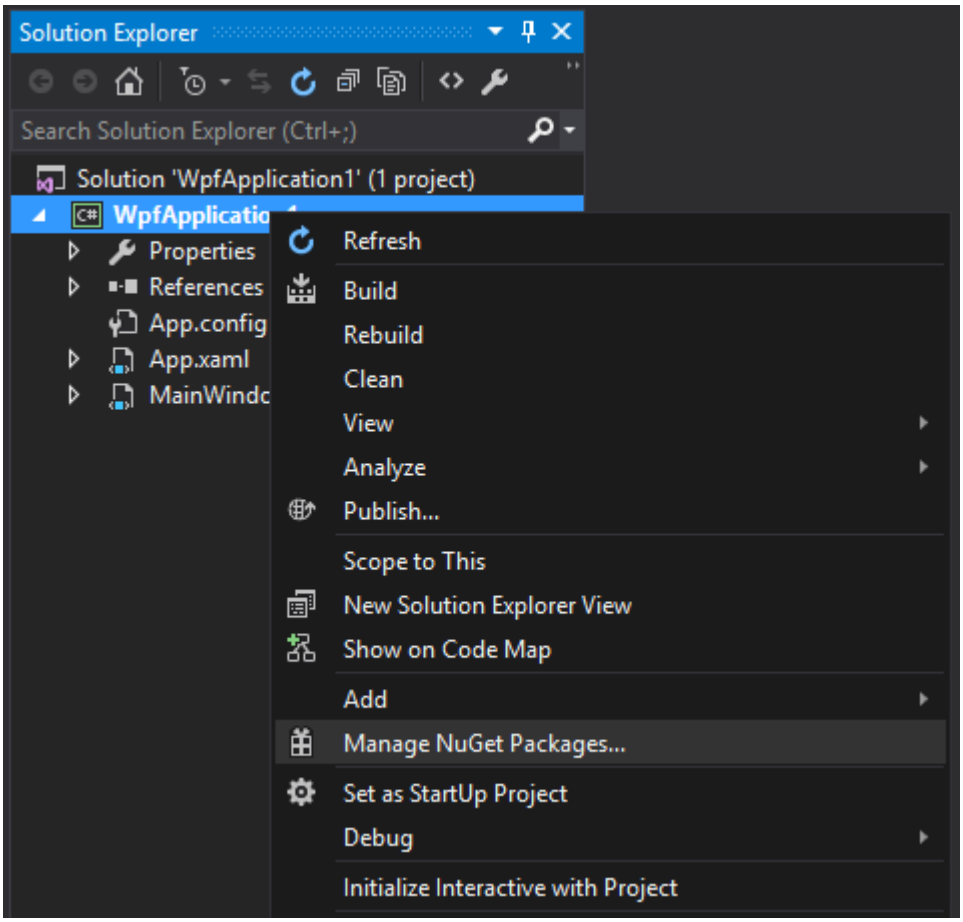
O podemos recuperar filas de la base de datos:

```
using(var context = new PlanetContext())
{
    var jupiter = context.Planets.Single(p => p.Name == "Jupiter");
    Console.WriteLine($"Jupiter is {jupiter.AverageDistanceFromSun} million km from the
sun.");
}
```

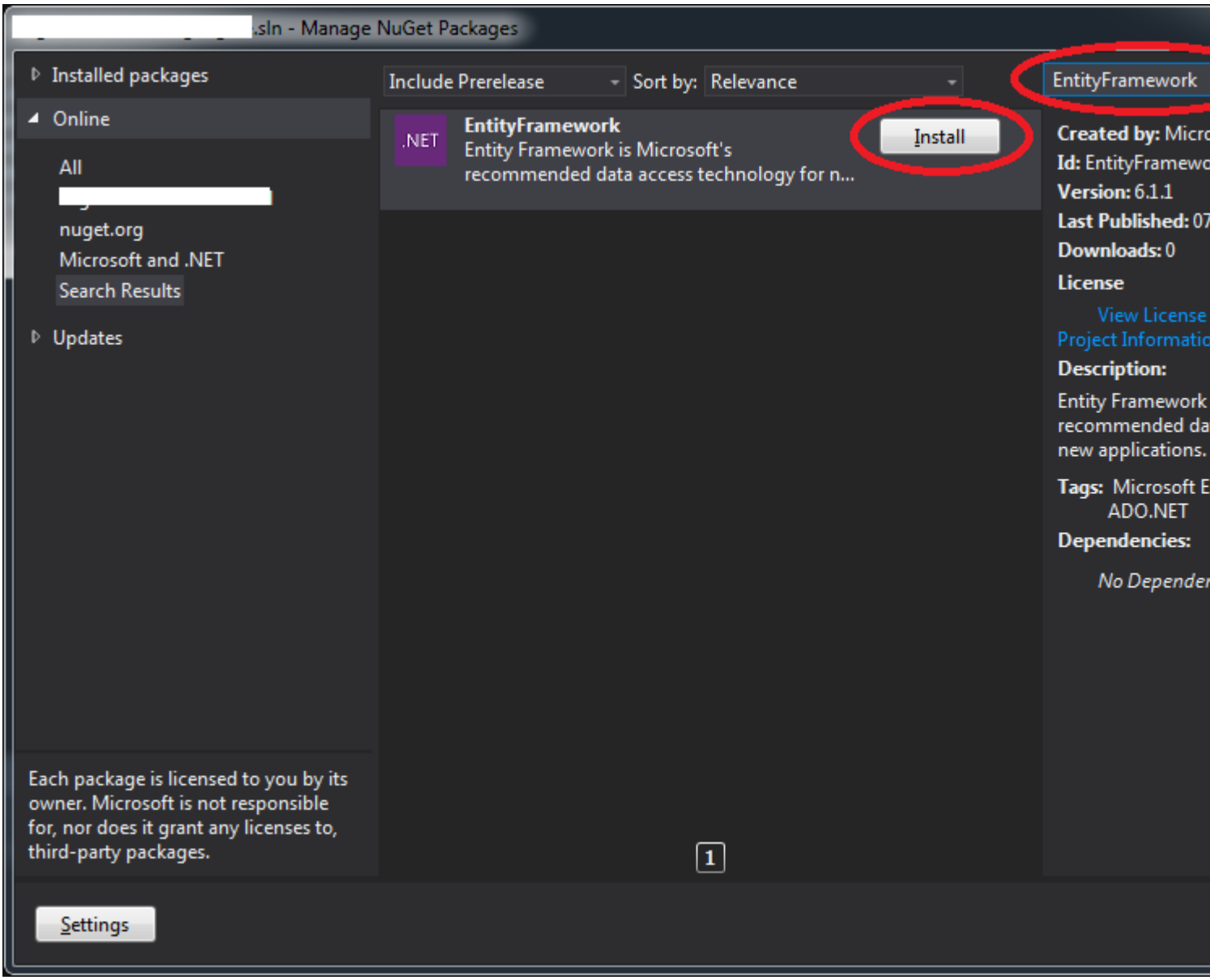
Instalación del paquete Entity Framework NuGet

En su Visual Studio, abra la ventana del **Explorador de soluciones y** luego `haga clic con` el `botón`

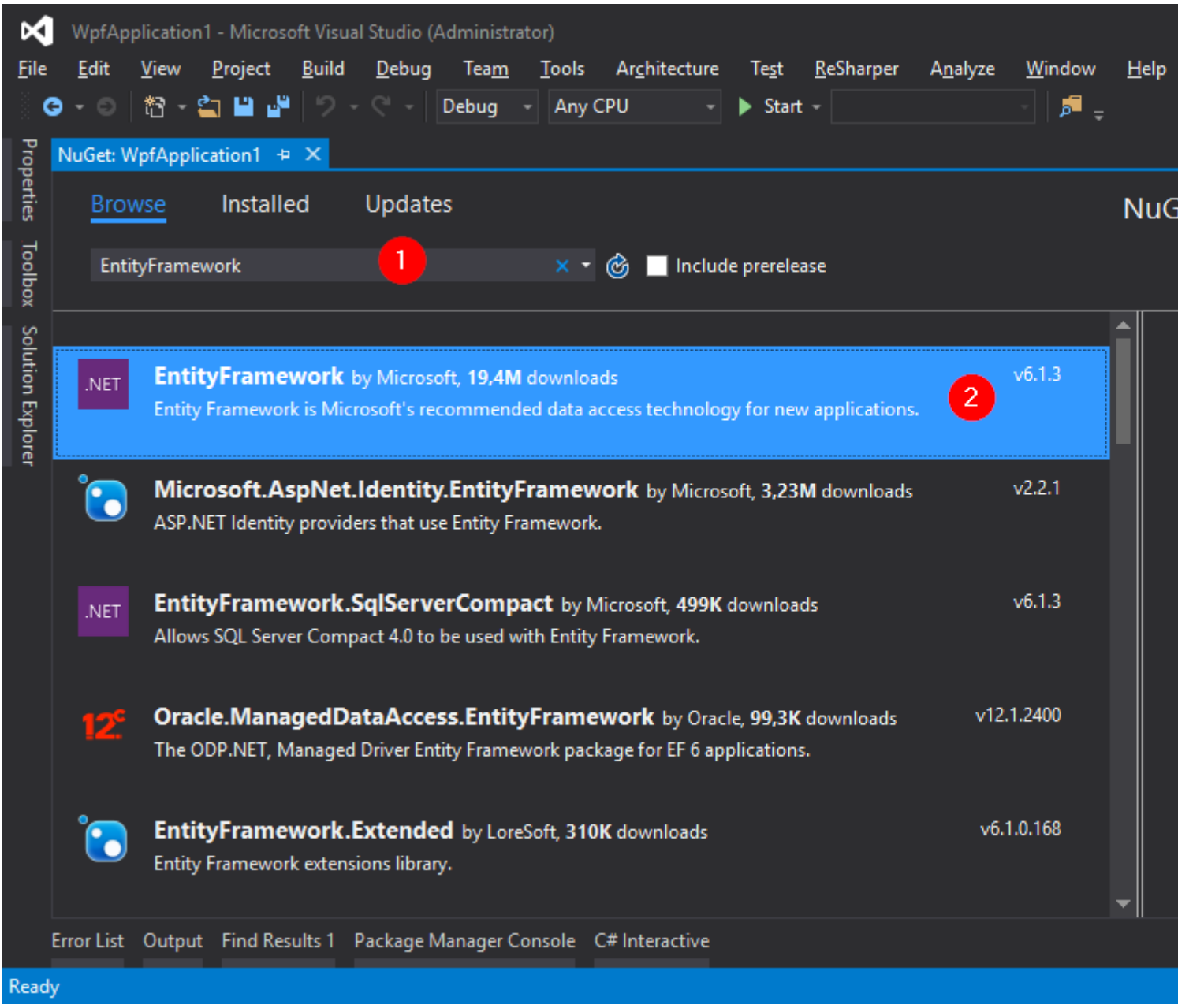
`derecho` en su proyecto, luego elija *Administrar paquetes NuGet* en el menú:



En la ventana que se abre, escriba `EntityFramework` en el cuadro de búsqueda en la parte superior derecha.



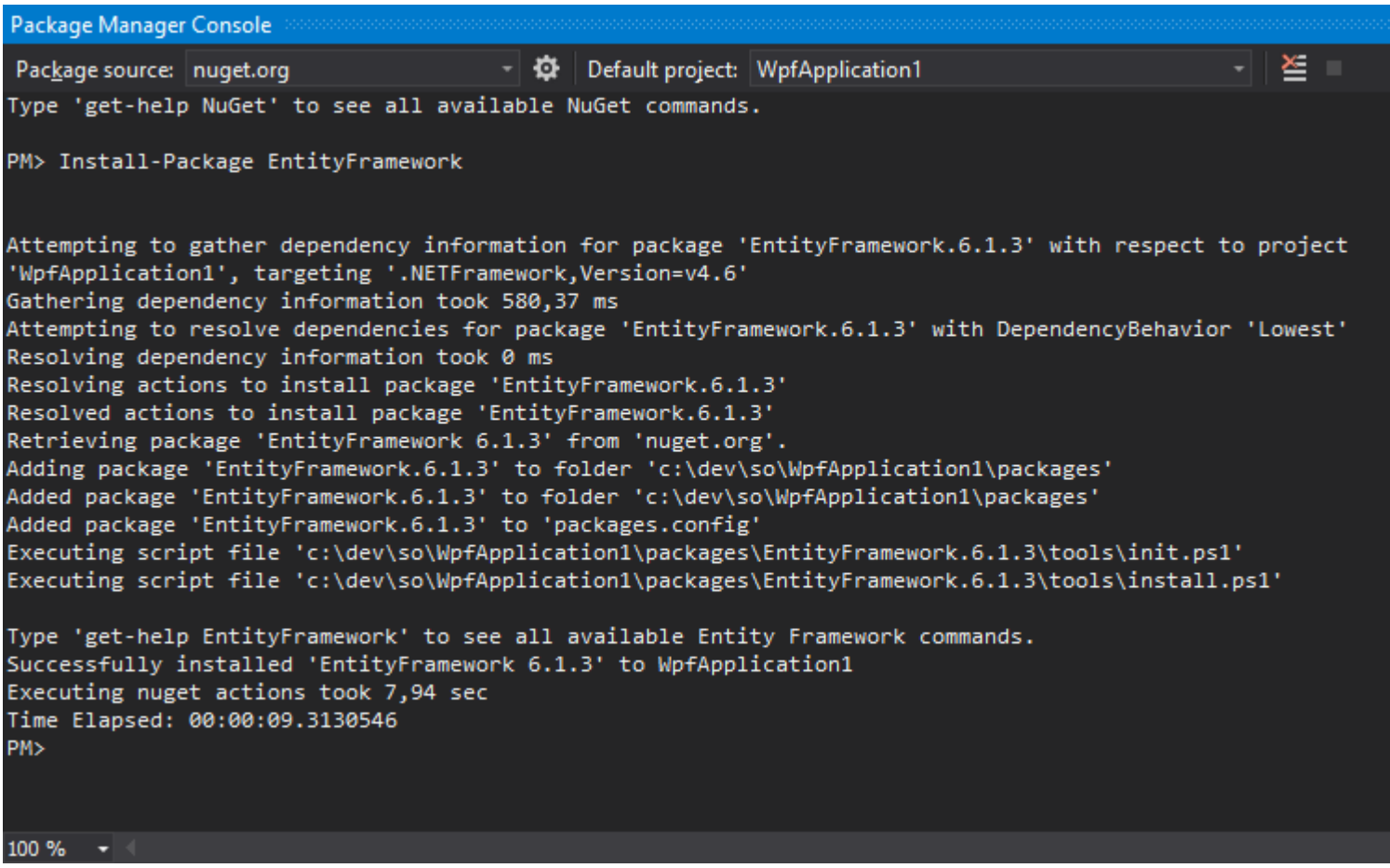
O si está utilizando Visual Studio 2015 verá algo como esto:



Luego haga clic en Instalar.

También podemos instalar el framework de entidades utilizando la consola del administrador de paquetes. Para hacerlo, primero debe abrirlo utilizando el *menú Herramientas -> NuGet Package Manager -> Package Manager Console y* luego ingrese esto:

```
Install-Package EntityFramework
```



Esto instalará Entity Framework y agregará automáticamente una referencia al ensamblaje en su proyecto.

¿Qué es Entity Framework?

Escribir y administrar el código ADO.Net para el acceso a datos es un trabajo tedioso y monótono. Microsoft ha proporcionado un **marco O / RM llamado "Entity Framework"** para automatizar las actividades relacionadas con la base de datos para su aplicación.

Entity framework es un marco Object / Relational Mapping (O / RM). Es una mejora de ADO.NET que brinda a los desarrolladores un mecanismo automatizado para acceder y almacenar los datos en la base de datos.

¿Qué es O / RM?

ORM es una herramienta para almacenar datos de objetos de dominio en la base de datos relacional como MS SQL Server, de manera automatizada, sin mucha programación. O / RM incluye tres partes principales:

1. Objetos de clase de dominio
2. Objetos de base de datos relacionales
3. Información de asignación sobre cómo los objetos de dominio se asignan a objetos de base de datos relacionales (por **ejemplo** , tablas, vistas y procedimientos almacenados)

ORM nos permite mantener el diseño de nuestra base de datos separado de nuestro diseño de

clase de dominio. Esto hace que la aplicación sea mantenible y extensible. También automatiza la operación estándar de CRUD (Crear, Leer, Actualizar y Eliminar) para que el desarrollador no tenga que escribirlo manualmente.

Lea Empezando con Entity Framework en línea: https://riptutorial.com/es/entity-framework/topic/815/empezando-con-entity-framework

Capítulo 2: Actas

Examples

Database.BeginTransaction ()

Se pueden ejecutar varias operaciones contra una sola transacción para que los cambios se puedan revertir si falla alguna de las operaciones.

```
using (var context = new PlanetContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            //Lets assume this works
            var jupiter = new Planet { Name = "Jupiter" };
            context.Planets.Add(jupiter);
            context.SaveChanges();

            //And then this will throw an exception
            var neptune = new Planet { Name = "Neptune" };
            context.Planets.Add(neptune);
            context.SaveChanges();

            //Without this line, no changes will get applied to the database
            transaction.Commit();
        }
        catch (Exception ex)
        {
            //There is no need to call transaction.Rollback() here as the transaction object
            //will go out of scope and disposing will roll back automatically
        }
    }
}
```

Tenga en cuenta que puede ser una convención de desarrolladores para llamar a `transaction.Rollback()` explícitamente, porque hace que el código sea más fácil de explicar. Además, *puede* haber proveedores de consultas (menos conocidos) para Entity Framework que no implementen `Dipsose` correctamente, lo que también requeriría una `transaction.Rollback()` explícita.Rollback `transaction.Rollback()` llamada.

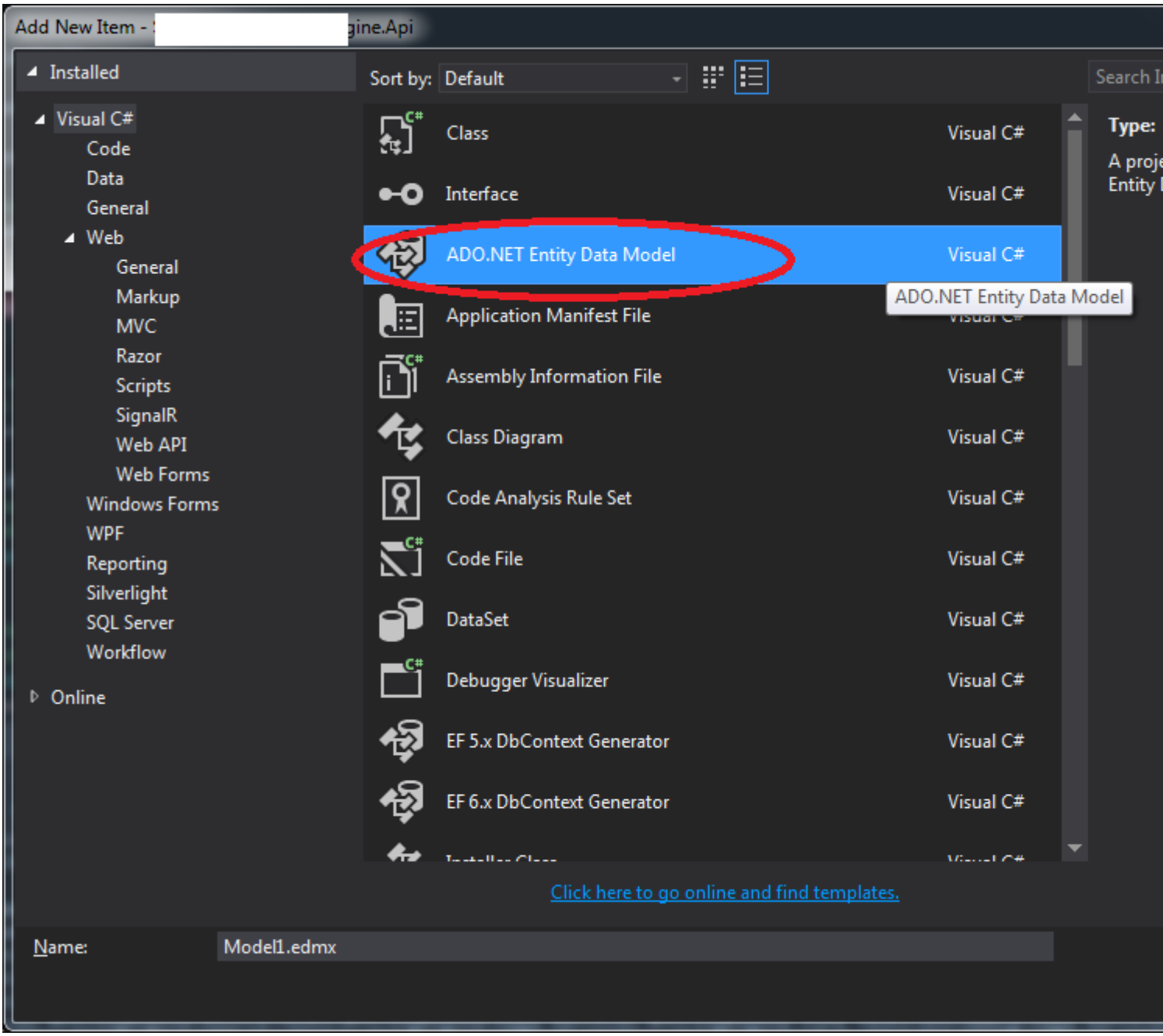
Lea Actas en línea: https://riptutorial.com/es/entity-framework/topic/4944/actas

Capítulo 3: Base de datos de primera generación de modelos

Examples

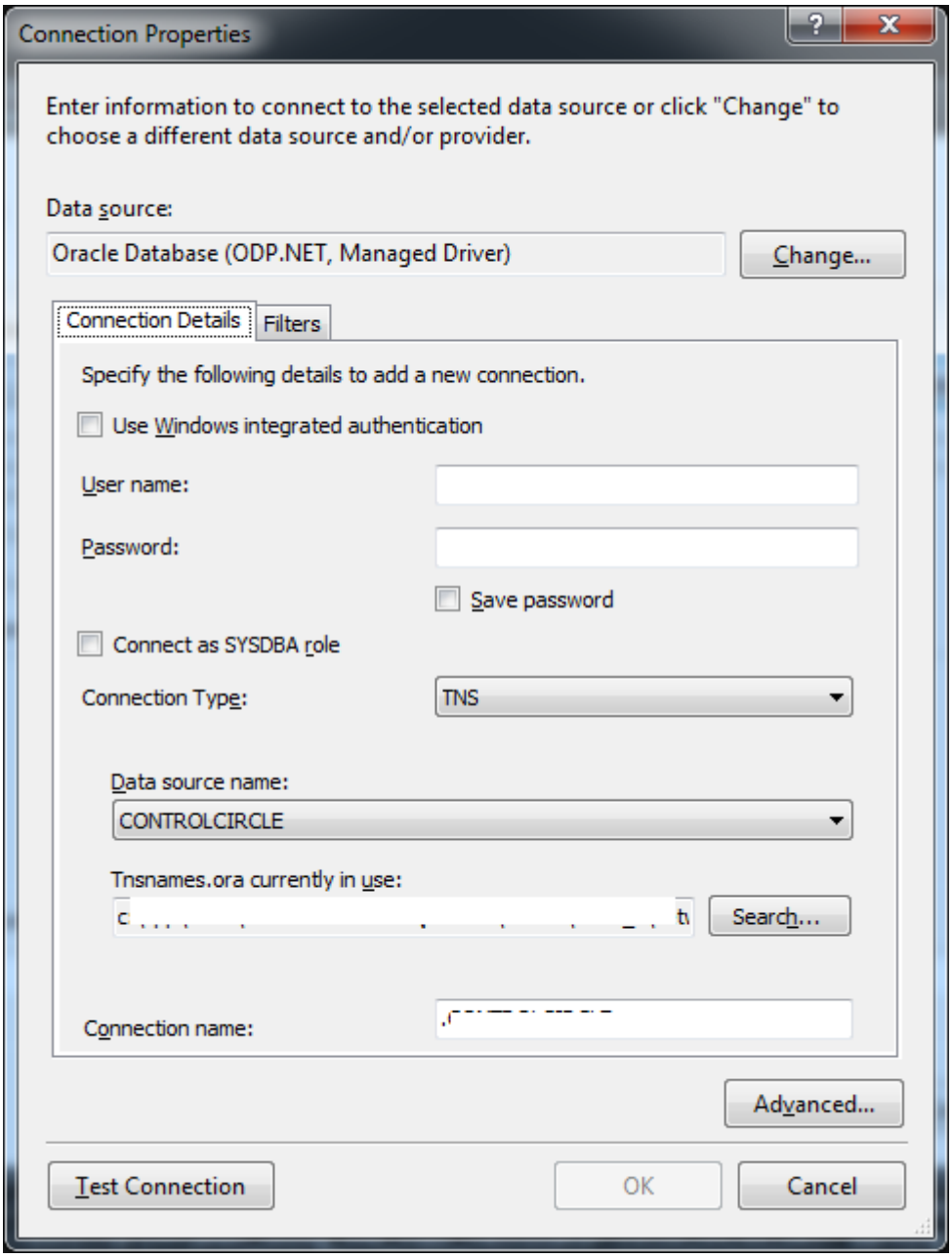
Generando modelo desde base de datos

En `Visual Studio` vaya a su `Solution Explorer` luego haga clic en `Project` y agregará el `mouse derecho del` modelo. Elija `ADO.NET Entity Data Model`



Luego, elija `Generate from database` y haga clic en `Next` en la siguiente ventana, haga clic en `New Connection...` y apunte a la base de datos desde la que desea generar el modelo (podría ser `MSSQL`

, `MySQL` u `Oracle`)



Después de hacer esto, haga clic en `Test Connection` para ver si ha configurado la conexión correctamente (no continúe si falla aquí).

Haga clic en `Next` luego elija las opciones que desee (como el estilo para generar nombres de entidades o para agregar claves externas).

Haga clic en `Next` nuevamente, en este punto debe tener un modelo generado desde la base de datos.

Agregando anotaciones de datos al modelo generado.

En la estrategia de generación de código T4 utilizada por Entity Framework 5 y superior, los atributos de anotación de datos no se incluyen de forma predeterminada. Para incluir anotaciones de datos en la parte superior de cierta propiedad en cada regeneración de modelo, abra el archivo

de plantilla incluido con EDMX (con la extensión `.tt`) y luego agregue una declaración de `using` bajo el método `UsingDirectives` como a continuación:

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>
(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code);
#>
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
using System.ComponentModel.DataAnnotations;  // --> add this line
```

Como ejemplo, suponga que la plantilla debe incluir `KeyAttribute` que indica una propiedad de clave principal. Para insertar `KeyAttribute` automáticamente mientras se regenera el modelo, encuentre parte del código que contiene `codeStringGenerator.Property` como se muestra a continuación:

```
var simpleProperties = typeMapper.GetSimpleProperties(entity);
    if (simpleProperties.Any())
    {
        foreach (var edmProperty in simpleProperties)
        {
#>
    <#=codeStringGenerator.Property(edmProperty)#>
<#
        }
    }
```

Luego, inserte una condición if para verificar la propiedad clave como esta:

```
var simpleProperties = typeMapper.GetSimpleProperties(entity);
    if (simpleProperties.Any())
    {
        foreach (var edmProperty in simpleProperties)
        {
             if (ef.IsKey(edmProperty)) {
#>    [Key]
<#      }
#>
    <#=codeStringGenerator.Property(edmProperty)#>
<#
        }
    }
```

Al aplicar los cambios anteriores, todas las clases de modelo generadas tendrán `KeyAttribute` en su propiedad de clave principal después de actualizar el modelo desde la base de datos.

antes de

```
using System;

public class Example
{
    public int Id { get; set; }
    public string Name { get; set; }
```

```
}
```

Después

```
using System;
using System.ComponentModel.DataAnnotations;

public class Example
{
    [Key]
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Lea Base de datos de primera generación de modelos en línea: https://riptutorial.com/es/entity-framework/topic/4414/base-de-datos-de-primera-generacion-de-modelos

Capítulo 4: Cargando entidades relacionadas

Observaciones

Si los modelos están relacionados correctamente, puede cargar fácilmente datos relacionados utilizando EntityFramework. Tiene tres opciones para elegir: *carga perezosa* , *carga impaciente* y *carga explícita* .

Modelos utilizados en ejemplos:

```
public class Company
{
    public int Id { get; set; }
    public string FullName { get; set; }
    public string ShortName { get; set; }

    // Navigation properties
    public virtual Person Founder { get; set; }
    public virtual ICollection<Address> Addresses { get; set; }
}

public class Address
{
    public int Id { get; set; }
    public int CompanyId { get; set; }
    public int CountryId { get; set; }
    public int CityId { get; set; }
    public string Street { get; set; }

    // Navigation properties
    public virtual Company Company { get; set; }
    public virtual Country Country { get; set; }
    public virtual City City { get; set; }
}
```

Examples

Carga lenta

La carga diferida está habilitada por defecto. La carga diferida se logra creando clases de proxy derivadas y anulando proeprties de navegación virtual. La carga perezosa se produce cuando se accede a la propiedad por primera vez.

```
int companyId = ...;
Company company = context.Companies
    .First(m => m.Id == companyId);
Person founder = company.Founder; // Founder is loaded
foreach (Address address in company.Addresses)
{
    // Address details are loaded one by one.
}
```

Para desactivar la carga diferida para propiedades de navegación específicas, simplemente elimine la palabra clave virtual de la declaración de propiedad:

```
public Person Founder { get; set; } // "virtual" keyword has been removed
```

Si desea desactivar completamente la carga diferida, debe cambiar la configuración, por ejemplo, en el *constructor de contexto* :

```
public class MyContext : DbContext
{
    public MyContext(): base("Name=ConnectionString")
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

Nota: recuerde *desactivar la* carga diferida si está utilizando la serialización. Debido a que los serializadores acceden a todas las propiedades, los cargará desde la base de datos. Además, puede ejecutarse en bucle entre las propiedades de navegación.

Carga impaciente

La carga impaciente le permite cargar todas las entidades necesarias a la vez. Si prefiere que todas sus entidades trabajen en una llamada a la base de datos, entonces el proceso de *carga impaciente* es el camino a seguir. También te permite cargar múltiples niveles.

Tiene *dos opciones* para cargar entidades relacionadas, puede elegir sobrecargas de *tipo fuertemente tipadas* o de *cadena* del método *Include* .

Fuertemente mecanografiado

```
// Load one company with founder and address details
int companyId = ...;
Company company = context.Companies
    .Include(m => m.Founder)
    .Include(m => m.Addresses)
    .SingleOrDefault(m => m.Id == companyId);

// Load 5 companies with address details, also retrieve country and city
// information of addresses
List<Company> companies = context.Companies
    .Include(m => m.Addresses.Select(a => a.Country));
    .Include(m => m.Addresses.Select(a => a.City))
    .Take(5).ToList();
```

Este método está disponible desde Entity Framework 4.1. Asegúrese de tener la referencia `using System.Data.Entity;` conjunto.

Sobrecarga de cuerdas.

```
// Load one company with founder and address details
int companyId = ...;
Company company = context.Companies
    .Include("Founder")
    .Include("Addresses")
    .SingleOrDefault(m => m.Id == companyId);

// Load 5 companies with address details, also retrieve country and city
// information of addresses
List<Company> companies = context.Companies
    .Include("Addresses.Country");
    .Include("Addresses.City"))
    .Take(5).ToList();
```

Carga explícita

Después de girar *perezoso carga* fuera puede cargar con pereza entidades llamando explícitamente al método *de carga* para las entradas. *La referencia* se usa para cargar propiedades de navegación individuales, mientras que la *colección* se usa para obtener colecciones.

```
Company company = context.Companies.FirstOrDefault();
// Load founder
context.Entry(company).Reference(m => m.Founder).Load();
// Load addresses
context.Entry(company).Collection(m => m.Addresses).Load();
```

Como está en la *carga Eager* , puede utilizar sobrecargas de los métodos anteriores para cargar entiteis por sus nombres:

```
Company company = context.Companies.FirstOrDefault();
// Load founder
context.Entry(company).Reference("Founder").Load();
// Load addresses
context.Entry(company).Collection("Addresses").Load();
```

Filtrar entidades relacionadas.

Usando el método de *consulta* podemos filtrar las entidades relacionadas cargadas:

```
Company company = context.Companies.FirstOrDefault();
// Load addresses which are in Baku
context.Entry(company)
    .Collection(m => m.Addresses)
    .Query()
    .Where(a => a.City.Name == "Baku")
    .Load();
```

Consultas de proyeccion

Si se necesitan datos relacionados en un tipo desnormalizado, o, por ejemplo, solo un subconjunto de columnas, se pueden usar consultas de proyección. Si no hay razón para usar un

tipo extra, existe la posibilidad de unir los valores en un tipo anónimo .

```
var dbContext = new MyDbContext();
var denormalizedType = from company in dbContext.Company
                       where company.Name == "MyFavoriteCompany"
                       join founder in dbContext.Founder
                       on company.FounderId equals founder.Id
                       select new
                       {
                           CompanyName = company.Name,
                           CompanyId = company.Id,
                           FounderName = founder.Name,
                           FounderId = founder.Id
                       };
```

O con sintaxis de consulta:

```
var dbContext = new MyDbContext();
var denormalizedType = dbContext.Company
                       .Join(dbContext.Founder,
                                   c => c.FounderId,
                                   f => f.Id ,
                                   (c, f) => new
                                   {
                                       CompanyName = c.Name,
                                       CompanyId = c.Id,
                                       FounderName = f.Name,
                                       FounderId = f.Id
                                   })
                       .Select(cf => cf);
```

Lea Cargando entidades relacionadas en línea: https://riptutorial.com/es/entity-framework/topic/4678/cargando-entidades-relacionadas

Capítulo 5: Código de primeras anotaciones de datos

Observaciones

Entity Framework Code-First proporciona un conjunto de atributos de Anotación de Datos, que puede aplicar a sus clases y propiedades de dominio. Los atributos de DataAnnotation anulan las convenciones predeterminadas de código primero.

1. **System.ComponentModel.DataAnnotations** incluye atributos que afectan la capacidad de nulos o el tamaño de la columna.
2. **System.ComponentModel.DataAnnotations.Schema** namespace incluye atributos que afectan el esquema de la base de datos.

Nota: las Anotaciones de datos solo le brindan un subconjunto de opciones de configuración. Fluent API proporciona un conjunto completo de opciones de configuración disponibles en Code-First.

Examples

Atributo [clave]

La clave es un campo en una tabla que identifica de forma única cada fila / registro en una tabla de base de datos.

Utilice este atributo para **anular la convención predeterminada de Code-First** . Si se aplica a una propiedad, se usará como la **columna de clave principal** para esta clase.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    [Key]
    public int PersonKey { get; set; }        // <- will be used as primary key

    public string PersonName { get; set; }
}
```

Si se requiere una clave primaria compuesta, el atributo [Clave] también se puede agregar a varias propiedades. El orden de las columnas dentro de la clave compuesta debe proporcionarse en el formulario [**Clave, Columna (Orden = x)]** .

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    [Key, Column(Order = 0)]
```

```
    public int PersonKey1 { get; set; }    // <- will be used as part of the primary key

    [Key, Column(Order = 1)]
    public int PersonKey2 { get; set; }    // <- will be used as part of the primary key

    public string PersonName { get; set; }
}
```

Sin el atributo [Clave] , EntityFramework volverá a la convención predeterminada, que consiste en utilizar la propiedad de la clase como clave principal que se denomina "Id" o "{ClassName} Id".

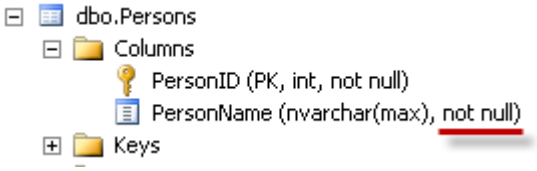
```
public class Person
{
    public int PersonID { get; set; }        // <- will be used as primary key

    public string PersonName { get; set; }
}
```

[Requerido] atributo

Cuando se aplica a una propiedad de una clase de dominio, la base de datos creará una columna NOT NULL.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    public int PersonID { get; set; }

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

La columna resultante con la restricción NOT NULL:



Nota: También se puede usar con asp.net-mvc como un atributo de validación.

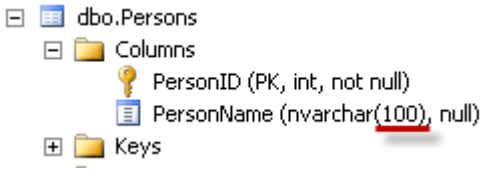
Atributos [MaxLength] y [MinLength]

El atributo [MaxLength (int)] se puede aplicar a una cadena o propiedad de tipo de matriz de una clase de dominio. Entity Framework establecerá el tamaño de una columna al valor especificado.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
```

```
    public int PersonID { get; set; }

    [MinLength(3), MaxLength(100)]
    public string PersonName { get; set; }
}
```

La columna resultante con la longitud de columna especificada:



El atributo [MinLength (int)] es un atributo de validación, no afecta la estructura de la base de datos. Si intentamos insertar / actualizar una Persona con Nombre de Persona con una longitud inferior a 3 caracteres, este compromiso fallará. `DbUpdateConcurrencyException` una `DbUpdateConcurrencyException` que necesitaremos manejar.

```
using (var db = new ApplicationDbContext())
{
     db.Staff.Add(new Person() { PersonName = "ng" });
     try
     {
          db.SaveChanges();
     }
     catch (DbEntityValidationException ex)
     {
          //ErrorMessage = "The field PersonName must be a string or array type with a minimum
length of '3'."
     }
}
```

Los **atributos [MaxLength]** y **[MinLength]** también se pueden usar con asp.net-mvc como atributo de validación.

Atributo [Rango (min, max)]

Especifica un rango numérico mínimo y máximo para una propiedad.

```
using System.ComponentModel.DataAnnotations;

public partial class Enrollment
{
    public int EnrollmentID { get; set; }

    [Range(0, 4)]
    public Nullable<decimal> Grade { get; set; }
}
```

Si intentamos insertar / actualizar un Grado con un valor fuera de rango, este compromiso fallará. `DbUpdateConcurrencyException` una `DbUpdateConcurrencyException` que necesitaremos manejar.

```
using (var db = new ApplicationDbContext())
```

```
{
    db.Enrollments.Add(new Enrollment() { Grade = 1000 });

    try
    {
        db.SaveChanges();
    }
    catch (DbEntityValidationException ex)
    {
        // Validation failed for one or more entities
    }
}
```

También se puede usar con asp.net-mvc como un atributo de validación.

Resultado:

Atributo [DatabaseGenerated]

Especifica cómo la base de datos genera valores para la propiedad. Hay tres valores posibles:

1. `None` especifica que los valores no son generados por la base de datos.
2. `Identity` especifica que la columna es una columna de identidad , que normalmente se utiliza para las claves primarias de enteros.
3. `Computed` especifica que la base de datos genera el valor para la columna.

Si el valor es distinto de `None` , Entity Framework no confirmará los cambios realizados en la propiedad en la base de datos.

De forma predeterminada (en función de `StoreGeneratedIdentityKeyConvention`) una propiedad de clave entera se tratará como una columna de identidad. Para anular esta convención y obligarla a ser tratada como una columna sin identidad, puede utilizar el atributo `DatabaseGenerated` con un valor de `None` .

```
using System.ComponentModel.DataAnnotations.Schema;

public class Foo
{
    [Key]
    public int Id { get; set; } // identity (auto-increment) column
}

public class Bar
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int Id { get; set; } // non-identity column
}
```

El siguiente SQL crea una tabla con una columna calculada:

```
CREATE TABLE [Person] (
    Name varchar(100) PRIMARY KEY,
    DateOfBirth Date NOT NULL,
    Age AS DATEDIFF(year, DateOfBirth, GETDATE())
)
GO
```

Para crear una entidad que represente los registros en la tabla anterior, necesitaría usar el atributo `DatabaseGenerated` con un valor de `Computed` .

```
[Table("Person")]
public class Person
{
    [Key, StringLength(100)]
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public int Age { get; set; }
}
```

Atributo [NotMapped]

Por la convención de Code-First, Entity Framework crea una columna para cada propiedad pública que es de un tipo de datos compatible y tiene tanto un getter como un setter. **La** anotación **[NotMapped]** debe aplicarse a cualquier propiedad para la cual **NO** deseamos una columna en una tabla de base de datos.

Un ejemplo de una propiedad que podríamos no querer almacenar en la base de datos es el nombre completo de un estudiante basado en su nombre y apellido. Esto se puede calcular sobre la marcha y no es necesario almacenarlo en la base de datos.

```
public string FullName => string.Format("{0} {1}", FirstName, LastName);
```

La propiedad "FullName" tiene solo un getter y ningún setter, por lo que Entity Framework **NO** creará una columna para él.

Otro ejemplo de una propiedad que tal vez no queramos almacenar en la base de datos es el "AverageGrade" de un estudiante. No queremos obtener el AverageGrade a pedido; en cambio, podríamos tener una rutina en otro lugar que la calcule.

```
[NotMapped]
public float AverageGrade { set; get; }
```

El "AverageGrade" debe estar marcado **como** anotación **[NotMapped]** , de lo contrario, Entity Framework creará una columna para él.

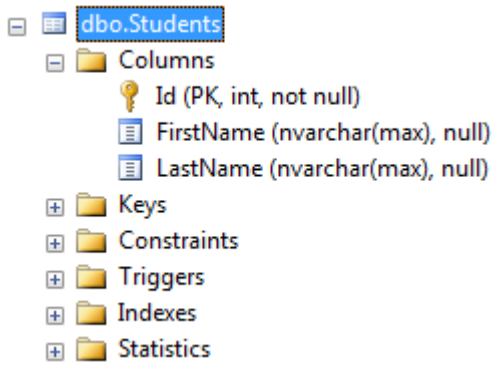
```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int Id { set; get; }
```

```
    public string FirstName { set; get; }

    public string LastName { set; get; }

    public string FullName => string.Format("{0} {1}", FirstName, LastName);

    [NotMapped]
    public float AverageGrade { set; get; }
}
```

Para la Entidad anterior veremos dentro de `DbMigration.cs`

```
CreateTable(
    "dbo.Students",
     c => new
         {
             Id = c.Int(nullable: false, identity: true),
             FirstName = c.String(),
             LastName = c.String(),
         })
     .PrimaryKey(t => t.Id);
```

y en SQL Server Management Studio



Atributo [tabla]

```
[Table("People")]
public class Person
{
    public int PersonID { get; set; }
    public string PersonName { get; set; }
}
```

Le dice a Entity Framework que use un nombre de tabla específico en lugar de generar uno (es decir, `Person` o `Persons`)

También podemos especificar un esquema para la tabla usando el atributo [Tabla]

```
[Table("People", Schema = "domain")]
```

Atributo [columna]

```
public class Person
{
    public int PersonID { get; set; }

    [Column("NameOfPerson")]
    public string PersonName { get; set; }
}
```

Le dice a Entity Framework que use un nombre de columna específico en lugar de usar el nombre de la propiedad. También puede especificar el tipo de datos de la base de datos y el orden de la columna en la tabla:

```
[Column("NameOfPerson", TypeName = "varchar", Order = 1)]
public string PersonName { get; set; }
```

Atributo [índice]

```
public class Person
{
    public int PersonID { get; set; }
    public string PersonName { get; set; }

    [Index]
    public int Age { get; set; }
}
```

Crea un índice de base de datos para una columna o conjunto de columnas.

```
[Index("IX_Person_Age")]
public int Age { get; set; }
```

Esto crea un índice con un nombre específico.

```
[Index(IsUnique = true)]
public int Age { get; set; }
```

Esto crea un índice único.

```
[Index("IX_Person_NameAndAge", 1)]
public int Age { get; set; }

[Index("IX_Person_NameAndAge", 2)]
public string PersonName { get; set; }
```

Esto crea un índice compuesto utilizando 2 columnas. Para hacer esto, debe especificar el mismo nombre de índice y proporcionar un orden de columna.

Nota : el atributo Índice se introdujo en Entity Framework 6.1. Si está utilizando una versión anterior, la información en esta sección no se aplica.

Atributo [ForeignKey (cadena)]

Especifica el nombre de la clave externa personalizada si se desea una clave externa que no siga la convención de EF.

```
public class Person
{
    public int IdAddress { get; set; }

    [ForeignKey(nameof(IdAddress))]
    public virtual Address HomeAddress { get; set; }
}
```

Esto también se puede usar cuando tienes varias relaciones con el mismo tipo de entidad.

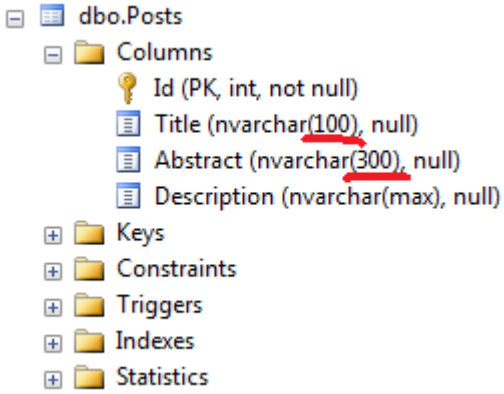
```
using System.ComponentModel.DataAnnotations.Schema;

public class Customer
{
    ...

    public int MailingAddressID { get; set; }
    public int BillingAddressID { get; set; }

    [ForeignKey("MailingAddressID")]
    public virtual Address MailingAddress { get; set; }
    [ForeignKey("BillingAddressID")]
    public virtual Address BillingAddress { get; set; }
}
```

Sin los atributos de `ForeignKey` , EF podría mezclarlos y usar el valor de `BillingAddressID` al obtener la `MailingAddress` , o simplemente podría tener un nombre diferente para la columna en función de sus propias convenciones de nombres (como `Address_MailingAddress_Id`) y tratar de usar eso en su lugar (lo que resultaría en un error si está usando esto con una base de datos existente).

Atributo [StringLength (int)]

```
using System.ComponentModel.DataAnnotations;

public class Post
{
    public int Id { get; set; }

    [StringLength(100)]
    public string Title { get; set;}

    [StringLength(300)]
    public string Abstract { get; set; }

    public string Description { get; set; }
}
```

Define una longitud máxima para un campo de cadena.

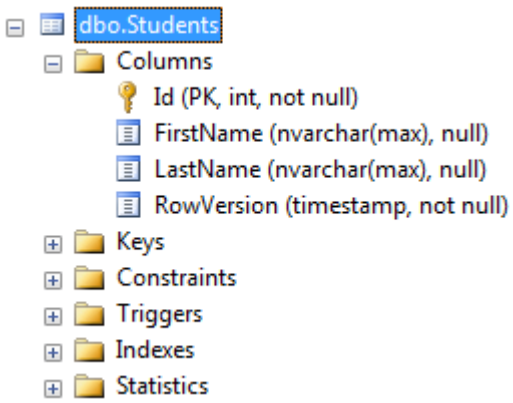


Nota : También se puede usar con asp.net-mvc como un atributo de validación.

Atributo [marca de tiempo]

El atributo **[TimeStamp]** se puede aplicar a solo una propiedad de matriz de bytes en una clase de entidad dada. Entity Framework creará una columna de marca de tiempo no anulable en la tabla de la base de datos para esa propiedad. Entity Framework utilizará automáticamente esta columna de TimeStamp en la verificación de concurrencia.

```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int Id { set; get; }

    public string FirstName { set; get; }

    public string LastName { set; get; }

    [Timestamp]
    public byte[] RowVersion { get; set; }
}
```



Atributo [ConcurrencyCheck]

Este atributo se aplica a la propiedad de clase. Puede usar el atributo ConcurrencyCheck cuando desee usar columnas existentes para la comprobación de concurrencia y no una columna de marca de tiempo separada para la concurrencia.

```
using System.ComponentModel.DataAnnotations;

public class Author
{
    public int AuthorId { get; set; }

    [ConcurrencyCheck]
    public string AuthorName { get; set; }
}
```

Del ejemplo anterior, el atributo ConcurrencyCheck se aplica a la propiedad AuthorName de la clase Author. Por lo tanto, Code-First incluirá la columna AuthorName en el comando de actualización (donde cláusula) para verificar la concurrencia optimista.

Atributo [InverseProperty (cadena)]

```
using System.ComponentModel.DataAnnotations.Schema;

public class Department
{
    ...

    public virtual ICollection<Employee> PrimaryEmployees { get; set; }
    public virtual ICollection<Employee> SecondaryEmployees { get; set; }
}

public class Employee
{
    ...

    [InverseProperty("PrimaryEmployees")]
    public virtual Department PrimaryDepartment { get; set; }

    [InverseProperty("SecondaryEmployees")]
    public virtual Department SecondaryDepartment { get; set; }
}
```

InverseProperty se puede usar para identificar relaciones de *dos vías* cuando existen **varias** relaciones de *dos vías* entre dos entidades.

Le dice a Entity Framework qué propiedades de navegación deben coincidir con las propiedades en el otro lado.

Entity Framework no sabe qué mapa de propiedades de navegación con qué propiedades en el otro lado cuando existen varias relaciones bidireccionales entre dos entidades.

Necesita el nombre de la propiedad de navegación correspondiente en la clase relacionada como parámetro.

Esto también puede usarse para entidades que tienen una relación con otras entidades del mismo tipo, formando una relación recursiva.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
```

```
public class TreeNode
{
    [Key]
    public int ID { get; set; }
    public int ParentID { get; set; }

    ...

    [ForeignKey("ParentID")]
    public TreeNode ParentNode { get; set; }
    [InverseProperty("ParentNode")]
    public virtual ICollection<TreeNode> ChildNodes { get; set; }
}
```

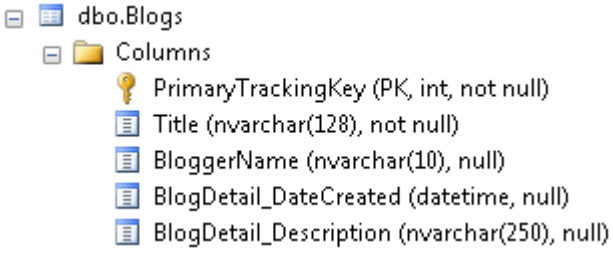
Tenga en cuenta también el uso del atributo `ForeignKey` para especificar la columna que se utiliza para la clave externa en la tabla. En el primer ejemplo, las dos propiedades en la clase `Employee` podrían haber tenido el atributo `ForeignKey` aplicado para definir los nombres de columna.

Atributo [ComplexType]

```
using System.ComponentModel.DataAnnotations.Schema;

[ComplexType]
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

    [MaxLength(250)]
    public string Description { get; set; }
}

public class Blog
{
    ...

    public BlogDetails BlogDetail { get; set; }
}
```

Marque la clase como tipo complejo en Entity Framework.

Los tipos complejos (o los *objetos de valor* en diseño impulsado por dominio) no se pueden rastrear por sí solos, pero se rastrean como parte de una entidad. Esta es la razón por la que BlogDetails en el ejemplo no tiene una propiedad clave.



Pueden ser útiles al describir entidades de dominio en varias clases y agrupar esas clases en una entidad completa.

Lea Código de primeras anotaciones de datos en línea: https://riptutorial.com/es/entity-framework/topic/4161/codigo-de-primeras-anotaciones-de-datos

Capítulo 6: Código de primeras convenciones

Observaciones

Convention es un conjunto de reglas predeterminadas para configurar automáticamente un modelo conceptual basado en definiciones de clase de dominio cuando se trabaja con Code-First. Las convenciones de Code-First se definen en *System.Data.Entity.ModelConfiguration.Conventions* espacio de nombres (EF 5 & EF 6).

Examples

Convención de Clave Primaria

Por defecto, una propiedad es una clave principal si una propiedad en una clase se denomina "ID" (no distingue entre mayúsculas y minúsculas), o el nombre de la clase seguido de "ID". Si el tipo de propiedad de clave principal es numérico o GUID, se configurará como una columna de identidad. Ejemplo simple:

```
public class Room
{
    // Primary key
    public int RoomId{ get; set; }
    ...
}
```

Eliminando Convenciones

Puede eliminar cualquiera de las convenciones definidas en el espacio de nombres System.Data.Entity.ModelConfiguration.Conventions, reemplazando el método `OnModelCreating` .

El siguiente ejemplo elimina PluralizingTableNameConvention.

```
public class EshopContext : DbContext
{
    public DbSet<Product> Products { set; get; }
    . . .

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

Por defecto, EF creará una tabla de base de datos con el nombre de la clase de la entidad con el sufijo 's'. En este ejemplo, Code First está configurado para ignorar la convención `dbo.Products` , por lo que, en lugar de `dbo.Products` table, se `dbo.Product` tabla `dbo.Product` .

Tipo de descubrimiento

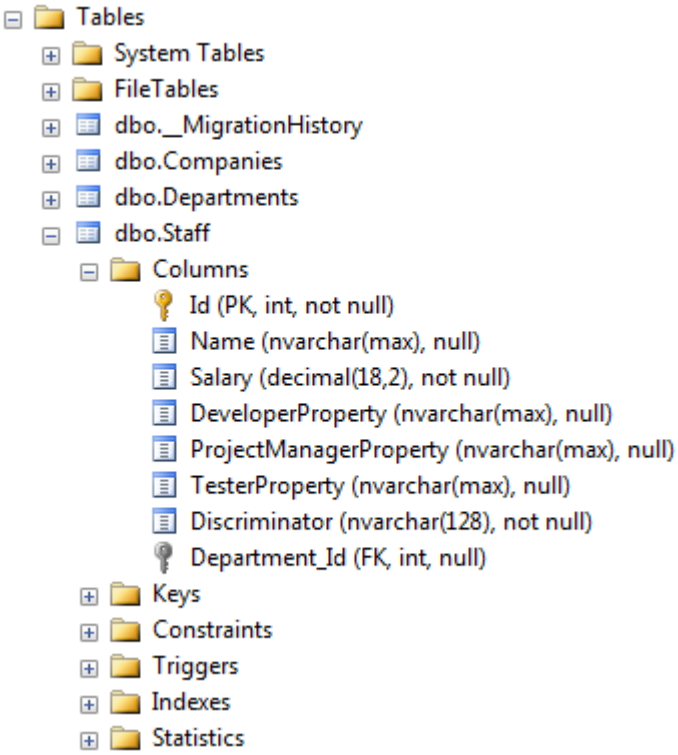
Por defecto el Código Primero incluye en el modelo

1. Tipos definidos como una propiedad DbSet en la clase de contexto.
2. Tipos de referencia incluidos en los tipos de entidad, incluso si se definen en un conjunto diferente.
3. Clases derivadas incluso si solo la clase base se define como propiedad DbSet

Aquí hay un ejemplo, que solo estamos agregando `Company` as `DbSet<Company>` en nuestra clase de contexto:

```
public class Company
{
    public int Id { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Department> Departments { set; get; }
}

public class Department
{
    public int Id { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Person> Staff { set; get; }
}

[Table("Staff")]
public class Person
{
    public int Id { set; get; }
    public string Name { set; get; }
    public decimal Salary { set; get; }
}

public class ProjectManager : Person
{
   public string ProjectManagerProperty { set; get; }
}

public class Developer : Person
{
    public string DeveloperProperty { set; get; }
}

public class Tester : Person
{
    public string TesterProperty { set; get; }
}

public class ApplicationDbContext : DbContext
{
    public DbSet<Company> Companies { set; get; }
}
```

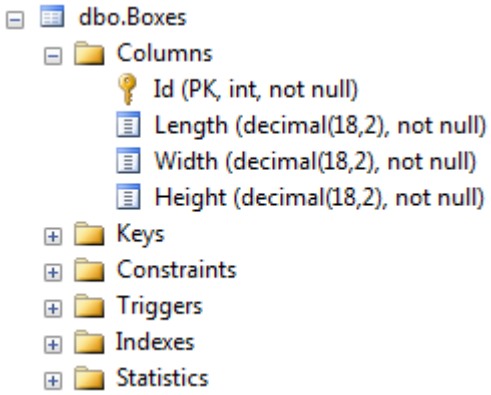
Podemos ver que todas las clases están incluidas en el modelo.



DecimalPropertyConvention

Por defecto, Entity Framework asigna las propiedades decimales a columnas decimales (18,2) en las tablas de la base de datos.

```
public class Box
{
    public int Id { set; get; }
    public decimal Length { set; get; }
    public decimal Width { set; get; }
    public decimal Height { set; get; }
}
```

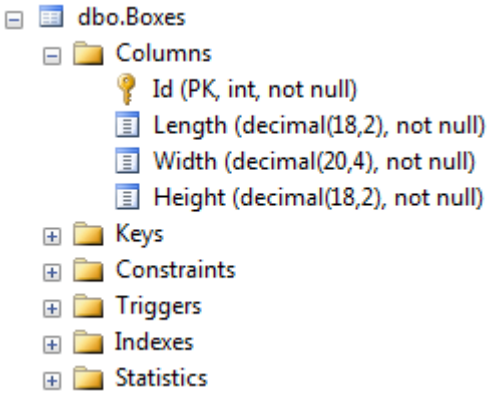


Podemos cambiar la precisión de las propiedades decimales:

1.Utilizar la API fluida:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Box>().Property(b => b.Width).HasPrecision(20, 4);
```

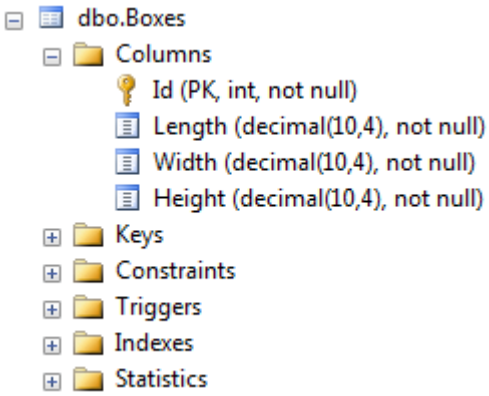
```
}
```



Sólo la propiedad "Ancho" se asigna a decimal (20, 4).

2. Reemplazar la convención:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<DecimalPropertyConvention>();
    modelBuilder.Conventions.Add(new DecimalPropertyConvention(10, 4));
}
```



Cada propiedad decimal se asigna a columnas decimales (10,4).

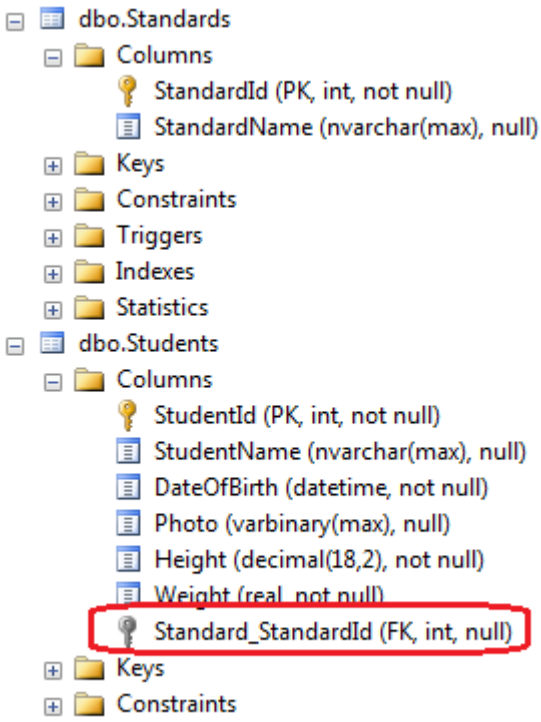
Convenio de relacion

Código Primero inferir la relación entre las dos entidades utilizando la propiedad de navegación. Esta propiedad de navegación puede ser un tipo de referencia simple o un tipo de colección. Por ejemplo, definimos la propiedad de navegación estándar en la clase de estudiante y la propiedad de navegación de ICollection en la clase estándar. Por lo tanto, Code First creó automáticamente una relación de uno a varios entre los estándares y la tabla de base de datos de estudiantes insertando la columna de clave foránea Standard_StandardId en la tabla de estudiantes.

```
public class Student
{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
```

```
    public DateTime DateOfBirth { get; set; }

    //Navigation property
    public Standard Standard { get; set; }
}

public class Standard
{

    public int StandardId { get; set; }
    public string StandardName { get; set; }

    //Collection navigation property
    public IList<Student> Students { get; set; }

}
```

Las entidades anteriores crearon la siguiente relación utilizando la clave foránea Standard_StandardId.



Convención de clave extranjera

Si la clase A está en relación con la clase B y la clase B tiene una propiedad con el mismo nombre y tipo que la clave principal de A, EF asume automáticamente que la propiedad es una clave externa.

```
public class Department
{
    public int DepartmentId { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Person> Staff { set; get; }
}
```

```
public class Person
{
    public int Id { set; get; }
    public string Name { set; get; }
    public decimal Salary { set; get; }
    public int DepartmentId { set; get; }
    public virtual Department Department { set; get; }
}
```

En este caso, DepartmentId es una clave externa sin especificación explícita.

Lea Código de primeras convenciones en línea: https://riptutorial.com/es/entity-framework/topic/2447/codigo-de-primeras-convenciones

Capítulo 7: Código primero - API fluida

Observaciones

Hay dos formas generales de especificar CÓMO Entity Framework asignará las clases de POCO a las tablas, columnas, bases de datos, etc .: **Anotaciones de datos** y **API de Fluent** .

Si bien las Anotaciones de datos son fáciles de leer y comprender, carecen de ciertas características, como la especificación del comportamiento "Cascada en eliminar" para una entidad. La API Fluent, por otro lado, es un poco más compleja de usar, pero proporciona un conjunto de características mucho más avanzadas.

Examples

Modelos de mapeo

EntityFramewok Fluent API es una forma potente y elegante de mapear los modelos de su dominio **primero en código** a la base de datos subyacente. Esto también se puede usar con el *código primero con la base de datos existente* . Tiene dos opciones al usar *Fluent API* : puede asignar directamente sus modelos en el método *OnModelCreating* o puede crear clases de asignadores que heredan de *EntityTypeConfiguration* y luego agregar esos modelos a *modelBuilder* en el método *OnModelCreating* . *La segunda* opción es la que yo prefiero y voy a mostrar un ejemplo de ello.

Paso uno: Crear modelo.

```
public class Employee
{
    public int Id { get; set; }
    public string Surname { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public short Age { get; set; }
    public decimal MonthlySalary { get; set; }

    public string FullName
    {
        get
        {
            return $"{Surname} {FirstName} {LastName}";
        }
    }
}
```

Paso dos: Crear una clase de asignador

```
public class EmployeeMap
```

```
    : EntityTypeConfiguration<Employee>
{
    public EmployeeMap()
    {
        // Primary key
        this.HasKey(m => m.Id);

        this.Property(m => m.Id)
            .HasColumnType("int")
            .HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);

        // Properties
        this.Property(m => m.Surname)
            .HasMaxLength(50);

        this.Property(m => m.FirstName)
            .IsRequired()
            .HasMaxLength(50);

        this.Property(m => m.LastName)
            .HasMaxLength(50);

        this.Property(m => m.Age)
            .HasColumnType("smallint");

        this.Property(m => m.MonthlySalary)
            .HasColumnType("number")
            .HasPrecision(14, 5);

        this.Ignore(m => m.FullName);

        // Table & column mappings
        this.ToTable("TABLE_NAME", "SCHEMA_NAME");
        this.Property(m => m.Id).HasColumnName("ID");
        this.Property(m => m.Surname).HasColumnName("SURNAME");
        this.Property(m => m.FirstName).HasColumnName("FIRST_NAME");
        this.Property(m => m.LastName).HasColumnName("LAST_NAME");
        this.Property(m => m.Age).HasColumnName("AGE");
        this.Property(m => m.MonthlySalary).HasColumnName("MONTHLY_SALARY");
    }
}
```

Vamos a explicar las asignaciones:

- **HasKey** - define la clave principal. También se pueden utilizar *claves primarias compuestas* . Por ejemplo: *this.HasKey (m => new {m.DepartmentId, m.PositionId})* .
- **Propiedad** - nos permite configurar las propiedades del modelo.
- **HasColumnType** : especifique el tipo de columna de nivel de base de datos. Tenga en cuenta que puede ser diferente para diferentes bases de datos como Oracle y MS SQL.
- **HasDatabaseGeneratedOption** : especifica si la propiedad se calcula a nivel de base de datos. Las PK numéricas son *DatabaseGeneratedOption.Identity* de forma predeterminada, debe especificar *DatabaseGeneratedOption.None* si no desea que lo sean.
- **HasMaxLength** - limita la longitud de la cadena.
- **IsRequired** : marca la propiedad como requerida.
- **HasPrecision** - nos permite especificar la precisión para decimales.
- **Ignorar** : ignora la propiedad completamente y no la asigna a la base de datos. Ignoramos

FullName, porque no queremos que esta columna esté en nuestra tabla.
- **ToTable** : especifique el nombre de la tabla y el nombre del esquema (opcional) para el modelo.
- **HasColumnName** - relaciona propiedad con nombre de columna. Esto no es necesario cuando los nombres de propiedades y los nombres de columnas son idénticos.

Paso tres: Agregar clase de mapeo a las configuraciones.

Necesitamos decirle a EntityFramework que use nuestra clase de asignador. Para hacerlo, debemos agregarlo a *modelBuilder.Configurations* en el método *OnModelCreating* :

```
public class DbContext()
    : base("Name=DbContext")
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new EmployeeMap());
    }
}
```

Y eso es todo. Estamos listos para irnos.

Clave primaria

Al usar el método .HasKey (), una propiedad puede configurarse explícitamente como clave principal de la entidad.

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

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>().HasKey(p => p.PersonKey);
    }
}
```

Clave primaria compuesta

Al usar el método .HasKey (), un conjunto de propiedades puede configurarse explícitamente como la clave primaria compuesta de la entidad.

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

public class PersonContext : DbContext
{
```

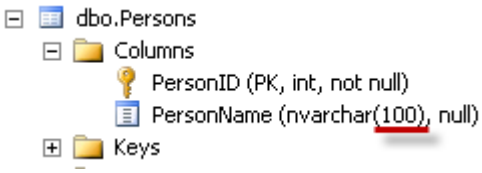
```
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>().HasKey(p => new { p.FirstName, p.LastName });
    }
}
```

Longitud maxima

Al usar el método .HasMaxLength (), el conteo máximo de caracteres se puede configurar para una propiedad.

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

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>()
                    .Property(t => t.Name)
                    .HasMaxLength(100);
    }
}
```

La columna resultante con la longitud de columna especificada:



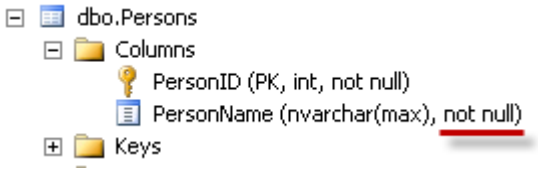
Propiedades requeridas (NO NULL)

Al usar el método .IsRequired (), las propiedades se pueden especificar como obligatorias, lo que significa que la columna tendrá una restricción NO NULA.

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

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..
```

```
        modelBuilder.Entity<Person>()
                    .Property(t => t.Name)
                    .IsRequired();
    }
}
```

La columna resultante con la restricción NOT NULL:



Nombramiento de clave externa explícita

Cuando existe una propiedad de navegación en un modelo, Entity Framework creará automáticamente una columna de clave externa. Si se desea un nombre de clave externa específico, pero no está contenido como una propiedad en el modelo, se puede establecer explícitamente mediante la API Fluent. Al utilizar el método de `Map` mientras se establece la relación de clave externa, se puede usar cualquier nombre único para claves externas.

```
public class Company
{
    public int Id { get; set; }
}

public class Employee
{
    property int Id { get; set; }
    property Company Employer { get; set; }
}

public class EmployeeContext : DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Employee>()
                    .HasRequired(x => x.Employer)
                    .WithRequiredDependent()
                    .Map(m => m.MapKey("EmployerId"));
    }
}
```

Después de especificar la relación, el método `Map` permite que el nombre de la clave externa se establezca explícitamente ejecutando `MapKey` . En este ejemplo, lo que habría resultado en un nombre de columna de Employer_Id ahora es EmployerId.

Lea Código primero - API fluida en línea: https://riptutorial.com/es/entity-framework/topic/4530/codigo-primero---api-fluida

Capítulo 8: Entidad marco código primero

Examples

Conectarse a una base de datos existente

Para lograr la tarea más sencilla en Entity Framework: para conectarse a una base de datos de `ExampleDatabase` Base de datos en su instancia local de MSSQL, debe implementar solo dos clases.

Primero está la clase de entidad, que se asignará a nuestra base de datos `dbo.People` .

```
class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
}
```

La clase utilizará las convenciones de Entity Framework y se `dbo.People` a la tabla `dbo.People` que se espera que tenga un clave principal, `PersonId` y una propiedad varchar (max), `FirstName` .

La segunda es la clase de contexto que deriva de `System.Data.Entity.DbContext` y que administrará los objetos de la entidad durante el tiempo de ejecución, los agrupará desde la base de datos, manejará la concurrencia y los guardará de nuevo en la base de datos.

```
class Context : DbContext
{
    public Context(string connectionString) : base(connectionString)
    {
        Database.SetInitializer<Context>(null);
    }

    public DbSet<Person> People { get; set; }
}
```

Tenga en cuenta que en el constructor de nuestro contexto debemos establecer el inicializador de la base de datos en nulo; no queremos que Entity Framework cree la base de datos, solo queremos acceder a ella.

Ahora puede manipular los datos de esa tabla, por ejemplo, cambiar el `FirstName` de la primera persona en la base de datos desde una aplicación de consola como esta:

```
class Program
{
    static void Main(string[] args)
    {
        using (var ctx = new Context("DbConnectionString"))
        {
            var firstPerson = ctx.People.FirstOrDefault();
            if (firstPerson != null) {
```

```
                    firstPerson.FirstName = "John";
                    ctx.SaveChanges();
                }
            }
        }
    }
```

En el código anterior creamos una instancia de contexto con un argumento "DbConnectionString". Esto se debe especificar en nuestro archivo `app.config` como este:

```
<connectionStrings>
  <add  name="DbConnectionString"
  connectionString="Data Source=.;Initial Catalog=ExampleDatabase;Integrated Security=True"
  providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Lea Entidad marco código primero en línea: https://riptutorial.com/es/entity-framework/topic/5337/entidad-marco-codigo-primero

Capítulo 9: Entidad-marco de código primeras migraciones

Examples

Habilitar las migraciones

Para habilitar las Migraciones de Code First en el marco de la entidad, use el comando

```
Enable-Migrations
```

en la *consola del administrador de paquetes* .

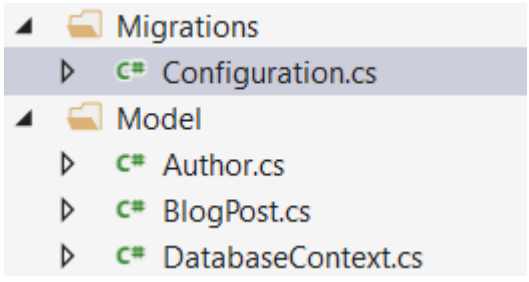
`DbContext` tener una implementación `DbContext` válida que contenga los objetos de su base de datos administrados por EF. En este ejemplo, el contexto de la base de datos contendrá los objetos `BlogPost` y `Author` :

```
internal class DatabaseContext: DbContext
{
    public DbSet<Author> Authors { get; set; }

    public DbSet<BlogPost> BlogPosts { get; set; }
}
```

Después de ejecutar el comando, debería aparecer la siguiente salida:

```
PM> Enable-Migrations
Checking if the context targets an existing database...
Code First Migrations enabled for project <YourProjectName>.
PM>
```

Además, debe aparecer una nueva carpeta de `Migrations` con un solo archivo `Configuration.cs` dentro de:



El siguiente paso sería crear su primer script de migración de base de datos que creará la base de datos inicial (ver el siguiente ejemplo).

Añade tu primera migración

Después de habilitar las migraciones (consulte este ejemplo), ahora puede crear su primera

migración que contenga una creación inicial de todas las tablas de base de datos, índices y conexiones.

Se puede crear una migración usando el comando

```
Add-Migration <migration-name>
```

Este comando creará una nueva clase que contiene dos métodos `Up` y `Down` que se utilizan para aplicar y eliminar la migración.

Ahora aplique el comando basado en el ejemplo anterior para crear una migración llamada *Inicial* :

```
PM> Add-Migration Initial
Scaffolding migration 'Initial'.
The Designer Code for this migration file includes a snapshot of your current Code
First model. This snapshot is used to calculate the changes to your model when you
scaffold the next migration. If you make additional changes to your model that you
want to include in this migration, then you can re-scaffold it by running
'Add-Migration Initial' again.
```

Se crea un nuevo archivo de fecha y *hora* _Initial.cs (solo se muestran las cosas importantes aquí):

```
public override void Up()
{
   CreateTable(
       "dbo.Authors",
        c => new
           {
               AuthorId = c.Int(nullable: false, identity: true),
               Name = c.String(maxLength: 128),
           })
        .PrimaryKey(t => t.AuthorId);

    CreateTable(
       "dbo.BlogPosts",
       c => new
           {
                Id = c.Int(nullable: false, identity: true),
                Title = c.String(nullable: false, maxLength: 128),
                Message = c.String(),
                Author_AuthorId = c.Int(),
            })
        .PrimaryKey(t => t.Id)
        .ForeignKey("dbo.Authors", t => t.Author_AuthorId)
        .Index(t => t.Author_AuthorId);
}

public override void Down()
{
    DropForeignKey("dbo.BlogPosts", "Author_AuthorId", "dbo.Authors");
    DropIndex("dbo.BlogPosts", new[] { "Author_AuthorId" });
    DropTable("dbo.BlogPosts");
    DropTable("dbo.Authors");
}
```

Como puede ver, en el método `Up()` se crean dos tablas `Authors` y `BlogPosts` y los campos se crean en consecuencia. Además, la relación entre las dos tablas se crea agregando el campo `Author_AuthorId` . Por otro lado, el método `Down()` intenta revertir las actividades de migración.

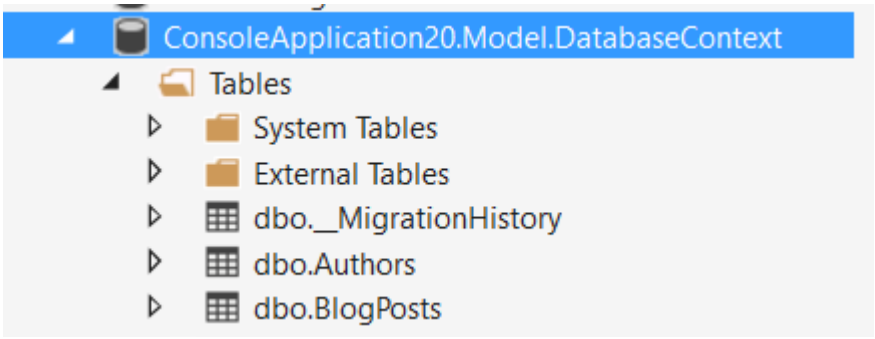
Si se siente seguro con su migración, puede aplicar la migración a la base de datos usando el comando:

```
Update-Database
```

Todas las migraciones pendientes (en este caso, la migración *inicial*) se aplican a la base de datos y luego se aplica el método de inicialización (el ejemplo apropiado)

```
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target
database.
Applying explicit migrations: [201609302203541_Initial].
Applying explicit migration: 201609302203541_Initial.
Running Seed method.
```

Puedes ver los resultados de las actividades en el explorador de SQL:



Para los comandos `Add-Migration` y `Update-Database` varias opciones disponibles que se pueden usar para modificar las actividades. Para ver todas las opciones, por favor use

```
get-help Add-Migration
```

y

```
get-help Update-Database
```

Siembra de datos durante las migraciones

Después de habilitar y crear migraciones, puede ser necesario llenar o migrar inicialmente los datos en su base de datos. Hay varias posibilidades, pero para migraciones simples puede usar el método 'Seed ()' en el archivo Configuración creada después de llamar a `enable-migrations` .

La función `Seed()` recupera un contexto de base de datos ya que es solo un parámetro y puede realizar operaciones EF dentro de esta función:

```
protected override void Seed(Model.DatabaseContext context);
```

Puedes realizar todo tipo de actividades dentro de `Seed()` . En caso de cualquier falla, la transacción completa (incluso los parches aplicados) se está revirtiendo.

Una función de ejemplo que crea datos solo si una tabla está vacía podría tener este aspecto:

```
protected override void Seed(Model.DatabaseContext context)
{
    if (!context.Customers.Any()) {
        Customer c = new Customer{ Id = 1, Name = "Demo" };
        context.Customers.Add(c);
        context.SaveData();
    }
}
```

Una buena característica proporcionada por los desarrolladores de EF es el método de extensión `AddOrUpdate()` . Este método permite actualizar los datos en función de la clave principal o insertar datos si aún no existen (el ejemplo se toma del código fuente generado de Configuration.cs):

```
protected override void Seed(Model.DatabaseContext context)
{
    context.People.AddOrUpdate(
      p => p.FullName,
      new Person { FullName = "Andrew Peters" },
      new Person { FullName = "Brice Lambson" },
      new Person { FullName = "Rowan Miller" }
    );
}
```

> Tenga en cuenta que se llama a `Seed()` después de que se haya aplicado el **último** parche. Si necesita migrar o generar datos durante los parches, debe usar otros enfoques.

Usando Sql () durante las migraciones

Por ejemplo: va a migrar una columna existente de no obligatorio a obligatorio. En este caso, es posible que deba completar algunos valores predeterminados en su migración para las filas en las que los campos modificados son realmente `NULL` . En caso de que el valor predeterminado sea simple (por ejemplo, "0"), puede usar una propiedad `default` o `defaultSql` en su definición de columna. En caso de que no sea tan fácil, puede usar la función `Sql()` en las funciones miembro `Up()` o `Down()` de sus migraciones.

Aquí hay un ejemplo. Suponiendo un *Autor de* clase que contiene una dirección de correo electrónico como parte del conjunto de datos. Ahora decidimos tener la dirección de correo electrónico como un campo obligatorio. Para migrar las columnas existentes, la empresa tiene la idea *inteligente* de crear direcciones de correo electrónico ficticias como `fullname@example.com` , donde el nombre completo es el nombre completo de los autores sin espacios. Al agregar el atributo `[Required]` al campo `Email` se crearía la siguiente migración:

```
public partial class AuthorsEmailRequired : DbMigration
{
    public override void Up()
```

```
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(nullable: false, maxLength: 512));
    }

    public override void Down()
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(maxLength: 512));
    }
}
```

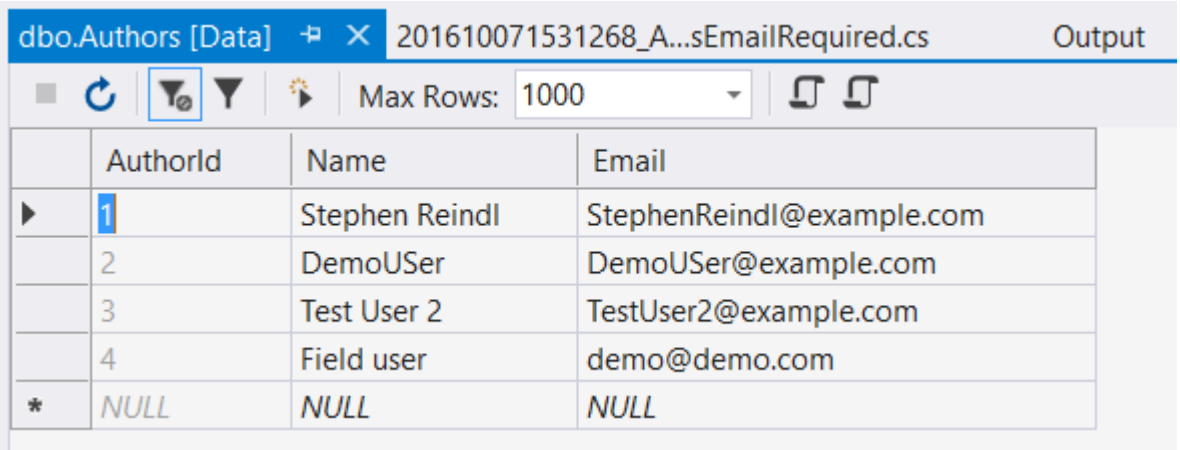
Esto fallaría en caso de que algunos campos NULOS estén dentro de la base de datos:

> No se puede insertar el valor NULL en la columna 'Correo electrónico', tabla 'App.Model.DatabaseContext.dbo.Authors'; La columna no permite nulos. La actualización falla

Agregar lo siguiente como **antes** del comando `AlterColumn` ayudará:

```
Sql(@"Update dbo.Authors
    set Email = REPLACE(name, ' ', '') + N'@example.com'
    where Email is null");
```

La llamada de `update-database` correctamente y la tabla tiene este aspecto (se muestran los datos de ejemplo):



Otro uso

Puede usar la función `Sql()` para todos los tipos de actividades DML y DDL en su base de datos. Se ejecuta como parte de la transacción de migración; Si el SQL falla, la migración completa falla y se realiza una reversión.

Haciendo "Update-Database" dentro de tu código

Las aplicaciones que se ejecutan en entornos que no son de desarrollo a menudo requieren actualizaciones de la base de datos. Después de usar el comando `Add-Migration` para crear los parches de la base de datos, existe la necesidad de ejecutar las actualizaciones en otros entornos, y también el entorno de prueba.

Los desafíos comúnmente enfrentados son:

- no se instala Visual Studio en entornos de producción, y
- No se permiten conexiones a entornos de conexión / cliente en la vida real.

Una solución es la siguiente secuencia de código que verifica las actualizaciones que se realizarán y las ejecuta en orden. Asegúrese de que las transacciones y el manejo de excepciones sean correctos para garantizar que no se pierdan datos en caso de errores.

```
void UpdateDatabase(MyDbConfiguration configuration) {
   DbMigrator dbMigrator = new DbMigrator( configuration);
   if ( dbMigrator.GetPendingMigrations().Any() )
   {
      // there are pending migrations run the migration job
      dbMigrator.Update();
   }
}
```

donde `MyDbConfiguration` es su configuración de migración en algún lugar de sus fuentes:

```
public class MyDbConfiguration : DbMigrationsConfiguration<ApplicationDbContext>
```

Código inicial de Entity Framework, primera migración paso a paso

1. Crear una aplicación de consola.
2. Instale el paquete nuget EntityFramework ejecutando `Install-Package EntityFramework` en "Package Manager Console"
3. Agregue su cadena de conexión en el archivo app.config. Es importante incluir `providerName="System.Data.SqlClient"` en su conexión.
4. Crea una clase pública como desees, algo como " `Blog` "
5. Cree su ContextClass que herede de DbContext, algo como " `BlogContext` "
6. Defina una propiedad en su contexto de tipo DbSet, algo como esto:

```
public class Blog
{
    public int Id { get; set; }

    public string Name { get; set; }
}
public class BlogContext: DbContext
{
    public BlogContext(): base("name=Your_Connection_Name")
    {
    }

    public virtual DbSet<Blog> Blogs{ get; set; }
}
```

7. Es importante pasar el nombre de la conexión en el constructor (aquí, Your_Connection_Name)
8. En la consola del Administrador de paquetes ejecute el comando `Enable-Migration` , esto creará una carpeta de migración en su proyecto

9. Ejecute el comando `Add-Migration Your_Arbitrary_Migraiton_Name` , esto creará una clase de migración en la carpeta de migraciones con dos métodos Up () y Down ()
10. Ejecute el comando `Update-Database` para crear una base de datos con una tabla de blog

Lea Entidad-marco de código primeras migraciones en línea: https://riptutorial.com/es/entity-framework/topic/7157/entidad-marco-de-codigo-primeras-migraciones

Capítulo 10: Entity Framework con SQLite

Introducción

SQLite es una base de datos SQL transaccional, autónoma, sin servidor. Se puede usar dentro de una aplicación .NET utilizando tanto una biblioteca de .NET SQLite disponible gratuitamente como el proveedor de Entity Framework SQLite. Este tema se centrará en la configuración y el uso del proveedor SQLite de Entity Framework.

Examples

Configuración de un proyecto para usar Entity Framework con un proveedor SQLite

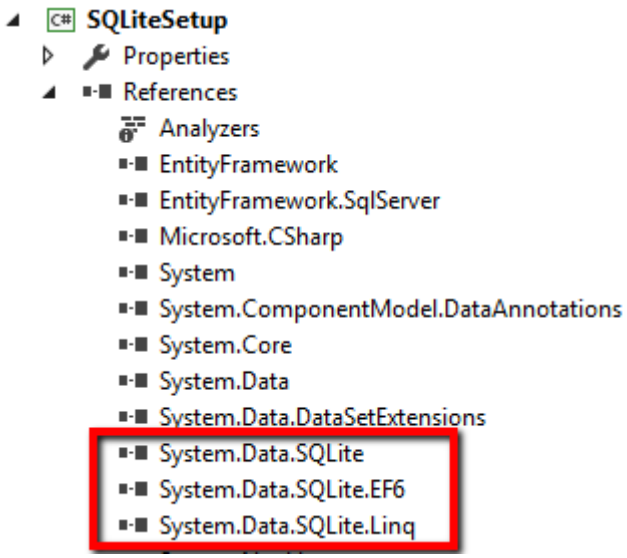
La biblioteca de Entity Framework solo viene con un proveedor de SQL Server. Para usar SQLite se requerirán dependencias y configuraciones adicionales. Todas las dependencias requeridas están disponibles en NuGet.

Instalar bibliotecas administradas de SQLite

Todas las dependencias modificadas pueden instalarse utilizando la consola de NuGet Package Manager. Ejecute el comando `Install-Package System.Data.SQLite` .

```
PM> Install-Package System.Data.SQLite
Attempting to gather dependency information for package 'System.Data.SQLite.1.0.104' with respect to proje
Attempting to resolve dependencies for package 'System.Data.SQLite.1.0.104' with DependencyBehavior 'Lowes
Resolving actions to install package 'System.Data.SQLite.1.0.104'
Resolved actions to install package 'System.Data.SQLite.1.0.104'
Adding package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Projec
Added package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Project
Added package 'EntityFramework.6.0.0' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Ent
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Ent

Type 'get-help EntityFramework' to see all available Entity Framework commands.
Successfully installed 'EntityFramework 6.0.0' to SQLiteSetup
Adding package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2
Added package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 20
Added package 'System.Data.SQLite.Core.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite.Core 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 20
Added package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 201
Added package 'System.Data.SQLite.EF6.1.0.104' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Sys
Successfully installed 'System.Data.SQLite.EF6 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2
Added package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 20
Added package 'System.Data.SQLite.Linq.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite.Linq 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.1.0.104', which only has dependencies, to project 'SQLiteSetup'.
Adding package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\P
Added package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Pr
Added package 'System.Data.SQLite.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite 1.0.104' to SQLiteSetup
```

Como se muestra arriba, al instalar `System.Data.SQLite` , todas las bibliotecas administradas relacionadas se instalan con él. Esto incluye `System.Data.SQLite.EF6` , el proveedor de EF para SQLite. El proyecto ahora también hace referencia a los ensamblajes necesarios para usar el proveedor SQLite.



Incluyendo la biblioteca no administrada

Las bibliotecas administradas de SQLite dependen de un ensamblado no administrado llamado `SQLite.Interop.dll` . Se incluye con los conjuntos de paquetes descargados con el paquete

SQLite, y se copian automáticamente en el directorio de compilación cuando se genera el proyecto. Sin embargo, debido a que no está administrado, no se incluirá en su lista de referencia. Pero tenga en cuenta que este ensamblaje se distribuye con la aplicación para que funcionen los ensamblajes de SQLite.

Nota: este ensamblaje depende de los bits, lo que significa que deberá incluir un ensamblaje específico para cada bitness que planee admitir (x86 / x64).

Edición de la aplicación del proyecto.config

El archivo `app.config` requerirá algunas modificaciones antes de que SQLite pueda usarse como proveedor de Entity Framework.

Arreglos requeridos

Al instalar el paquete, el archivo `app.config` se actualiza automáticamente para incluir las entradas necesarias para SQLite y SQLite EF. Lamentablemente estas entradas contienen algunos errores. Deben ser modificados antes de que funcione correctamente.

Primero, ubique el elemento `DbProviderFactories` en el archivo de configuración. Está dentro del elemento `system.data` y contendrá lo siguiente

```
<DbProviderFactories>
  <remove invariant="System.Data.SQLite.EF6" />
  <add name="SQLite Data Provider (Entity Framework 6)" invariant="System.Data.SQLite.EF6"
description=".NET Framework Data Provider for SQLite (Entity Framework 6)"
type="System.Data.SQLite.EF6.SQLiteProviderFactory, System.Data.SQLite.EF6" />
  <remove invariant="System.Data.SQLite" /><add name="SQLite Data Provider"
invariant="System.Data.SQLite" description=".NET Framework Data Provider for SQLite"
type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite" />
</DbProviderFactories>
```

Esto se puede simplificar para contener una sola entrada.

```
<DbProviderFactories>
    <add name="SQLite Data Provider" invariant="System.Data.SQLite.EF6" description=".NET
Framework Data Provider for SQLite" type="System.Data.SQLite.SQLiteFactory,
System.Data.SQLite" />
</DbProviderFactories>
```

Con esto, hemos especificado que los proveedores de EF6 SQLite deben usar la fábrica de SQLite.

Añadir cadena de conexión SQLite

Las cadenas de conexión se pueden agregar al archivo de configuración dentro del elemento raíz. Agregue una cadena de conexión para acceder a una base de datos SQLite.

```
<connectionStrings>
    <add name="TestContext" connectionString="data source=testdb.sqlite;initial
catalog=Test;App=EntityFramework;" providerName="System.Data.SQLite.EF6"/>
</connectionStrings>
```

Lo importante a tener en cuenta aquí es el `provider` . Se ha establecido en `System.Data.SQLite.EF6` . Esto le dice a EF que cuando usamos esta cadena de conexión, queremos usar SQLite. El `data source` especificado es solo un ejemplo y dependerá de la ubicación y el nombre de su base de datos SQLite.

Tu primer SQLb DbContext

Con toda la instalación y configuración completadas, ahora puede comenzar a usar un `DbContext` que funcionará en su base de datos SQLite.

```
public class TestContext : DbContext
{
    public TestContext()
        : base("name=TestContext") { }
}
```

Al especificar `name=TestContext` , he indicado que la cadena de conexión TestContext ubicada en el archivo `app.config` debe usarse para crear el contexto. Esa cadena de conexión se configuró para usar SQLite, por lo que este contexto usará una base de datos SQLite.

Lea Entity Framework con SQLite en línea: https://riptutorial.com/es/entity-framework/topic/9280/entity-framework-con-sqlite

Capítulo 11: Entity-Framework con Postgresql

Examples

Pasos previos necesarios para utilizar Entity Framework 6.1.3 con PostgresSql usando Npgsqlddexprovider

1) Tomó copia de seguridad de Machine.config de las ubicaciones C: \ Windows \ Microsoft.NET \ Framework \ v4.0.30319 \ Config y C: \ Windows \ Microsoft.NET \ Framework64 \ v4.0.30319 \ Config

2) Copielas en diferentes ubicaciones y edítelas como

a) `<system.data> <DbProviderFactories>` y agregue bajo `<system.data> <DbProviderFactories>`

```
            <add name="Npgsql Data Provider" invariant="Npgsql" support="FF"
            description=".Net Framework Data Provider for Postgresql Server"
            type="Npgsql.NpgsqlFactory, Npgsql, Version=2.2.5.0, Culture=neutral,
PublicKeyToken=5d8b90d52f46fda7" />
```

b) Si ya existe sobre la entrada, verifique y actualice.

3. Reemplace los archivos originales con otros modificados.
4. ejecute el símbolo del sistema del desarrollador para VS2013 como administrador.
5. si Npgsql ya está instalado, use el comando "gacutil -u Npgsql" para desinstalar y luego instale la nueva versión de Npgsql 2.5.0 con el comando "gacutil -i [ruta de acceso de dll]"
6. Hacer arriba para Mono.Security 4.0.0.0
7. Descargue NpgsqlDdexProvider-2.2.0-VS2013.zip y ejecute NpgsqlDdexProvider.vsix desde allí (cierre todas las instancias de Visual Studio)
8. Encontró EFTools6.1.3-beta1ForVS2013.msi y ejecútelo.
9. Después de crear un nuevo proyecto, instale la versión de EntityFramework (6.1.3), NpgSql (2.5.0) y NpgSql.EntityFramework (2.5.0) desde Manage Nuget Packages.10) Está listo. Agregue un nuevo modelo de datos de entidad en tu proyecto MVc

Lea Entity-Framework con Postgresql en línea: https://riptutorial.com/es/entity-framework/topic/7647/entity-framework-con-postgresql

Capítulo 12: Escenarios de mapeo avanzados: división de entidades, división de tablas

Introducción

Cómo configurar su modelo EF para admitir la división de entidades o la división de tablas.

Examples

División de la entidad

Digamos que tienes una clase de entidad como esta:

```
public class Person
{
   public int PersonId { get; set; }
   public string Name { get; set; }
   public string ZipCode { get; set; }
   public string City { get; set; }
   public string AddressLine { get; set; }
}
```

Y luego digamos que desea asignar esta entidad de Persona a dos tablas: una con el PersonId y el Nombre, y otra con los detalles de la dirección. Por supuesto, también necesitaría el PersonId aquí para identificar a qué persona pertenece la dirección. Básicamente, lo que quieres es dividir la entidad en dos (o incluso más) partes. De ahí el nombre, división de entidades. Puede hacer esto asignando cada una de las propiedades a una tabla diferente:

```
public class MyDemoContext : DbContext
{
  public DbSet<Person> Products { get; set; }

  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
      modelBuilder.Entity<Person>().Map(m =>
      {
         m.Properties(t => new { t.PersonId, t.Name });
         m.ToTable("People");
      }).Map(m =>
      {
        m.Properties(t => new { t.PersonId, t.AddressLine, t.City, t.ZipCode });
        m.ToTable("PersonDetails");
      });
  }
}
```

Esto creará dos tablas: People y PersonDetails. La persona tiene dos campos, PersonId y Name,

PersonDetails tiene cuatro columnas, PersonId, AddressLine, City y ZipCode. En Personas, PersonId es la clave principal. En PersonDetails, la clave principal también es PersonId, pero también es una clave externa que hace referencia a PersonId en la tabla Person.

Si consulta el DbSet de personas, EF realizará una unión en los PersonIds para obtener los datos de ambas tablas para completar las entidades.

También puedes cambiar el nombre de las columnas:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
  modelBuilder.Entity<Person>().Map(m =>
  {
    m.Properties(t => new { t.PersonId });
    m.Property(t => t.Name).HasColumnName("PersonName");
    m.ToTable("People");
  }).Map(m =>
  {
    m.Property(t => t.PersonId).HasColumnName("ProprietorId");
    m.Properties(t => new { t.AddressLine, t.City, t.ZipCode });
    m.ToTable("PersonDetails");
  });
}
```

Esto creará la misma estructura de tabla, pero en la tabla Personas habrá una columna PersonName en lugar de la columna Nombre, y en la tabla PersonDetails habrá un ProprietorId en lugar de la columna PersonId.

División de la mesa

Y ahora digamos que desea hacer lo contrario a la división de entidades: en lugar de asignar una entidad a dos tablas, le gustaría asignar una tabla a dos entidades. Esto se llama división de tabla. Digamos que tienes una tabla con cinco columnas: PersonId, Name, AddressLine, City, ZipCode, donde PersonId es la clave principal. Y luego te gustaría crear un modelo EF como este:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public Address Address { get; set; }
}

public class Address
{
  public string ZipCode { get; set; }
  public string City { get; set; }
  public string AddressLine { get; set; }
  public int PersonId { get; set; }
  public Person Person { get; set; }
}
```

Una cosa salta de inmediato: no hay AddressId en la dirección. Esto se debe a que las dos entidades se asignan a la misma tabla, por lo que también deben tener la misma clave principal. Si haces la división de tablas, esto es algo con lo que tienes que lidiar. Entonces, además de la

división de tablas, también debe configurar la entidad de Dirección y especificar la clave primaria. Y aquí está cómo:

```
public class MyDemoContext : DbContext
{
  public DbSet<Person> Products { get; set; }
  public DbSet<Address> Addresses { get; set; }

  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
     modelBuilder.Entity<Address>().HasKey(t => t.PersonId);
     modelBuilder.Entity<Person>().HasRequired(t => t.Address)
                                  .WithRequiredPrincipal(t => t.Person);

     modelBuilder.Entity<Person>().Map(m => m.ToTable("People"));
     modelBuilder.Entity<Address>().Map(m => m.ToTable("People"));
  }
}
```

Lea Escenarios de mapeo avanzados: división de entidades, división de tablas en línea: https://riptutorial.com/es/entity-framework/topic/9362/escenarios-de-mapeo-avanzados--division-de-entidades--division-de-tablas

Capítulo 13: Estado de la entidad gestora

Observaciones

Las entidades en Entity Framework pueden tener varios estados enumerados por la enumeración System.Data.Entity.EntityState . Estos estados son:

```
Added
Deleted
Detached
Modified
Unchanged
```

Entity Framework trabaja con POCOs. Eso significa que las entidades son clases simples que no tienen propiedades ni métodos para administrar su propio estado. El estado de la entidad es administrado por un contexto en sí mismo, en el `ObjectStateManager` .

Este tema cubre varias formas de establecer el estado de la entidad.

Examples

Estado de configuración Agregado de una sola entidad

`EntityState.Added` se puede configurar de dos maneras totalmente equivalentes:

1. Al establecer el estado de su entrada en el contexto:

   ```
   context.Entry(entity).State = EntityState.Added;
   ```

2. `DbSet` a un `DbSet` del contexto:

   ```
   context.Entities.Add(entity);
   ```

Al llamar a `SaveChanges` , la entidad se insertará en la base de datos. Cuando tiene una columna de identidad (una clave principal de auto-establecimiento, auto-incremento), luego, después de `SaveChanges` , la propiedad de la clave primaria de la entidad contendrá el valor recién generado, *incluso cuando esta propiedad ya tuviera un valor* .

Estado de configuración Agregado de un gráfico de objeto

Establecer el estado de un *gráfico de objeto* (una colección de entidades relacionadas) en `Added` es diferente a establecer una sola entidad como `Added` (ver este ejemplo).

En el ejemplo, almacenamos planetas y sus lunas:

Modelo de clase

```
public class Planet
{
    public Planet()
    {
        Moons = new HashSet<Moon>();
    }
    public int ID { get; set; }
    public string Name { get; set; }
    public ICollection<Moon> Moons { get; set; }
}

public class Moon
{
    public int ID { get; set; }
    public int PlanetID { get; set; }
    public string Name { get; set; }
}
```

Contexto

```
public class PlanetDb : DbContext
{
    public property DbSet<Planet> Planets { get; set; }
}
```

Usamos una instancia de este contexto para agregar planetas y sus lunas:

Ejemplo

```
var mars = new Planet { Name = "Mars" };
mars.Moons.Add(new Moon { Name = "Phobos" });
mars.Moons.Add(new Moon { Name = "Deimos" });

context.Planets.Add(mars);

Console.WriteLine(context.Entry(mars).State);
Console.WriteLine(context.Entry(mars.Moons.First()).State);
```

Salida:

```
Added
Added
```

Lo que vemos aquí es que agregar un `Planet` también establece el estado de una luna en `Added` .

Al configurar el estado de una entidad como `Added` , todas las entidades en sus propiedades de navegación (propiedades que "navegan" hacia otras entidades, como `Planet.Moons`) también se marcan como `Added` , a *menos que ya estén adjuntas al contexto* .

Lea Estado de la entidad gestora en línea: https://riptutorial.com/es/entity-framework/topic/5256/estado-de-la-entidad-gestora

Capítulo 14: Herencia con EntityFramework (primero el código)

Examples

Tabla por jerarquía

Este enfoque generará una tabla en la base de datos para representar toda la estructura de herencia.

Ejemplo:

```
public abstract class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
}

public class Employee : Person
{
    public DateTime AdmissionDate { get; set; }
    public string JobDescription { get; set; }
}

public class Customer : Person
{
    public DateTime LastPurchaseDate { get; set; }
    public int TotalVisits { get; set; }
}

// On DbContext
public DbSet<Person> People { get; set; }
public DbSet<Employee> Employees { get; set; }
public DbSet<Customer> Customers { get; set; }
```

La tabla generada será:

Tabla: Campos de personas: Id. Nombre Fecha de nacimiento. Disimitador. Fecha de admisión. Fecha de trabajo. Descripción. Última compra. Fecha de visita. Total.

Donde 'Discriminador' guardará el nombre de la subclase en la herencia y 'AdmissionDate', 'JobDescription', 'LastPurchaseDate', 'TotalVisits' son anulables.

Ventajas

- Mejor rendimiento ya que no se requieren combinaciones, aunque para muchas columnas la base de datos puede requerir muchas operaciones de paginación.
- Fácil de usar y crear
- Fácil de agregar más subclases y campos

Desventajas

- Viola la tercera forma normal de Wikipedia: tercera forma normal
- Crea lotes de campos anulables.

Tabla por tipo

Este enfoque generará (n + 1) tablas en la base de datos para representar toda la estructura de herencia donde n es el número de subclases.

Cómo:

```
public abstract class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
}

[Table("Employees")]
public class Employee : Person
{
    public DateTime AdmissionDate { get; set; }
    public string JobDescription { get; set; }
}

[Table("Customers")]
public class Customer : Person
{
    public DateTime LastPurchaseDate { get; set; }
    public int TotalVisits { get; set; }
}

// On DbContext
public DbSet<Person> People { get; set; }
public DbSet<Employee> Employees { get; set; }
public DbSet<Customer> Customers { get; set; }
```

La tabla generada será:

Tabla: Campos de personas: Id. Nombre Fecha de nacimiento

Tabla: Campos de empleados: PersonId AdmissionDate JobDescription

Tabla: Clientes: Campos: PersonId LastPurchaseDate TotalVisits

Donde 'PersonId' en todas las tablas será una clave principal y una restricción para People.Id

Ventajas

- Tablas normalizadas
- Fácil de agregar columnas y subclases
- Columnas sin nulos

Desventajas

- Se requiere unirse para recuperar los datos
- La inferencia de subclases es más cara

Lea Herencia con EntityFramework (primero el código) en línea: https://riptutorial.com/es/entity-framework/topic/7715/herencia-con-entityframework--primero-el-codigo-

Capítulo 15: Inicializadores de bases de datos

Examples

CreateDatabaseIfNotExists

Implementación de `IDatabaseInitializer` que se utiliza en EntityFramework de forma predeterminada. Como su nombre lo indica, crea la base de datos si no existe. Sin embargo, cuando se cambia el modelo, se produce una excepción.

Uso:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new CreateDatabaseIfNotExists<MyContext>());
    }
}
```

DropCreateDatabaseIfModelChanges

Esta implementación de `IDatabaseInitializer` elimina y `IDatabaseInitializer` crear la base de datos si el modelo cambia automáticamente.

Uso:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<MyContext>());
    }
}
```

DropCreateDatabaseSiempre

Esta implementación de `IDatabaseInitializer` elimina y recrea la base de datos cada vez que se usa su contexto en el dominio de aplicación de aplicaciones. Tenga cuidado con la pérdida de datos debido al hecho de que la base de datos se vuelve a crear.

Uso:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new DropCreateDatabaseAlways<MyContext>());
    }
}
```

Inicializador de base de datos personalizado

Puede crear su propia implementación de `IDatabaseInitializer` .

Ejemplo de implementación de un inicializador, que migrará la base de datos a 0 y luego migrará hasta la migración más reciente (útil, por ejemplo, cuando se ejecutan pruebas de integración). Para hacer eso necesitarías un tipo `DbMigrationsConfiguration` también.

```
public class RecreateFromScratch<TContext, TMigrationsConfiguration> :
IDatabaseInitializer<TContext>
where TContext : DbContext
where TMigrationsConfiguration : DbMigrationsConfiguration<TContext>, new()
{
    private readonly DbMigrationsConfiguration<TContext> _configuration;

    public RecreateFromScratch()
    {
        _configuration = new TMigrationsConfiguration();
    }

    public void InitializeDatabase(TContext context)
    {
        var migrator = new DbMigrator(_configuration);
        migrator.Update("0");
        migrator.Update();
    }
}
```

MigrateDatabaseToLatestVersion

Una implementación de `IDatabaseInitializer` que utilizará las migraciones de Code First para actualizar la base de datos a la última versión. Para usar este inicializador tienes que usar `DbMigrationsConfiguration` tipo `DbMigrationsConfiguration` .

Uso:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(
            new MigrateDatabaseToLatestVersion<MyContext, Configuration>());
    }
}
```

Lea Inicializadores de bases de datos en línea: https://riptutorial.com/es/entity-framework/topic/5526/inicializadores-de-bases-de-datos

Capítulo 16: Mejores Prácticas para Entity Framework (Simple y Profesional)

Introducción

Este artículo presenta una práctica simple y profesional para utilizar Entity Framework.

Simple: porque solo necesita una clase (con una interfaz)

Profesional: porque aplica los principios de la arquitectura SÓLIDA.

No quiero hablar más ... ¡Vamos a disfrutarlo!

Examples

1- Entity Framework @ Data layer (Conceptos básicos)

En este artículo usaremos una base de datos simple llamada "Compañía" con dos tablas:

[dbo]. [Categorías] ([CategoryID], [CategoryName])

[dbo]. [Productos] ([ProductID], [CategoryID], [ProductName])

1-1 Generar código Entity Framework

En esta capa generamos el código de Entity Framework (en la biblioteca del proyecto) (vea este artículo en cómo puede hacerlo) y luego tendrá las siguientes clases

```
public partial class CompanyContext : DbContext
public partial class Product
public partial class Category
```

1-2 Crear interfaz básica

Crearemos una interfaz para nuestras funciones básicas.

```
public interface IDbRepository : IDisposable
{
    #region Tables and Views functions

    IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class;
    TEntity Add<TEntity>(TEntity entity) where TEntity : class;
    TEntity Delete<TEntity>(TEntity entity) where TEntity : class;
    TEntity Attach<TEntity>(TEntity entity) where TEntity : class;
    TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class;

    #endregion Tables and Views functions
```

```
    #region Transactions Functions

    int Commit();
    Task<int> CommitAsync(CancellationToken cancellationToken = default(CancellationToken));

    #endregion Transactions Functions

    #region Database Procedures and Functions

    TResult Execute<TResult>(string functionName, params object[] parameters);

    #endregion Database Procedures and Functions
}
```

1-3 Implementando la interfaz básica

```
/// <summary>
/// Implementing basic tables, views, procedures, functions, and transaction functions
/// Select (GetAll), Insert (Add), Delete, and Attach
/// No Edit (Modify) function (can modify attached entity without function call)
/// Executes database procedures or functions (Execute)
/// Transaction functions (Commit)
/// More functions can be added if needed
/// </summary>
/// <typeparam name="TEntity">Entity Framework table or view</typeparam>
public class DbRepository : IDbRepository
{
    #region Protected Members

    protected DbContext _dbContext;

    #endregion Protected Members

    #region Constractors

    /// <summary>
    /// Repository constructor
    /// </summary>
    /// <param name="dbContext">Entity framework databse context</param>
    public DbRepository(DbContext dbContext)
    {
        _dbContext = dbContext;

        ConfigureContext();
    }

    #endregion Constractors

    #region IRepository Implementation

    #region Tables and Views functions

    /// <summary>
    /// Query all
    /// Set noTracking to true for selecting only (read-only queries)
    /// Set noTracking to false for insert, update, or delete after select
    /// </summary>
    public virtual IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult :
class
    {
```

```
        var entityDbSet = GetDbSet<TResult>();

        if (noTracking)
            return entityDbSet.AsNoTracking();

        return entityDbSet;
    }

    public virtual TEntity Add<TEntity>(TEntity entity) where TEntity : class
    {
        return GetDbSet<TEntity>().Add(entity);
    }

    /// <summary>
    /// Delete loaded (attached) or unloaded (Detached) entitiy
    /// No need to load object to delete it
    /// Create new object of TEntity and set the id then call Delete function
    /// </summary>
    /// <param name="entity">TEntity</param>
    /// <returns></returns>
    public virtual TEntity Delete<TEntity>(TEntity entity) where TEntity : class
    {
        if (_dbContext.Entry(entity).State == EntityState.Detached)
        {
            _dbContext.Entry(entity).State = EntityState.Deleted;
            return entity;
        }
        else
            return GetDbSet<TEntity>().Remove(entity);
    }

    public virtual TEntity Attach<TEntity>(TEntity entity) where TEntity : class
    {
        return GetDbSet<TEntity>().Attach(entity);
    }

    public virtual TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class
    {
        if (_dbContext.Entry(entity).State == EntityState.Detached)
            return Attach(entity);

        return entity;
    }

    #endregion Tables and Views functions

    #region Transactions Functions

    /// <summary>
    /// Saves all changes made in this context to the underlying database.
    /// </summary>
    /// <returns>The number of objects written to the underlying database.</returns>
    public virtual int Commit()
    {
        return _dbContext.SaveChanges();
    }

    /// <summary>
    /// Asynchronously saves all changes made in this context to the underlying database.
    /// </summary>
    /// <param name="cancellationToken">A System.Threading.CancellationToken to observe while
```

```
waiting for the task to complete.</param>
    /// <returns>A task that represents the asynchronous save operation.  The task result
contains the number of objects written to the underlying database.</returns>
    public virtual Task<int> CommitAsync(CancellationToken cancellationToken =
default(CancellationToken))
    {
        return _dbContext.SaveChangesAsync(cancellationToken);
    }

    #endregion Transactions Functions

    #region Database Procedures and Functions

    /// <summary>
    /// Executes any function in the context
    /// use to call database procesdures and functions
    /// </summary>>
    /// <typeparam name="TResult">return function type</typeparam>
    /// <param name="functionName">context function name</param>
    /// <param name="parameters">context function parameters in same order</param>
    public virtual TResult Execute<TResult>(string functionName, params object[] parameters)
    {
        MethodInfo method = _dbContext.GetType().GetMethod(functionName);

        return (TResult)method.Invoke(_dbContext, parameters);
    }

    #endregion Database Procedures and Functions

    #endregion IRepository Implementation

    #region IDisposable Implementation

    public void Dispose()
    {
        _dbContext.Dispose();
    }

    #endregion IDisposable Implementation

    #region Protected Functions

    /// <summary>
    /// Set Context Configuration
    /// </summary>
    protected virtual void ConfigureContext()
    {
        // set your recommended Context Configuration
        _dbContext.Configuration.LazyLoadingEnabled = false;
    }

    #endregion Protected Functions

    #region Private Functions

    private DbSet<TEntity> GetDbSet<TEntity>() where TEntity : class
    {
        return _dbContext.Set<TEntity>();
    }

    #endregion Private Functions
```

```
}
```

2- Entity Framework @ Business layer

En esta capa escribiremos el negocio de la aplicación.

Se recomienda para cada pantalla de presentación, cree la interfaz de negocios y la clase de implementación que contengan todas las funciones necesarias para la pantalla.

A continuación escribiremos el negocio para pantalla de producto como ejemplo.

```
/// <summary>
/// Contains Product Business functions
/// </summary>
public interface IProductBusiness
{
    Product SelectById(int productId, bool noTracking = true);
    Task<IEnumerable<dynamic>> SelectByCategoryAsync(int CategoryId);
    Task<Product> InsertAsync(string productName, int categoryId);
    Product InsertForNewCategory(string productName, string categoryName);
    Product Update(int productId, string productName, int categoryId);
    Product Update2(int productId, string productName, int categoryId);
    int DeleteWithoutLoad(int productId);
    int DeleteLoadedProduct(Product product);
    IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId);
}



/// <summary>
/// Implementing Product Business functions
/// </summary>
public class ProductBusiness : IProductBusiness
{
    #region Private Members

    private IDbRepository _dbRepository;

    #endregion Private Members

    #region Constructors

    /// <summary>
    /// Product Business Constructor
    /// </summary>
    /// <param name="dbRepository"></param>
    public ProductBusiness(IDbRepository dbRepository)
    {
        _dbRepository = dbRepository;
    }

    #endregion Constructors

    #region IProductBusiness Function

    /// <summary>
    /// Selects Product By Id
```

```
    /// </summary>
    public Product SelectById(int productId, bool noTracking = true)
    {
        var products = _dbRepository.GetAll<Product>(noTracking);

        return products.FirstOrDefault(pro => pro.ProductID == productId);
    }

    /// <summary>
    /// Selects Products By Category Id Async
    /// To have async method, add reference to EntityFramework 6 dll or higher
    /// also you need to have the namespace "System.Data.Entity"
    /// </summary>
    /// <param name="CategoryId">CategoryId</param>
    /// <returns>Return what ever the object that you want to return</returns>
    public async Task<IEnumerable<dynamic>> SelectByCategoryAsync(int CategoryId)
    {
        var products = _dbRepository.GetAll<Product>();
        var categories = _dbRepository.GetAll<Category>();

        var result = (from pro in products
                      join cat in categories
                      on pro.CategoryID equals cat.CategoryID
                      where pro.CategoryID == CategoryId
                      select new
                      {
                          ProductId = pro.ProductID,
                          ProductName = pro.ProductName,
                          CategoryName = cat.CategoryName
                      }
                              );

        return await result.ToListAsync();
    }

    /// <summary>
    /// Insert Async new product for given category
    /// </summary>
    public async Task<Product> InsertAsync(string productName, int categoryId)
    {
        var newProduct = _dbRepository.Add(new Product() { ProductName = productName,
CategoryID = categoryId });

        await _dbRepository.CommitAsync();

        return newProduct;
    }

    /// <summary>
    /// Insert new product and new category
    /// Do many database actions in one transaction
    /// each _dbRepository.Commit(); will commit one transaction
    /// </summary>
    public Product InsertForNewCategory(string productName, string categoryName)
    {
        var newCategory = _dbRepository.Add(new Category() { CategoryName = categoryName });
        var newProduct = _dbRepository.Add(new Product() { ProductName = productName, Category
= newCategory });

        _dbRepository.Commit();
```

```
        return newProduct;
    }

    /// <summary>
    /// Update given product with tracking
    /// </summary>
    public Product Update(int productId, string productName, int categoryId)
    {
        var product = SelectById(productId,false);
        product.CategoryID = categoryId;
        product.ProductName = productName;

        _dbRepository.Commit();

        return product;
    }

    /// <summary>
    /// Update given product with no tracking and attach function
    /// </summary>
    public Product Update2(int productId, string productName, int categoryId)
    {
        var product = SelectById(productId);
        _dbRepository.Attach(product);

        product.CategoryID = categoryId;
        product.ProductName = productName;

        _dbRepository.Commit();

        return product;
    }

    /// <summary>
    /// Deletes product without loading it
    /// </summary>
    public int DeleteWithoutLoad(int productId)
    {
        _dbRepository.Delete(new Product() { ProductID = productId });

        return _dbRepository.Commit();
    }

    /// <summary>
    /// Deletes product after loading it
    /// </summary>
    public int DeleteLoadedProduct(Product product)
    {
        _dbRepository.Delete(product);

        return _dbRepository.Commit();
    }

    /// <summary>
    /// Assuming we have the following procedure in database
    /// PROCEDURE [dbo].[GetProductsCategory] @CategoryID INT, @OrderBy VARCHAR(50)
    /// </summary>
    public IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId)
    {
        return
_dbRepository.Execute<IEnumerable<GetProductsCategory_Result>>("GetProductsCategory",
```

```
categoryId, "ProductName DESC");
    }


    #endregion IProductBusiness Function
}
```

3- Usando capa de presentación @ capa de negocios (MVC)

En este ejemplo usaremos la capa de negocios en la capa de presentación. Y usaremos MVC como ejemplo de la capa de presentación (pero puede usar cualquier otra capa de presentación).

Primero necesitamos registrar el IoC (usaremos Unity, pero puede usar cualquier IoC), luego escribir nuestra capa de presentación

3-1 Registrar tipos de Unity dentro de MVC

3-1-1 Agregue el respaldo NuGet de "Unity bootstrapper for ASP.NET MVC"

3-1-2 Agregar UnityWebActivator.Start (); en el archivo Global.asax.cs (función Application_Start ())

3-1-3 Modificar la función UnityConfig.RegisterTypes de la siguiente manera

```
    public static void RegisterTypes(IUnityContainer container)
    {
        // Data Access Layer
        container.RegisterType<DbContext, CompanyContext>(new PerThreadLifetimeManager());
        container.RegisterType(typeof(IDbRepository), typeof(DbRepository), new
PerThreadLifetimeManager());

        // Business Layer
        container.RegisterType<IProductBusiness, ProductBusiness>(new
PerThreadLifetimeManager());

    }
```

3-2 Usando capa de presentación @ Business layer (MVC)

```
public class ProductController : Controller
{
    #region Private Members

    IProductBusiness _productBusiness;

    #endregion Private Members

    #region Constractors

    public ProductController(IProductBusiness productBusiness)
    {
        _productBusiness = productBusiness;
    }

    #endregion Constractors
```

```
    #region Action Functions

    [HttpPost]
    public ActionResult InsertForNewCategory(string productName, string categoryName)
    {
        try
        {
            // you can use any of IProductBusiness functions
            var newProduct = _productBusiness.InsertForNewCategory(productName, categoryName);

            return Json(new { success = true, data = newProduct });
        }
        catch (Exception ex)
        {   /* log ex*/
            return Json(new { success = false, errorMessage = ex.Message});
        }
    }

    [HttpDelete]
    public ActionResult SmartDeleteWithoutLoad(int productId)
    {
        try
        {
            // deletes product without load
            var deletedProduct = _productBusiness.DeleteWithoutLoad(productId);

            return Json(new { success = true, data = deletedProduct });
        }
        catch (Exception ex)
        {   /* log ex*/
            return Json(new { success = false, errorMessage = ex.Message });
        }
    }

    public async Task<ActionResult> SelectByCategoryAsync(int CategoryId)
    {
        try
        {
            var results = await _productBusiness.SelectByCategoryAsync(CategoryId);

            return Json(new { success = true, data = results },JsonRequestBehavior.AllowGet);
        }
        catch (Exception ex)
        {   /* log ex*/
            return Json(new { success = false, errorMessage = ex.Message
},JsonRequestBehavior.AllowGet);
        }
    }
    #endregion Action Functions
}
```

4- Entity Framework @ Unit Test Layer

En la capa Prueba de unidad usualmente probamos las funcionalidades de la capa de negocios. Y para hacer esto, eliminaremos las dependencias de la capa de datos (Entity Framework).

Y la pregunta ahora es: ¿Cómo puedo eliminar las dependencias de Entity Framework para probar de forma unitaria las funciones de la capa empresarial?

Y la respuesta es simple: haremos una implementación falsa para IDbRepository Interface y luego podremos hacer nuestra prueba de unidad.

4-1 Implementando la interfaz básica (implementación falsa)

```
class FakeDbRepository : IDbRepository
{
    #region Protected Members

    protected Hashtable _dbContext;
    protected int _numberOfRowsAffected;
    protected Hashtable _contextFunctionsResults;

    #endregion Protected Members

    #region Constractors

    public FakeDbRepository(Hashtable contextFunctionsResults = null)
    {
        _dbContext = new Hashtable();
        _numberOfRowsAffected = 0;
        _contextFunctionsResults = contextFunctionsResults;
    }

    #endregion Constractors

    #region IRepository Implementation

    #region Tables and Views functions

    public IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class
    {
        return GetDbSet<TResult>().AsQueryable();
    }

    public TEntity Add<TEntity>(TEntity entity) where TEntity : class
    {
        GetDbSet<TEntity>().Add(entity);
        ++_numberOfRowsAffected;
        return entity;
    }

    public TEntity Delete<TEntity>(TEntity entity) where TEntity : class
    {
        GetDbSet<TEntity>().Remove(entity);
        ++_numberOfRowsAffected;
        return entity;
    }

    public TEntity Attach<TEntity>(TEntity entity) where TEntity : class
    {
        return Add(entity);
    }

    public TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class
    {
        if (!GetDbSet<TEntity>().Contains(entity))
            return Attach(entity);

        return entity;
```

```
    }

    #endregion Tables and Views functions

    #region Transactions Functions


    public virtual int Commit()
    {
        var numberOfRowsAffected = _numberOfRowsAffected;
        _numberOfRowsAffected = 0;
        return numberOfRowsAffected;
    }

    public virtual Task<int> CommitAsync(CancellationToken cancellationToken =
default(CancellationToken))
    {
        var numberOfRowsAffected = _numberOfRowsAffected;
        _numberOfRowsAffected = 0;
        return new Task<int>(() => numberOfRowsAffected);
    }

    #endregion Transactions Functions

    #region Database Procedures and Functions

    public virtual TResult Execute<TResult>(string functionName, params object[] parameters)
    {
        if (_contextFunctionsResults != null &&
_contextFunctionsResults.Contains(functionName))
            return (TResult)_contextFunctionsResults[functionName];

        throw new NotImplementedException();
    }

    #endregion Database Procedures and Functions

    #endregion IRepository Implementation

    #region IDisposable Implementation

    public void Dispose()
    {

    }

    #endregion IDisposable Implementation

    #region Private Functions

    private List<TEntity> GetDbSet<TEntity>() where TEntity : class
    {
        if (!_dbContext.Contains(typeof(TEntity)))
            _dbContext.Add(typeof(TEntity), new List<TEntity>());

        return (List<TEntity>)_dbContext[typeof(TEntity)];
    }

    #endregion Private Functions
}
```

4-2 Ejecuta tu unidad de pruebas

```
[TestClass]
public class ProductUnitTest
{
    [TestMethod]
    public void TestInsertForNewCategory()
    {
        // Initialize repositories
        FakeDbRepository _dbRepository = new FakeDbRepository();

        // Initialize Business object
        IProductBusiness productBusiness = new ProductBusiness(_dbRepository);

        // Process test method
        productBusiness.InsertForNewCategory("Test Product", "Test Category");

        int _productCount = _dbRepository.GetAll<Product>().Count();
        int _categoryCount = _dbRepository.GetAll<Category>().Count();

        Assert.AreEqual<int>(1, _productCount);
        Assert.AreEqual<int>(1, _categoryCount);
    }

    [TestMethod]
    public void TestProceduresFunctionsCall()
    {
        // Initialize Procedures / Functions result
        Hashtable _contextFunctionsResults = new Hashtable();
        _contextFunctionsResults.Add("GetProductsCategory", new
List<GetProductsCategory_Result> {
            new GetProductsCategory_Result() { ProductName = "Product 1", ProductID = 1,
CategoryName = "Category 1" },
            new GetProductsCategory_Result() { ProductName = "Product 2", ProductID = 2,
CategoryName = "Category 1" },
            new GetProductsCategory_Result() { ProductName = "Product 3", ProductID = 3,
CategoryName = "Category 1" }});

        // Initialize repositories
        FakeDbRepository _dbRepository = new FakeDbRepository(_contextFunctionsResults);

        // Initialize Business object
        IProductBusiness productBusiness = new ProductBusiness(_dbRepository);

        // Process test method
        var results = productBusiness.GetProductsCategory(1);

        Assert.AreEqual<int>(3, results.Count());
    }
}
```

Lea Mejores Prácticas para Entity Framework (Simple y Profesional) en línea: https://riptutorial.com/es/entity-framework/topic/8879/mejores-practicas-para-entity-framework--simple-y-profesional-

Capítulo 17: Modelo de restricciones

Examples

Relaciones uno a muchos

UserType pertenece a muchos Usuarios <-> Los usuarios tienen un UserType

Propiedad de navegación unidireccional con requerido

```
public class UserType
{
    public int UserTypeId {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int UserTypeId {get; set;}
    public virtual UserType UserType {get; set;}
}

Entity<User>().HasRequired(u => u.UserType).WithMany().HasForeignKey(u => u.UserTypeId);
```

Propiedad de navegación unidireccional con opcional (la clave externa debe ser de tipo `Nullable`)

```
public class UserType
{
    public int UserTypeId {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int? UserTypeId {get; set;}
    public virtual UserType UserType {get; set;}
}

Entity<User>().HasOptional(u => u.UserType).WithMany().HasForeignKey(u => u.UserTypeId);
```

Propiedad de navegación bidireccional con (requerido / opcional, cambiar la propiedad de clave externa según sea necesario)

```
public class UserType
{
    public int UserTypeId {get; set;}
    public virtual ICollection<User> Users {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int UserTypeId {get; set;}
    public virtual UserType UserType {get; set;}
}
```

Necesario

```
Entity<User>().HasRequired(u => u.UserType).WithMany(ut => ut.Users).HasForeignKey(u =>
u.UserTypeId);
```

Opcional

```
Entity<User>().HasOptional(u => u.UserType).WithMany(ut => ut.Users).HasForeignKey(u =>
u.UserTypeId);
```

Lea Modelo de restricciones en línea: https://riptutorial.com/es/entity-framework/topic/4528/modelo-de-restricciones

Capítulo 18: Plantillas .t4 en Entidad-marco

Examples

Agregando dinámicamente interfaces al modelo

Cuando se trabaja con un modelo existente que es bastante grande y se regenera con bastante frecuencia en los casos en que la abstracción es necesaria, puede ser costoso ir redecorando manualmente el modelo con interfaces. En tales casos, es posible que desee agregar algún comportamiento dinámico a la generación de modelos.

El siguiente ejemplo mostrará cómo agregar interfaces automáticamente en clases que tienen nombres de columna específicos:

En su modelo de ir a `.tt` archivo de modificar la `EntityClassOpening` método en el siguiente manera, esto añadirá `IPolicyNumber` interfaz en entidades que tienen `POLICY_NO` columna y `IUniqueId` en `UNIQUE_ID`

```
public string EntityClassOpening(EntityType entity)
{
    var stringsToMatch = new Dictionary<string,string> { { "POLICY_NO", "IPolicyNumber" }, {
"UNIQUE_ID", "IUniqueId" } };
    return string.Format(
        CultureInfo.InvariantCulture,
        "{0} {1}partial class {2}{3}{4}",
        Accessibility.ForType(entity),
        _code.SpaceAfter(_code.AbstractOption(entity)),
        _code.Escape(entity),
        _code.StringBefore(" : ", _typeMapper.GetTypeName(entity.BaseType)),
        stringsToMatch.Any(o => entity.Properties.Any(n => n.Name == o.Key)) ? " : " +
string.Join(", ", stringsToMatch.Join(entity.Properties, l => l.Key, r => r.Name, (l,r) =>
l.Value)) : string.Empty);
}
```

Este es un caso específico, pero muestra la capacidad de poder modificar plantillas `.tt` .

Adición de documentación XML a las clases de entidad

En todas las clases de modelo generadas no hay comentarios de documentación agregados por defecto. Si desea utilizar comentarios de la documentación XML para cada clases de entidad generados, se encontró dentro de esta parte .tt *[ModelName] (ModelName* es el nombre del archivo EDMX actual):

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code); // used to write model namespace
#>
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
```

Puede agregar los comentarios de la documentación XML antes de `UsingDirectives` línea `UsingDirectives` como se muestra en el siguiente ejemplo:

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code);
#>
/// <summary>
/// <#=entity.Name#> model entity class.
/// </summary>
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
```

El comentario de documentación generado debe incluir el nombre de la entidad como se indica a continuación.

```
/// <summary>
/// Example model entity class.
/// </summary>
public partial class Example
{
    // model contents
}
```

Lea Plantillas .t4 en Entidad-marco en línea: https://riptutorial.com/es/entity-framework/topic/3964/plantillas--t4-en-entidad-marco

Capítulo 19: Relación de mapeo con Entity Framework Code First: One-to-many y Many-to-many

Introducción

El tema trata sobre cómo puede mapear relaciones de uno a muchos y de muchos a muchos usando el Código Entity Framework Primero.

Examples

Mapeo uno a muchos

Así que digamos que tienes dos entidades diferentes, algo como esto:

```
public class Person
{
   public int PersonId { get; set; }
   public string Name { get; set; }
}

public class Car
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
}

public class MyDemoContext : DbContext
{
   public DbSet<Person> People { get; set; }
   public DbSet<Car> Cars { get; set; }
}
```

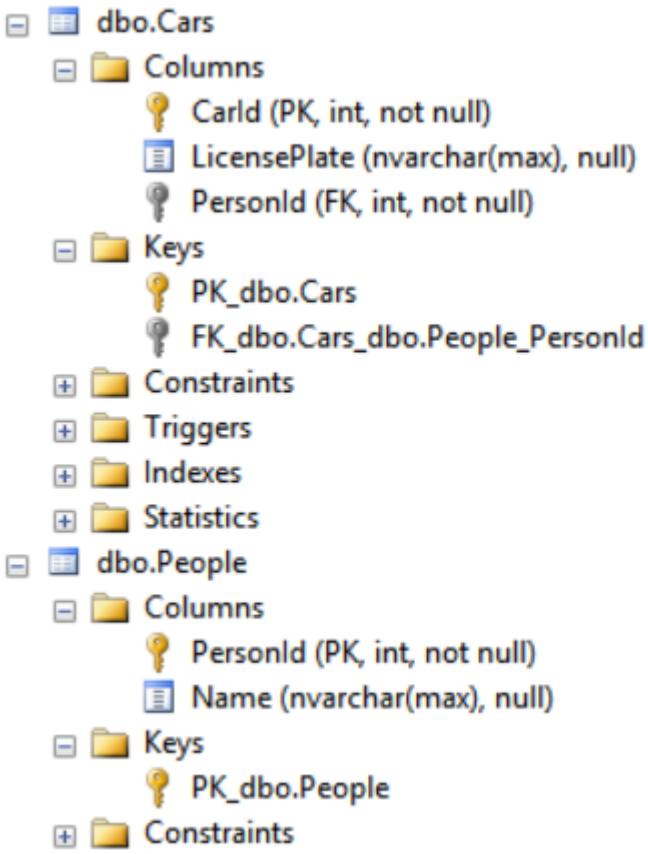
Y desea configurar una relación de uno a varios entre ellos, es decir, una persona puede tener cero, uno o más automóviles, y un automóvil pertenece a una sola persona exactamente. Toda relación es bidireccional, por lo que si una persona tiene un automóvil, el automóvil pertenece a esa persona.

Para hacer esto solo modifica tus clases modelo:

```
public class Person
{
   public int PersonId { get; set; }
   public string Name { get; set; }
   public virtual ICollection<Car> Cars { get; set; } // don't forget to initialize (use
HashSet)
}
```

```
public class Car
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
   public int PersonId { get; set; }
   public virtual Person Person { get; set; }
}
```

Y eso es :) Ya tienes tu relación configurada. En la base de datos, esto se representa con claves externas, por supuesto.



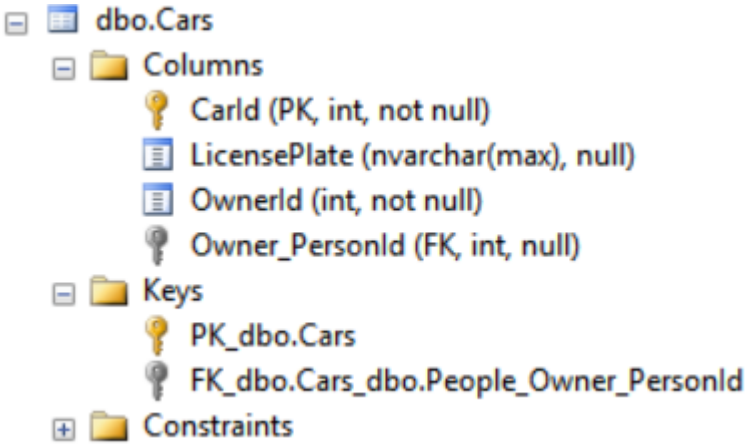
Mapeo de uno a muchos: contra la convención

En el último ejemplo, puede ver que EF determina qué columna es la clave externa y hacia dónde debe apuntar. ¿Cómo? Mediante el uso de convenciones. Tener una propiedad de tipo `Person` que se denomina `Person` con una propiedad `PersonId` lleva a EF a concluir que `PersonId` es una clave externa y apunta a la clave principal de la tabla representada por el tipo `Person` .

Pero, ¿qué *sucedería* si cambiara *PersonId* a *OwnerId* y *Person* a *Owner* en el tipo de **automóvil** ?

```
public class Car
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
   public int OwnerId { get; set; }
   public virtual Person Owner { get; set; }
}
```

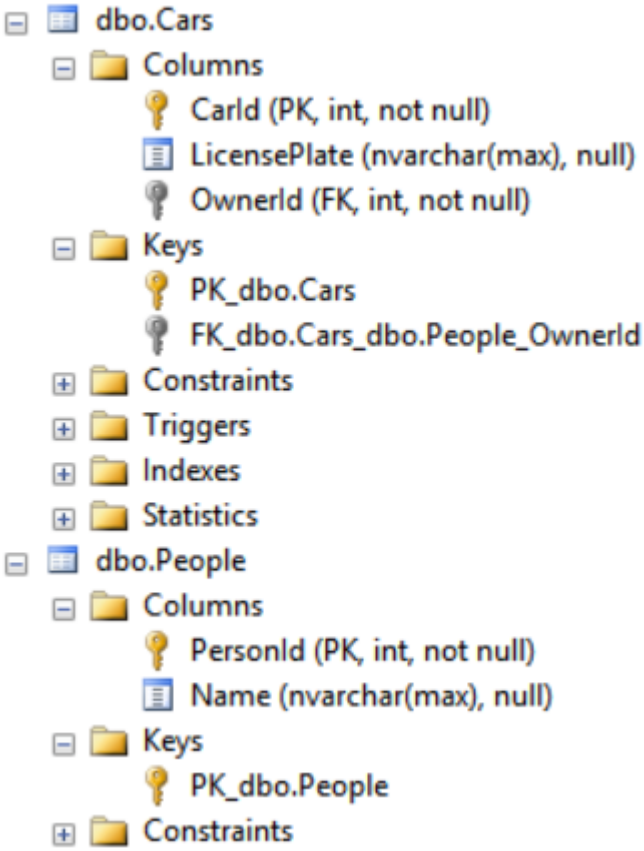
Bueno, desafortunadamente en este caso, las convenciones no son suficientes para producir el esquema de base de datos correcto:



Sin preocupaciones; puede ayudar a EF con algunos consejos sobre sus relaciones y claves en el modelo. Simplemente configure su tipo de `Car` para usar la propiedad `OwnerId` como FK. Cree una configuración de tipo de entidad y aplíquela en su `OnModelCreating()` :

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasRequired(c => c.Owner).WithMany(p => p.Cars).HasForeignKey(c => c.OwnerId);
  }
}
```

Básicamente, esto dice que el `Car` tiene una propiedad requerida, el `Owner` (*HasRequired ()*) y en el tipo de `Owner` , la propiedad `Cars` se usa para referirse a las entidades del automóvil (*WithMany ()*). Y finalmente se especifica la propiedad que representa la clave externa (*HasForeignKey ()*). Esto nos da el esquema que queremos:



También puedes configurar la relación desde el lado de la `Person` :

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
  public PersonEntityTypeConfiguration()
  {
    this.HasMany(p => p.Cars).WithRequired(c => c.Owner).HasForeignKey(c => c.OwnerId);
  }
}
```

La idea es la misma, solo los lados son diferentes (tenga en cuenta cómo puede leerlo todo: 'esta persona tiene muchos autos, cada uno con un propietario requerido'). No importa si configura la relación desde el lado `Person` o el lado `Car` . Incluso puede incluir ambos, pero en este caso, ¡tenga cuidado de especificar la misma relación en ambos lados!

Mapeo cero o uno a muchos

En los ejemplos anteriores, un coche no puede existir sin una persona. ¿Qué pasaría si quisiera que la persona fuera opcional desde el lado del automóvil? Bueno, es algo fácil, saber cómo hacer uno a muchos. Solo cambia el `PersonId` en `Car` para que sea nullable:

```
public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public int? PersonId { get; set; }
    public virtual Person Person { get; set; }
}
```

Y luego use *HasOptional ()* (o *WithOptional ()* , dependiendo de qué lado hace la configuración):

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasOptional(c => c.Owner).WithMany(p => p.Cars).HasForeignKey(c => c.OwnerId);
  }
}
```

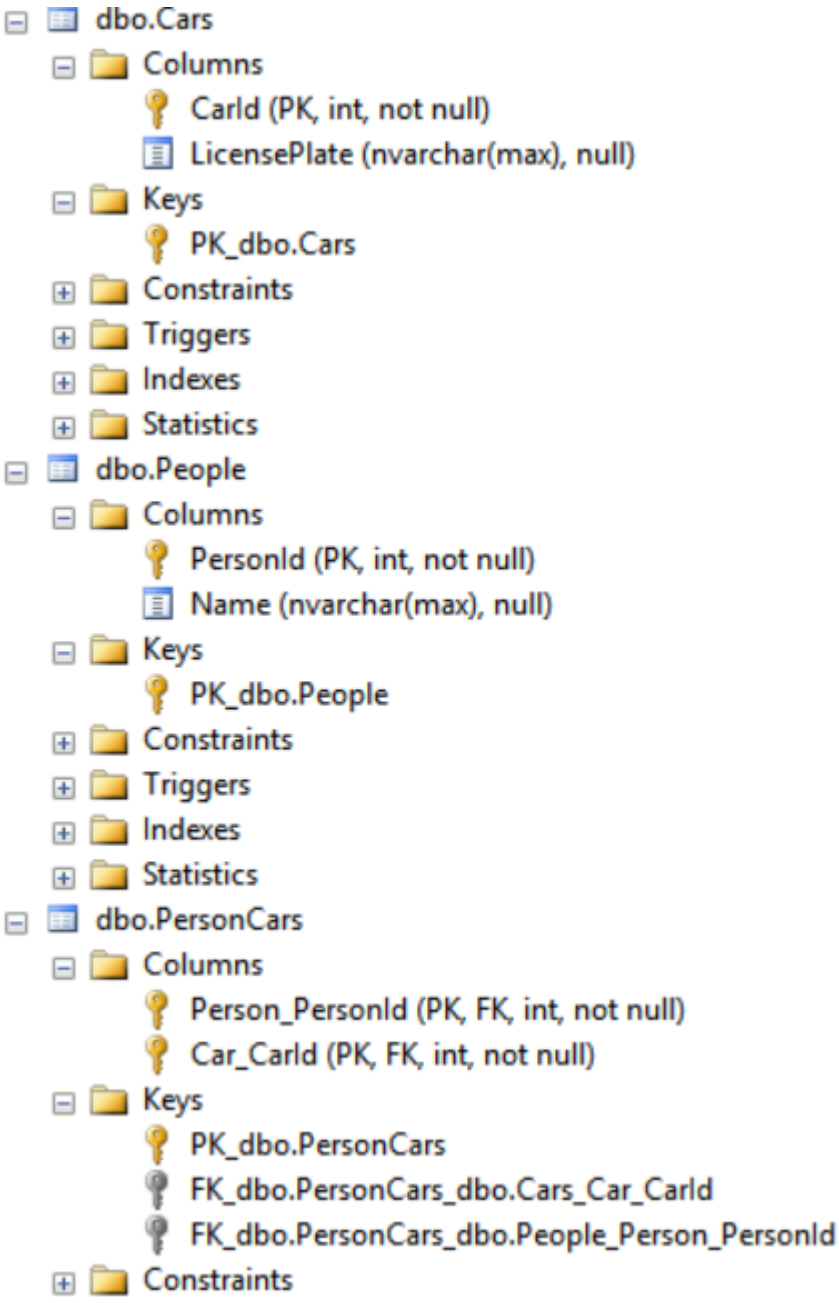
Muchos a muchos

Vayamos al otro escenario, donde cada persona puede tener múltiples autos y cada automóvil puede tener múltiples dueños (pero, nuevamente, la relación es bidireccional). Esta es una relación de muchos a muchos. La forma más fácil es dejar que EF haga su magia usando convenciones.

Solo cambia el modelo de esta manera:

```
 public class Person
{
   public int PersonId { get; set; }
   public string Name { get; set; }
   public virtual ICollection<Car> Cars { get; set; }
}

public class Car
{
   public int CarId { get; set; }
   public string LicensePlate { get; set; }
   public virtual ICollection<Person> Owners { get; set; }
}
```

Y el esquema:



Casi perfecto. Como puede ver, EF reconoció la necesidad de una tabla de unión, donde pueda realizar un seguimiento de los emparejamientos persona-automóvil.

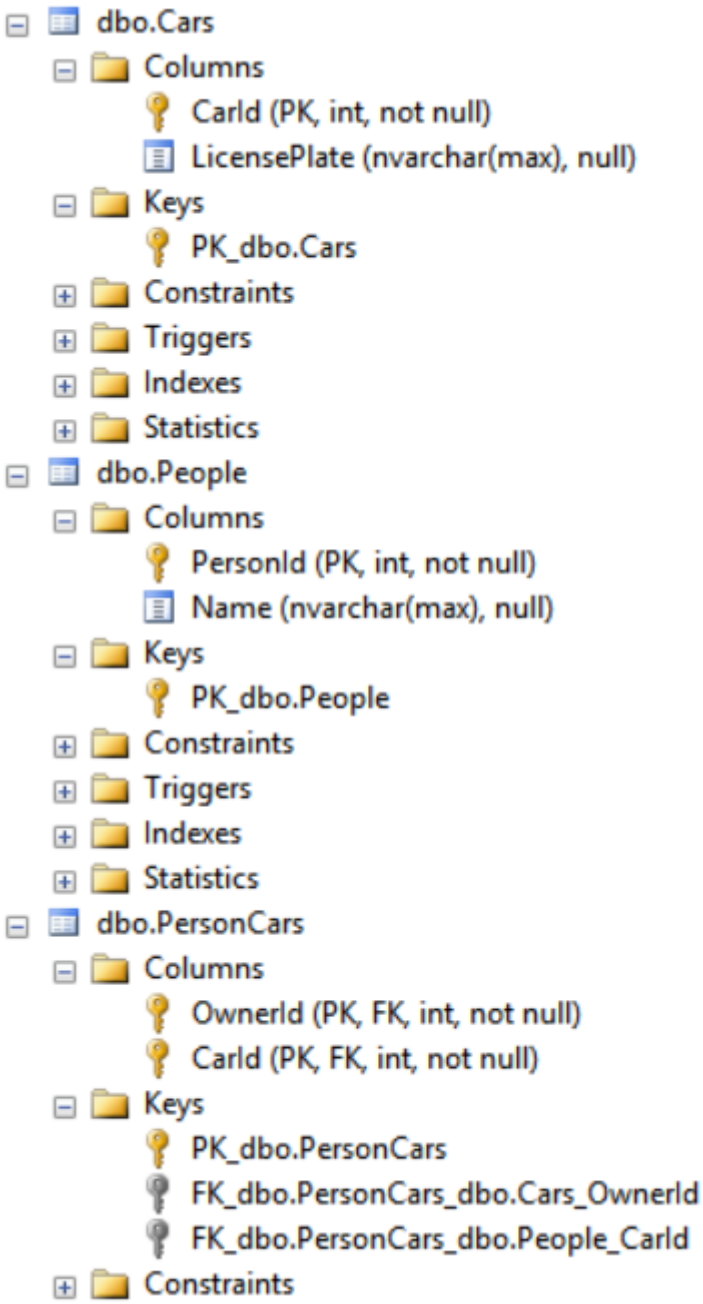
Muchos a muchos: personalizando la tabla de unión

Es posible que desee cambiar el nombre de los campos en la tabla de unión para que sea un poco más amigable. Puede hacerlo utilizando los métodos de configuración habituales (de nuevo, no importa de qué lado realice la configuración):

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
   public CarEntityTypeConfiguration()
   {
      this.HasMany(c => c.Owners).WithMany(p => p.Cars)
          .Map(m =>
              {
                 m.MapLeftKey("OwnerId");
                 m.MapRightKey("CarId");
```

```
                m.ToTable("PersonCars");
            }
        );
    }
}
```

Incluso muy fácil de leer: este automóvil tiene muchos propietarios (*HasMany ()*), y cada propietario tiene muchos automóviles (*WithMany ()*). Asigne esto de modo que asigne la clave izquierda a OwnerId (*MapLeftKey ()*), la clave derecha a CarId (*MapRightKey ()*) y todo a la tabla PersonCars (*ToTable ()*). Y esto te da exactamente ese esquema:



Many-to-many: entidad de unión personalizada

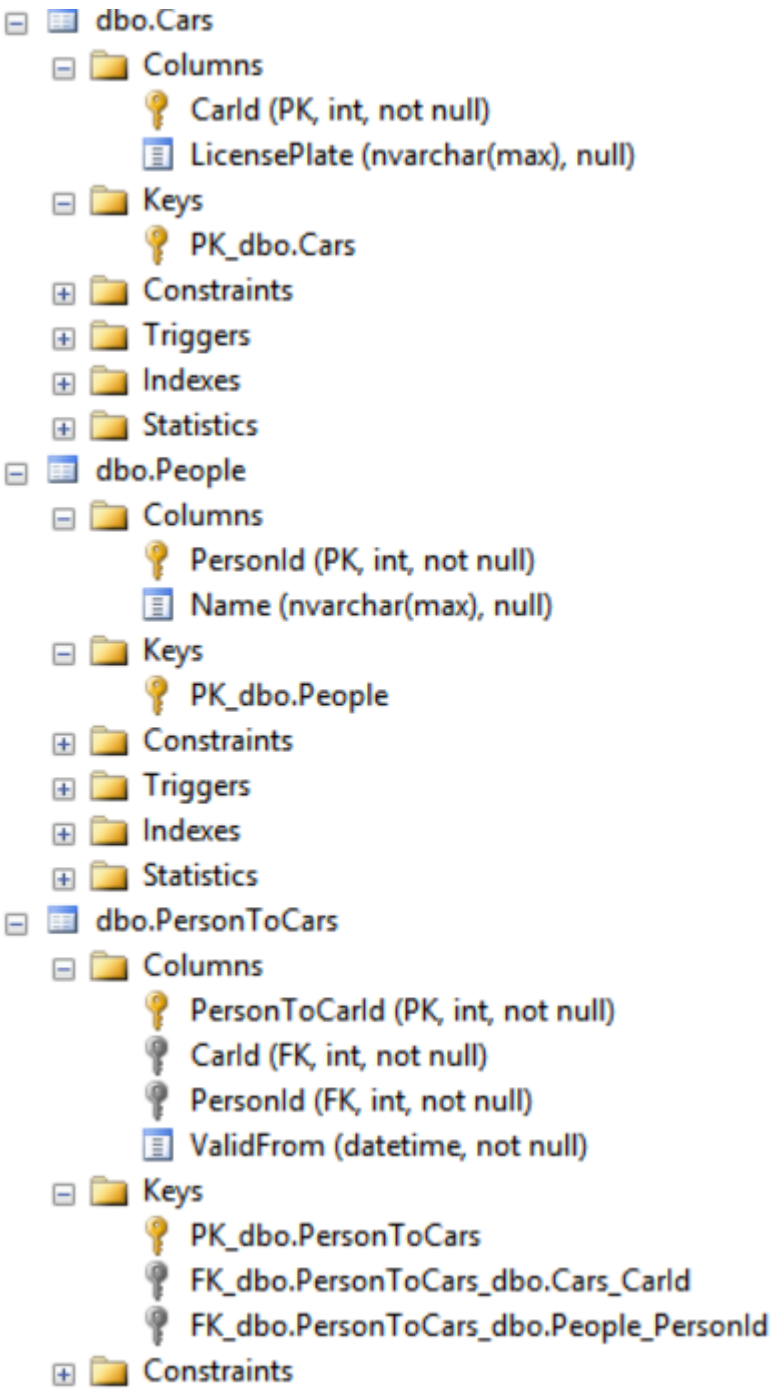
Debo admitir que no soy realmente un fanático de permitir que EF infiera la tabla de unión sin una entidad de unión. No puede rastrear información adicional a una asociación persona-automóvil (digamos la fecha a partir de la cual es válida), porque no puede modificar la tabla.

Además, el `CarId` en la tabla de unión es parte de la clave principal, por lo que si la familia compra un auto nuevo, primero debe eliminar las asociaciones antiguas y agregar otras nuevas. EF oculta esto de usted, pero esto significa que tiene que hacer estas dos operaciones en lugar de una simple actualización (sin mencionar que las inserciones / eliminaciones frecuentes pueden llevar a la fragmentación del índice, lo que es una solución fácil para eso).

En este caso, lo que puede hacer es crear una entidad de unión que tenga una referencia tanto a un automóvil específico como a una persona específica. Básicamente, usted ve su asociación de muchos a muchos como una combinación de dos asociaciones de uno a muchos:

```
public class PersonToCar
{
   public int PersonToCarId { get; set; }
   public int CarId { get; set; }
   public virtual Car Car { get; set; }
   public int PersonId { get; set; }
   public virtual Person Person { get; set; }
   public DateTime ValidFrom { get; set; }
}

public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public virtual ICollection<PersonToCar> CarOwnerShips { get; set; }
}

public class Car
{
  public int CarId { get; set; }
  public string LicensePlate { get; set; }
  public virtual ICollection<PersonToCar> Ownerships { get; set; }
}

public class MyDemoContext : DbContext
{
  public DbSet<Person> People { get; set; }
  public DbSet<Car> Cars { get; set; }
  public DbSet<PersonToCar> PersonToCars { get; set; }
}
```

Esto me da mucho más control y es mucho más flexible. Ahora puedo agregar datos personalizados a la asociación y cada asociación tiene su propia clave principal, por lo que puedo actualizar el auto o la referencia del propietario en ellos.



Tenga en cuenta que esto es solo una combinación de dos relaciones de uno a varios, por lo que puede usar todas las opciones de configuración que se analizaron en los ejemplos anteriores.

Lea Relación de mapeo con Entity Framework Code First: One-to-many y Many-to-many en línea: https://riptutorial.com/es/entity-framework/topic/9413/relacion-de-mapeo-con-entity-framework-code-first--one-to-many-y-many-to-many

Capítulo 20: Relación de mapeo con Entity Framework Code First: One-to-one y variaciones

Introducción

Este tema explica cómo asignar relaciones de tipo uno a uno mediante Entity Framework.

Examples

Mapeo de uno a cero o uno

Así que digamos de nuevo que tienes el siguiente modelo:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
}

public class Car
{
  public int CarId { get; set; }
  public string LicensePlate { get; set; }
}

public class MyDemoContext : DbContext
{
  public DbSet<Person> People { get; set; }
  public DbSet<Car> Cars { get; set; }
}
```

Y ahora desea configurarlo para que pueda expresar la siguiente especificación: una persona puede tener uno o cero automóviles, y cada automóvil pertenece exactamente a una persona (las relaciones son bidireccionales, por lo tanto, si CarA pertenece a PersonA, entonces PersonA posee 'CarA).

Así que modifiquemos un poco el modelo: agregue las propiedades de navegación y las propiedades de clave externa:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public int CarId { get; set; }
  public virtual Car Car { get; set; }
}
```

```
public class Car
{
  public int CarId { get; set; }
  public string LicensePlate { get; set; }
  public int PersonId { get; set; }
  public virtual Person Person { get; set; }
}
```

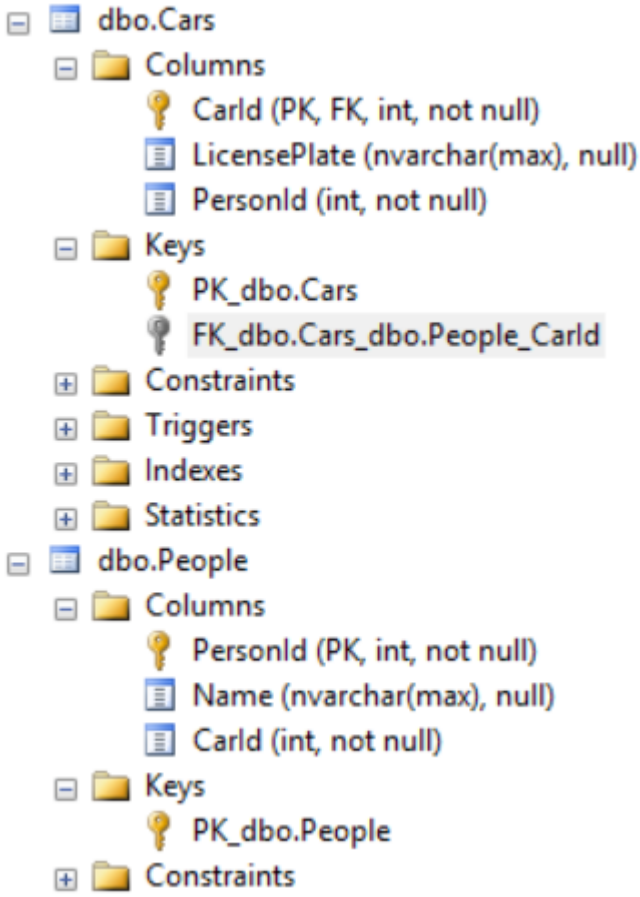
Y la configuración:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasRequired(c => c.Person).WithOptional(p => p.Car);
  }
}
```

En este momento esto debería ser autoexplicativo. El auto tiene una persona requerida (*HasRequired ()*), y la persona tiene un auto opcional (*WithOptional ()*). Nuevamente, no importa de qué lado configures esta relación, solo ten cuidado cuando uses la combinación correcta de Has / With y Required / Optional. Desde el lado de la `Person` , se vería así:

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
  public PersonEntityTypeConfiguration()
  {
     this.HasOptional(p => p.Car).WithOptional(c => c.Person);
  }
}
```

Ahora vamos a ver el esquema de db:



Mire detenidamente: puede ver que no hay FK en `People` para referirse a `Car` . Además, el FK en `Car` no es el `PersonId` , sino el `CarId` . Aquí está el script real para el FK:

```
ALTER TABLE [dbo].[Cars]  WITH CHECK ADD  CONSTRAINT [FK_dbo.Cars_dbo.People_CarId] FOREIGN
KEY([CarId])
REFERENCES [dbo].[People] ([PersonId])
```

Por lo tanto, esto significa que las propiedades clave de `CarId` y `PersonId` foregn que tenemos en el modelo se ignoran básicamente. Están en la base de datos, pero no son claves externas, como podría esperarse. Esto se debe a que las asignaciones uno a uno no admiten agregar el FK a su modelo EF. Y eso se debe a que las asignaciones uno a uno son bastante problemáticas en una base de datos relacional.

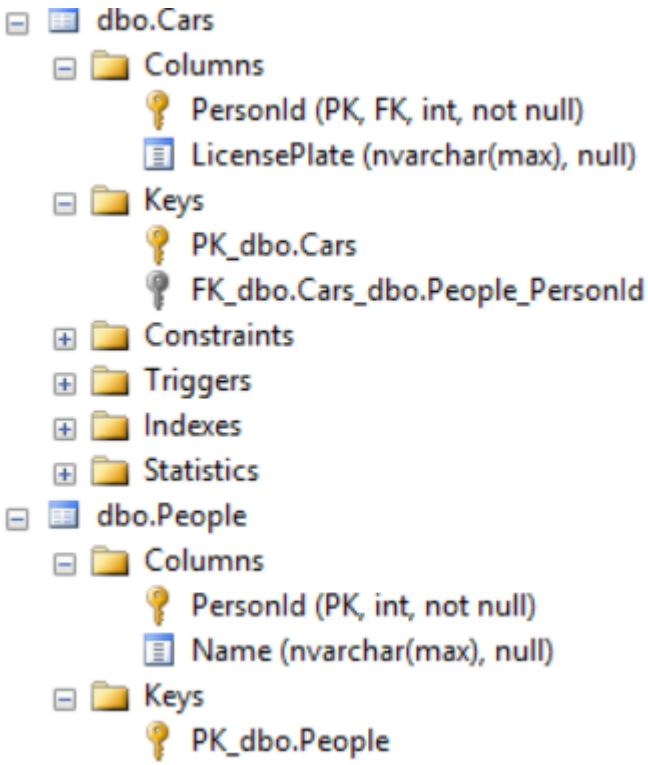
La idea es que cada persona puede tener exactamente un automóvil, y ese automóvil solo puede pertenecer a esa persona. O puede haber registros personales, que no tienen automóviles asociados con ellos.

Entonces, ¿cómo podría ser representado con claves externas? Obviamente, podría haber un `PersonId` en el `Car` y un `CarId` en las `People` . Para imponer que cada persona pueda tener solo un auto, `PersonId` tendría que ser único en `Car` . Pero si `PersonId` es único en `People` , ¿cómo puede agregar dos o más registros donde `PersonId` es `NULL` (más de un auto que no tiene dueño)? Respuesta: no puede (bueno, en realidad, puede crear un índice único filtrado en SQL Server 2008 y más reciente, pero olvidemos este tecnicismo por un momento; por no mencionar otros RDBMS). Sin mencionar el caso donde se especifican ambos extremos de la relación ...

La única forma real de hacer cumplir esta regla si las tablas de `People` y `Car` tienen la clave principal 'misma' (los mismos valores en los registros conectados). Y para hacer esto, `CarId` in `Car` debe ser tanto PK como FK para PK de People. Y esto hace que todo el esquema sea un desastre. Cuando uso esto prefiero nombrar el PK / FK en `Car PersonId` , y lo configuro en consecuencia:

```
public class Person
{
  public int PersonId { get; set; }
  public string Name { get; set; }
  public virtual Car Car { get; set; }
}

public class Car
{
  public string LicensePlate { get; set; }
  public int PersonId { get; set; }
  public virtual Person Person { get; set; }
}

public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
     this.HasRequired(c => c.Person).WithOptional(p => p.Car);
     this.HasKey(c => c.PersonId);
  }
}
```

No ideal, pero tal vez un poco mejor. Aún así, debes estar alerta cuando utilices esta solución, porque va en contra de las convenciones de nombres habituales, lo que podría desviarte. Aquí está el esquema generado a partir de este modelo:



Entonces, esta relación no es impuesta por el esquema de la base de datos, sino por Entity Framework. Es por eso que debes tener mucho cuidado al usar esto, no permitir que nadie se enoje directamente con la base de datos.

Mapeo uno a uno

Mapear uno a uno (cuando se requieren ambos lados) también es algo complicado.

Imaginemos cómo podría representarse esto con claves foráneas. De nuevo, un `CarId` in `People` que se refiere a `CarId` in `Car` , y un `PersonId` in Car que se refiere al `PersonId` in `People` .

Ahora, ¿qué pasa si quieres insertar un registro de coche? Para que esto tenga éxito, debe haber un `PersonId` especificado en este registro de auto, porque es necesario. Y para que este `PersonId` sea válido, debe existir el registro correspondiente en `People` . OK, entonces sigamos adelante e insertemos el registro de persona. Pero para que esto `CarId` éxito, un `CarId` válido debe estar en el registro de la persona, ¡pero el auto aún no está insertado! No puede ser, porque primero tenemos que insertar el registro de la persona referida. Pero no podemos insertar el registro de la persona referida, porque se refiere al registro del automóvil, por lo que debe insertarse primero (clave-clave extranjera :)).

Entonces, esto tampoco puede ser representado de la manera 'lógica'. De nuevo, tienes que soltar una de las claves foráneas. La que dejes caer depende de ti. El lado que queda con una clave externa se llama "dependiente", el lado que queda sin una clave externa se llama "principal". Y nuevamente, para garantizar la singularidad en el dependiente, el PK debe ser el FK, por lo que no se admite la adición de una columna FK ni la importación a su modelo.

Así que aquí está la configuración:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
    this.HasRequired(c => c.Person).WithRequiredDependent(p => p.Car);
    this.HasKey(c => c.PersonId);
  }
}
```

En este momento, realmente debería haber obtenido la lógica de la misma :) Solo recuerde que también puede elegir el otro lado, solo tenga cuidado de usar las versiones Dependiente / Principal de WithRequired (y todavía tiene que configurar la PK en el automóvil).

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
  public PersonEntityTypeConfiguration()
  {
    this.HasRequired(p => p.Car).WithRequiredPrincipal(c => c.Person);
  }
}
```

Si verifica el esquema de DB, encontrará que es exactamente igual que en el caso de la solución uno a uno o cero. Eso es porque, de nuevo, esto no se aplica por el esquema, sino por el propio

EF. Así que de nuevo, ten cuidado :)

Mapeo de uno o cero a uno o cero

Y para terminar, veamos brevemente el caso cuando ambos lados son opcionales.

A estas alturas, debería estar realmente aburrido de estos ejemplos :), así que no voy a entrar en detalles y jugar con la idea de tener dos FK-s y los posibles problemas y le advierto sobre los peligros de no aplicar estas reglas en el esquema pero solo en EF mismo.

Aquí está la configuración que necesita aplicar:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
    this.HasOptional(c => c.Person).WithOptionalPrincipal(p => p.Car);
    this.HasKey(c => c.PersonId);
  }
}
```

De nuevo, también puede configurar desde el otro lado, solo tenga cuidado de usar los métodos correctos :)

Lea Relación de mapeo con Entity Framework Code First: One-to-one y variaciones en línea: https://riptutorial.com/es/entity-framework/topic/9412/relacion-de-mapeo-con-entity-framework-code-first--one-to-one-y-variaciones

Capítulo 21: Seguimiento vs No-seguimiento

Observaciones

El comportamiento de seguimiento controla si Entity Framework mantendrá información sobre una instancia de entidad en su rastreador de cambios. Si se realiza un seguimiento de una entidad, todos los cambios detectados en la entidad se conservarán en la base de datos durante `SaveChanges()` .

Examples

Consultas de seguimiento

- De forma predeterminada, las consultas que devuelven tipos de entidad están **siguiendo**
- Esto significa que puede realizar cambios en esas instancias de entidad y que `SaveChanges()` esos cambios `SaveChanges()`

Ejemplo:

- El cambio en la calificación del `book` se detectará y se conservará en la base de datos durante `SaveChanges()` .

```
using (var context = new BookContext())
{
  var book = context.Books.FirstOrDefault(b => b.BookId == 1);
  book.Rating = 5;
  context.SaveChanges();
}
```

Consultas sin seguimiento

- Ninguna consulta de seguimiento es útil cuando los resultados se utilizan en un escenario de `read-only`
- Son `quicker to execute` porque no es necesario configurar la información de seguimiento de cambios

Ejemplo:

```
using (var context = new BookContext())
{
    var books = context.Books.AsNoTracking().ToList();
}
```

Con EF Core 1.0, también puede cambiar el comportamiento de seguimiento predeterminado en el nivel de `context instance` .

Ejemplo:

```
using (var context = new BookContext())
{
    context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

    var books = context.Books.ToList();
}
```

Seguimiento y proyecciones.

- Incluso si el tipo de resultado de la consulta no es un tipo de entidad, si el resultado `contains entity` tipos de `contains entity` , todavía se `tracked by default`

Ejemplo:

- En la siguiente consulta, que devuelve un `anonymous type` , `will be tracked` las instancias de `Book` en el conjunto de resultados

```
using (var context = new BookContext())
{
   var book = context.Books.Select(b => new { Book = b, Authors = b.Authors.Count() });
}
```

- Si el conjunto de resultados `does not` contiene ningún tipo de `entity` , `no tracking` se realiza `no tracking`

Ejemplo:

- En la siguiente consulta, que devuelve un `anonymous type` con algunos de los valores de la entidad (pero `no instances` del tipo de `entity` real), no se realiza **ningún seguimiento** .

```
using (var context = new BookContext())
{
   var book = context.Books.Select(b => new { Id = b.BookId, PublishedDate = b.Date });
}
```

Lea Seguimiento vs No-seguimiento en línea: https://riptutorial.com/es/entity-framework/topic/6836/seguimiento-vs-no-seguimiento

Capítulo 22: Técnicas de optimización en EF

Examples

Usando AsNoTracking

Mal ejemplo:

```
var location =  dbContext.Location
                     .Where(l => l.Location.ID == location_ID)
                     .SingleOrDefault();

return location;
```

Dado que el código anterior simplemente devuelve una entidad sin modificarla ni agregarla, podemos evitar el seguimiento del costo.

Buen ejemplo:

```
var location =  dbContext.Location.AsNoTracking()
                     .Where(l => l.Location.ID == location_ID)
                     .SingleOrDefault();

return location;
```

Cuando usamos la función `AsNoTracking()` le estamos diciendo explícitamente a Entity Framework que el contexto no rastrea a las entidades. Esto puede ser especialmente útil cuando recupera grandes cantidades de datos de su almacén de datos. Sin embargo, si desea realizar cambios en las entidades sin seguimiento, debe recordar adjuntarlos antes de llamar a `SaveChanges` .

Cargando solo los datos requeridos

Un problema que se ve a menudo en el código es cargar todos los datos. Esto aumentará en gran medida la carga en el servidor.

Digamos que tengo un modelo llamado "ubicación" que tiene 10 campos, pero no todos los campos son obligatorios al mismo tiempo. Digamos que solo quiero el parámetro 'LocationName' de ese modelo.

Mal ejemplo

```
var location =  dbContext.Location.AsNoTracking()
                        .Where(l => l.Location.ID == location_ID)
                        .SingleOrDefault();

return location.Name;
```

Buen ejemplo

```
var location =  dbContext.Location
                          .Where(l => l.Location.ID == location_ID)
                          .Select(l => l.LocationName);
                          .SingleOrDefault();

return location;
```

El código en el "buen ejemplo" solo buscará 'LocationName' y nada más.

Tenga en cuenta que como no se materializa ninguna entidad en este ejemplo, `AsNoTracking()` no es necesario. No hay nada que rastrear de todos modos.

Obteniendo más campos con tipos anónimos

```
var location = dbContext.Location
                    .Where(l => l.Location.ID == location_ID)
                    .Select(l => new { Name = l.LocationName, Area = l.LocationArea })
                    .SingleOrDefault();

return location.Name + " has an area of " + location.Area;
```

Igual que en el ejemplo anterior, solo los campos 'LocationName' y 'LocationArea' se recuperarán de la base de datos, el Tipo Anónimo puede contener tantos valores que desee.

Ejecute las consultas en la base de datos cuando sea posible, no en la memoria.

Supongamos que queremos contar cuántos condados hay en Texas:

```
var counties = dbContext.States.Single(s => s.Code == "tx").Counties.Count();
```

La consulta es correcta, pero ineficiente. `States.Single(…)` carga un estado desde la base de datos. A continuación, los `Counties` cargan los 254 condados con todos sus campos en una segunda consulta. `.Count()` luego se realiza *en memoria* en la colección de `Counties` cargados. Hemos cargado una gran cantidad de datos que no necesitamos, y podemos hacerlo mejor:

```
var counties = dbContext.Counties.Count(c => c.State.Code == "tx");
```

Aquí solo hacemos una consulta, que en SQL se traduce en un recuento y una combinación. Solo devolvemos el recuento de la base de datos; hemos guardado filas, campos y creación de objetos.

Es fácil ver dónde se realiza la consulta mirando el tipo de colección: `IQueryable<T>` frente a `IEnumerable<T>`.

Ejecutar múltiples consultas asíncronas y en paralelo.

Al utilizar consultas asíncronas, puede ejecutar varias consultas al mismo tiempo, pero no en el mismo contexto. Si el tiempo de ejecución de una consulta es 10s, el tiempo para el mal ejemplo

será 20s, mientras que el tiempo para el buen ejemplo será 10s.

Mal ejemplo

```
IEnumerable<TResult1> result1;
IEnumerable<TResult2> result2;

using(var context = new Context())
{
    result1 = await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
    result2 = await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
}
```

Buen ejemplo

```
public async Task<IEnumerable<TResult>> GetResult<TResult>()
{
    using(var context = new Context())
    {
        return await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
    }
}



IEnumerable<TResult1> result1;
IEnumerable<TResult2> result2;

var result1Task = GetResult<TResult1>();
var result2Task = GetResult<TResult2>();

await Task.WhenAll(result1Task, result2Task).ConfigureAwait(false);

var result1 = result1Task.Result;
var result2 = result2Task.Result;
```

Deshabilitar el seguimiento de cambios y la generación de proxy

Si solo desea obtener datos, pero no modificar nada, puede desactivar el seguimiento de cambios y la creación de proxy. Esto mejorará su rendimiento y también evitará la carga perezosa.

Mal ejemplo:

```
using(var context = new Context())
{
    return await context.Set<MyEntity>().ToListAsync().ConfigureAwait(false);
}
```

Buen ejemplo:

```
using(var context = new Context())
```

```
{
    context.Configuration.AutoDetectChangesEnabled = false;
    context.Configuration.ProxyCreationEnabled = false;

    return await context.Set<MyEntity>().ToListAsync().ConfigureAwait(false);
}
```

Es particularmente común desactivar esto dentro del constructor de su contexto, especialmente si desea que estos se configuren en su solución:

```
public class MyContext : DbContext
{
    public MyContext()
        : base("MyContext")
    {
        Configuration.AutoDetectChangesEnabled = false;
        Configuration.ProxyCreationEnabled = false;
    }

    //snip
}
```

Trabajando con entidades de stub

Digamos que tenemos `Product` y `Category` en una relación de muchos a muchos:

```
public class Product
{
    public Product()
    {
        Categories = new HashSet<Category>();
    }
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public virtual ICollection<Category> Categories { get; private set; }
}

public class Category
{
    public Category()
    {
        Products = new HashSet<Product>();
    }
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public virtual ICollection<Product> Products { get; set; }
}
```

Si queremos agregar una `Category` a un `Product` , tenemos que cargar el producto y agregar la categoría a sus `Categories` , por ejemplo:

Mal ejemplo:

```
var product = db.Products.Find(1);
var category = db.Categories.Find(2);
```

```
product.Categories.Add(category);
db.SaveChanges();
```

(donde `db` es una subclase `DbContext`).

Esto crea un registro en la tabla de unión entre `Product` y `Category` . Sin embargo, esta tabla solo contiene dos valores de `Id` . Es una pérdida de recursos cargar dos entidades completas para crear un registro pequeño.

Una forma más eficiente es usar *entidades de código auxiliar* , es decir, objetos de entidad, creados en la memoria, que contienen solo el mínimo de datos, generalmente solo un valor de `Id` . Esto es lo que parece:

Buen ejemplo:

```
// Create two stub entities
var product = new Product { ProductId = 1 };
var category = new Category { CategoryId = 2 };

// Attach the stub entities to the context
db.Entry(product).State = System.Data.Entity.EntityState.Unchanged;
db.Entry(category).State = System.Data.Entity.EntityState.Unchanged;

product.Categories.Add(category);
db.SaveChanges();
```

El resultado final es el mismo, pero evita dos viajes de ida y vuelta a la base de datos.

Prevenir duplicados

Si desea comprobar si la asociación ya existe, una consulta barata es suficiente. Por ejemplo:

```
var exists = db.Categories.Any(c => c.Id == 1 && c.Products.Any(p => p.Id == 14));
```

De nuevo, esto no cargará entidades completas en la memoria. Consulta eficazmente la tabla de unión y solo devuelve un valor booleano.

Lea Técnicas de optimización en EF en línea: https://riptutorial.com/es/entity-framework/topic/2714/tecnicas-de-optimizacion-en-ef

Capítulo 23: Tipos complejos

Examples

Codificar primero los tipos complejos

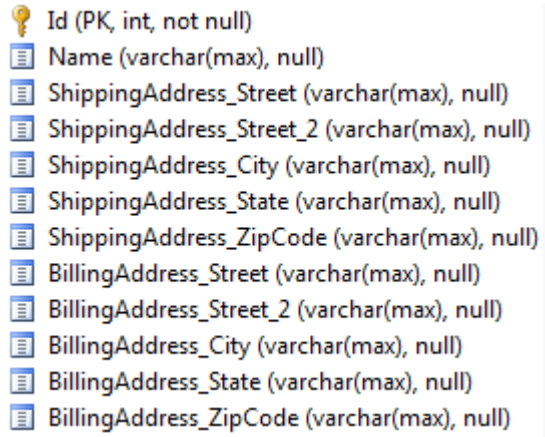
Un tipo complejo le permite asignar los campos seleccionados de una tabla de base de datos a un solo tipo que es un elemento secundario del tipo principal.

```
[ComplexType]
public class Address
{
    public string Street { get; set; }
    public string Street_2 { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string ZipCode { get; set; }
}
```

Este tipo complejo se puede usar en múltiples tipos de entidades. Incluso se puede utilizar más de una vez en el mismo tipo de entidad.

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    ...
    public Address ShippingAddress { get; set; }
    public Address BillingAddress { get; set; }
}
```

Este tipo de entidad se almacenaría en una tabla en la base de datos que se vería así.



Por supuesto, en este caso, una asociación 1: n (Dirección del cliente) sería el modelo preferido, pero el ejemplo muestra cómo se pueden usar los tipos complejos.

Lea Tipos complejos en línea: https://riptutorial.com/es/entity-framework/topic/5527/tipos-complejos

Creditos

S. No	Capítulos	Contributors
1	Empezando con Entity Framework	Adil Mammadov, Community, DavidG, Eldho, Jacob Linney, Martin4ndersen, Matas Vaitkevicius, Nasreddine, NovaDev, Parth Patel, Stephen Reindl, tmg
2	Actas	CptRobby, DavidG, Gert Arnold
3	Base de datos de primera generación de modelos	Matas Vaitkevicius, Tetsuya Yamamoto
4	Cargando entidades relacionadas	Adil Mammadov, Florian Haider, Gert Arnold, hasan, Joshit, Matas Vaitkevicius, tmg
5	Código de primeras anotaciones de datos	bubi, CptRobby, Daniel A. White, Daniel Lemke, DavidG, Diego, Gert Arnold, Jozef Lačný, Mark Shevchenko, Matas Vaitkevicius , Parth Patel, Piotrek, tmg, Tushar patel
6	Código de primeras convenciones	MacakM, Parth Patel, Sivanantham Padikkasu, Stephen Reindl, tmg
7	Código primero - API fluida	Adil Mammadov, Daniel Lemke, Jason Tyler, tmg
8	Entidad marco código primero	Balázs Nagy, Jozef Lačný
9	Entidad-marco de código primeras migraciones	CGritton, hasan, Joshit, Mostafa, RamenChef, Stephen Reindl
10	Entity Framework con SQLite	Jason Tyler
11	Entity-Framework con Postgresql	skj123
12	Escenarios de mapeo avanzados: división de entidades, división de tablas	Akos Nagy

13	Estado de la entidad gestora	Gert Arnold
14	Herencia con EntityFramework (primero el código)	lucavgobbi
15	Inicializadores de bases de datos	Jozef Lačný
16	Mejores Prácticas para Entity Framework (Simple y Profesional)	Braiam, Mina Matta
17	Modelo de restricciones	SOfanatic, Tushar patel
18	Plantillas .t4 en Entidad-marco	Matas Vaitkevicius, Tetsuya Yamamoto
19	Relación de mapeo con Entity Framework Code First: One-to-many y Many-to-many	Akos Nagy
20	Relación de mapeo con Entity Framework Code First: One-to-one y variaciones	Akos Nagy
21	Seguimiento vs No-seguimiento	hasan, Sampath, Stephen Reindl, tmg
22	Técnicas de optimización en EF	Amit Shahani, Anshul Nigam, DavidG, Gert Arnold, Jacob Linney, Kobi, lucavgobbi, Stephen Reindl, tmg, wertzui
23	Tipos complejos	CptRobby, Gert Arnold