Herencia en Code First

POO

has a

is a

SQL

has a

Table per Hierarchy (TPH)

1 tabla para toda la jerarquía con un campo discriminador por tipo

Estrategia de herencia por defecto

Table per Type (TPT)

1 tabla por clase concreta

Table per Concrete class (TPC)

1 tabla por clase concreta, pero ninguna para la clase abstracta

# TPH

public DbSet<Payment> Payments { get; set; }

class CreditCardPayment : Payment

{

public string Number { get; set; }

}

class PayPalPayment : Payment

{

public string Email { get; set; }

}

internal class Order

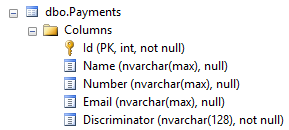
{

// ...

public int PaymentId { get; set; }

public virtual Payment Payment { get; set; }

}



Discriminator es el nombre por defecto y el valor es el nombre de la clase

Con este mapeo se puede cambiar el nombre del campo y el valor

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

const string discriminator = "MyDiscriminator";

modelBuilder.Entity<Payment>()

.Map<CreditCardPayment>(configuration => configuration.Requires(discriminator).HasValue("CCP"))

.Map<PayPalPayment>(configuration => configuration.Requires(discriminator).HasValue("PPP"));

base.OnModelCreating(modelBuilder);

}

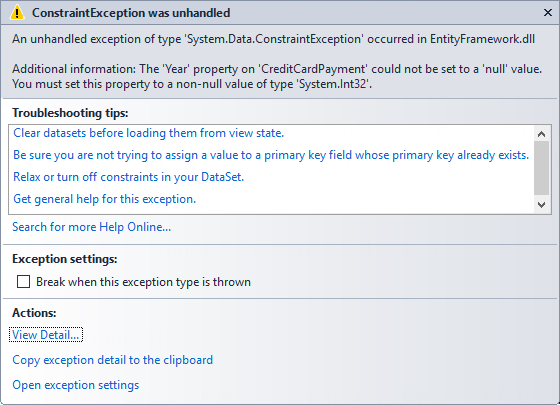
Incluso con HasValue (que acepta un object) se puede cambiar el tipo del campo en SQL:

modelBuilder.Entity<Payment>()

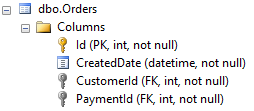
.Map<CreditCardPayment>(configuration => configuration.Requires(discriminator).**HasValue(1)**)

.Map<PayPalPayment>(configuration => configuration.Requires(discriminator).**HasValue(2)**);

Las propiedades de las subclases serán declaradas en BD como NULL (con independencia de si son nulables o no en el tipo), si metemos datos en BD podría pasar lo siguiente:



Es una buena opción en términos de simplicidad. Además, si hay que tirar SQL a mano no es muy complicado (sin JOIN). Sin embargo, desnormaliza y eso podría no gustarte.



LINQ por defecto hace consultas polimórficas (polymorphic queries) para recuperar instancias de una clase o instancias de todas sus subclases.

Por el contrario, con OfType<> podemos hacer consultas no polimórficas (Non-polymorphic queries)

using System;

using System.Data.Entity;

using System.Linq;

namespace DomainModel

{

class Program

{

static void Main(string[] args)

{

Database.SetInitializer(new DropCreateDatabaseAlways<ShopContext>());

using (var context = new ShopContext())

{

context.Database.Initialize(true);

context.Payments.Add(new CreditCardPayment() { Name = "Sergio", Number = "12345" });

context.Payments.Add(new PayPalPayment() { Name = "Sergio", Email = "panicoenlaxbox@gmail.com" });

context.SaveChanges();

context.Database.Log = Console.WriteLine;

IQueryable<Payment> payments = from p in context.Payments select p;

foreach (var payment in payments)

{

Console.WriteLine(payment.ToString());

}

IQueryable<CreditCardPayment> creditCards = from p in context.Payments.OfType<CreditCardPayment>() select p;

foreach (var creditCard in creditCards)

{

Console.WriteLine(creditCard.ToString());

}

}

}

}

}

La SQL es sencilla pero por el contrario nos traemos siempre todos los campos de todas las subclases (aunque usemos incluso OfType<>).

# TPT

Cada clase tiene su propia tabla.

Una clase para la abstracta con propiedades comunes.

Una clase por clase concreta con propiedades propias y una foreign key a la tabla con la clase padre.

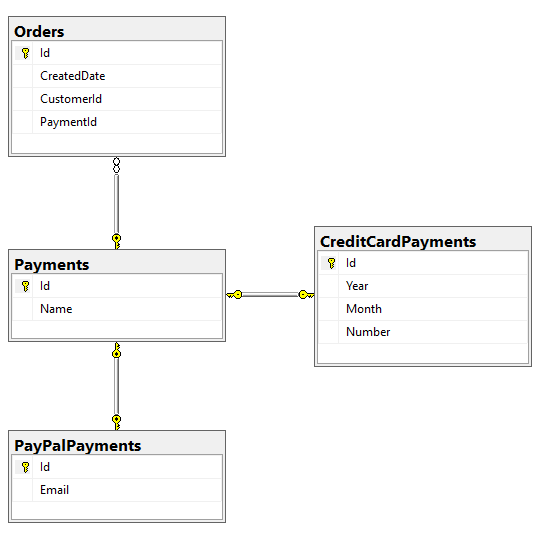
Ahora la base de datos está normalizada.

Las columnas en base de datos no tienen que ser nulables por defecto.

Lo único que hay que hacer en Code First es poner [Table] a una tabla distinta en cada clase derivada.

modelBuilder.Entity<CreditCardPayment>().ToTable("CreditCardPayments");

modelBuilder.Entity<PayPalPayment>().ToTable("PayPalPayments");



La SQL que tira EF es muy fea…

SELECT

CASE WHEN ([UnionAll1].[C2] = 1) THEN '0X0X' ELSE '0X1X' END AS [C1],

[UnionAll1].[Id] AS [C2],

[Extent3].[Name] AS [Name],

CASE WHEN ([UnionAll1].[C2] = 1) THEN [UnionAll1].[Year] END AS [C3],

CASE WHEN ([UnionAll1].[C2] = 1) THEN [UnionAll1].[Month] END AS [C4],

CASE WHEN ([UnionAll1].[C2] = 1) THEN [UnionAll1].[Number] END AS [C5],

CASE WHEN ([UnionAll1].[C2] = 1) THEN CAST(NULL AS varchar(1)) ELSE [UnionAll1].[C1] END AS [C6]

FROM (SELECT

[Extent1].[Id] AS [Id],

[Extent1].[Year] AS [Year],

[Extent1].[Month] AS [Month],

[Extent1].[Number] AS [Number],

CAST(NULL AS varchar(1)) AS [C1],

cast(1 as bit) AS [C2]

FROM [dbo].[CreditCardPayments] AS [Extent1]

UNION ALL

SELECT

[Extent2].[Id] AS [Id],

CAST(NULL AS int) AS [C1],

CAST(NULL AS int) AS [C2],

CAST(NULL AS varchar(1)) AS [C3],

[Extent2].[Email] AS [Email],

cast(0 as bit) AS [C4]

FROM [dbo].[PayPalPayments] AS [Extent2]) AS [UnionAll1]

INNER JOIN [dbo].[Payments] AS [Extent3] ON [UnionAll1].[Id] = [Extent3].[Id]

Como problemas está que la SQL es muy fea (y peor en rendimiento que TPH), que hacer SQL a mano se complica. Además, DML también se complica.

# TPC

La más simple de todas.

Una clase por cada clase concreta (incluyendo tanto sus propiedades propias como las heredadas).

En BD no hay relación entre estas tablas, ninguna.

//TPC

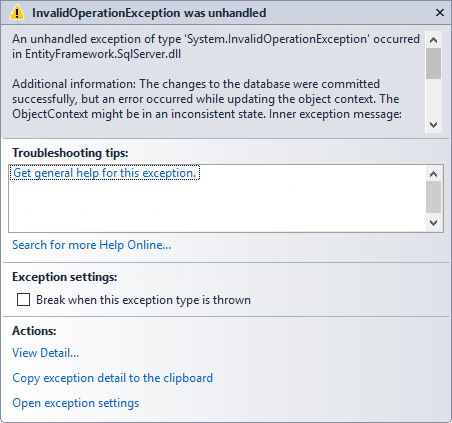
modelBuilder.Entity<Payment>()

.Map<CreditCardPayment>(

configuration => configuration.MapInheritedProperties().ToTable("CreditCardPayments"))

.Map<PayPalPayment>(

configuration => configuration.MapInheritedProperties().ToTable("PayPalPayments"));



Ha dado el mismo Id a 2 registros de tipo distinto, pero ambas heredando de la misma clase base (porque identity empieza en 0) y ChangeTracker no puede seguir la pista a 2 objetos del mismo tipo (la base) con la misma clave (cuando actualiza las claves temporales con las creó la bd, 2 unos).

modelBuilder.Entity<Payment>()

.Property(p => p.Id)

.HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);

modelBuilder.Entity<Payment>()

.Map<CreditCardPayment>(

configuration => configuration.MapInheritedProperties().ToTable("CreditCardPayments"))

.Map<PayPalPayment>(

configuration => configuration.MapInheritedProperties().ToTable("PayPalPayments"));

Pero igualmente y aunque sea a mano el problema persiste si le damos mismo Id a varios subtipos… luego cambiamos a Guid o cambiamos semilla de identity (peligro) o usamos índices que no se pisen (peligro)…

modelBuilder.Entity<Payment>()

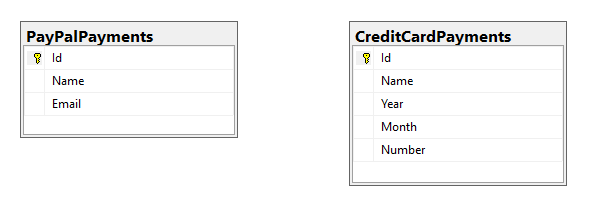
.Property(p => p.Id)

.HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);

public Guid Id { get; set; }

NEWID() vs NEWSEQUENTIALID() (este usa EF)

uniqueidentifier es un GUID, luego único en el mundo mundial… se supone



De todas formas, la SQL generada también fea (JOIN, UNION…)

Lo peor de todo es que no hay relaciones en BD, luego la cascada sólo funcionará en memoria (y eso asumiendo los registros relacionados estén cargados), luego nadie garantiza la integridad de la base de datos (te puedes quedar con registros hijos colgados)

# Complex Type

¿Cuándo infiere Code First un tipo complejo?

* No se puede inferir (ni tiene) una PK
* Sólo tiene propiedades simples
* No tiene propiedades de navegación
* Ninguna entidad la referencia a través de una propiedad de navegación de colección

Si no se infiere podemos usar el atributo ComplexTypeAttribute o modelBuilder.ComplexType<Address>()

internal class Customer

{

public Customer()

{

Orders = new Collection<Order>();

}

public int Id { get; set; }

public string Name { get; set; }

public virtual ICollection<Order> Orders { get; set; }

public Address Address { get; set; }

}

public class Address

{

public string Street { get; set; }

public string City { get; set; }

public string PostalCode { get; set; }

}

No se puede insertar null en Address, al menos un new Address() es requerido, aunque luego todas sus propiedades sean nulas, por eso el HasValue…

internal class Customer

{

public Customer()

{

Orders = new Collection<Order>();

Address = new Address();

}

public int Id { get; set; }

public string Name { get; set; }

public virtual ICollection<Order> Orders { get; set; }

public Address Address { get; set; }

}

public class Address

{

public int Id { get; set; }

public string Street { get; set; }

public string City { get; set; }

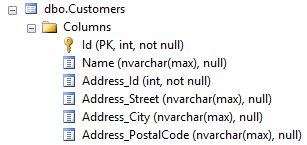
public string PostalCode { get; set; }

public bool HasValue => !string.IsNullOrWhiteSpace(Street) ||

!string.IsNullOrWhiteSpace(City) ||

!string.IsNullOrWhiteSpace(PostalCode);

}



modelBuilder.ComplexType<Address>();

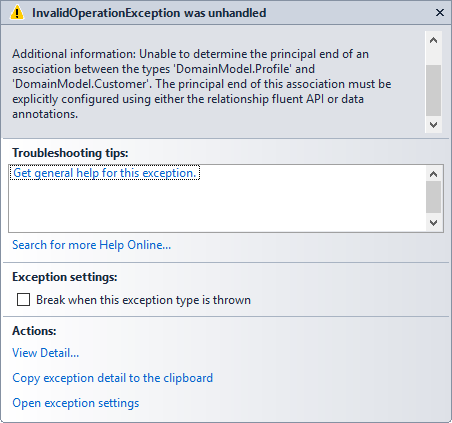
Por Id hay que especificar que es un Complex Type (si no crearía una tabla Address), en cualquier caso, mejor ponerlo siempre…

# Table Splitting

**Table Splitting** es “mapear 1 tabla a 2 o más entidades de nuestro modelo”.   
“la **tabla** está spliteada en varias entidades”.

¿Por qué?

* Para dividir nuestra entidad “grande” a nivel organizativo en varias entidades y así poder trabajar mejor
* Hacer Lazy Loading a propiedades “caras” (image por ejemplo) sólo cuando proceda (y no penalizar así la carga de la entidad inicialmente).



modelBuilder.Entity<Customer>()

.HasRequired(e => e.Profile)

.WithRequiredDependent(e => e.Customer)

.WillCascadeOnDelete(true);

internal class Customer

{

public Customer()

{

Orders = new Collection<Order>();

Address = new Address();

}

public int Id { get; set; }

public string Name { get; set; }

public virtual ICollection<Order> Orders { get; set; }

public Address Address { get; set; }

public int ProfileId { get; set; }

public Profile Profile { get; set; }

}

class Profile

{

public int Id { get; set; }

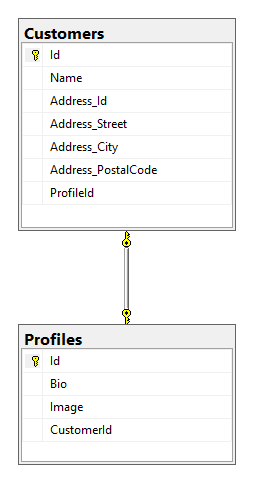
public string Bio { get; set; }

public byte[] Image { get; set; }

public int CustomerId { get; set; }

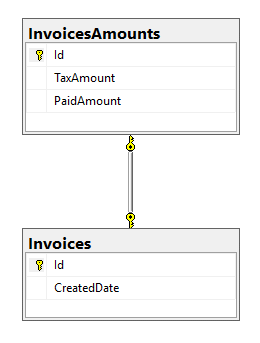
public Customer Customer { get; set; }

}



# Entity Splitting

**Entity Splitting** es “mapear 2 o más tablas a 1 entidad de nuestro modelo”.  
“la **entidad** está spliteado en varias tablas”.



class Invoice

{

public int Id { get; set; }

public DateTime CreatedDate { get; set; }

public decimal TaxAmount { get; set; }

public decimal PaidAmount { get; set; }

}

modelBuilder.Entity<Invoice>()

.Map(map =>

{

map.Properties(p => new

{

p.Id,

p.CreatedDate

});

map.ToTable("Invoices");

})

.Map(map =>

{

map.Properties(p => new

{

p.TaxAmount,

p.PaidAmount

});

map.ToTable("InvoicesAmounts");

});

No hay forma de hacer Cascade Delete en BD, en memoria si funciona, pero…

# Guión

**Sin Profiler no hay paraíso**

**EF Power Tools 4**

<https://marketplace.visualstudio.com/items?itemName=EntityFrameworkTeam.EntityFrameworkPowerToolsBeta4>

<http://thedatafarm.com/data-access/installing-ef-power-tools-into-vs2015/>

**Context.DumpAsHtml()**

**LINQPad**

**AddRange**

**AutoDetectChanges**

**Log**

<interceptors>

<interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">

<parameters>

<parameter value="C:\Path\To\My\LogOutput.txt"/>

</parameters>

</interceptor>

</interceptors>

context.Database.Log = Console.WriteLine;

q.ToString()

<http://www.entityframeworktutorial.net/entityframework6/database-command-interception.aspx>

<https://msdn.microsoft.com/en-us/library/dn469464(v=vs.113).aspx>

EntityFramework Extended Library

https://github.com/loresoft/EntityFramework.Extended

- Batch Update and Delete

- Audit Log

- agregar ctor (sin quitar el de por defecto) - ok

- agregar propiedad virtual

- persistiendo - ok

- sin persistir - ok

SaveChanges

- string.Empty

- Audit

EntityFramework.MappingAPI

https://efmappingapi.codeplex.com/

Pitfalls entidades desconectadas

Bounded Context

<http://p2p.wrox.com/book-beginning-asp-net-3-5-c-vb-book-isbn-978-0-470-18759-3/71802-cannot-create-database-diagram-error-code-0x54b.html>   
ALTER AUTHORIZATION ON DATABASE ::[Inheritance] TO [sa];