

Entity Framework Code-First

Announcing EF Code-First

Scott Guthrie

Announcing Entity Framework Code-First (CTP5 release)

This week the data team [released the CTP5 build](#) of the new Entity Framework Code-First library. EF Code-First enables a pretty sweet *code-centric* development workflow for working with data. It enables you to:

- Develop without ever having to open a designer or define an XML mapping file
- Define model objects by simply writing “plain old classes” with no base classes required
- Use a “convention over configuration” approach that enables database persistence without explicitly configuring anything
- Optionally override the convention-based persistence and use a fluent code API to fully customize the persistence mapping

I'm a big fan of the EF Code-First approach, and wrote several blog posts about it this summer:

- [Code-First Development with Entity Framework 4](#) (July 16th)
- [EF Code-First: Custom Database Schema Mapping](#) (July 23rd)
- [Using EF Code-First with an Existing Database](#) (August 3rd)

Today's new CTP5 release delivers several nice improvements over the CTP4 build, and will be the last preview build of Code First before the final release of it. We will ship the final EF Code First release in the first quarter of next year (Q1 of 2011). It works with all .NET application types (including both ASP.NET Web Forms and ASP.NET MVC projects).

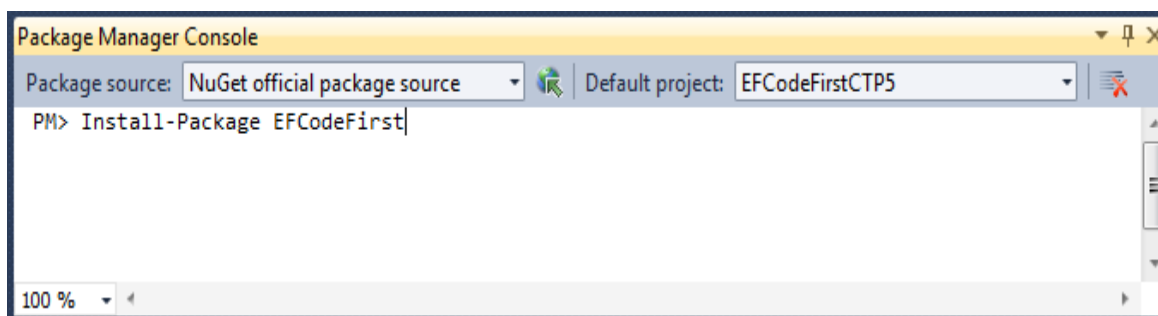
Installing EF Code First

You can install and use EF Code First CTP5 using one of two ways:

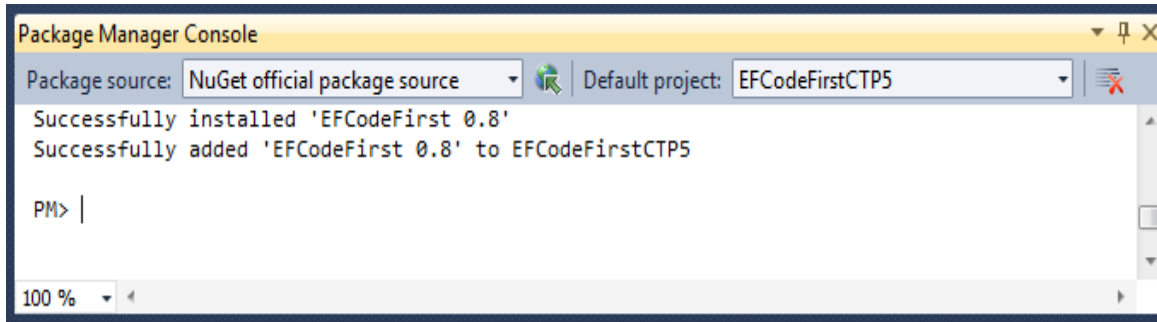
Approach 1) By [downloading and running a setup program](#). Once installed you can reference the EntityFramework.dll assembly it provides within your projects.

or:

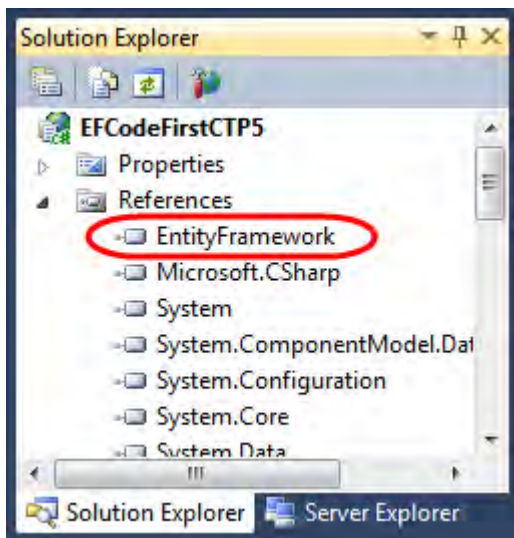
Approach 2) By using the [NuGet Package Manager](#) within Visual Studio to download and install EF Code First within a project. To do this, simply bring up the NuGet Package Manager Console within Visual Studio (View->Other Windows->Package Manager Console) and type “Install-Package EFCodeFirst”:



Typing “Install-Package EFCodeFirst” within the Package Manager Console will cause NuGet to download the EF Code First package, and add it to your current project:



Doing this will automatically add a reference to the EntityFramework.dll assembly to your project:



NuGet enables you to have EF Code First setup and ready to use within seconds. When the final release of EF Code First ships you'll also be able to just type "Update-Package EFCodeFirst" to update your existing projects to use the final release.

EF Code First Assembly and Namespace

The CTP5 release of EF Code First has an updated assembly name, and new .NET namespace:

- Assembly Name: **EntityFramework.dll**
- Namespace: **System.Data.Entity**

These names match what we plan to use for the final release of the library.

Nice New CTP5 Improvements

The new CTP5 release of EF Code First contains a bunch of nice improvements and refinements. Some of the highlights include:

- Better support for Existing Databases
- Built-in Model-Level Validation and DataAnnotation Support
- Fluent API Improvements
- Pluggable Conventions Support
- New Change Tracking API
- Improved Concurrency Conflict Resolution
- Raw SQL Query/Command Support

The rest of this blog post contains some more details about a few of the above changes.

Better Support for Existing Databases

EF Code First makes it really easy to create model layers that work against existing databases. CTP5 includes some refinements that further streamline the developer workflow for this scenario.

Below are the steps to use EF Code First to create a model layer for the [Northwind sample database](#):

Step 1: Create Model Classes and a DbContext class

Below is all of the code necessary to implement a simple model layer using EF Code First that goes against the Northwind database:

```
//
// Plain Old CLR Objects (aka POCO)

public class Product
{
    public int      ProductID { get; set; }
    public int      CategoryID { get; set; }
    public string   ProductName { get; set; }
    public Decimal? UnitPrice { get; set; }
    public bool     Discontinued { get; set; }

    public virtual Category Category { get; set; }
}

public class Category
{
    public int      CategoryID { get; set; }
    public string   CategoryName { get; set; }
    public string   Description { get; set; }
    public byte[]   Picture { get; set; }

    public virtual ICollection<Product> Products { get; set; }
}

//
// Northwind EF Code First Context Class

public class Northwind : DbContext
{
    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }
}
```

EF Code First enables you to use “POCO” – Plain Old CLR Objects – to represent entities within a database. This means that you do not need to derive model classes from a base class, nor implement any interfaces or data persistence attributes on them. This enables the model classes to be kept clean, easily testable, and “persistence ignorant”. The Product and Category classes above are examples of POCO model classes.

EF Code First enables you to easily connect your POCO model classes to a database by creating a “DbContext” class that exposes public properties that map to the tables within a database. The Northwind class above illustrates how this can be done. It is mapping our Product and Category classes to the “Products” and “Categories” tables within the database. The properties within the Product and Category classes in turn map to the columns within the Products and Categories tables – and each instance of a Product/Category object maps to a row within the tables.

The above code is all of the code required to create our model and data access layer! Previous CTPs of EF Code First required an additional step to work against existing databases (a call to `Database.Initializer<Northwind>(null)` to tell EF Code First to not create the database) – this step is no longer required with the CTP5 release.

Step 2: Configure the Database Connection String

We've written all of the code we need to write to define our model layer. Our last step before we use it will be to setup a connection-string that connects it with our database. To do this we'll add a "Northwind" connection-string to our web.config file (or App.Config for client apps) like so:

```
<connectionStrings>

  <add name="Northwind"
        connectionString="data source=.\SQLEXPRESS;Integrated Security=SSl
        providerName="System.Data.SqlClient" />

</connectionStrings>
```

EF "code first" uses a convention where `DbContext` classes by default look for a connection-string that has the same name as the context class. Because our `DbContext` class is called "Northwind" it by default looks for a "Northwind" connection-string to use. Above our Northwind connection-string is configured to use a local SQL Express database (stored within the `\App_Data` directory of our project). You can alternatively point it at a remote SQL Server.

Step 3: Using our Northwind Model Layer

We can now easily query and update our database using the strongly-typed model layer we just built with EF Code First.

The code example below demonstrates how to use LINQ to query for products within a specific product category. This query returns back a sequence of strongly-typed `Product` objects that match the search criteria:

```
Northwind northwind = new Northwind();

var products = from p in northwind.Products
               where p.Category.CategoryName == "Beverages"
               select p;
```

The code example below demonstrates how we can retrieve a specific `Product` object, update two of its properties, and then save the changes back to the database:


```

Northwind northwind = new Northwind();

// Retrieve specific product by primary key
Product product = northwind.Products.Find(1);

// Update two properties
product.UnitPrice = 2.33M;
product.Discontinued = false;

// Persist changes to Database
northwind.SaveChanges();

```

EF Code First handles all of the change-tracking and data persistence work for us, and allows us to focus on our application and business logic as opposed to having to worry about data access plumbing.

Built-in Model Validation

EF Code First allows you to use any validation approach you want when implementing business rules with your model layer. This enables a great deal of flexibility and power.

Starting with this week's CTP5 release, EF Code First also now includes built-in support for both the DataAnnotation and IValidatorObject validation support built-into .NET 4. This enables you to easily implement validation rules on your models, and have these rules automatically be enforced by EF Code First whenever you save your model layer. It provides a very convenient "out of the box" way to enable validation within your applications.

Applying DataAnnotations to our Northwind Model

The code example below demonstrates how we could add some declarative validation rules to two of the properties of our "Product" model:

```

public class Product
{
    public int ProductID { get; set; }

    public int CategoryID { get; set; }

    [Required(ErrorMessage="Product Name must be specified")]
    public string ProductName { get; set; }

    [Required(ErrorMessage="Price must be specified")]
    [Range(0.01, Double.MaxValue, ErrorMessage="Can't be free!")]
    public Decimal? UnitPrice { get; set; }

    public virtual Category Category { get; set; }
}

```

We are using the [Required] and [Range] attributes above. These validation attributes live within the System.ComponentModel.DataAnnotations namespace that is built-into .NET 4, and can be used independently of EF. The error messages specified on them can either be explicitly defined (like above) – or retrieved from resource files (which makes localizing applications easy).

Validation Enforcement on SaveChanges()

EF Code-First (starting with CTP5) now automatically applies and enforces DataAnnotation rules when a model object is updated or saved. You do not need to write any code to enforce this – this support is now enabled by default.

This new support means that the below code – which violates our above rules – will automatically throw an exception when we call the “SaveChanges()” method on our Northwind DbContext:

```
// Retrieve a Product
var product = northwind.Products.Find(id);

// Set invalid property values
product.ProductName = null;
product.UnitPrice = 0.00M;

// Will throw error!!!
northwind.SaveChanges(); ← Validation Exception!
```

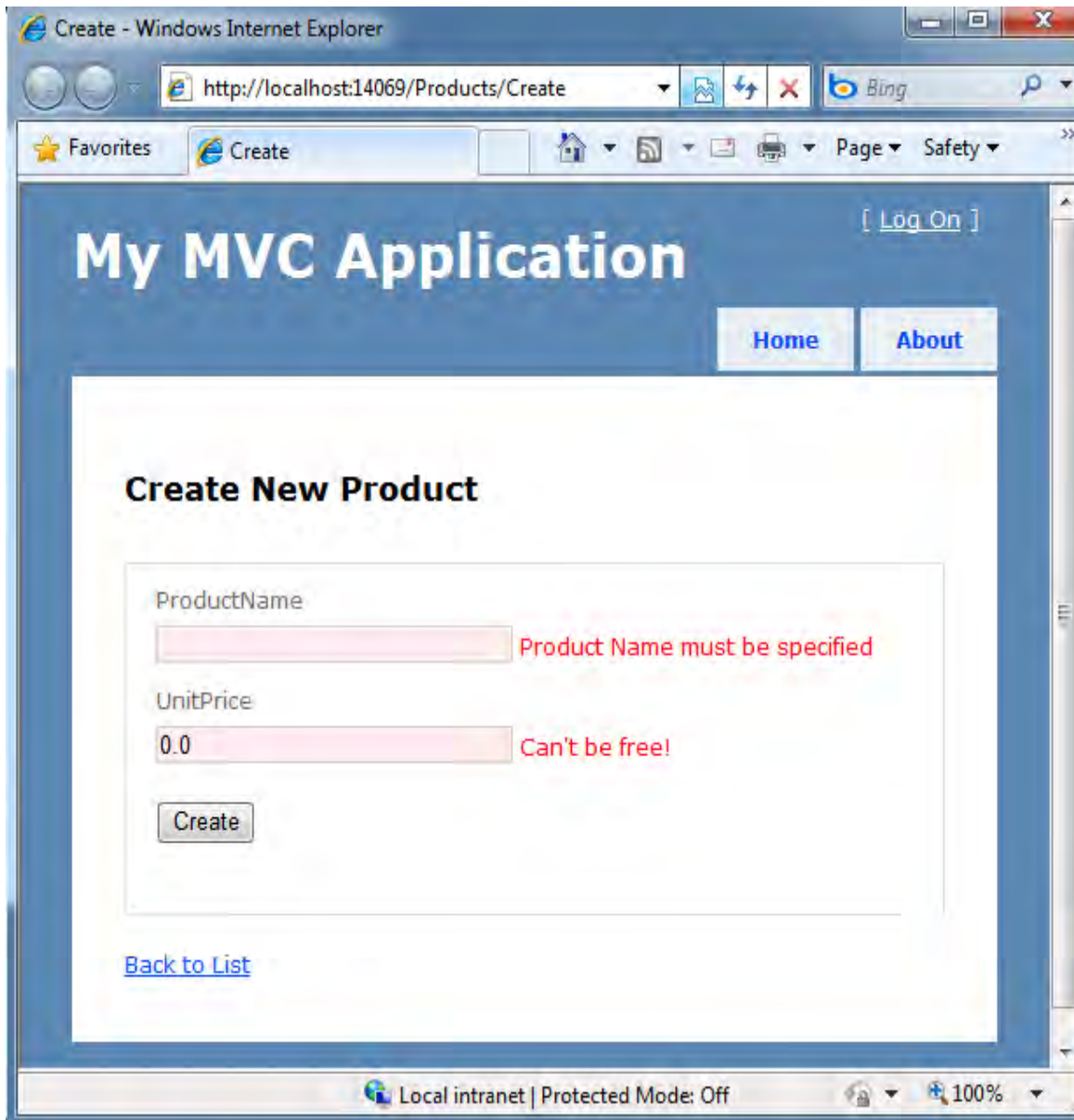
The *DbEntityValidationException* that is raised when the SaveChanges() method is invoked contains a “EntityValidationErrors” property that you can use to retrieve the list of all validation errors that occurred when the model was trying to save. This enables you to easily guide the user on how to fix them. Note that EF Code-First will abort the entire transaction of changes if a validation rule is violated – ensuring that our database is always kept in a valid, consistent state.

EF Code First's validation enforcement works both for the built-in .NET DataAnnotation attributes (like Required, Range, RegularExpression, StringLength, etc), as well as for any custom validation rule you create by sub-classing the *System.ComponentModel.DataAnnotations.ValidationAttribute* base class.

UI Validation Support

A lot of our UI frameworks in .NET also provide support for DataAnnotation-based validation rules. For example, ASP.NET MVC, ASP.NET Dynamic Data, and Silverlight (via WCF RIA Services) all provide support for displaying client-side validation UI that honor the DataAnnotation rules applied to model objects.

The screen-shot below demonstrates how using the default “Add-View” scaffold template within an ASP.NET MVC 3 application will cause appropriate validation error messages to be displayed if appropriate values are not provided:



ASP.NET MVC 3 supports both client-side and server-side enforcement of these validation rules. The error messages displayed are automatically picked up from the declarative validation attributes – eliminating the need for you to write any custom code to display them.

Keeping things DRY

The “DRY Principle” stands for “Do Not Repeat Yourself”, and is a best practice that recommends that you avoid duplicating logic/configuration/code in multiple places across your application, and instead specify it only once and have it apply everywhere.

EF Code First CTP5 now enables you to apply declarative DataAnnotation validations on your model classes (and specify them only once) and then have the validation logic be enforced (and corresponding error messages displayed) across all applications scenarios – including within controllers, views, client-side scripts, and for any custom code that updates and manipulates model classes.

This makes it much easier to build good applications with clean code, and to build applications that can rapidly iterate and evolve.

Other EF Code First Improvements New to CTP5

EF Code First CTP5 includes a bunch of other improvements as well. Below are a few short descriptions of some of them:

- [Fluent API Improvements](#)

EF Code First allows you to override an “OnModelCreating()” method on the DbContext class to further refine/override the schema mapping rules used to map model classes to underlying database schema. CTP5 includes some refinements to the modelBuilder class that is passed to this method which can make defining mapping rules cleaner and more concise. The ADO.NET Team blogged [some samples](#) of how to do this [here](#).

- [Pluggable Conventions Support](#)

EF Code First CTP5 provides new support that allows you to override the “default conventions” that EF Code First honors, and optionally replace them with your own set of conventions.

- [New Change Tracking API](#)

EF Code First CTP5 exposes a new set of change tracking information that enables you to access Original, Current & Stored values, and State (e.g. Added, Unchanged, Modified, Deleted). This support is useful in a variety of scenarios.

- [Improved Concurrency Conflict Resolution](#)

EF Code First CTP5 provides better exception messages that allow access to the affected object instance and the ability to resolve conflicts using current, original and database values.

- [Raw SQL Query/Command Support](#)

EF Code First CTP5 now allows raw SQL queries and commands (including SPROC's) to be executed via the SqlQuery and SqlCommand methods exposed off of the DbContext.Database property. The results of these method calls can be materialized into object instances that can be optionally change-tracked by the DbContext. This is useful for a variety of advanced scenarios.

- [Full Data Annotations Support](#)

EF Code First CTP5 now supports all standard DataAnnotations within .NET, and can use them both to perform validation as well as to automatically create the appropriate database schema when EF Code First is used in a database creation scenario.


Summary

EF Code First provides an elegant and powerful way to work with data. I really like it because it is extremely clean and supports best practices, while also enabling solutions to be implemented very, very rapidly. The code-only approach of the library means that model layers end up being flexible and easy to customize.

This week's CTP5 release further refines EF Code First and helps ensure that it will be really sweet when it ships early next year. I recommend using NuGet to install and give it a try today. I think you'll be pleasantly surprised by how awesome it is.

Hope this helps,

Scott

Published Wednesday, December 08, 2010 1:39 AM by [ScottGu](#) 
Filed under: [ASP.NET](#), [.NET](#), [LINQ](#), [Data](#)

Comments

Associations in EF Code First

Morteza Manavi

Associations in EF Code First: Part 1 – Introduction and Basic Concepts

Earlier this month the data team shipped the [Release Candidate](#) of EF 4.1. The most exciting feature of EF 4.1 is *Code First*, a new development pattern for EF which provides a really elegant and powerful code-centric way to work with data as well as an alternative to the existing *Database First* and *Model First* patterns. Code First is designed based on [Convention over Configuration](#) paradigm and focused around defining your model using C#/VB.NET classes, these classes can then be mapped to an existing database or be used to generate a database schema. Additional configuration can be supplied using Data Annotations or via a fluent API.

I'm a big fan of the EF Code First approach, and wrote several blog posts about it based on its CTP5 build:

- [Associations in EF Code First CTP5: Part 1 – Complex Types](#)
- [Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations](#)
- [Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations](#)
- [Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy \(TPH\)](#)
- [Inheritance with EF Code First CTP5: Part 2 – Table per Type \(TPT\)](#)
- [Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type \(TPC\)](#)

Compare to CTP5, EF 4.1 release is more about bug fixing and bringing it to a go-live quality level than anything else. Pretty much all of the API that has been introduced in CTP5 is still exactly the same (except very few changes including renaming of `DbContext` and `ModelBuilder` classes as well as consolidation of `IsIndependent` fluent API method). Therefore, the above blog posts are still usable and can (hopefully) help you in your Code First development. Having said that, I decided to complete my Code First articles by starting a whole new series instead of doing post maintenance on the current CTP5 ones.

A Note For Those Who are New to EF and Code-First

If you choose to learn EF you've chosen well. If you choose to learn EF with Code First you've done even better. To get started, you can find an EF 4.1 Code First walkthrough by ADO.NET team [here](#). In this series, I assume you already setup your machine to do Code First development and also that you are familiar with Code First fundamentals and basic concepts.

Code First And Associations

I will start my EF 4.1 Code First articles by a series on entity association mappings. You will see

that when it comes to associations, Code First brings ultimate power and flexibility. This series will come in several parts including:

- [Part 1 – Introduction and Basic Concepts](#)
- [Part 2 – Complex Types](#)
- [Part 3 – Shared Primary Key Associations](#)
- [Part 4 – Table Splitting](#)
- [Part 5 – One-to-One Foreign Key Associations](#)
- [Part 6 – Many-valued Associations](#)

Why Starting with Association Mappings?

From my experience with the EF user community, I know that the first thing many developers try to do when they begin using EF (specially when having a Code First approach) is a mapping of a parent/children relationship. This is usually the first time you encounter collections. It's also the first time you have to think about the differences between entities and value types, or the type of relationships between your entities. Managing the associations between classes and the relationships between tables is at the heart of ORM. Most of the difficult problems involved in implementing an ORM solution relate to association management.

In order to build a solid foundation for our discussion, we will start by learning about some of the core concepts around the relationship mapping and will leave the discussion for each type of entity associations to the next posts in this series.

What is Mapping?

Mapping is the act of determining how objects and their relationships are persisted in permanent data storage, in our case, relational databases.

What is Relationship Mapping?

A mapping that describes how to persist a relationship (association, aggregation, or composition) between two or more objects.

Types of Relationships

There are two categories of object relationships that we need to be concerned with when mapping associations. The first category is based on *multiplicity* and it includes three types:

- **One-to-one relationships:** This is a relationship where the maximums of each of its multiplicities is one.
- **One-to-many relationships:** Also known as a many-to-one relationship, this occurs when the maximum of one multiplicity is one and the other is greater than one.
- **Many-to-many relationships:** This is a relationship where the maximum of both multiplicities is greater than one.

The second category is based on *directionality* and it contains two types:

- **Uni-directional relationships:** when an object knows about the object(s) it is related to but the other object(s) do not know of the original object. To put this in EF terminology, when a navigation property exists only on one of the association ends and not on the both.
- **Bi-directional relationships:** When the objects on both end of the relationship know of each other (i.e. a navigation property defined on both ends).

How Object Relationships are Implemented in POCO Object Models?


When the multiplicity is one (e.g. 0..1 or 1) the relationship is implemented by defining a *navigation property* that reference the other object (e.g. an *Address* property on *User* class). When the multiplicity is many (e.g. 0..*, 1..*) the relationship is implemented via an [ICollection](#) of the type of other object.

How Relational Database Relationships are Implemented?

Relationships in relational databases are maintained through the use of *Foreign Keys*. A foreign key is a data attribute(s) that appears in one table and must be the primary key or other candidate key in another table. With a one-to-one relationship the foreign key needs to be implemented by one of the tables. To implement a one-to-many relationship we implement a foreign key from the “one table” to the “many table”. We could also choose to implement a one-to-many relationship via an *associative table* (aka *Join table*), effectively making it a many-to-many relationship.

References

- [ADO.NET team blog](#)
- [Mapping Objects to Relational Databases](#)
- [Java Persistence with Hibernate book](#)

Published Sunday, March 27, 2011 6:04 AM by [mortezaam](#) 
Filed under: [C#](#), [Code First](#), [Entity Framework 4.1](#)

Comments

re: Associations in EF 4.1 Code First: Part 1 – Introduction and Basic Concepts

Monday, March 28, 2011 8:55 AM by [Jan C. de Graaf](#)

I'm eagerly awaiting "Part 7 – Many-to-Many Associations"!

Ассоциации в EF 4.1 Code First: часть 1

Monday, March 28, 2011 12:40 PM by [progg.ru](#)

Thank you for submitting this cool story - Trackback from [progg.ru](#)

re: Associations in EF 4.1 Code First: Part 1 – Introduction and Basic Concepts

Wednesday, March 30, 2011 12:44 PM by Cristian Peraferrer

I'm also awaiting "Part 7 - Many-to-Many Associations"

Any other resources to look for?

re: Associations in EF 4.1 Code First: Part 1 – Introduction and Basic Concepts

Wednesday, March 30, 2011 1:09 PM by [morteza](#)

@Cristian Peraferrer: There are not many resources on associations with Code First yet, and it will take me a while to get to the many to many associations. For the time being, if you have any questions, please post it here and I'll try my best to help you with that. Thanks :)

Что почитать на выходных? Рекомендуем, выпуск №5

Friday, April 01, 2011 2:59 AM by [Блог Владимира Юнева](#)

Это подборка статей на тему веб-разработки на платформе .NET (и не только). За день перед выходными я

Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Tuesday, May 17, 2011 12:18 AM by [Enterprise .Net](#)

This is the sixth and last post in a series that explains entity association mappings with EF Code First

Associations in EF 4.1 Code First: Part 4 ??? Table Splitting - Enterprise .Net

Tuesday, May 17, 2011 12:30 AM by [Associations in EF 4.1 Code First: Part 4 ??? Table Splitting - Enterprise .Net](#)

Pingback from [Associations in EF 4.1 Code First: Part 4 ??? Table Splitting - Enterprise .Net](#)

Associations in EF 4.1 Code First: Part 6 ??? Many-valued Associations

Tuesday, May 17, 2011 2:20 AM by [Associations in EF 4.1 Code First: Part 6 ??? Many-valued Associations](#)

Pingback from [Associations in EF 4.1 Code First: Part 6 ??? Many-valued Associations](#)

What does multiplicity 1, 0..1, * mean? | Code First :: Entity Framework

Thursday, May 19, 2011 8:38 PM by [What does multiplicity "1", "0..1", "*" mean? | Code First :: Entity Framework](#)

Pingback from [What does multiplicity 1, 0..1, * mean? | Code First :: Entity Framework](#)

re: Associations in EF 4.1 Code First: Part 1 – Introduction and Basic Concepts

Saturday, June 11, 2011 10:43 AM by shaggy

Examples man. Examples.

What is Entity Framework fluent api? - Programmers Goodies

Sunday, August 07, 2011 5:47 AM by [What is Entity Framework fluent api? - Programmers Goodies](#)

Pingback from [What is Entity Framework fluent api? - Programmers Goodies](#)

SMO + LINQ for Entity Framework Code First Database Modifications

Saturday, September 03, 2011 5:21 PM by [SMO + LINQ for Entity Framework Code First Database Modifications](#)

Pingback from SMO + LINQ for Entity Framework Code First Database Modifications

EF: Code First Tutorial/Sample « Alvin's Blog

Saturday, November 05, 2011 1:34 PM by [EF: Code First Tutorial/Sample « Alvin's Blog](#)

Pingback from EF: Code First Tutorial/Sample « Alvin's Blog

re: Associations in EF 4.1 Code First: Part 1 – Introduction and Basic Concepts

Thursday, November 10, 2011 1:32 AM by dliss

terrific. I am going to use EF in my project. Basic Concepts is clear and straightforward for me.

thanks.

Associations in EF 4.1 Code First: Part 3 ??? Shared Primary Key Associations - Enterprise .Net

Wednesday, November 16, 2011 10:25 AM by [Associations in EF 4.1 Code First: Part 3 ??? Shared Primary Key Associations - Enterprise .Net](#)

Pingback from Associations in EF 4.1 Code First: Part 3 ??? Shared Primary Key Associations - Enterprise .Net

EF Code First « Default Constructor

Saturday, January 28, 2012 3:02 AM by [EF Code First « Default Constructor](#)

Pingback from EF Code First « Default Constructor

[Terms of Use](#)

Associations in EF Code First CTP5: Part 1 – Complex Types

Last week the CTP5 build of the new Entity Framework Code First has been [released](#) by data team Microsoft. Entity Framework Code-First provides a pretty powerful code-centric way to work with the databases. When it comes to associations, it brings ultimate flexibility. I'm a big fan of the EF Code First approach and I am planning to explain association mapping with code first in a series of blog posts and this one is dedicated to Complex Types.

A Note For Those Who are New to Entity Framework and Code-First

If you choose to learn EF you've chosen well. If you choose to learn EF with Code First you've done even better. To get started, you can find a great walkthrough by Scott Guthrie [here](#) and another one by ADO.NET team [here](#). In this post, I assume you already setup your machine to do Code First development and also that you are familiar with Code First fundamentals and basic concepts.

In order to build a solid foundation for our discussion, we will start by learning about some of the core concepts around the relationship mapping.

What is Mapping?

Mapping is the act of determining how objects and their relationships are persisted in permanent data storage, in our case, relational databases.

What is Relationship Mapping?

A mapping that describes how to persist a relationship (association, aggregation, or composition) between two or more objects.

Types of Relationships

There are two categories of object relationships that we need to be concerned with when mapping associations. The first category is based on *multiplicity* and it includes three types:

- **One-to-one relationships:** This is a relationship where the maximums of each of its multiplicities is one.
- **One-to-many relationships:** Also known as a many-to-one relationship, this occurs when the maximum of one multiplicity is one and the other is greater than one.
- **Many-to-many relationships:** This is a relationship where the maximum of both multiplicities

is greater than one.

The second category is based on *directionality* and it contains two types:

- **Uni-directional relationships:** when an object knows about the object(s) it is related to but 1 other object(s) do not know of the original object. To put this in EF terminology, when a navigation property exists only on one of the association ends and not on the both.
- **Bi-directional relationships:** When the objects on both end of the relationship know of each other (i.e. a navigation property defined on both ends).

How Object Relationships are Implemented in POCO Object Models?

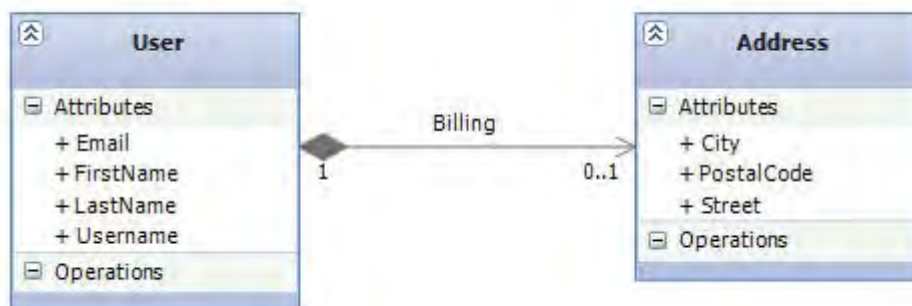
When the multiplicity is one (e.g. 0..1 or 1) the relationship is implemented by defining a *navigation property* that reference the other object (e.g. an *Address* property on *User* class). When the multiplicity is many (e.g. 0..*, 1..*) the relationship is implemented via an [ICollection](#) of the type of other object.

How Relational Database Relationships are Implemented?

Relationships in relational databases are maintained through the use of *Foreign Keys*. A foreign key a data attribute(s) that appears in one table and must be the primary key or other candidate key in another table. With a one-to-one relationship the foreign key needs to be implemented by one of the tables. To implement a one-to-many relationship we implement a foreign key from the “one table” to the “many table”. We could also choose to implement a one-to-many relationship via an *associative table* (aka *Join table*), effectively making it a many-to-many relationship.

Introducing the Model

Now, let's review the model that we are going to use in order to implement Complex Type with Code First. It's a simple object model which consist of two classes: *User* and *Address*. Each user *could* have one billing address. The Address information of a User is modeled as a separate class as you can see in the UML model below:



In object-modeling terms, this association is a kind of *aggregation*—a *part-of* relationship. Aggregation is a strong form of association; it has some additional semantics with regard to the lifecycle of object. In this case, we have an even stronger form, *composition*, where the lifecycle of the part is fully dependent upon the lifecycle of the whole.

Fine-grained Domain Models

The motivation behind this design was to achieve *Fine-grained domain models*. In crude terms, fine-grained means “more classes than tables”. For example, a user may have both a billing address and a home address. In the database, you may have a single User table with the columns BillingStreet, BillingCity, and BillingPostalCode along with HomeStreet, HomeCity, and HomePostalCode. There are good reasons to use this somewhat *denormalized* relational model (performance, for one). In our object model, we can use the same approach, representing the two addresses as six string-valued properties of the User class. But it’s much better to model this using Address class, where User has the BillingAddress and HomeAddress properties. This object model achieves improved *cohesion* and *greater code reuse* and is more understandable.

Complex Types: Splitting a Table Across Multiple Types

Back to our model, there is no difference between this composition and other weaker styles of association when it comes to the actual C# implementation. But in the context of ORM, there is a big difference: A composed class is often a candidate *Complex Type*. But C# has no concept of composition—a class or property can’t be marked as a composition. The only difference is the object identifier: a complex type has *no* individual identity (i.e. no AddressId defined on Address class) which make sense because when it comes to the database everything is going to be saved into one single table.

How to implement a Complex Type with Code First

Code First has a concept of *Complex Type Discovery* that works based on a set of *Conventions*. The convention is that if Code First discovers a class where a primary key cannot be inferred, and no primary key is registered through Data Annotations or the fluent API, then the type will be *automatically* registered as a complex type. Complex type detection also requires that the type does not have properties that reference entity types (i.e. all the properties must be scalar types) and is not referenced from a collection property on another type. Here is the implementation:

```
public class User
{
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Username { 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; }
}
```

```

}

public class EntityMappingContext : DbContext
{
    public DbSet<User> Users { get; set; }
}

```

With code first, this is all of the code we need to write to create a complex type, we do not need to configure any additional database schema mapping information through Data Annotations or the fluent API.

Database Schema

The mapping result for this object model is as follows:



Complex Types are Required

As a limitation of EF in general, complex types are always *considered* required. To see this limitation in action, let's try to add a record to our database:

```

using (var context = new EntityMappingContext())
{
    User user = new User()
    {
        FirstName = "Morteza",
        LastName = "Manavi",
        Username = "mmanavi"
    };

    context.Users.Add(user);
    context.SaveChanges();
}

```

Surprisingly, this code throws a [System.Data.UpdateException](#) at runtime with this message:

Null value for non-nullable member. Member: 'Address'.

If we initialize the address object, the exception would go away and user will be successfully saved into database:


```

using (var context = new EntityMappingContext())
{
    User user = new User()
    {
        FirstName = "Morteza",
        LastName = "Manavi",
        Username = "mmanavi",
        Address = new Address()
    };

    context.Users.Add(user);
    context.SaveChanges();
}

```



When we read back the inserted record from the database, EF will return an Address object with all the properties (Street, City and PostalCode) have null values. This means that if you store a complex type object with all null property values, EF returns a initialized complex type when the owning entity is retrieved from the database.

Explicitly Register a Type as Complex

You saw that in our model, we did not use any data annotation or fluent API code to designate the Address as a complex type, yet Code First perfectly detects it as a complex type based on Complex Type Discovery concept. But what if our domain model requires a new property called Id on Address class? This new Id property is just a scalar non-primary key property that represents let's say another piece of information about address. In this case, Code First actually *can* infer a key and therefore marks Address as an entity that has its own mapping table unless we specify otherwise. This is where explicit complex type registration comes into play. CTP5 defined a new attribute in System.ComponentModel.DataAnnotations namespace called [ComplexTypeAttribute](#). All we need to do is to use this attribute on our Address class:

```

[ComplexType]
public class Address
{
    public int Id { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
}

```

This will cause Address to remain as a complex type in our model. As always, we can do the same with fluent API. In CTP5 a new generic method has been added to modelBuilder class which is called `ComplexType` and has the following signature (when working with fluent API, we don't really care about the method's return values):

```

public virtual ComplexTypeConfiguration<TComplexType> ComplexType<TComplexType>(
    where TComplexType : class;

```

Here is how we can register our Address type as complex in fluent API:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ComplexType<Address>();
}

```

```
}

```

Best Practices When Working with Complex Types

- **Always initialize the complex type:** Because of the problem we just saw, I recommended to always initialize the complex type inside its owning entity's constructor.
- **Add a read only property to the complex type for null value checking:** Defining a non-persistent read only property like *HasValue* will help to test for null values.
- **Consider always using ComplexType attribute:** Even if your class is automatically detected as a complex type by Code First, I still recommend to mark it with `[ComplexType]` attribute. Not only that helps your object model to be more readable but also ensures that your complex types will stay as complex type as your model evolves in your project. Furthermore, there is a bug in CTP5 and that is if you put `Required` attribute (a data annotation that Code First supports for validation) on any of the complex type's properties (e.g. `PostalCode`) then Code First will stop thinking that it is a complex type and will throw a `ModelValidationException`. The workaround for this bug is to explicitly mark `Address` with `ComplexType` attribute. Hence, it will be beneficial in such cases as well.

Therefore, our final object model will be:

```
public class User
{
    public User()
    {
        Address = new Address();
    }

    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Username { get; set; }
    public Address Address { get; set; }
}

[ComplexType]
public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }

    public bool HasValue
    {
        get
        {
            return (Street != null || PostalCode != null || City != null);
        }
    }
}

```

The interesting point is that we do not have to explicitly exclude *HasValue* property from the mapping. Since this property does not have a setter, EF Code First will be ignoring it based on a convention which makes sense since a read only property is most probably represents a computed value and does not need to be persist in the database.

Complex Types and the New Change Tracking API

EF Code First CTP5 exposes a new set of change tracking information that enables us to access Original, Current & Stored values, and State (e.g. Added, Unchanged, Modified, Deleted) of our entities. The *Original Values* are the values the entity had when it was queried from the database. *Current Values* are the values the entity has now. This feature also fully supports complex types:

```
using (var context = new EntityMappingContext())
{
    var user = context.Users.Find(1);

    Address originalValues = context.Entry(user)
        .ComplexProperty(u => u.Address)
        .OriginalValue;

    Address currentValues = context.Entry(user)
        .ComplexProperty(u => u.Address)
        .CurrentValue;
}
```

The entry point for accessing the new change tracking API is *DbContext*'s *Entry* method which return an object of type *DbEntityEntry*. *DbEntityEntry* contains a *ComplexProperty* method that returns a *DbComplexPropertyEntry* object where we can access the original and current values:

```
namespace System.Data.Entity.Infrastructure
{
    public class DbEntityEntry<TEntity> where TEntity : class
    {
        public DbComplexPropertyEntry<TEntity, TComplexProperty>
            ComplexProperty<TComplexProperty>
                (Expression<Func<TEntity, TComplexProperty>> property);
    }
}
```

Limitations of This Mapping

There are two important limitations to classes mapped as Complex Types:


- **Shared references is not possible:** The Address Complex Type doesn't have its own database identity (primary key) and so can't be referred to by any object other than the containing instance of User (e.g. a Shipping class that also needs to reference the same User Address).
- **No elegant way to represent a null reference:** As we saw there is no elegant way to represent a null reference to an Address. When reading from database, EF Code First *always* initialize Address object even if values in all mapped columns of the complex type are null.

Summary

In this post we learned about fine-grained domain models which complex type is just one example of it. Fine-grained is fully supported by EF Code First and is known as the most important requirement for a rich domain model. Complex type is usually the simplest way to represent one-to-one relationships and because the lifecycle is almost always dependent in such a case, it's either an aggregation or a composition in UML. In the next posts we will revisit the same domain model and we will learn about other ways to map a one-to-one association that does not have the limitations of the complex types.

References

- [ADO.NET team blog](#)
- [Mapping Objects to Relational Databases](#)
- [Java Persistence with Hibernate book](#)

Published Saturday, December 11, 2010 8:21 AM by [mortezam](#) 
 Filed under: [Entity Framework](#), [C#](#), [Code First](#), [CTP5](#), [.NET](#)

Comments

[# re: Entity Association Mapping with Code First Part 1 : Mapping Complex Types](#)

Sunday, December 12, 2010 12:22 PM by [ali62b](#)

Very useful post. Keep up the great work !

[# re: Entity Association Mapping with Code First Part 1 : Mapping Complex Types](#)

Sunday, December 12, 2010 6:57 PM by [Paul](#)

Great read! Cannot wait to read the rest of this series.

[# re: Entity Association Mapping with Code First Part 1 : Mapping Complex Types](#)

Monday, December 13, 2010 6:30 AM by HariOm

hmm..

I create applications using an EF that I created myself.

Never bothered to track what Microsoft is doing with their EF.

Anyway, mine doesn't even need the code to be typed in.

Also, a query. In the above example, how does the EF track change in Address of the User.

Since it is normalised, how does one retrieve the old Address for a given User once the Address object is updated.

[# re: Entity Association Mapping with Code First Part 1 : Mapping Complex Types](#)

Tuesday, December 14, 2010 2:11 PM by [mortezam](#)

@HariOm: EF keeps track of the original and current values for every entity. The original values are the

values the entity had when it was queried from the database. The current values are the ones the entity has now. EF doesn't keep track of any more history than this. Therefore, you can access the old Address values for a given User by reading its original values. I just updated the post to show how it is done by using the new change tracking API when it comes to Complex Types. Thanks for your great question :)

[# re: Entity Association Mapping with Code First Part 1 : Mapping Complex Types](#)

Tuesday, December 14, 2010 8:45 PM by Yuvan

How do I add validation via dataannotation. When I try adding a Required Validation on PostalCode Property it breaks and throws an error. However when I add a Regular expression validation on that property it works. Is this a bug in the CTP5??

[# re: Entity Association Mapping with Code First Part 1 : Mapping Complex Types](#)

Wednesday, December 15, 2010 12:00 AM by [Paul](#)

Great additions, was not aware of the Entry() method. Learn something new everyday!

[# re: Entity Association Mapping with Code First Part 1: Complex Types](#)

Sunday, December 19, 2010 10:11 PM by [mortezam](#)

@Yuvan: Yes, I checked this with EF team, this is a bug in CTP5. Putting [Required] on a property of the complex type should not cause Code First to stop thinking that it is a complex type. The exception that you are getting would go away if you explicitly mark Address class with [ComplexType] attribute. Thanks :)

[# re: Entity Association Mapping with Code First Part 1: Complex Types](#)

Tuesday, December 21, 2010 5:59 AM by [Batslhor](#)

Thank, very nice explanation, I am learning EF4CF and this series really help me!

[# re: Entity Association Mapping with Code First Part 1: Complex Types](#)

Tuesday, December 21, 2010 9:01 AM by anonymous

```

If I have
Person
{
  Id
  Name
  ContactInfo
}
ContactInfo
{
  Phone
  Address
}
Address
{
  city
}

```

it will be 2 complex types (ContactInfo and Address)

This logic is not working with ctp5 and says (and if only contactInfo it is working)

The server encountered an error processing the request. The exception message is 'One or more validation errors were detected during model generation: System.Data.Edm.EdmEntityType: : EntityType 'ContactInfo' has no key defined. Define the key for this EntityType. System.Data.Edm.EdmEntityType: The

EntitySet ContactInfoes is based on type ContactInfo that has no keys defined. '. See server logs for more details. The exception stack trace is:

re: Entity Association Mapping with Code First Part 1: Complex Types

Tuesday, December 21, 2010 9:08 AM by Anonymous

Sorry. The above code will work if we do the constructor with creation of ComplexTypes in the constructor thingy:

```
Person
{
    Id
    Name
    ContactInfo
    public Person()
    {
        contactInfo=new ContactInfo();
    }
}
```

similarly for ContactInfo and Address (create Address in ContactInfo constructor)

re: Entity Association Mapping with Code First Part 1: Complex Types

Wednesday, December 22, 2010 6:24 PM by Frank

By the way, excellent stuff. I have a class User which has ICollection<Token> and ICollection<Role>. User to Tokens is 1..many. User to Roles table is many..many. My mapping is (probably wrong):

```
HasKey(u => u.UserId).HasOptional(u => u.Tokens).WithRequired().WillCascadeOnDelete();
```

```
HasMany(u => u.Roles).WithMany(r => r.Users);
```

When I create a new User, in its ctor I instantiate a new List of Tokens and then simply add a new Token class to that collection later as User.Tokens.Add(some_token) and I get:

"Unable to cast object of type 'System.Collections.Generic.List`1[Token]' to type 'Token'."

Any help is appreciated.

re: Entity Association Mapping with Code First Part 1: Complex Types

Wednesday, December 22, 2010 8:21 PM by [mortezam](#)

@Frank: The reason you are getting the exception is because the fluent API code for User-Token association is incorrect. That code is good for a one-to-one association while the association is one-to-many (like you mentioned). The following will do the trick:

```
public class User
{
    public User()
    {
        Tokens = new List<Token>();
        Roles = new List<Role>();
    }
    public int UserId { get; set; }
    public virtual ICollection<Token> Tokens { get; set; }
    public virtual ICollection<Role> Roles { get; set; }
}

public class Token
{
    public int TokenId { get; set; }
```

```

}

[Table("Role")]
public class Role
{
    public int RoleId { get; set; }
    public int RoleTypeId { get; set; }
    public virtual ICollection<User> Users { get; set; }
    public virtual RoleType RoleType { get; set; }
}

public class RoleType
{
    public int RoleTypeId { get; set; }
    public string RoleName { get; set; }
}

public class CTP5Context : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Token> Tokens { get; set; }
    public DbSet<Role> Roles { get; set; }
    public DbSet<RoleType> RoleType { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // 1:* between User and Token:
        modelBuilder.Entity<User>().HasMany(u => u.Tokens).WithRequired().WillCascadeOnDelete();

        // *: * between User and Role
        modelBuilder.Entity<User>().HasMany(u => u.Roles).WithMany(r => r.Users).Map(m =>
        {
            m.ToTable("Roles");
        });
    }
}

```

Please note that I've not use any fluent API code to designate UserId as a key for User entity. This will be configured automatically by Code First based on the convention. Now, if you run your code that adds a new User object with a Token against this object model, you'll see that it will successfully save into the database. I'll fully explain one-to-many and many-to-many associations in my next posts in this series. Thanks :)

re: Entity Association Mapping with Entity Framework Code First CTP5 Part 1: Complex Types

Thursday, December 23, 2010 12:24 PM by Frank

Perfect. So for the roles, I'm trying to have a table called Roles that has RoleId and UserId, the actual role name should be in a separate table called RoleTypes with RoleId and Name.

1) How do you set that up?

2) At what point do I insert static values for the role types. Say my app needs 5 roles which are always static. How would you configure this.

Thank you again.

re: Entity Association Mapping with Entity Framework Code First CTP5 Part 1: Complex Types

Thursday, December 23, 2010 1:10 PM by Frank

Looks like my original is working where I have `HasMany(u => u.Roles).WithMany(r => r.Users)`. The only issue is when I create a new user twice with the same role, I do

```
User user = new User()
user.Roles.Add(new Role() { Name = "RoleX" })

User user = new User()
user.Roles.Add(new Role() { Name = "RoleX" })
```

I need the Roles table to have one row for RoleX and have UserRoles to have two rows pointing to the same RoleId. Roles should be a static table with RoleX as a unique value.

re: Entity Association Mapping with Entity Framework Code First CTP5 Part 1: Complex Types

Saturday, December 25, 2010 11:12 AM by [mortezaam](#)

@Frank: I've updated the object model above and now it handles all your requirements. Please note that the only reason I've started using fluent API for the many-to-many association between User and Role is to customize the join table's name to be "Roles". For that I also had to change the name of the table for Role entity to avoid name collision. In addition, the following code snippet shows how you can add multiple Users for a particular Role which the Role is new as well:

```
using (var context = new CTP5Context())
{
    RoleType roleType = context.RoleType.Single(rt => rt.RoleName.Equals("Role1"));
    Role role = new Role()
    {
        RoleType = roleType
    };

    User user1 = new User();
    user1.Roles.Add(role);
    User user2 = new User();
    user2.Roles.Add(role);

    context.Users.Add(user1);
    context.Users.Add(user2);
    context.SaveChanges();
}
```

Regarding to your first question, as you can see in the object model above, I set up a one-to-one *foreign key* association between RoleType and Role entities. About your second question, probably the best way to populate RoleTypes table is to override the Seed() method:

```
public class DatabaseInitializer : DropCreateDatabaseIfModelChanges<CTP5Context>
{
    protected override void Seed(CTP5Context context)
    {
        RoleType roleType = new RoleType()
        {
            RoleName = "Role1"
        };
    };
}
```



```

context.RoleType.Add(roleType);
context.SaveChanges();

int result1 = context.Database.SqlCommand("ALTER TABLE Role ADD CONSTRAINT
uc_RoleTypeId UNIQUE(RoleTypeId)");
int result2 = context.Database.SqlCommand("ALTER TABLE RoleTypes ADD CONSTRAINT
uc_RoleName UNIQUE(RoleName)");
}
}

```

I also want to point out that in the Seed method, I took advantage of the new CTP5's SqlCommand method on DbContext.Database which allows raw SQL commands to be executed against the database. Using that, I defined a unique constraint on RoleName column in RoleTypes table as per your requirement and also another one on RoleTypeId FK in Role table to make sure that the relationship between Role and RoleType will be remained as one-to-one. Hope this helps :)

[# re: Entity Association Mapping with Entity Framework Code First CTP5 Part 1: Complex Types](#)

Sunday, December 26, 2010 10:56 PM by Frank

Thank you for taking the time to do this. I appreciate it. The only problem I have is the manual SqlCommand logic. To me, a big benefit with CF is the ability to switch out to another db fast. With manual SqlCommand stuff, it is now tightly coupled to SQL Server. Just wondering if there's a better approach.

[# re: Entity Association Mapping with Entity Framework Code First CTP5 Part 1: Complex Types](#)

Monday, December 27, 2010 12:01 PM by Frank

Another point. Would a TPH scenario work better may be? I was thinking may be create an

AdminRole : Role that doesn't add any additional properties but then the discriminator would act as the key of what type of role the user would have. Thoughts about my two comments?

[# re: Entity Association Mapping with Entity Framework Code First CTP5 Part 1: Complex Types](#)

Monday, December 27, 2010 3:51 PM by [mortezam](#)

@Frank: The reason that I put the code for creating unique constraints in the Seed method is because CTP5 (and EF in general) does not natively support one-to-one FK associations. It's likely to be supported in the next RTM and until then, you can manually add the constraints to your DB if you don't like the idea of doing it with SqlCommand method.

Regarding your second point, I don't think introducing inheritance in this scenario would be a good idea because defining 5 new subclasses inheriting from Role class without introducing any specialized attribute or behavior is a bit overkill and also won't scale well (e.g. consider a scenario that you want to add more RoleTypes in the future). In addition, by using TPH (or any other inheritance mapping strategy for that matter) you are technically introducing *polymorphic associations* (an association to a base class, hence to all classes in the hierarchy with dynamic resolution of the concrete class at runtime) in your object model. In other words, User will have a polymorphic association to an abstract Role class. The dynamic resolution of the Role Type at runtime means a more complex query being sent to the database and also more work at the time of object materialization which both come with a bit of performance penalty. Polymorphic association is an important topic and I'll explain it in a separate post once I finish my inheritance mapping strategies series. Hope this helps :)

[# re: Entity Association Mapping with Entity Framework Code First CTP5: Part 1 – Complex Types](#)

Friday, December 31, 2010 11:21 AM by Andrew

Excellent post and also some very helpful comments here. Keep it up!

re: Entity Association Mapping with Entity Framework Code First CTP5: Part 1 – Complex Types

Tuesday, January 04, 2011 11:22 AM by [Daniel](#)

Excellent post! I am new to EF code first and have been banging my head against the wall trying to setup a relationship between four tables without putting data in all the tables every time I need to update one table. I think complex types is what I have been missing. Please keep up the good work. This is highly appreciated.

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Monday, January 24, 2011 7:40 PM by [milanCHE](#)

Thx, for the great post! The only problem I have is to set complex type as primary key in code first development. For example Key has Id, Name, RepositoryName and TypeName and every strong object has key. Help please!

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Monday, January 24, 2011 11:33 PM by [mortezaam](#)

@milanCHE: Complex Types are not meant to be used in this scenario and they wouldn't be able to handle that. Like I said in the post they are only for mapping a special type of one-to-one association (i.e. composition) between objects, period. If your intention by doing that is to reuse the key properties instead of repeating them in every single entity, then I recommend using inheritance instead of a complex type in your object model. That inheritance would be best represented by a [Table per Concrete Type \(TPC\)](#) strategy. For example, consider the following implementation for your scenario:

```
public abstract class EntityBase
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string RepositoryName { get; set; }
    public string TypeName { get; set; }
}

public class User : EntityBase
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Username { get; set; }
}

public class Address : EntityBase
{
    public string Street { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
}

public class EntityMappingContext : DbContext
{
    public DbSet<EntityBase> Entities { get; set; }
}
```

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<EntityBase>().HasKey(e => new
    {
        e.Id,
        e.Name,
        e.RepositoryName,
        e.TypeName
    });

    modelBuilder.Entity<User>().Map(m =>
    {
        m.MapInheritedProperties();
        m.ToTable("Users");
    });

    modelBuilder.Entity<Address>().Map(m =>
    {
        m.MapInheritedProperties();
        m.ToTable("Addresses");
    });
}
}

```

As a result every concrete subclass will end up having their own table with key columns defined in EntityBase. Hope this helps :)

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Tuesday, January 25, 2011 6:50 AM by [MilanCHE](#)

Thanks for the response, very much.

I need this for passing keys, for example in XAML

```
{Binding Key}
```

But I can extend EntityBase with read only Key property

```

public abstract class EntityBase
{
    ...

    [NotMapped]
    public Key Key
    {
        get { return this;}
    }

    public static implicit operator Key(EntityBase input)
    {
        Key k = new Key();
        k.Id = input.Id;
        k.Name = input.Name;
        k.TypeName = input.TypeName;
        k.RepositoryName = input.RepositoryName;
        return k;
    }
}

```

```

    }
}

```

I have one questions for now, if you do not mind:

If I want to implement week reference for association as Complex type, for example:

```

public class Reference<T> where T : EntityBase
{
    public Reference(T input)
    {
        this.Id = input.Id;
        this.Name = input.Name;
        this.TypeName = input.TypeName;
        this.RepositoryName = input.RepositoryName;
        Value = input;
    }

    public string Id { get; set; }
    public string Name { get; set; }
    public string TypeName { get; set; }
    public string RepositoryName { get; set; }

    [XmlIgnore][NotMapped]
    public T Value {get;set;}
}

public class User : EntityBase
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Username { get; set; }
    public Reference<Address> Address{ get; set; }
}

public class Address : EntityBase
{
    public string Street { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
}

```

How to configure this, but in db to get table for User, with columns :

```

Id
Name
TypeName
RepositoryName
FirstName
LastName
Username
Address_Id
Address_Name
Address_TypeName
Address_RepositoryName

```

Once again I appreciate your help and the time you have spent posting these articles. They are a HUGE help for me.

Milan

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Tuesday, January 25, 2011 9:59 PM by [morteza](#)

@Milan: I can see that you've used *EntityBase* as a base class for both an entity (e.g. User) and a

complex type (e.g. Address). In this case we need to modify my inheritance solution a little bit (the way you defined Address property as type of Reference<Address> on User class is not going to work.):

```
public abstract class EntityBase
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string RepositoryName { get; set; }
    public string TypeName { get; set; }
}

public class User : EntityBase
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Username { get; set; }

    public Address Address { get; set; }
}

public class Address : EntityBase
{
    public string Street { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
}

public class Context : DbContext
{
    public DbSet<User> Users { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>().HasKey(e => new { e.Id, e.Name, e.RepositoryName, e.TypeName });

        modelBuilder.ComplexType<Address>().Property(a => a.Id).HasColumnName("Address_Id");
        modelBuilder.ComplexType<Address>().Property(a =>
a.Name).HasColumnName("Address_Name");
        modelBuilder.ComplexType<Address>().Property(a =>
a.RepositoryName).HasColumnName("Address_RepositoryName");
        modelBuilder.ComplexType<Address>().Property(a =>
a.TypeName).HasColumnName("Address_TypeName");

        modelBuilder.ComplexType<Address>().Property(a =>
a.Street).HasColumnName("Address_Street");
        modelBuilder.ComplexType<Address>().Property(a => a.City).HasColumnName("Address_City");
        modelBuilder.ComplexType<Address>().Property(a =>
a.PostalCode).HasColumnName("Address_PostalCode");
    }
}
```

This gives you the desired schema. Basically we used EntityBase as a complex type for Address which is also a complex type by itself. However, the way we changed the column names for Address properties (e.g. Address_Street for Address.Street) doesn't seem to be very elegant. There is a way that we can

refactor this code: CTP5 introduced a new feature called *Pluggable conventions* that allows us to override the “default conventions” that EF Code First honors, and optionally replace them with our own set of conventions. For example, we can use this feature to override the convention for complex types and prefix their DB column names with the Complex Type name. The following code shows how this can be done (only our fluent API codes in the DbContext would be changed to the following):

```
public class Context : DbContext
{
    public DbSet<User> Users { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>().HasKey(e => new { e.Id, e.Name, e.RepositoryName, e.TypeName });

        modelBuilder.ComplexType<Address>().Property(a => a.Id).HasColumnName("Address_Id");
        modelBuilder.ComplexType<Address>().Property(a =>
a.Name).HasColumnName("Address_Name");
        modelBuilder.ComplexType<Address>().Property(a =>
a.RepositoryName).HasColumnName("Address_RepositoryName");
        modelBuilder.ComplexType<Address>().Property(a =>
a.TypeName).HasColumnName("Address_TypeName");

        modelBuilder.Conventions.Add(new MyComplexTypeConvention(typeof(Address)));
    }
}

public class MyComplexTypeConvention : IConfigurationConvention<PropertyInfo,
PrimitivePropertyConfiguration>
{
    private Type[] knownComplexTypes;

    public MyComplexTypeConvention(params Type[] complexTypes)
    {
        this.knownComplexTypes = complexTypes;
    }

    public void Apply(PropertyInfo memberInfo, Func<PrimitivePropertyConfiguration> configuration)
    {
        if (this.knownComplexTypes.Contains(memberInfo.DeclaringType))
        {
            var config = configuration.Invoke();
            config.ColumnName = string.Format("{0}_{1}", memberInfo.DeclaringType.Name,
memberInfo.Name);
        }
    }
}
```

By the way, thanks for your comment regarding the articles, I'm glad you find them helpful :)

FreeSNAP, Day 12: More Partial Views, and EFCodeFirst | The Fae Magic of Programming

Friday, February 11, 2011 11:09 PM by [FreeSNAP, Day 12: More Partial Views, and EFCodeFirst | The Fae Magic of Programming](#)

Pingback from [FreeSNAP, Day 12: More Partial Views, and EFCodeFirst | The Fae Magic of Programming](#)

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Thursday, February 17, 2011 4:43 AM by Nekketsu

Is it possible to automatically prefix complex types table name with the name of variable instead of name of type? I mean:

```
public class User
{
    Address DeliveryAddress;
    Address HomeAddress;
}
```

And I want to generate automatically the names for table fields:

```
DeliveryAddress_Street
DeliveryAddress_City
DeliveryAddress_PostalCode
HomeAddress_Street
HomeAddress_City
HomeAddress_PostalCode
```

Thank you!

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Thursday, February 17, 2011 12:33 PM by [morte zam](#)

@Nekketsu: For now this can be achieved by using fluent API:

```
public class EntityMappingContext : DbContext
{
    public DbSet<User> Users { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new UserConfiguration());
    }
}

class UserConfiguration : EntityTypeConfiguration<User>
{
    public UserConfiguration()
    {
        Property(u => u.DeliveryAddress.Street).HasColumnName("DeliveryAddress_Street");
        Property(u => u.DeliveryAddress.City).HasColumnName("DeliveryAddress_City");
        Property(u => u.DeliveryAddress.PostalCode).HasColumnName("DeliveryAddress_PostalCode");

        Property(u => u.HomeAddress.Street).HasColumnName("HomeAddress_Street");
        Property(u => u.HomeAddress.City).HasColumnName("HomeAddress_City");
        Property(u => u.HomeAddress.PostalCode).HasColumnName("HomeAddress_PostalCode");
    }
}
```

I realize that this code is not very elegant and could become tedious and buggy if there are so many complex types out there. The EF team is looking into ways to enable us achieving this with *Pluggable Conventions* but they are still very early in the implementation process. Hope this helps.

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Friday, February 18, 2011 1:35 AM by Jack

How is it that the complex type column naming (as per your comment directly prior) used to work just fine in CTP4? And now its broken?

It really destroys faith in the EF team when things constantly regress.

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Friday, February 18, 2011 11:21 AM by [morteza](#)

@Jack: I admit that CTP4 had a better naming convention when it comes to complex type's column names and I understand the source of your frustration in this regard, however, you should also take this into account that these are merely CTPs and being released just to get an early feedback during development. Therefore, it is normal that the API get changed from time to time as it is not final yet. In the case of CTP5, they did some large refactoring on the relationship API and they didn't re-enable everything in time for CTP5 so you can find few inconsistencies and limitations comparing to CTP4.

Entity Framework complex types « enthudev

Saturday, February 19, 2011 4:03 PM by [Entity Framework complex types « enthudev](#)

Pingback from Entity Framework complex types « enthudev

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Friday, April 15, 2011 5:12 PM by [ryanse](#)

If you are interested in using the Entity Framework with Domain Driven Design, checkout my new "Domain Driver" framework on CodePlex.

This framework provides developers with a way to quickly and easily implement a full domain model, to test that domain model using pre-built generic tests, and to prototype user-interface components against that domain model to help achieve customer validation. Initially, the domain model will function as an in-memory database, but at any point developers can add the ability to persist to a database, a file, or any other non-volatile data store.

Domain Driver itself is decoupled from any persistence technology, but I implemented an example that clearly shows how to use it with EF Code-First to accomplish Database persistence.

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Wednesday, May 18, 2011 8:32 AM by dencio

Thanks. Great post.

Profil sayfas?? ki??iselle??tirme ve ComplexType | asp.net, jquery ve di??er web teknolojileri ??zerine

Wednesday, June 29, 2011 5:41 AM by [Profil sayfas?? ki??iselle??tirme ve ComplexType | asp.net, jquery ve di??er web teknolojileri ??zerine](#)

Pingback from Profil sayfas?? ki??iselle??tirme ve ComplexType | asp.net, jquery ve di??er web teknolojileri ??zerine

How to create an inherited class when the base comes from an EF4.1 DbContext? - Programmers Goodies

Wednesday, September 14, 2011 12:05 AM by [How to create an inherited class when the base comes from an EF4.1 DbContext? - Programmers Goodies](#)

Pingback from [How to create an inherited class when the base comes from an EF4.1 DbContext?](#) - Programmers Goodies

re: Associations in EF Code First CTP5: Part 1 – Complex Types

Thursday, October 06, 2011 3:03 AM by Jviaches

Great post ! Simple and clear explain regarding Code First approach.

Thank you!

[Terms of Use](#)

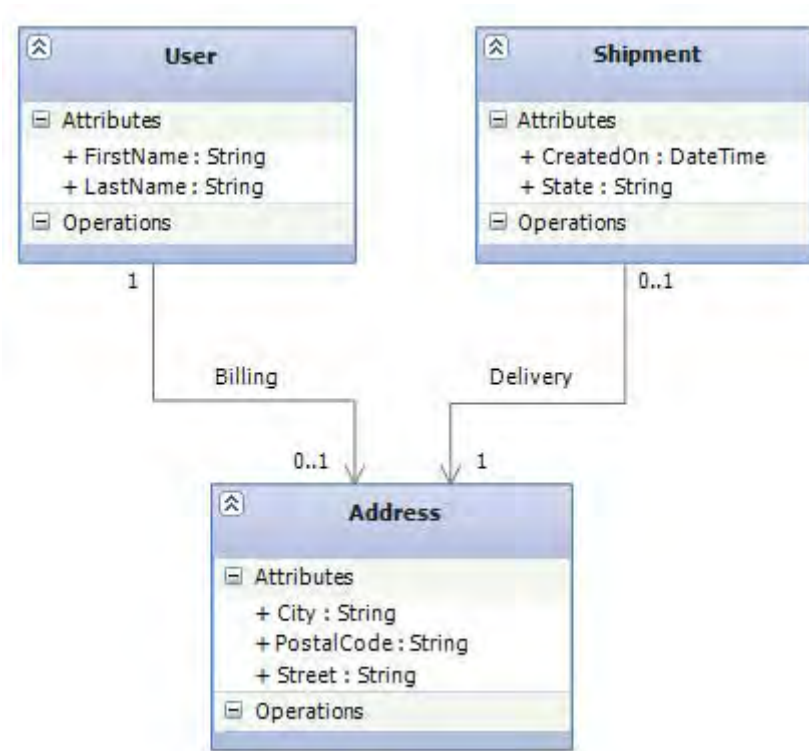
Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

In the previous [blog post](#) I demonstrated how to map a special kind of one-to-one association—a complex types as the first post in a series about entity association mapping with EF Code First. We relationships between User and Address are best represented with a complex type mapping and we usually the simplest way to represent one-to-one relationships but comes with some limitations.

In today's blog post I'm going to discuss how we can address those limitations by changing our map is particularly useful for scenarios that we want a dedicated table for Address, so that we can map to Address as entities. One benefit of this model is the possibility for *shared references*— another entity (Shipment) can also have a reference to a particular Address instance. If a User has a reference to her BillingAddress, the Address instance has to support shared references and needs its own identity. User and Address classes have a true *one-to-one* association.

Introducing the Revised Model

In this revised version, each User *could* have one BillingAddress (Billing Association). Also Shipmer delivered to an address so it *always* has one Delivery Address (Delivery Association). Here is the class diagram for this domain model (note the multiplicities on association lines):



In this model we assumed that the billing address of the user is the same as her delivery address. Next we'll look at the association mappings for this domain model. There are several choices, the first being a *One-to-One Association*.

Shared Primary Associations

Also known as *One-to-One Primary Key Associations*, means two related tables share the same primary key. The primary key of one table is also a foreign key of the other. Let's see how we map the primary key with Code First.

How to Implement a One-to-One Primary Key Association with Code First

First, we start with the POCO classes. As you can see, we've defined *BillingAddress* as a navigation class and another one on *Shipment* class named *DeliveryAddress*. Both associations are unidirectional. We didn't define related navigation properties on *Address* class as for *User* and *Shipment*.

```

public class User
{
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual Address BillingAddress { get; set; }
}

public class Address
{
    public int AddressId { get; set; }
}
  
```

```

    public string Street { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
}

public class Shipment
{
    public int ShipmentId { get; set; }
    public DateTime CreatedOn { get; set; }
    public string State { get; set; }
    public virtual Address DeliveryAddress { get; set; }
}

public class EntityMappingContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Address> Addresses { get; set; }
    public DbSet<Shipment> Shipments { get; set; }
}

```

How Code First Reads This Object Model: One-to-Many

Code First reads the model and tries to figure out the multiplicity of the associations. Since the associations are unidirectional, Code First takes this as if one Address has many Users and many Shipments and will create a *one-to-many* association for each of them. So, what we were hoping for — a one-to-one association — is not the convention.

How to Change the Multiplicity to One-to-One by Using the Conventions

One way to turn our associations to be one-to-one is by making them bidirectional. That is, adding a property to the Address class of type User and another one of type Shipment. By doing that we basically tell Code First that we are looking to have one-to-one associations since for example User has an Address and Address has a User. Based on the conventions, Code First will change the multiplicity to one-to-one and this will solve the problem.

Should We Make This Association Bidirectional?

As always, the decision is up to us and depends on whether we need to navigate through our objects in the application code. In this case, we'd probably conclude that the bidirectional association doesn't make sense. If we call *anAddress.User*, we are saying "give me the user who has this address", not a very useful request. So this is not a good option. Instead we'll keep our object model as it is and will resort to fluent API.

How to Change the Multiplicity to One-to-One with Fluent API

The following code is all that is needed to make the associations to be one-to-one. Note how the multiplicity in the UML class diagram (e.g. 1 on User and 0..1 on address) has been translated to the fluent API code using *HasRequired* and *HasOptional* methods:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{

```

```

modelBuilder.Entity<User>().HasOptional(u => u.BillingAddress)
    .WithRequired();

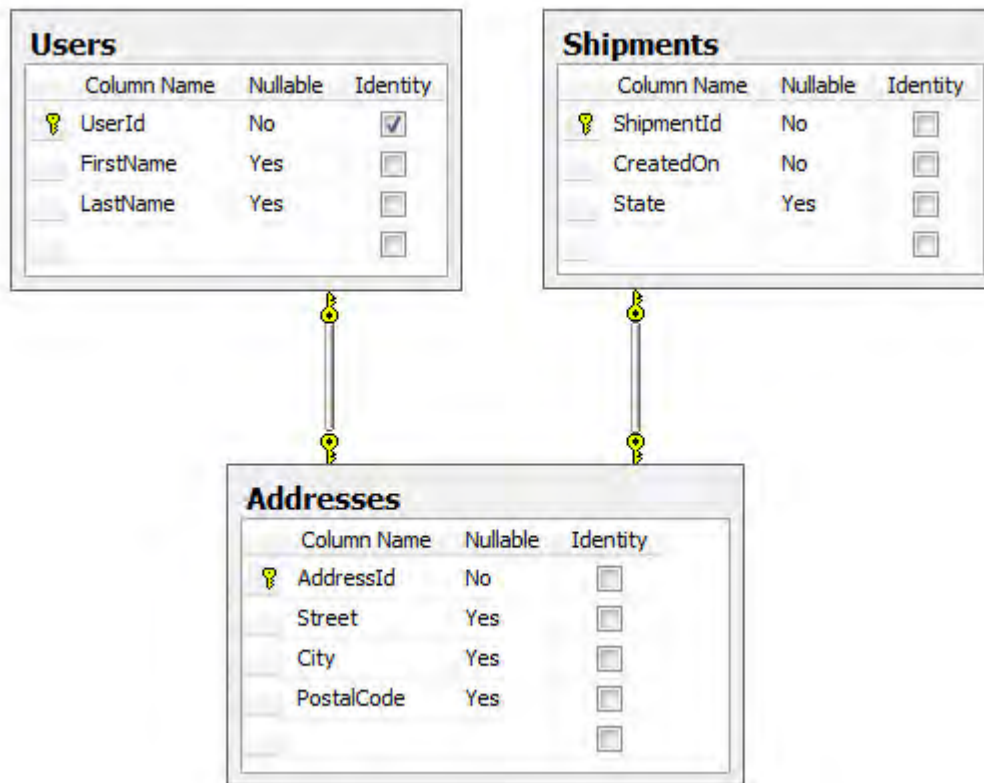
modelBuilder.Entity<Shipment>().HasRequired(u => u.DeliveryAddress)
    .WithOptional();
}

```

Also it worth mentioning that in CTP5, when we are mapping a one-to-one association with fluent API to specify the foreign key as we would do when mapping a one-to-many association with *HasForeign*. Since EF only supports one-to-one primary key associations it will automatically create the relations database based on the primary keys and we don't need to state the obvious as we did in CTP4.

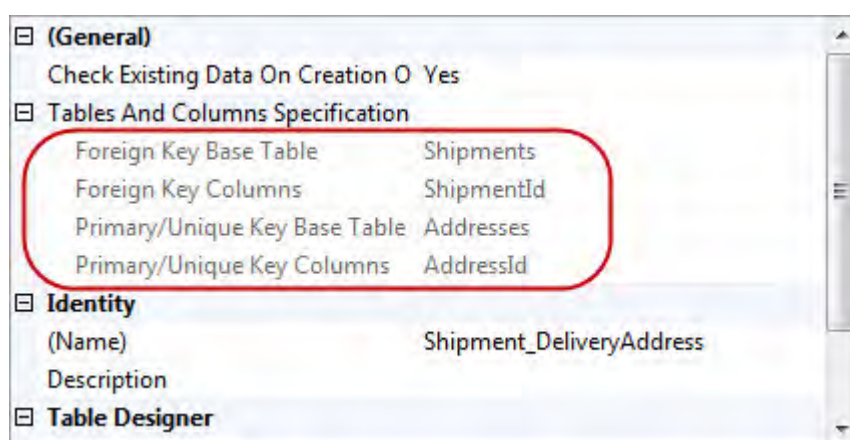
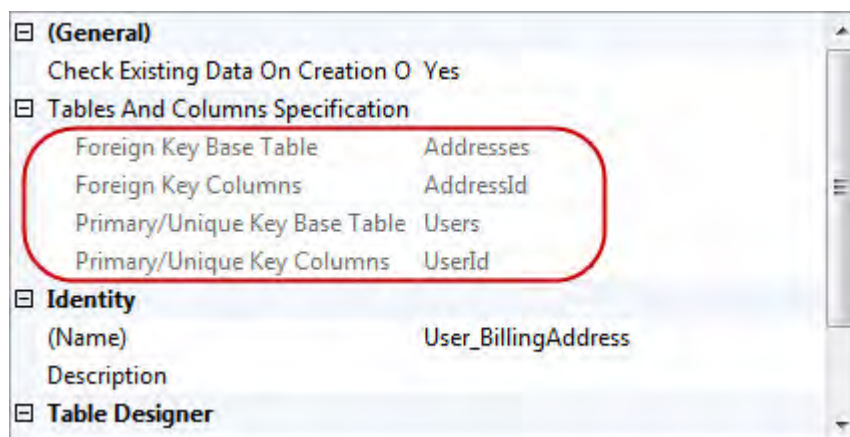
Database Schema

The mapping result for our object model is as follows (note the Identity column):



Referential Integrity

In relational database design the referential integrity rule states that each non-null value of a foreign key must be the value of some primary key. But wait, how does it even apply here? All we have is just three primary keys referencing each other. Who is the primary key and who is the foreign key? The best way to find the answer to the question is to take a look at the properties of the relationships in the database that has been created.



As you can see, Code First adds a foreign key constraint which links the primary key of the Address table to the primary key of the Users table and adds another foreign key constraint that links the primary key of the Shipments table to the primary key of the Addresses table. The foreign key constraint means that a user has a particular address but not the other way around. In other words, the database guarantees that an Address primary key references a valid Users primary key and a Shipments row's primary key references a valid Address primary key.

How Code First Determines Principal and Dependent?

Code First has rules to determine the principal and dependent ends of an association. For one-to-many the many end is always the dependent, but it gets a little tricky in one-to-one associations. In one-to-one Code First decides based on our object model, and possible data annotations or fluent API that we use. In our example in our case, we wrote this fluent API code to configure User-Address association:

```
modelBuilder.Entity<User>().HasOptional(u => u.BillingAddress).WithRequired();
```

This reads as "User entity has an optional association with one Address object but this association is required on the Address entity."

For Code First this is good enough to make the decision: It marked User as the principal end and Address as the dependent end in the association. Since we have the same fluent API code for the second association between Address and Shipment, it marks Address as the principal end and Shipment as the dependent end in the association as well.

The referential integrity that we saw, is the first result of this Code First's principal/dependent decisio

Second Result of Code First's Principal/Dependent Decision: Database Identity

If you take a closer look at the above DB schema, you'll notice that only `UserId` has a regular identifi (*Identity or Sequence*) and `AddressId` and `ShipmentId` does not. This is a very important consequent principal/dependent decision for one-to-one associations: *the dependent primary key will become a default*. This makes sense because they share their primary key values and only one of them can be and we need to take care of providing valid keys for the rest.

What about Cascade Deletes?

As we saw, each `Address` always belongs to one `User` and each `Shipment` always delivered to one `User`. We want to make sure that when we delete a `User` the possible dependent rows on `Address` and `Shipment` are deleted in the database. In fact, this is one of the [Referential Integrity Refactorings](#) which called *Introduce Cascading Delete*. The primary reason we would apply "Introduce Cascading Delete" is to preserve the referential data by ensuring that related rows are appropriately deleted when a parent row is deleted. By default, EF does not enable cascade delete when it creates a relationship in the database. As always we can override this with the fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>().HasOptional(u => u.BillingAddress)
        .WithRequired()
        .WillCascadeOnDelete();

    modelBuilder.Entity<Shipment>().HasRequired(u => u.DeliveryAddress)
        .WithOptional()
        .WillCascadeOnDelete();
}
```

What If Both Ends are Required?

We saw that the only reason Code First could figure out principal and dependent in our 1:1 association was that our fluent API code clearly specified one end as Required and the other as Optional. But what if both ends are required in the association? For example, what if in our domain model, `User` always has one `Address` and `Address` always has one `User` (required on both ends)? The answer is that ultimately, the relationship needs to be configured by the fluent API and the interesting point is that the fluent API is designed in a way that allows us to explicitly specify who is principal and who is dependent in such cases that this cannot be inferred.

To illustrate the idea, let's see how we can configure mapping for this `User-Address` association (Required on both ends) with the fluent API:

```
modelBuilder.Entity<User>().HasRequired(u => u.BillingAddress).WithRequiredDependent();
```

So we invoke `WithRequiredDependent()` after `HasRequired()` method. To see the reason, we need to look at the `RequiredNavigationPropertyConfiguration` type which is returned by `HasRequired()`:

```
public class RequiredNavigationPropertyConfiguration<TEntityType, TTargetEntityType>
{
    public DependentNavigationPropertyConfiguration<TEntityType, TTargetEntityType>
```



```

    public CascadableNavigationPropertyConfiguration WithOptional();
    public CascadableNavigationPropertyConfiguration WithRequiredDependent();
    public CascadableNavigationPropertyConfiguration WithRequiredPrincipal();
}

```

As you can see, if you want to go another Required after HasRequired() method, you have to either WithRequiredDependent() or WithRequiredPrincipal() since there is no WithRequired() method in the RequiredNavigationPropertyConfiguration class which is returned by HasRequired() method. Both WithRequired and WithOptional methods return a CascadableNavigationPropertyConfiguration which has a WillCascadeOnDelete() method. Now if we run the code and check the database, we can see that cascade delete on both relationships are switched on.

Working with the Model

Here is an example for adding a new user along with its billing address. EF is smart enough to use the generated UserId for the AddressId as well:

```

using (var context = new EntityMappingContext())
{
    Address billingAddress = new Address()
    {
        Street = "Yonge St.",
        City = "Toronto"
    };
    User morteza = new User()
    {
        FirstName = "Morteza",
        LastName = "Manavi",
        BillingAddress = billingAddress
    };

    context.Users.Add(morteza);
    context.SaveChanges();
}

```

The following code is an example of adding a new Address and Shipment for an existing User (assume a User with UserId=2 in the database):

```

using (var context = new EntityMappingContext())
{
    Address deliveryAddress = new Address()
    {
        AddressId = 2,
        Street = "Main St.",
        City = "Seattle"
    };
    Shipment shipment = new Shipment()
    {
        ShipmentId = 2,
        State = "Shipped",
        CreatedOn = DateTime.Now,
        DeliveryAddress = deliveryAddress
    };
}

```

```

};

context.Shipments.Add(shipment);
context.SaveChanges();
}

```

Limitations of This Mapping

There are two important limitations to associations mapped as shared primary key:


- **Difficulty in saving related objects:** The main difficulty with this approach is ensuring that related instances are assigned the same primary key value when the objects are saved. For example, if you create a new Address object, it's our responsibility to provide a unique AddressId that is also valid (and not in conflict with such a value as UserId.)
- **Multiple addresses for User is not possible:** With this mapping we cannot have more than one address for a User. At the beginning of this post, when we introduced our model, we assumed that the user would have one address for billing and delivery. But what if that's not the case? What if we also want to add a User for the deliveries? In the current setup, each row in the User table has a corresponding row in the Address table. Two addresses would require an additional address table, and this mapping style there is not adequate.

Summary

In this post we learned about one-to-one associations which shared primary key is just one way to implement one-to-one associations. Shared primary key associations aren't uncommon but are relatively rare. In many schemas, a one-to-one association is represented with a foreign key field and a unique constraint. In the next posts we will revisit the same topic and will learn about other ways to map one-to-one associations that does not have the limitations of shared primary key association mapping.

References

- [ADO.NET team blog](#)
- [Java Persistence with Hibernate book](#)

Published Sunday, December 19, 2010 3:09 AM by [morteza](#) 
 Filed under: [Entity Framework](#), [C#](#), [Code First](#), [CTP5](#), [.NET](#)

Comments

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Saturday, December 18, 2010 11:23 PM by [rickj1](#)

Great article! I can't get enough of code first. I have a problem and have been looking for a solution, it's with an e-commerce database that has a Product table and a Category table, it also has a ProductCategory table with columns ProductID (ASC), CategoryID (ASC), FK_ProductCategory_Category

Tables And Columns Specification:

Foreign Key Base Table: ProductCategory

Foreign Key Columns: CategoryID

Primary/Unique Key Base Table: Category
Primary/Unique Key Columns: CategoryID
And the same for ProductID. My problem is how do you work with such a table in code first?

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Sunday, December 19, 2010 12:46 AM by [Paul](#)

Where was this post 7 hours ago when I was moving my repo to cpt5!??? Great info and immediately applicable in real world aps.

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Sunday, December 19, 2010 6:37 PM by Designation_One

Interesting post !

Took me a moment to grok what you're doing here. The db-schema made me scratch my head. (You explained it later though)

Linking UserId to AddressId 'looks' wrong to me. I would at least rename AddressId to UserId, because that's what it really is, but that's just my opinion.

Just discovered your blog through Twitter, looking forward for your next posts !

Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Sunday, December 19, 2010 10:25 PM by [progg.ru](#)

Thank you for submitting this cool story - Trackback from progg.ru

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Sunday, December 19, 2010 11:51 PM by [mortezam](#)

@Designation_One: In database design, a one-to-one relationship on primary keys usually represents *Entity Splitting* (splitting a single entity into two or more tables), in that scenario choosing one name for the primary keys makes the best sense and is recommended (e.g. Individual and Customer tables in AdventureWorks DB where both have a CustomerID column as their PK). However, we're dealing with a different situation here: Users, Addresses and Shipments tables are mapping back to three completely *different* entities that just happen to share their primary key values in this particular type of mapping. Like I said in the post, shared primary key associations are relatively rare and a one-to-one relationship is usually represented with a foreign key field and a unique constraint. Thanks :)

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Monday, December 20, 2010 2:00 PM by [mortezam](#)

@ rickj1: Thanks! About your question, it's a classic model of a many-to-many association. Code First maps this by creating a join table (i.e. ProductCategory) that has a one-to-many relationship with each Product and Category tables. I will explain this association type in my future posts but for now an object model like this will do the trick:

```
public class Product
{
```

```

public int ProductId { get; set; }
public virtual ICollection<Category> Categories { get; set; }
}
public class Category
{
public int CategoryID { get; set; }
public virtual ICollection<Product> Products { get; set; }
}

```

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Monday, December 20, 2010 2:50 PM by [Koistya `Navin](#)

Having AddressId and ShipmentId keys might be confusing. Why not to call them just UserId?

User (UserId, FirstName, LastName)

Address (UserId, Street, City, PostalCode)

Shipment (UserId, CreatedOn, State)

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Monday, December 20, 2010 3:19 PM by [Koistya `Navin](#)

Is there a way to make one-to-one unidirectional association work with conventions? For example by adding the following properties to Address entity:

```
protected virtual User User { get; set; }
```

```
protected virtual Shipment Shipment { get; set; }
```

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Monday, December 20, 2010 9:16 PM by [mortezam](#)

@Koistya `Navin: If you add navigation properties for User and Shipment to Address type then your associations are going to be bidirectional and yes, Code First will change the multiplicity to one-to-one like I explained in the post under the title "*How to Change the Multiplicity to One-to-One by using the Conventions*". If you want to keep your associations unidirectional then fluent API is the *only* way to make them one-to-one. Regarding your first question, choosing different names for primary keys are intentional. Please read my reply to *Designation_One* above. Thanks :)

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Tuesday, December 21, 2010 4:16 AM by steve

Know this is just an example to illustrate how, but one other limitation must be that a user can only have 1 shipment, or am I missing something.

Great post though, looking forward to this coming out of ctp

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Tuesday, December 21, 2010 7:36 PM by Yusuf Demirag

First of all thank you for this great post that you covered a very common real-world scenario having shared primary keys for one-to-one relationships.

My question is, when one design a database with such association, it is also very common to set cascade delete operation in database itself. So to say, when I delete a User, it automatically deletes the Address, and that deletion of Address cascades to Shipment. Is this automatically handled or will it throw foreign key violation exception? If it is not handled automatically is there a way to tell this to model binder? Thanks.

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Wednesday, December 22, 2010 2:32 PM by [mortezam](#)

@Yusuf Demirag: In CTP5, by default Code First does *not* enable cascade delete when it creates a relationship in the database. As a result, if you try to delete a User, you'll get a `SQLException` because of the foreign key constraint violation if an Address holds a reference to that particular User. We can enable cascade delete by chaining `WillCascadeOnDelete()` method at the end of our fluent API code. I've added a new section to the post and explained how to do this. Please find it under the title "*What about Cascade Deletes?*". Thanks for your great question!

re: Entity Association Mapping with Code First Part 2: One-to-One Shared Primary Key Associations

Wednesday, December 22, 2010 2:56 PM by [mortezam](#)

@Steve: Like I explained in the post under the *Referential Integrity* title, the database guarantees that there is always one user exists for an address and also one address always exists for a Shipment because of the foreign key constraints that links the primary keys. Therefore, if you have a Shipment, it holds a reference to exactly one Address and that particular Address always holds a reference to one User. Hence, a User can only have one Shipment. Thanks :)

re: Entity Association Mapping with Entity Framework Code First CTP5: Part 2 – One-to-One Shared Primary Key Associations

Saturday, January 01, 2011 3:02 PM by [whoever](#)

I'm wondering if it's possible to do entity splitting in the following scenario.

```
class {Id; ItemId; ItemName; ItemDisplayName}
```

```
MainTable {Id, ItemId, ItemName}
```

```
OptionalTable{ ItemId, ItemDisplayName}
```

Some of the items have optional display names that need to be used if present. In SQL term, it's MainTable left join OptionalTable on ItemId, so you have something like this

```
Id ItemId ItemName ItemDisplayName
```

```
1 aaa NameA
```

```
2 bbb NameB DisplayNameB
```

```
3 ccc NameC
```

How do I map the class to the two underlying table? Thanks

re: Entity Association Mapping with Entity Framework Code First CTP5: Part 2 – One-to-One Shared Primary Key Associations

Saturday, January 01, 2011 5:38 PM by [mortezam](#)

@whoever: Yes, EF Code First fully supports Entity Splitting. In CTP5, you can make use of the *EntityTypeConfiguration* class that you access through the *Map()* method to split an entity across multiple tables:

```
public class FoolItem
{
    public int Id { get; set; }
    public string ItemId { get; set; }
    public string ItemName { get; set; }
    public string ItemDisplayName { get; set; }
}

public class CTP5Context : DbContext
{
    public DbSet<FoolItem> FoolItems { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<FoolItem>()
            .Map(m =>
            {
                m.Properties(p => new
                {
                    p.Id,
                    ItemId = p.ItemId,
                    p.ItemName
                });
                m.ToTable("MainTable");
            })
            .Map(m =>
            {
                m.Properties(p => new
                {
                    p.Id,
                    ItemId = p.ItemId,
                    p.ItemDisplayName
                });
                m.ToTable("OptionalTable");
            });
    }
}
```

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Wednesday, January 12, 2011 3:11 PM by [whoever](#)

@mortezaam. Thanks for the update. The OptionalTable does not have field Id, it linked to MainTable by ItemId only.

When I try as you suggested, without p.Id in the second mapping. I got "Schema specified is not valid. Errors: (0,0) : error 0005: Root element is missing"

I added HasKey(x=>x.Id) in first mapping and HasKey(x=>x.ItemId) in the second mapping and try again, I got

(40,10) : error 3007: Problem in mapping fragments starting at lines 26, 40:Column(s) [ItemDisplayName]

are being mapped in both fragments to different conceptual side properties."

Any suggestions? Thanks.

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Wednesday, January 12, 2011 3:29 PM by [whoever](#)

To be accurate, because all my tables have different name than the classes, so my mapping is like this

```
.Map(m =>
{
    m.Properties(p => new
    {
        A_Id = p.Id,
        A_ItemId = p.ItemId,
        A_ItemName = p.ItemName
    });
    m.ToTable("MainTable");
    HasKey(x=> x.Id);
})

.Map(m =>
{
    m.Properties(p => new
    {
        B_ItemId = p.ItemId,
        B_ItemDisplayName = p.ItemDisplayName
    });
    m.ToTable("OptionalTable");
    HasKey(x=> x.ItemId);
});
```

Not sure if any of these extras cause the problem. I was hoping this will generate the equivalent of
SELECT FROM MainTable LEFT JOIN OptionaTable ON ItemId

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Thursday, January 13, 2011 9:24 PM by [mortezaam](#)

@whoever: What you are trying to accomplish is impossible with Entity Splitting. You should be aware that even though you end up getting two tables in this mapping, you still have one single entity (Fooltem) which only can have one primary key and not two (i.e. ItemId and Id). In other words, when mapping an entity to two different tables, you must map the same primary key for both tables (Of course you can rename the primary key column in the second table but it still has to refer the same primary key property). It also worth noting that EF uses INNER JOIN (and not LEFT JOIN) to read the split entity from the database. For example, EF submits the following SQL statements to the database as a result of this query: context.Fooltems.ToList();

```
SELECT
[Extent1].[Id] AS [Id],
[Extent2].[ItemId] AS [ItemId],
[Extent2].[ItemName] AS [ItemName],
[Extent1].[ItemDisplayName] AS [ItemDisplayName]
FROM [dbo].[OptionalTable] AS [Extent1]
INNER JOIN [dbo].[MainTable] AS [Extent2] ON [Extent1].[Id] = [Extent2].[Id]
```

Therefore, Entity splitting is not meant to be used for your particular scenario. Your desired schema would be best achieved by a one-to-one foreign key association which involves two separate entities.

Hope this helps :)

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Monday, January 17, 2011 7:25 AM by Ihar Bury

I'm trying to use shared primary key one-to-one association in our project. In our tests we have FixupE table with required reference to FixupA table via primary key sharing. Entity classes have bidirectional relationship properties.

```
private class ConfigurationE : EntityTypeConfiguration<FixupE>
{
    public ConfigurationE()
    {
        HasKey(e => e.Id);
        Property(e => e.Id)
        .HasDatabaseGenerationOption(DatabaseGenerationOption.None)
        .HasColumnName("id");
        HasRequired(e => e.A)
        .WithOptional(a => a.E);
        ToTable("FixupE", "dbo");
    }
}
```

However, EF treats FixupE as having "FixupAId" column (int not null) referencing FixupA table instead of using shared primary key ("id").

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Monday, January 17, 2011 7:39 AM by Ihar Bury

Ah, got it. We removed *OneToOneConstraintIntroductionConvention*. Should such a thing be a convention really?

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Monday, January 17, 2011 8:58 PM by [mortezaam](#)

@Ihar Bury: Good question! Let me clarify it. I agree that having a convention like *OneToOneConstraintIntroductionConvention* doesn't make a good sense given that EF only supports Shared Primary Keys for one-to-one associations. However, you have to be aware that this is just one way for creating a 1:1 association and EF team is actively working on a new feature that will enable us to create 1:1 associations in another (better) way which is called *One-to-One Foreign Key Associations*. Now let's assume the RTM version ships with this One-to-One Foreign Key association support and Shared Primary Keys still remains as the default mapping for one-to-one associations. As a result, we have to use fluent API every time we want to create a Foreign Key association for our one-to-one relationships. In this case, being able to override this convention by removing this *OneToOneConstraintIntroductionConvention* would be really helpful. Hope this helps :)

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Tuesday, March 08, 2011 10:28 PM by [Victor Ponce](#)

First of all, great article. Just starting on code-first and loving it. I have a problem similar to one you already answered. In the example you gave (below) there are two tables Product and Category. My problem is that I'm trying to map with tables that have prefixes but the classes don't. So in the example you answered, my table names would be tblProduct and tblCategory so I have to use the *ToTable()* method to map them. It's trying to create a FK relationship called ProductCategory and it doesn't use my mapping table to make the association.

@rickj1: Thanks! About your question, it's a classic model of a many-to-many association. Code First maps

this by creating a join table (i.e. *ProductCategory*) that has a one-to-many relationship with each *Product* and *Category* tables. I will explain this association type in my future posts but for now an object model like this will do the trick:

```
public class Product
{
    public int ProductId { get; set; }
    public virtual ICollection<Category> Categories { get; set; }
}

public class Category
{
    public int CategoryId { get; set; }
    public virtual ICollection<Product> Products { get; set; }
}
```

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Thursday, March 10, 2011 11:01 AM by [morteza](#)

@Victor Ponce: Using the *ToTable* method to customize *Product* and *Category* table names wouldn't stop Code First to create the join table to map the many to many association between them but you can still customize the join table name and columns using fluent API like the following:

```
public class Product
{
    public int ProductId { get; set; }
    public virtual ICollection<Category> Categories { get; set; }
}

public class Category
{
    public int CategoryId { get; set; }
    public virtual ICollection<Product> Products { get; set; }
}

public class Context : DbContext
{
    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Product>().ToTable("tblProduct");
        modelBuilder.Entity<Category>().ToTable("tblCategory");

        modelBuilder.Entity<Product>().HasMany(p => p.Categories).WithMany(c => c.Products).Map(c =>
        {
            c.MapLeftKey(p => p.ProductId, "ProductId");
            c.MapRightKey(p => p.CategoryId, "CategoryId");
            c.ToTable("tblProductCategory");
        });
    }
}
```

That said, I'm not sure what you are exactly trying to achieve but if this does not answer your question, then please post your desired database schema and I'll create a Code First object model to match that for you. Thanks :)

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Sunday, March 20, 2011 10:02 PM by bob

Hey!

GRRRRRRRRRRRRRRRR...

This got me confused.. I got I'm trying to add a new entity as an item in a collection but its failing miserably

I got an entity Session with an ICollection<Discussion> Discussions{get;set;}

how do I add a new discuss object?

I'm getting a PK violation? cannot understand why

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Monday, March 21, 2011 10:59 AM by [mortezaam](#)

@bob: If you show your object model as well as the code that throws the exception then I will have a better idea of the reason you get an exception while trying to save a new discuss object. Thanks.

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Friday, May 06, 2011 9:00 AM by [Sagi Fogel](#)

I'm trying to configure one-to-one association.

I have an Apartment class and a Contract class.

```
public class Apartment
{
    public int ID { get; set; }
    public virtual Contract Contract { get; set; }
}
```

```
public class Contract
{
    public int ID { get; set; }
}
```

I've tried using

```
modelBuilder.Entity<Apartment>()
    .HasOptional(u => u.Contract)
    .WithRequired();
```

But the contract is always null and the query performs 2 left joins.

```
SELECT
[Extent1].[ID] AS [ID]
.
.
.
FROM [dbo].[Apartments] AS [Extent1]
LEFT OUTER JOIN [dbo].[Contracts] AS [Extent2] ON [Extent1].[ID] = [Extent2].[ID]
LEFT OUTER JOIN [dbo].[Contracts] AS [Extent3] ON [Extent2].[ID] = [Extent3].[ID]
```

What am I missing and how can I change the mapping name for the association to be

```
[Contracts] AS [Extent3] ON [Extent1].[ContractID] = [Extent3].[ID]
```

10x,
Sagi.

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Monday, July 25, 2011 9:44 AM by [TarekShawadfy](#)

I have a similar one to one relationship. the problem happens when updating entities of the table with the foreign key by calling:

```
dbcontext.Entry(Product).State = EntityState.Modified;
```

this returns "Make sure that the key values are unique".

what do you recommend?

re: Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations

Wednesday, July 27, 2011 9:02 PM by [morteza](#)

@TarekShawadfy: I couldn't repro the exception you are getting; you have duplicate keys, most probably in some of your associations, which isn't allowed. That's all I can say without seeing your code. Any chance you could post your complete code here? You can also send it to me at bmanavi@gmail.com, if it's a project.

Use Entity Framework and WCF Ria Services development of SilverLight 3: the Map

Wednesday, September 07, 2011 5:43 AM by [Use Entity Framework and WCF Ria Services development of SilverLight 3: the Map](#)

Pingback from [Use Entity Framework and WCF Ria Services development of SilverLight 3: the Map](#)

[Terms of Use](#)

Associations in EF Code First: Part 3 – Shared Primary Key Associations

This is the third post in a series that explains entity association mappings with EF Code First. This s

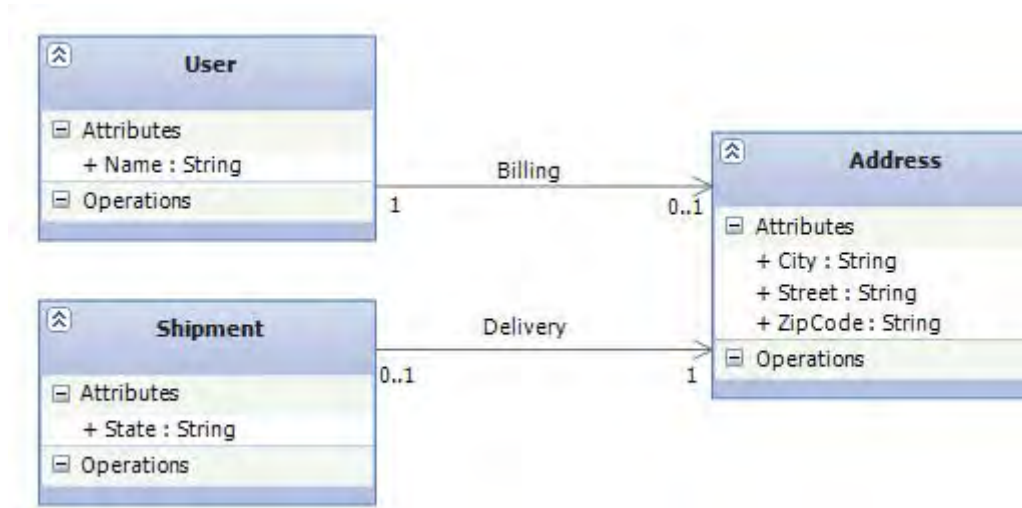
- [Part 1 – Introduction and Basic Concepts](#)
- [Part 2 – Complex Types](#)
- Part 3 – Shared Primary Key Associations
- [Part 4 – Table Splitting](#)
- [Part 5 – One-to-One Foreign Key Associations](#)
- [Part 6 – Many-valued Associations](#)

In the previous [blog post](#) I demonstrated how to map a special kind of one-to-one association—a cc complex types. We argued that the relationship between User and Address is best represented with mapping and we saw that this is usually the simplest way to represent one-to-one relationships but limitations.

In today's blog post I'm going to discuss how we can address those limitations by changing our map is particularly useful for scenarios that we want a dedicated table for Address, so that we can map b Address as entities. One benefit of this model is the possibility for *shared references*— another enti Shipment) can also have a reference to a particular Address instance. If a User has a reference to t her BillingAddress, the Address instance has to support shared references and needs its own identi User and Address classes have a true *one-to-one association*.

Introducing the Revised Model

In this revised version, each User could have one BillingAddress (Billing Association). Also a Shipm a destination address for delivery (Delivery Association). The following shows the class diagram for (note the multiplicities on association lines):



In this model we assumed that the billing address of the user is the same as her delivery address. Next we'll look at the association mappings for this domain model. There are several choices, the first being a *One-to-One Association*.

Shared Primary Key Associations

Also known as *One-to-One Primary Key Associations*, means two related tables share the same primary key. The primary key of one table is also a foreign key of the other. Let's see how we can create a primary key mapping with Code First.

How to Implement a One-to-One Primary Key Association with Code First

First, we start with the POCO classes. As you can see, we've defined *BillingAddress* as a navigation class and another one on Shipment class named *DeliveryAddress*. Both associations are unidirectional. We didn't define related navigation properties on Address class as for User and Shipment.

```

public class User
{
    public int UserId { get; set; }
    public string Name { get; set; }

    public virtual Address BillingAddress { get; set; }
}

public class Address
{
    public int AddressId { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public string ZipCode { get; set; }
}

public class Shipment

```

```

{
    public int ShipmentId { get; set; }
    public string State { get; set; }

    public virtual Address DeliveryAddress { get; set; }
}

public class Context : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Address> Addresses { get; set; }
    public DbSet<Shipment> Shipments { get; set; }
}

```

How Code First Sees the Associations in our Object Model: One-to-Many

Code First reads the model and tries to figure out the multiplicity of the associations. Since the associations are unidirectional, Code First takes this as if one Address has many Users and many Shipments and infers a *one-to-many* association for each of them. In other words, a unidirectional association is always inferred as One-to-Many by Code First. So, what we were hoping for—a one-to-one association, is not inline with the conventions.

How to Change the Multiplicity of the Associations to One-to-One by Using the Conventions

Obviously, one way to turn our associations to one-to-one is by making them bidirectional. That is, a navigation property to Address class of type User and another one of type Shipment. By doing that, Code First infers that we are looking to have one-to-one associations since for example User has an Address and Address has a User. Therefore, Code First will change the multiplicity to one-to-one and this will solve the problem.

Should We Make the Associations Bidirectional?

As always, the decision is up to us and depends on whether we need to navigate through our object model in the application code. In this case, we'd probably conclude that the bidirectional association doesn't make sense. If we call `anAddress.User`, we are saying "give me the user who has this address", not a very common request. So this is not a good option. Instead we'll keep our object model as it is and will explicitly make our associations one-to-one.

How to Change the Multiplicity to One-to-One with Fluent API

The following code is all that is needed to make the associations to be one-to-one. Note how the multiplicity in the UML class diagram (e.g. 1 on User and 0..1 on address) has been translated to the fluent API code using [HasRequired](#) and [HasOptional](#) methods:

```

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>().HasOptional(u => u.BillingAddress)
        .WithRequired();

    modelBuilder.Entity<Shipment>().HasRequired(u => u.DeliveryAddress)

```

```

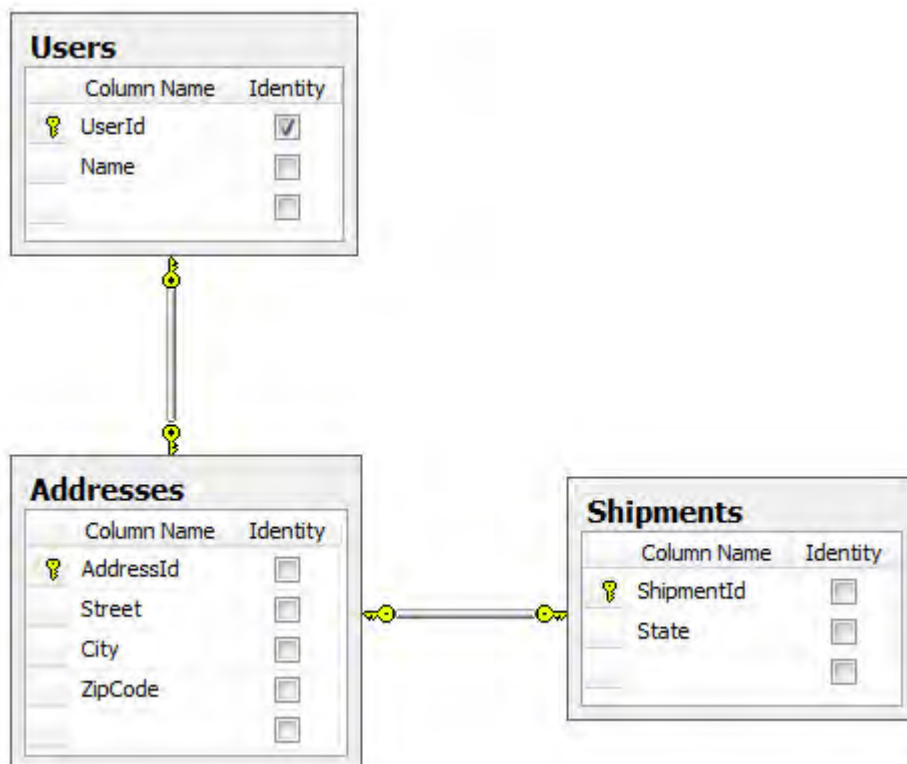
        .WithOptional();
    }

```

Also it worth noting that when we are mapping a one-to-one association with fluent API, we don't need foreign key as we would do when mapping a one-to-many association with [HasForeignKey](#) method. `HasForeignKey` supports one-to-one associations on primary keys, it will automatically create the relationship in the primary keys.

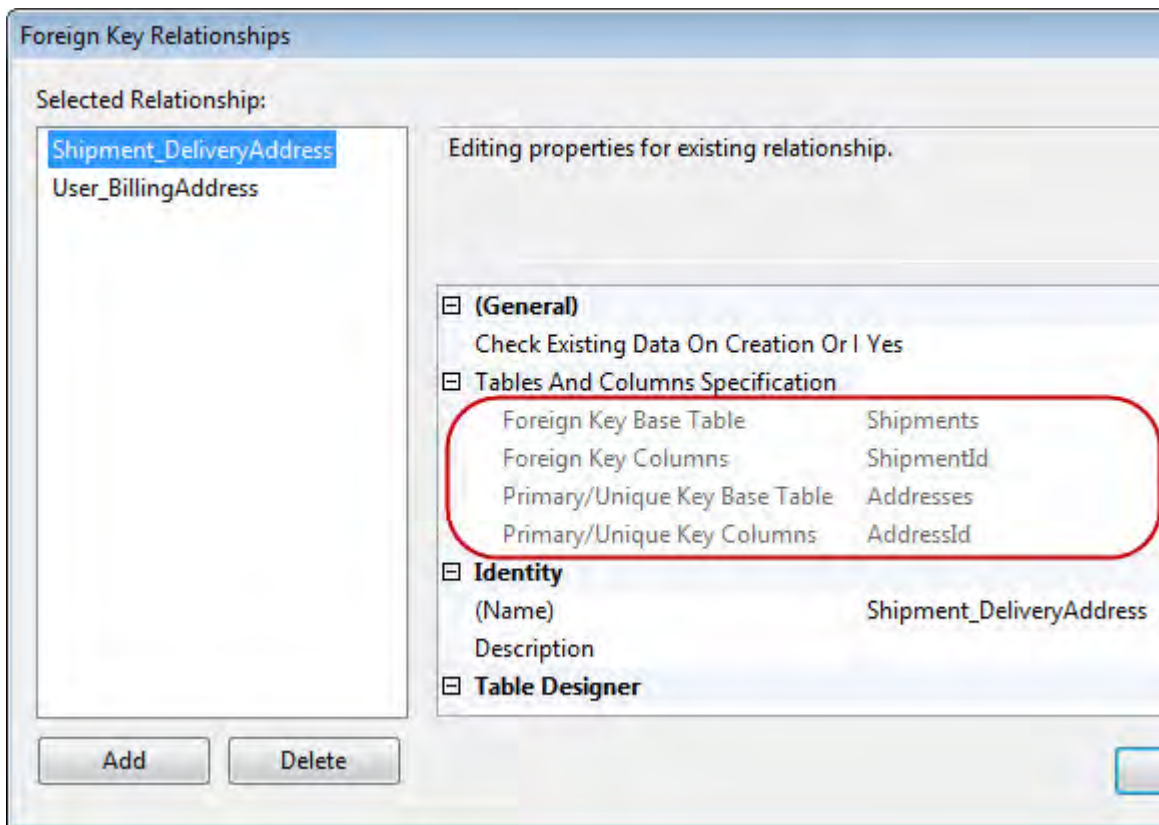
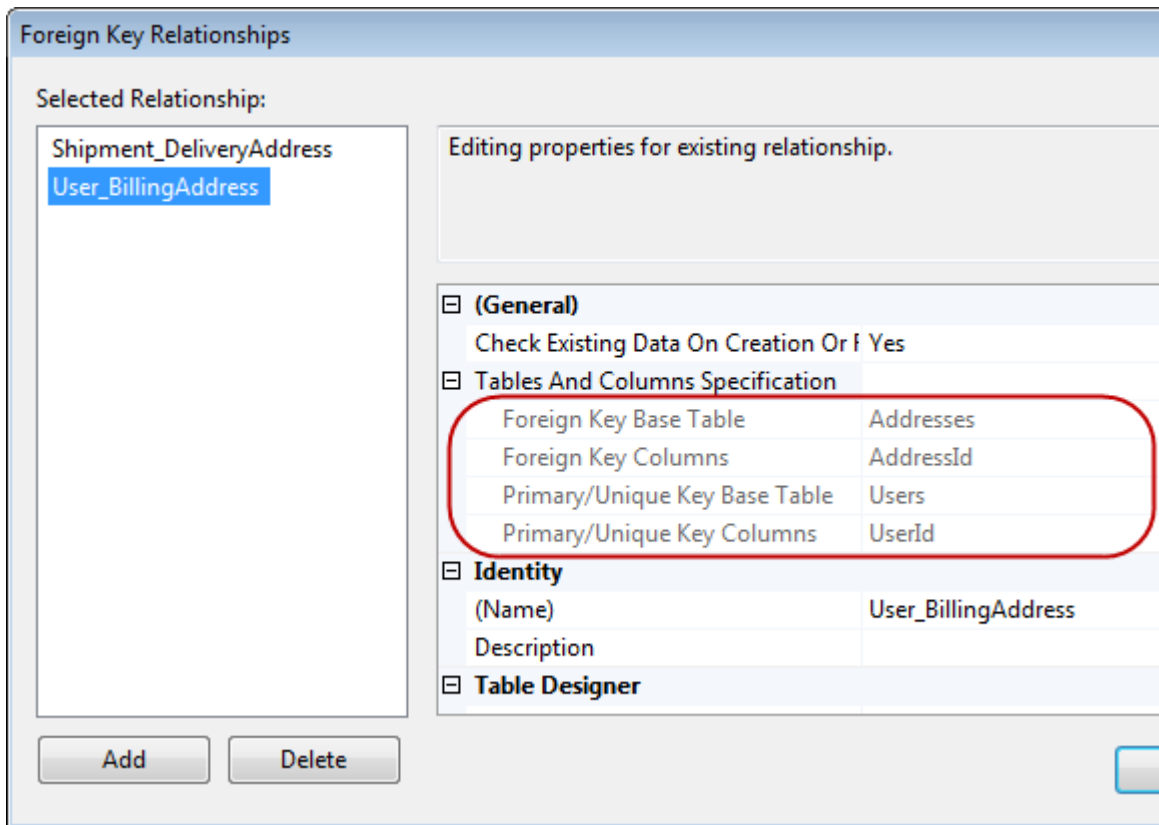
Database Schema

The mapping result for our object model is as follows (note the Identity column on Users table):



Referential Integrity

In relational database design the referential integrity rule states that each non-null value of a foreign key must be the value of some primary key. But wait, how does it even apply here? All we have is just three primary keys referencing each other! Who is the primary key and who is the foreign key? The best way to find the answer to this question is to take a look at the properties of the relationships in the database that has been created.



As you can see, Code First adds a foreign key constraint which links the primary key of the Address primary key of the Users table and adds another foreign key constraint that links the primary key of table to the primary key of the Addresses table. The foreign key constraint means that a user has to particular address but not the other way around. In other words, the database guarantees that an Address primary key references a valid Users primary key and a Shipments row's primary key references a valid Users primary key.

How Code First Determines the Principal and Dependent Ends in an Association?

Code First has rules to determine the principal and dependent ends of an association. For one-to-many the many end is always the dependent, but it gets a little tricky in one-to-one associations. In one-to-one Code First decides based on our object model, and possible data annotations or fluent API code that we use. For example in this case, we used the following fluent API code to configure the User-Address association:

```
modelBuilder.Entity<User>().HasOptional(u => u.BillingAddress).WithRequired();
```

This reads as "User entity has an optional association with one Address object but this association is required on the Address entity". For Code First this is good enough to make the decision: It marked User as the principal end and Address as the dependent end in the association. Since we have the same fluent API code for the association between Address and Shipment, it marks Address as the principal end and Shipment as the dependent end in the association as well.

This decision has some consequences. In fact, the referential integrity that we saw, is the first result of Code First's principal/dependent decision.

Second Result of Code First's Principal/Dependent Decision: Database Identity

If you take a closer look at the above DB schema, you'll notice that only UserId has a regular identifier (*Identity* or *Sequence*) and AddressId and ShipmentId does not. This is a very important consequence of Code First's principal/dependent decision for one-to-one associations: the dependent primary key will become non-unique and non-default. This makes sense because they share their primary key values and only one of them can be the primary key and we need to take care of providing valid keys for the rest.

What about Cascade Deletes?

As we saw, each Address always belongs to one User and each Shipment always delivered to one User. We want to make sure that when we delete a User the possible dependent rows on Address and Shipment are also deleted in the database. In fact, this is one of the [Referential Integrity Refactorings](#) which called [Introduce Cascading Delete](#). The primary reason we would apply "Introduce Cascading Delete" is to preserve the referential data by ensuring that related rows are appropriately deleted when a parent row is deleted. By default, Code First does not enable cascade delete when it creates a one-to-one relationship in the database. As always we can enable it by fluent API:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>().HasOptional(u => u.BillingAddress)
        .WithRequired()
        .WillCascadeOnDelete();
}
```

```

modelBuilder.Entity<Shipment>().HasRequired(u => u.DeliveryAddress)
    .WithOptional()
    .WillCascadeOnDelete();
}

```

What the Additional Methods Like `WithRequiredDependent` are for?

The `HasRequired` method returns an object of type `RequiredNavigationPropertyConfiguration` which has special methods called `WithRequiredDependent` and `WithRequiredPrincipal` in addition to the typical `WithOptional` methods that we usually use. We saw that the only reason Code First could figure out dependent in our associations was because our fluent API code clearly specified one end as Required and the other as Optional. But what if both endpoints are required or both are optional in the association? For example, a scenario that a User always has one Address and Address always has one User (required on both ends). Code First cannot pick up the principal and dependent ends on its own and that's exactly where methods like `WithRequiredDependent` come into play. In other words, this scenario ultimately needs to be configured using the fluent API and the fluent API is designed in a way that will force you to explicitly specify who is dependent and who is principal in a required-required or optional-optional association scenario.

For example, this fluent API code shows how we can configure the User-Address association where both ends are required:

```

modelBuilder.Entity<User>().HasRequired(u => u.BillingAddress).WithRequiredDependent()

```

Taking a closer look at the `RequiredNavigationPropertyConfiguration` type also shows the idea:

```

public class RequiredNavigationPropertyConfiguration<EntityType, TTargetEntityType>
{
    public DependentNavigationPropertyConfiguration<EntityType, TTargetEntityType>
        WithOptional();
    public CascadableNavigationPropertyConfiguration WithRequiredDependent();
    public CascadableNavigationPropertyConfiguration WithRequiredPrincipal();
}

```

As you can see, if you want to go another Required after `HasRequired` method, you have to either use `WithRequiredDependent` or `WithRequiredPrincipal` since there is no `WithRequired` method defined in the `RequiredNavigationPropertyConfiguration` class.

Working with the Model

Here is an example for adding a new user along with its billing address. EF is smart enough to use the generated `UserId` for the `AddressId` as well:

```

using (var context = new Context())
{
    Address billingAddress = new Address()
    {
        Street = "Main St.",
        City = "Seattle"
    };

    User user = new User()

```

```

{
    Name = "Morteza",
    BillingAddress = billingAddress
};

context.Users.Add(user);
context.SaveChanges();
}

```

The following code is an example of adding a new Address and Shipment for an existing User (assume a User with UserId = 1 in the database):

```

using (var context = new Context())
{
    Address deliveryAddress = new Address()
    {
        AddressId = 1,
        Street = "Main St.",
    };

    Shipment shipment = new Shipment()
    {
        ShipmentId = 1,
        State = "Shipped",
        DeliveryAddress = deliveryAddress
    };

    context.Shipments.Add(shipment);
    context.SaveChanges();
}

```

Limitations of This Mapping

There are two important limitations to associations mapped as shared primary key:

- **Difficulty in Saving Related Objects**

The main difficulty with this approach is ensuring that associated instances are assigned the value when the objects are saved. For example, when adding a new Address object, it's our job to provide a unique AddressId that is also valid (a User can be found with such a value as User

- **Multiple Addresses for User is Not Possible**

With this mapping we cannot have more than one Address for User. At the beginning of this post, when we introduced our model, we assumed that the user has the same address for billing and delivery. But is that not the case? What if we also want to add a Home address to User for the deliveries? In the current mapping, each row in the User table has at most one corresponding row in the Address table. To add a second address, we would need to add another address table, and this mapping style therefore wouldn't be adequate.


Summary

In this post we learned about one-to-one associations which shared primary key is just one way to implement a one-to-one association. Shared primary key associations aren't uncommon but are relatively rare. In many schemas, a one-

is represented with a foreign key field and a unique constraint. In the next posts we will revisit the same and will learn about other ways to map one-to-one associations that does not have the limitations of primary key association mapping.

References

- [ADO.NET team blog](#)
- [Java Persistence with Hibernate book](#)

Published Thursday, April 14, 2011 7:12 AM by [mortezaam](#) 
Filed under: [C#](#), [Code First](#), [Entity Framework 4.1](#)

Comments

[# re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations](#)

Friday, April 15, 2011 2:23 AM by TweeZz

Hi,

I think you have a tiny mistake in your post.

"assuming that we have a User with UserId=2 in the database"

Shouldn't this be "UserId=1"? Or am I missing something?

Manu.

[# re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations](#)

Friday, April 15, 2011 12:03 PM by [mortezaam](#)

@Manu: Yes, you are correct, it should be UserId = 1. I corrected this on the post. Thank you very much!
:)

[# re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations](#)

Friday, April 15, 2011 5:43 PM by Satish

Great article. Waiting for your next post as i am working on a mapping where i have two address fields for a user.

[# Associations in EF 4.1 Code First: Part 5 ??? One-to-One Foreign Key Associations](#)

Monday, May 02, 2011 6:21 PM by [Associations in EF 4.1 Code First: Part 5 ??? One-to-One Foreign Key Associations](#)

Pingback from [Associations in EF 4.1 Code First: Part 5 ??? One-to-One Foreign Key Associations](#)

[# re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations](#)

Monday, May 16, 2011 11:25 PM by [Johnny Fee](#)

Hi, "adding a new Address and Shipment for an existing User (assuming that we have a User with UserId = 1 in the database):..." Are you forget one line: "user.BillingAddress = billingAddress" or whether I ignore anything?

[# re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations](#)

Wednesday, May 18, 2011 10:12 AM by dencio

Thanks again for great series of posts.

Multiplicity constraint violated. | Code First :: Entity Framework

Thursday, May 19, 2011 7:53 PM by [Multiplicity constraint violated. | Code First :: Entity Framework](#)

Pingback from [Multiplicity constraint violated. | Code First :: Entity Framework](#)

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Friday, May 20, 2011 10:25 AM by [morte zam](#)

@Johnny Fee: No, we didn't really forget anything in there. The second code snippet has nothing to do with the previous one so there is neither a billingAddress nor a user. Like I said in the post, we know for a fact that "1" is a valid UserId in the database (let's say we retrieved this UserId earlier from another business process) so now we can use this UserId to relate an address along with a Shipment object to it. Hope this helps.

Code First ??????????????

Tuesday, May 24, 2011 11:31 AM by [Code First ??????????????](#)

Pingback from [Code First ??????????????](#)

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Wednesday, June 15, 2011 2:21 PM by [Preetham Reddy](#)

Hi Manavi,

Here is my Data Model...

```
public class User
{
    [Key]
    public string Email { get; set; }
    public string Name { get; set; }
}

public class Profile
{
    public int ProfileID { get; set; }
    public string ProfileName { get; set; }

    public virtual User User { get; set; }
    public virtual Address BillingAddress { get; set; }
}

public class Address
{
    public string AddressId { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public string ZipCode { get; set; }
}

public class Context : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Address> Addresses { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
```

```

{
    modelBuilder.Entity<Profile>().HasRequired(p => p.User).WithOptional();
    modelBuilder.Entity<Profile>().HasOptional(p => p.BillingAddress).WithRequired();
}
}

```

If Profile ID is string, then it creates one - one mapping. But if ProfileID is int, EF is creating another column User_Email as foreign key (expected). But this time, it's Many to One between user and profile. Why do you think that is happening?

Can't we have a new foreign key and yet have One-One?

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Thursday, June 16, 2011 8:57 PM by [mortezam](#)

@Preetham Reddy: Your fluent API code precisely creates an independent one-to-one association. In EF we usually create a 1:1 association by making the PK of the dependent entity to be also a FK of the principal entity, something that I explained in this post as a shared primary key association. EF also lets you create your one-to-one association on a column other than the child entity's primary key. This can be possible with one restriction though: you can't expose the FK property in your object model and that's exactly what you did with your fluent API code. For example, your code will add a new column to the Addresses table called Profile_ProfileID which represents a FK back to the Profiles table but your Address entity does not (and cannot) have such a property. This mapping has a catch though, if you look at your database you'll see that while your object model showing a 1:1 association (yes, you can even make your associations bidirectional, for example by adding a Profile property to the Address class and it would still work), the resulting relationship on the database is not, it's a typical 1:* relationship and if you reverse engineer the generated database into an EDMX file, you'll see that the relationship will be even picked up as a 1:* association by EF! The reason for that is because the only way EF can guarantee a 1:1 relationship on a column other than a PK is to create a unique constraint on that column but EF is not currently supporting unique constraints (and doesn't recognize them). As a result, you still have to create a unique constraint on the FK (e.g. Addresses.Profile_ProfileID) yourself, if you want to ensure the database consistency. Take a look at my answer to Fred at the end of [this post](#) where I answered the exact same question.

I discussed with a member of EF team about this somehow exotic one-to-one independent association at the end of [this thread](#) and it turns out that they originally allowed this mapping for using EF against existing databases that have a unique constraint defined. Long story short, this mapping should be avoided for a number of reasons and [one-to-one foreign key associations](#) should be used instead. Hope this helps :)

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Sunday, June 19, 2011 5:26 AM by [Preetham Reddy](#)

Hi Morteza,

The entity model that you've described only works if there is no other column in Junction Table.

In my case, I need to capture EventId too.

Users attend Events. During the event, User A can express interest in multiple users. User B can express interest in multiple users. If there is a match (ie., if a user A like user B and user B like User A) then their contact info will be exchanged.

Here it is a self referencing Many - Many relationship but I also need to know the EventID of the event they attended. That way, I can determine where they met.

Here is my model but it is not working.

```

public class User
{
    public User()
    {
        UserLike = new UserLike();
    }

    public int UserId { get; set; }
    public string Email { get; set; }
    public virtual UserLike UserLike { get; set; }
}

public class UserLike
{
    public UserLike()
    {
        LikesUsers = new List<User>();
    }

    public int EventId { get; set; }
    public virtual Event Event { get; set; }

    [Key, ForeignKey("User")]
    public int UserLiked { get; set; }
    public virtual User User { get; set; }
    public virtual ICollection<User> LikesUsers { get; set; }
}

```

Event is a simple class. What am I doing wrong?

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Sunday, June 19, 2011 3:40 PM by [mortezaam](#)

@Preetham Reddy: As I explained in the [sixth part](#) of this series, a many-to-many association cannot have a payload (e.g EventId), and if that's the case then we have to break it down to two one-to-many associations to an intervening class and I can see you've correctly created this class (UserLike) to represent the extra information attached to your self-referencing many-to-many association but the associations from this intermediate class are not correct as we need to define exactly 2 many-to-one association from UserLike to User like I showed in the following object model:

```

public class User
{
    public int UserId { get; set; }
    public string Email { get; set; }

    public virtual ICollection<UserLike> ThisUserLikes { get; set; }
    public virtual ICollection<UserLike> UsersLikeThisUser { get; set; }
}

public class UserLike
{
    public int UserLiked { get; set; }
    public int LikerId { get; set; }
    public int LikeeId { get; set; }
    public int EventId { get; set; }

    public User Liker { get; set; }
    public User Likee { get; set; }
    public virtual Event Event { get; set; }
}

```



```

public class Event
{
    public int EventId { get; set; }
    public string Name { get; set; }
}

public class Context : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Event> Events { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>()
            .HasMany(u => u.ThisUserLikes)
            .WithRequired(ul => ul.Liker)
            .HasForeignKey(ul => ul.LikerId);

        modelBuilder.Entity<User>()
            .HasMany(u => u.UsersLikeThisUser)
            .WithRequired(ul => ul.Likee)
            .HasForeignKey(ul => ul.LikeeId)
            .WillCascadeOnDelete(false);
    }
}

```

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Monday, June 20, 2011 12:32 AM by [Preetham Reddy](#)

Awesome... I've got my model working now...

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Monday, June 20, 2011 12:41 AM by [morteza](#)

@Preetham Reddy: One last thing before you go, you can use the following LINQ query to retrieve all the users who like each other:

```

using (var context = new Context())
{
    var friends = (from u1 in context.Users
                  from likers in u1.UsersLikeThisUser
                  from u2 in u1.ThisUserLikes
                  where u2.LikeeId == likers.LikerId
                  select new
                  {
                      OurUser = u1.UserId,
                      HerFriend = u2.LikeeId
                  })
    .ToList();
}

```

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Monday, June 20, 2011 4:08 AM by Tony

Hi,

What about the case where a User can have a list of Addresses but an Address can only belong to a single User? As with the above [original] example, an Address can be associated to a Shipment. I was thinking of modelling this through a association table (e.g. UserAddress which has UserId and AddressId; ShipmentAddress which has ShipmentId and AddressId). Would very much appreciate any guidance on how to model this in EF.

Thanks,

Tony

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key AssociationsMonday, June 20, 2011 10:57 AM by [mortezam](#)

@Tony: You don't really need to create a join table like UserAddress unless you require a many-to-many association between your entities. The scenario you described can be mapped with a simple one-to-many association between User and Address as I showed in the following object model:

```
public class User
{
    public int UserId { get; set; }
    public string Name { get; set; }

    public virtual ICollection<Address> Addresses { get; set; }
}

public class Address
{
    public int AddressId { get; set; }
    public int UserId { get; set; }

    public User User { get; set; }
}

public class Context : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Address> Addresses { get; set; }
}
```

Note that we didn't even need fluent API code for this as everything will be picked up by convention.

You can configure the same for the association between Address and Shipment entities if a 1:* relationship is required between them. Please take a look at the [last part](#) of this series where I explained many-valued associations in detail. Hope this helps.

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Monday, June 20, 2011 6:50 PM by Tony

Thanks Morteze.

Sorry I wasn't clear. I'll give my exact example instead...

I have the following classes...

```

public class Contact {
    public int Id { get; set; }
    <some other Contact details here>
}

public class BusinessUnit {
    public int Id { get; set; }
    public List<Contact> Contacts { get; set; }
    <some other Business Unit details here>
}

public class Counterparty {
    public int Id { get; set; }
    public List<Contact> Contacts { get; set; }
    <some other Counterparty details here>
}

```

Business Unit and Counterparty (and possibly some other Entities in the future) can have an arbitrary number of Contacts hence I am not able to put an inverse key (i.e. BusinessUnitId + BusinessUnit, CounterpartyId + Counterparty) on the Contact class.

Thanks heaps.

Tony

[# re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations](#)

Tuesday, June 21, 2011 9:22 PM by [morteza](#)

@Tony: Ok, if that's the case then using an associative table to map a one-to-many association would make sense. In order to create this mapping, you need to set up a many-to-many association between Contact and BusinessUnit (same thing for Counterparty as well). The following fluent API code shows how:

```

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

public class BusinessUnit
{
    public int Id { get; set; }
    public ISet<Contact> Contacts { get; set; }
}

public class Counterparty
{
    public int Id { get; set; }
    public ISet<Contact> Contacts { get; set; }
}

public class Context : DbContext
{
    public DbSet<Contact> Contacts { get; set; }
    public DbSet<BusinessUnit> BusinessUnits { get; set; }
    public DbSet<Counterparty> Counterparties { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<BusinessUnit>()

```

```

        .HasMany(u => u.Contacts)
        .WithMany()
        .Map(c =>
        {
            c.MapLeftKey("BusinessUnitId");
            c.MapRightKey("ContactId");
            c.ToTable("BusinessUnitContact");
        });

modelBuilder.Entity<Counterparty>()
    .HasMany(u => u.Contacts)
    .WithMany()
    .Map(c =>
    {
        c.MapLeftKey("CounterpartyId");
        c.MapRightKey("ContactId");
        c.ToTable("CounterpartyContact");
    });
    }
}

```

After running this object model you need to enforce the relationships as a real one-to-many. For that you have to go into your database and manually create unique constraints on ContactId in both join tables (CounterpartyContact and BusinessUnitContact). In other words, you want to make sure that for example once a BusinessUnit references a Contact, this Contact won't be referenced by any other BusinessUnit. That said, this design still has a problem: the data model allows a single Contact to be referenced by a BusinessUnit and a Counterparty at the same time. If that's an issue then we have to come up with other ways to map the association between Contact and other entities, ways that ensures a Contact would be referenced by one single entity at any time.

[# re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations](#)

Wednesday, June 22, 2011 2:40 AM by Tony

@morteza,

ABSOLUTELY BRILLIANT!!!

That's fixed my problem and you've saved me a lot of time with this.

re: "The Data Model allows a single Contact to be referenced by a BusinessUnit and a Counterparty" at the same time" issue, I'll have a think about it on my end. I'm sure I'll be able to come up with a solution but would more than welcome any tips you can provide.

Thank you so much for your help. It is very much appreciated!

Tony

[# re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations](#)

Wednesday, June 22, 2011 9:36 AM by Tony

Hi @morteza,

ABSOLUTELY BRILLIANT!

Thanks heaps for your guidance. This saved me a lot of time indeed!

I should be able to work out the single Contact able to be referenced by a Business Unit or Counterparty at the same time but am always open to any suggestions/tips you might have on this.

Very much appreciate and indebted to your expertise.

Tony

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Wednesday, June 22, 2011 5:07 PM by [morteza](#)

@Tony: You are very welcome, I'm glad you found it helpful :)

As for other ideas, you can introduce different types of Contacts in your object model, each associated with a different type of entity (e.g. BusinessUnitContact, CounterpartyContact, etc.). To make it more elegant, you can use inheritance and define an abstract base class like Contact (to hold the common Contact properties) which all the specialized contact classes inherit from it. Then depend on your preference, you can use different strategies to map this inheritance hierarchy to your database. For example, you can use [TPC](#) to have a completely separate table per each Contact subtype or you can choose [TPH](#) to map everything to one single table. Either ways, different entities would hold references to their specific contact type and the problem we saw with the previous approach regarding multiple references to a single Contact from different entities would never happen.

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Tuesday, June 28, 2011 10:49 PM by [ddredstar@hotmail.com](#)

I'm a little confused, i think the `modelBuilder.Entity<User>().HasRequired(u => u.BillingAddress).WithRequiredDependent()` should be `modelBuilder.Entity<User>().HasRequired(u => u.BillingAddress).WithRequiredPrincipal()`

i mean here we should use `WithRequiredPrincipal()` instead of `WithRequiredDependent()`

please help to clarify,thanks!

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Wednesday, June 29, 2011 6:27 PM by [morteza](#)

@ddredstar@hotmail.com: Great question, you are absolutely right, it has to be `WithRequiredPrincipal`.

To see why let's start with the documentation for the [WithRequiredPrincipal](#) method:

"WithRequiredPrincipal method Configures the relationship to be required:required with a navigation property on the other side of the relationship. The entity type being configured will be the principal in the relationship. The entity type that the relationship targets will be the dependent and contain a foreign key to the principal."

And this is the fluent API code which you correctly proposed:

```
modelBuilder.Entity<User>().HasRequired(u => u.BillingAddress).WithRequiredPrincipal();
```

The important point is that the entity type being configured is always the one we start our fluent API code with (i.e. the type parameter of the generic Entity method), which in this example is the User entity. The entity type that the relationship targets is always the one we specify in the first fluent API method (after the Entity method) which is HasRequired in this example and it targets BillingAddress entity. As a result of this fluent API code, Code First will mark User as the principal and BillingAddress as the dependent in this relationship which is exactly what we are looking for. Later on when you generate the database, Code First will make the UserId an identity for the Users table. If you take a look at the documentation for the [WithRequiredDependent](#) method and apply the same logic to my fluent API code in the post, you'll see that User entity will be inferred as the dependent end which will result in incorrect SQL schema being generated. I'll correct this in the post. Thank you so much for pointing this out, I really appreciate it :)

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Monday, August 22, 2011 10:10 AM by yaron

```
class tree
{
    int id;
    string title;
}

class file
{
    int id;
    string FileName;
    public virtual tree Tree;
}
```

file f = db.file.Find(id);

f.tree.title is ALWAYS NULL ,

why the tree instance is not loaded with the file ?

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key AssociationsSaturday, August 27, 2011 10:29 PM by [morteza](#)

@yaron: First make sure that tree and every other class member on both entities is a property and not a field, since EF does not support fields yet. For example, your file entity should be look like this:

```
class file
{
    public int id { get; set; }
    public string FileName { get; set; }
    public virtual tree Tree { get; set; }
}
```

Now if you are reading the f.tree.title while your DbContext instance is not yet disposed then lazy loading should kick in and retrieve the tree property for you, but if you are reading the f.tree.title when the DbContext is already disposed then you need to eager load the Tree property when retrieving the file entity. One way to eager load a navigation property is to use the Include method:

```
file f = db.file.Where(f => f.id == id).Include(f => f.Tree).Single();
```

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Friday, September 16, 2011 7:38 PM by Jon

Hello Morteza,

I happen to be working on a similar database design. By following your way, I was able to get one step closer. However, now I'm stuck and I'm in dire need of help.

Here's my scenario: I have 2 tables in a 1:1 relationship, UserLogin and UserProfile.

UserLogin consists of UserID set to autoincrement ON whereas UserProfile consists of UserID with autoincrement OFF.

I have also overridden OnModelCreating by adding the following 2 lines:

```
modelBuilder.Entity<UserProfile>().HasKey(u => u.UserID);
```

```
modelBuilder.Entity<UserLogin>().HasOptional(u => u.UserProfile).WithRequired();
```

By right, this should put them in a 1:1 relationship. However, this is the error I get:

```
{"Invalid column name 'UserProfile_UserID'."}
```

Am I missing something?

In UserLogin class, I have the following:

```
[Key()]  
public long UserID { get; set; }  
public virtual UserProfile UserProfile { get; set; }
```

And in UserProfile class, I have this:

```
public long UserID { get; set; }
```

Cascade Delete Rule in EF 4.1 Code First when using Shared Primary Key Association - Programmers Goodies

Wednesday, September 28, 2011 10:21 PM by [Cascade Delete Rule in EF 4.1 Code First when using Shared Primary Key Association - Programmers Goodies](#)

Pingback from [Cascade Delete Rule in EF 4.1 Code First when using Shared Primary Key Association - Programmers Goodies](#)

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Monday, November 14, 2011 8:07 PM by Chase

Just noticed at the end of the article (2nd limitation) that you stated, "in the current setup, each row in the User table has a corresponding row in the Address table." That isn't true since you have a 1 -> 0...1 relationship for User -> Address, right? Just want to be clear.

Great article by the way! Best code-first article on the internet IMO.

re: Associations in EF 4.1 Code First: Part 3 – Shared Primary Key Associations

Wednesday, November 16, 2011 10:27 AM by [mortezaam](#)

@Chase: You are correct and I changed the text a little bit so that it better reflects the model we discussed in this article. Thank you so much for your comment! :)

re: Associations in EF Code First: Part 3 – Shared Primary Key Associations

Monday, November 28, 2011 2:55 PM by [Yasir Atabani](#)

This is exactly the kind of article I was looking for. Great work.

[Terms of Use](#)

Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

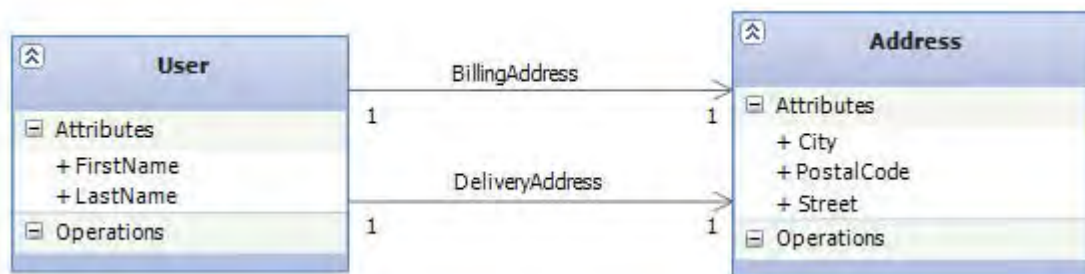
This is the third post in a series that explains entity association mappings with EF Code First. I've done so far:

- [Part 1 – Complex Types](#)
- [Part 2 – Shared Primary Key Associations](#)

In the previous [blog post](#) we saw the limitations of shared primary key association and argued that that is relatively rare and in many schemas, a one-to-one association is represented with a foreign key field. In this post we are going to discuss how this is done by learning about one-to-one foreign key associations.

Introducing the Revised Model

In this revised version, each User always have two addresses: one billing address and another one for delivery. The following diagram shows the class diagram for this domain model:



One-to-One Foreign Key Association

Instead of [sharing a primary key](#), two rows can have a foreign key relationship. One table has a foreign key to the primary key of the associated table (The source and target of this foreign key constraint can even be called a *self-referencing* relationship.). An additional constraint enforces this relationship as a real one-to-one association by making the BillingAddressId column unique, we declare that a particular address can be referenced by at most one user, period. With several foreign key columns (which is the case in our case where we have a foreign key for DeliveryAddress), we can reference the same address target row several times, but we can't share the same address for the same purpose.

The Object Model

Let's start by creating an object model for our domain:

```
public class User
{
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int BillingAddressId { get; set; }
    public int DeliveryAddressId { get; set; }

    public Address BillingAddress { get; set; }
    public Address DeliveryAddress { get; set; }
}

public class Address
{
    public int AddressId { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
}

public class EntityMappingContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Address> Addresses { get; set; }
}
```

As you can see, User class has introduced two new scalar properties as *BillingAddressId* and *DeliveryAddressId* related navigation properties (*BillingAddress* and *DeliveryAddress*).

Configuring Foreign Keys With Fluent API

BillingAddressId and *DeliveryAddressId* are foreign key scalar properties and representing the actual relationships are established on. However, Code First will not recognize them as the foreign keys for names are not aligned with the [conventions](#) that it has to infer foreign keys. Therefore, we need to use [Data Annotations](#) to tell Code First about the foreign keys. Here is the fluent API code to identify the fore

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>()
        .IsRequired(a => a.BillingAddress)
        .WithMany()
        .HasForeignKey(u => u.BillingAddressId);

    modelBuilder.Entity<User>()
        .IsRequired(a => a.DeliveryAddress)
        .WithMany()
        .HasForeignKey(u => u.DeliveryAddressId);
}
```

Alternatively, we can use Data Annotations to achieve this. CTP5 introduced a new attribute in [System.ComponentModel.DataAnnotations](#) namespace which is called `ForeignKeyAttribute` and property to specify the property that represents the foreign key of the relationship:

```
public class User
{
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int BillingAddressId { get; set; }
    public int DeliveryAddressId { get; set; }

    [ForeignKey("BillingAddressId")]
    public Address BillingAddress { get; set; }

    [ForeignKey("DeliveryAddressId")]
    public Address DeliveryAddress { get; set; }
}
```

However, we will not use this Data Annotation and will stick with our fluent API code for a reason that

Creating a SQL Server Schema

The object model seems to be ready to give us the desired SQL schema, however, if we try to create it, we will get an [InvalidOperationException](#) with this message:

"The database creation succeeded, but the creation of the database objects did not. See InnerException details."

The inner exception is a [System.Data.SqlClient.SqlException](#) containing this message:

"Introducing FOREIGN KEY constraint 'User_DeliveryAddress' on table 'Users' may cause cycles or multiple cascade paths. Specify ON DELETE NO ACTION or ON UPDATE NO ACTION, or modify other FOREIGN constraints. Could not create constraint. See previous errors."

As you can tell from the type of the inner exception (`SqlException`), it has nothing to do with EF or Code First purely by SQL Server when Code First was trying to create a database based on our object model.

SQL Server and Multiple Cascade Paths

A *Multiple cascade path* happens when a cascade path goes from column `col1` in table A to table B and then from table B to table A. So it seems that Code First tried to turn on Cascade Delete for both `BillingAddressId` and `DeliveryAddressId` columns in `Users` table. In fact, Code First was trying to use [Declarative Referential Integrity \(DRI\)](#) but the problem is that SQL Server is not fully [ANSI SQL-92](#) compliant when it comes to the cascading updates or deletes in a multiple cascade path scenario.

A [KB article](#) also explains why we received this error: *"In SQL Server, a table cannot appear more than once in a cascading referential actions that are started by either a DELETE or an UPDATE statement. For example, cascading referential actions must only have one path to a particular table on the cascading referential actions appeared twice in a list of cascading referential actions started by a DELETE). Basically, SQL Server does not support multiple cascade paths and, rather than trying to work out whether any cycles actually exist, it assumes the*

referential actions (Cascades).

Therefore, depend on our database engine, we may or may not get this exception (For example, bo create Cascades in this scenario.).

Overriding Code First Convention To Resolve the Problem

As you saw, Code First automatically turns on Cascade Deletes on *required* one-to-many associati However, in order to resolve the exception that we got from SQL Server, we have no choice other th and switching cascade deletes off on at least one of the associations and as of CTP5, the only way fluent API. Let's switch it off on DeliveryAddress Association:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>()
        .HasRequired(a => a.BillingAddress)
        .WithMany()
        .HasForeignKey(u => u.BillingAddressId);

    modelBuilder.Entity<User>()
        .HasRequired(a => a.DeliveryAddress)
        .WithMany()
        .HasForeignKey(u => u.DeliveryAddressId).WillCascadeOnDelete(false);
}
```

One-to-One Foreign Key Associations in EF Code First

As you may have noticed, both associations in the fluent API code has been configured as a *many*-might have expected. The reason is simple: Code First (and EF in general) does not natively suppo associations. In fact, EF does not support any association scenario that involves unique constraints we don't care what's on the target side of the association, so we can treat it like a *to-one* associati want is to express "This entity (User) has a property that is a reference to an instance of another en key field to represent that relationship. Basically EF still thinks that the relationship is many-to-one. current EF limitation which comes with two consequences: First, EF won't create any additional con relationship as a one to one, we need to manually create it ourselves. The second limitation that thi more important: one to one foreign key associations cannot be bidirectional (i.e. we cannot define a class).

Create a Unique Constraint To Enforce the Relationship as a Real One to One

We can manually create unique constraints on the foreign keys in the database after Code First cre me and prefer to create your database in one shot then there is a way in CTP5 to have Code First c its database creation process. For that we can take advantage of the new CTP5's SqlCommand metl allows raw SQL commands to be executed against the database. The best place to invoke SqlComr inside a Seed method that has been overridden in a custom Initializer class:

```
protected override void Seed(EntityMappingContext context)
{
    context.Database.SqlCommand("ALTER TABLE Users ADD CONSTRAINT uc_Billing UNI
    context.Database.SqlCommand("ALTER TABLE Users ADD CONSTRAINT uc_Delivery UN
```

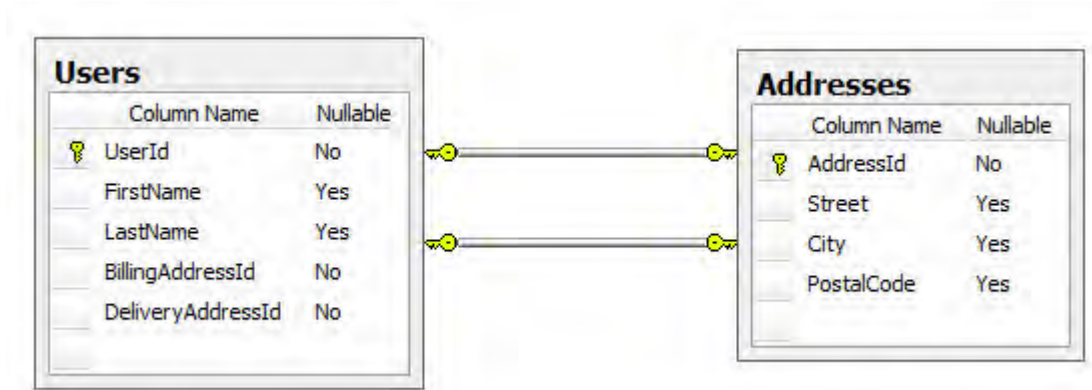
```
}

```

This code adds unique constraints to the BillingAddressId and DeliveryAddressId columns in the DI

SQL Schema

The object model is ready now and Code First will create the following database schema for us:



It is worth mentioning that we can still enforce cascade deletes for DeliveryAddress relationship. SC Referential Integrity in two different ways. DRI that we just saw is the most basic yet least flexible w/ *Triggers*. We can write a [Delete Triggers](#) on the primary table that either deletes the rows in the dep corresponding foreign keys to NULL (In our case the foreign keys are Non-Nullable so it has to dele

Download

[Click here](#) to download and run the one-to-one foreign key association sample that we have built in

Summary

In this blog post we learned about one-to-one foreign key associations as a better way to represent However, we saw some limitations such as the need for manual creation of unique constraints and ; associations cannot be bidirectional, all due to the lack of unique constraint support in EF. Support f require changes to the whole EF stack and it won't happen in the RTM targeted for this year as that the current .NET 4.0 functionality. That said, EF team has this feature on their list for the future, so f later release of EF and until then the workaround that I showed here is going to be the way to imple associations in EF Code First.

References

- [ADO.NET team blog](#)
- [Java Persistence with Hibernate book](#)

Published Sunday, January 23, 2011 8:22 PM by [morteza](#) ★★★★★

Filed under: [Entity Framework](#), [C#](#), [Code First](#), [CTP5](#), [.NET](#)

Comments

Twitter Trackbacks for Associations in EF Code First CTP5: Part 3 ??? One-to-One Foreign Key Associations - Morteza Manavi [asp.net] on Topsy.com

Sunday, January 23, 2011 9:01 PM by [Twitter Trackbacks for Associations in EF Code First CTP5: Part 3 ??? One-to-One Foreign Key Associations - Morteza Manavi \[asp.net\] on Topsy.com](#)

Pingback from [Twitter Trackbacks for Associations in EF Code First CTP5: Part 3 ??? One-to-One Foreign Key Associations - Morteza Manavi \[asp.net\] on Topsy.com](#)

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Tuesday, January 25, 2011 12:02 PM by [bonder](#)

Hi Morteza,

Great article - thanks for sharing this information, because my team and I have been struggling with 1:1 relationships in EF4 for a couple weeks now. :)

One question - instead of a unique constraint, do you think we could also specify a Primary Key constraint on the Foreign Key?

Our situation is this: we have a master entity table that is pretty simple: an EntityId (key) and a Name.

Then, we have specific entity tables, let's say Book. Book has BookId (key) and Title.

But we want to have Book.BookId have a 1:1 FK relationship with entity.

I am thinking we'll take your article and just try using a [Key] attribute instead of the Unique Constraint generation.

But, your thoughts are much appreciated!

Thanks again for the article!

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Tuesday, January 25, 2011 4:40 PM by [morteza](#)

@bonder: I believe what you are looking for can be best represented by a *Shared Primary Key Association* where two related tables share the same primary key values (The primary key of one table is also a foreign key of the other.). For example, consider this object model:

```
public class Entity
{
    public int EntityId { get; set; }
    public string Name { get; set; }

    public Book Book { get; set; }
}

public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }

    public Entity Entity { get; set; }
}

public class Ctp5Context : DbContext
{
    public DbSet<Book> Books { get; set; }
    public DbSet<Entity> Entities { get; set; }
}
```

As a result, Code First will add a foreign key constraint which links the primary key of the Books table to the primary key of the Entities table which enforces a strong 1:1 relationship. If this is what you want to achieve, then please take a look at my other article on [Shared Primary Key Associations with Code First](#). I hope I didn't misunderstand your question, but if did then please provide me with some more info about your domain model and I'll be more than happy to help out. Also thanks for your comment; I'm glad you found it useful :)

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Tuesday, February 01, 2011 7:38 PM by fred

Your solution is very clever, and I enjoy read your blog!

But, the association has Address as the primary end. That means that when you delete Address, it will cascade delete the associated User.

However it is desirable for User to be the primary end, so that when you delete a User, it will delete the associated Address. How to do that?

My only guess is to model a many:many so you can choose the primary end to be User, and then "pretend" the other side is a collection of only one Address (rather than many Addresses).

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Wednesday, February 02, 2011 12:23 AM by [mortezam](#)

@fred: Good point! You are right and I totally agree with you on this. In a real world scenario, having a User deleted from the DB as a result of deleting its BillingAddress (which still has the cascade enabled in our example) does not make much of a sense, but please note that even deleting an Address as a result of deleting its associated User is not always desirable. For example, consider a *shared reference* scenario where another entity (let's say a Shipment class) has also a reference to the Address entity like

the domain model we discussed [here](#). In this case we cannot just cascade delete an Address as a result of a User being deleted since at the same time it could have been referenced by a Shipment object. Having said that, let's assume this is not the case and we really want to get rid of the associated Addresses (Billing and Delivery) as we delete the associated User. In this case, the best way to achieve cascade deletes is to implement Delete Triggers on the User table that deletes the corresponding rows in the primary table (Addresses) as we delete a User from the dependent table (Users).

Another point to be aware of is that if you switch off cascade deletes on the associations to prevent the deletion of a User as a result of deleting its Address, you'd better off make the associations optional as well, since otherwise it would be impossible to delete an Address without first deleting its associated user (or having the User reference another Address). To make our associations optional (right now they are required) we would have to make our foreign keys (BillingAddressId and DeliveyAddressId) nullable as well as changing the fluent API by using *HasOptional* method instead of *HasRequired* method that we are using now.

Regarding your proposed solution, I have to tell you that using a Join table –like what we have in Many:Many associations– is in fact the fourth way to map a one:one association (we have seen 3 ways so far). For example in this case, we can use a Join table that basically has two columns, one of the columns is a primary key/foreign key that references Users.UserId and the other one is just a foreign key referencing Addresses.AddressId. In fact, it is an *Entity Splitting* on the User entity with the FK (e.g. AddressId) in a separate table. The interesting point is that even though database would take this as a one:many relationship, EF will still let you narrow this to a one:one association in the object model. Having said all that, a one:one association is mapped with a Join table for reasons unrelated to cascade delete and I don't recommend using it in this case. Hope this helps.

Thanks for your comment by the way, I really appreciate your feedback :)

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Wednesday, February 02, 2011 7:47 PM by fred

Yes the trigger idea is good!

I will make Address optional/nullable, but is that necessary? Can't you have it as required?

PS you maybe update article to show ".WillCascadeOnDelete(false)" for BOTH Addresses, otherwise you get a bad surprise when you delete an Address and the User is automatically deleted.

PS there is another "gotcha". In my scenario instead of your User->Address entities I have instead Project->Manager entities, where Manager is TPT "inherited table" from User (there is also Employee, Supervisor, Contractor, etc.) So in my trigger code I have to delete Manager entity AND User entity, because EF does not automatically set cascade delete on TPT relationships (the deletions are done by EF, not by database). If you don't do this, you get orphaned records. For TPC/TPH should not be a problem. If you update your article in the future, maybe show this briefly for benefit of readers.

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Friday, February 04, 2011 2:20 PM by [mortezam](#)

@fred: No, it's not necessary; of course you can cascade delete an address by defining a delete trigger on User table without having to make the associations optional. The only reason I said that was because you didn't seem to be very happy with the fact that a User gets deleted as a result of deleting an Address and wanted to have the cascades switched off on both associations. But switching cascade off on a required association has an important consequence on deleting the principal end: we can no longer delete an Address that has a User referencing it unless we either delete the referencing user or have it

reference another address because for example User.DeliveryAddressId cannot be null. In other words, a User cannot live without an Address. However, if we make the associations optional, then we can just remove an Address and update its User to simply not referencing it anymore (since now we can do this: User.DeliveryAddressId = null). I think my original answer was not clear enough, I will update it to avoid any confusion in this regard. Thanks :)

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Thursday, February 10, 2011 12:21 PM by [Dan Ludwig](#)

Awesome series. The only part I didn't like was on this one, where you say it's the last :(

I second the notion if you writing a book. I would buy it. Your writing style is easy to follow and you explain the concepts well.

I was particularly interested in one of the comments where the person was trying to implement the Layer Supertype pattern (a.k.a. public abstract class BaseEntity). I'm doing the same thing, but I'm uneasy choosing TPC for the mapping because of the bugs and problems you mentioned, at least until there is a non-CTP release.

Looking forward to reading more of your ramblings. Is this the best blog to add to my feed reader?

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Saturday, February 12, 2011 9:29 PM by [morteza](#)

@Dan Ludwig: Thank you for your nice comment; I appreciate you reading the articles :)
I am not sure I understand your question "Is this the best blog to add to my feed reader?", but [this blog](#) is the only place that I publish my articles.

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Saturday, February 19, 2011 7:04 PM by [Search Engine Optimization Seattle](#)

DUDE! you are the man. I've been working on a project and using the asp.net MVC 2 Music store as a guide, but then I ran into this issue as I'm utilizing CTP5 to create the database whereas they're using an already created database (and maybe for this lack of functionality in CTP5)... anyways thank you SOOOOOOOO much I'm soooo grateful!

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Monday, February 21, 2011 6:24 PM by [morteza](#)

@Search Engine Optimization Seattle: You're very welcome and thanks for your kind words, it is such a pleasure to hear that this blog post could help you in your Code First development :)

Entity Framework 4.1 Code First: Complex Many-to-Many Relationships | Christian Reumann

Wednesday, March 23, 2011 7:57 AM by [Entity Framework 4.1 Code First: Complex Many-to-Many Relationships | Christian Reumann](#)

Pingback from [Entity Framework 4.1 Code First: Complex Many-to-Many Relationships | Christian Reumann](#)

Storing objects with Code First that are related as classes and subclasses | Coding Answers

Friday, May 13, 2011 11:02 AM by [Storing objects with Code First that are related as classes and subclasses | Coding Answers](#)

Pingback from [Storing objects with Code First that are related as classes and subclasses | Coding Answers](#)

Storing objects with Code First that are related as classes and subclasses - Programmers Goodies

Sunday, September 11, 2011 11:31 PM by [Storing objects with Code First that are related as classes and subclasses - Programmers Goodies](#)

Pingback from [Storing objects with Code First that are related as classes and subclasses - Programmers Goodies](#)

Entity Framework 0..1 to 0 relation - Programmers Goodies

Tuesday, September 13, 2011 1:10 AM by [Entity Framework 0..1 to 0 relation - Programmers Goodies](#)

Pingback from [Entity Framework 0..1 to 0 relation - Programmers Goodies](#)

Same navigation types one more in EF 4.1 - Programmers Goodies

Sunday, October 23, 2011 10:45 PM by [Same navigation types one more in EF 4.1 - Programmers Goodies](#)

Pingback from [Same navigation types one more in EF 4.1 - Programmers Goodies](#)

How to Establish Foreign Key Relationships in EF 4.1 Code First - Programmers Goodies

Tuesday, October 25, 2011 7:30 PM by [How to Establish Foreign Key Relationships in EF 4.1 Code First - Programmers Goodies](#)

Pingback from [How to Establish Foreign Key Relationships in EF 4.1 Code First - Programmers Goodies](#)

Associations in EF Code First CTP5: Part 3 ??? One-to-One Foreign Key Associations – Enterprise .Net | ObondO Labs

Monday, November 14, 2011 9:56 AM by [Associations in EF Code First CTP5: Part 3 ??? One-to-One Foreign Key Associations – Enterprise .Net | ObondO Labs](#)

Pingback from [Associations in EF Code First CTP5: Part 3 ??? One-to-One Foreign Key Associations – Enterprise .Net | ObondO Labs](#)

re: Associations in EF Code First CTP5: Part 3 – One-to-One Foreign Key Associations

Thursday, December 08, 2011 6:08 AM by [dannz](#)

Thank you very much for your wonderful post, and keep on posting such useful things !!!

EF Code First Migrations 2 « mabbled

Friday, January 27, 2012 1:14 PM by [EF Code First Migrations 2 « mabbled](#)

Pingback from [EF Code First Migrations 2 « mabbled](#)

[Terms of Use](#)

Associations in EF Code First: Part 4 – Table Splitting

This is the fourth post in a series that explains entity association mappings with EF Code First. This series includes:

- [Part 1 – Introduction and Basic Concepts](#)
- [Part 2 – Complex Types](#)
- [Part 3 – Shared Primary Key Associations](#)
- Part 4 – Table Splitting
- [Part 5 – One-to-One Foreign Key Associations](#)
- [Part 6 – Many-valued Associations](#)

In the [second part](#) of this series we saw how to map a special kind of one-to-one association—a composition with complex types. We argued that this is usually the simplest way to represent one-to-one relationships which comes with some [limitations](#). We addressed the first limitation (shared references) by introducing shared primary key associations in the [previous blog post](#). In today's blog post we are going to address the third limitation of the complex types by learning about *Table Splitting* as yet another way to map a one-to-one association.

The Motivation Behind this Mapping: A Complex Type That Can be Lazy Loaded

A shared primary key association does not expose us to the third limitation of the complex types regarding [Lazy Loading](#), we can of course lazy/defer load the Address information of a given user but at the same time, it does not give us the same SQL schema as the complex type mapping. After all, it adds a new table for the Address entity to the schema while mapping the Address with a complex type stores the address information in the Users table. So the question still remains there: How can we keep everything (e.g. User and Address) in one single table yet be able to lazy load the complex type part (Address) after reading the principal entity (User)? In other words, how can we have lazy loading with a complex type?

Splitting a Single Table into Multiple Entities

Table splitting (a.k.a. horizontal splitting) enables us to map a single table to multiple entities. This is particularly useful for scenarios that we have a table with many columns where some of those columns might not be needed as frequently as others or some of the columns are expensive to load (e.g. a column with a binary data type).

An Example From the Northwind Database

Unlike the other parts of this series, where we start with an object model and then derive a SQL schema afterwards, in this post we are going to do the reverse, for a reason that you'll see, we will start with an existing schema and will try to create an object model that matches the schema. For that we are going to use the Employees table from the Northwind database. You can download and install Northwind database from [here](#) if you don't have it already installed on your SQL Server. The following shows the Employees table from the Northwind database that we are going to use:

| Column Name | Data Type | Allow Nulls |
|-----------------|---------------|-------------------------------------|
| EmployeeID | int | <input type="checkbox"/> |
| LastName | nvarchar(20) | <input type="checkbox"/> |
| FirstName | nvarchar(10) | <input type="checkbox"/> |
| Title | nvarchar(30) | <input checked="" type="checkbox"/> |
| TitleOfCourtesy | nvarchar(25) | <input checked="" type="checkbox"/> |
| BirthDate | datetime | <input checked="" type="checkbox"/> |
| HireDate | datetime | <input checked="" type="checkbox"/> |
| Address | nvarchar(60) | <input checked="" type="checkbox"/> |
| City | nvarchar(15) | <input checked="" type="checkbox"/> |
| Region | nvarchar(15) | <input checked="" type="checkbox"/> |
| PostalCode | nvarchar(10) | <input checked="" type="checkbox"/> |
| Country | nvarchar(15) | <input checked="" type="checkbox"/> |
| HomePhone | nvarchar(24) | <input checked="" type="checkbox"/> |
| Extension | nvarchar(4) | <input checked="" type="checkbox"/> |
| Photo | image | <input checked="" type="checkbox"/> |
| Notes | ntext | <input checked="" type="checkbox"/> |
| ReportsTo | int | <input checked="" type="checkbox"/> |
| PhotoPath | nvarchar(255) | <input checked="" type="checkbox"/> |

As you can see, this table has a Photo column of image type which makes it a good candidate to be lazy loaded each time we read an Employee from this table.

The Object Model

As the following object model shows, I created two entities: Employee and EmployeePhoto. I also created a unidirectional association between these two by defining a navigation property on the Employee class called EmployeePhoto:

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string LastName { get; set; }
```

```

    public string FirstName { get; set; }
    public string Title { get; set; }
    public string TitleOfCourtesy { get; set; }
    public DateTime? BirthDate { get; set; }
    public DateTime? HireDate { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string Region { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
    public string HomePhone { get; set; }
    public string Extension { get; set; }
    public string Notes { get; set; }
    public int? ReportsTo { get; set; }

    public virtual EmployeePhoto EmployeePhoto { get; set; }
}

public class EmployeePhoto
{
    [Key]
    public int EmployeeID { get; set; }
    public byte[] Photo { get; set; }
    public string PhotoPath { get; set; }
}

public class NorthwindContext : DbContext
{
    public DbSet<Employee> Employees { get; set; }
    public DbSet<EmployeePhoto> EmployeePhoto { get; set; }
}

```

How to Create a Table Splitting with Fluent API?

As also mentioned in the previous post, by convention, Code First always takes a unidirectional association as one-to-many unless we specify otherwise with fluent API. However, the fluent API codes that we have seen so far in this series won't let us create a table splitting. If we mark EmployeePhoto class as a complex type, we wouldn't be able to lazy load it anymore or if we create a shared primary key association then it will look for a separate table for the EmployeePhoto entity which we don't have in the Northwind database. The trick is to create a shared primary key association between Employee and EmployeePhoto entities but then instruct Code First to map them both to the same table. The following code shows how:

```

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Employee>()
        .HasRequired(e => e.EmployeePhoto)
        .WithRequiredPrincipal();

    modelBuilder.Entity<Employee>().ToTable("Employees");
}

```

```

    modelBuilder.Entity<EmployeePhoto>().ToTable("Employees");
}

```

Note how we made both ends of the association required by using [HasRequired](#) and [WithRequiredPrincipal](#) methods, even though both the Photo and PhotoPath columns has been defined to allow NULLs.

See the Lazy Loading of the Dependent Entity in Action

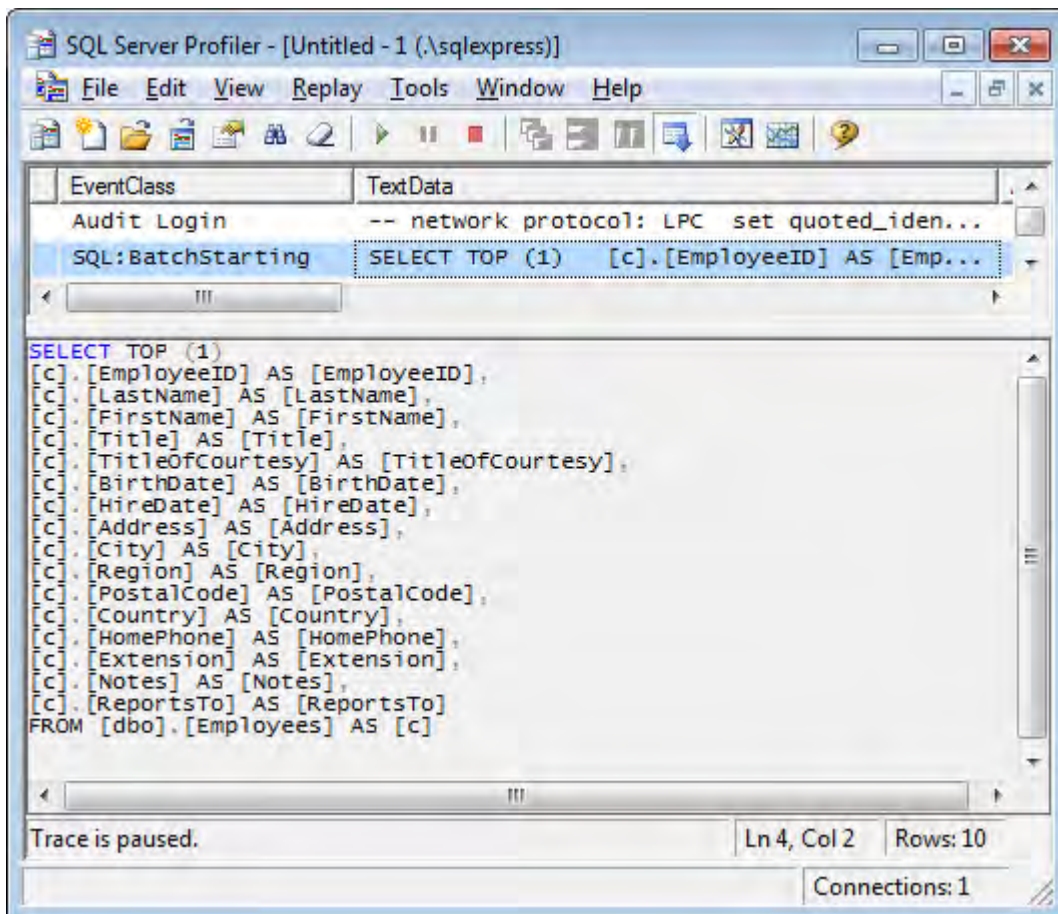
Now it's time to write a test to make sure that EF does not select the Photo column each time we query for an employee:

```

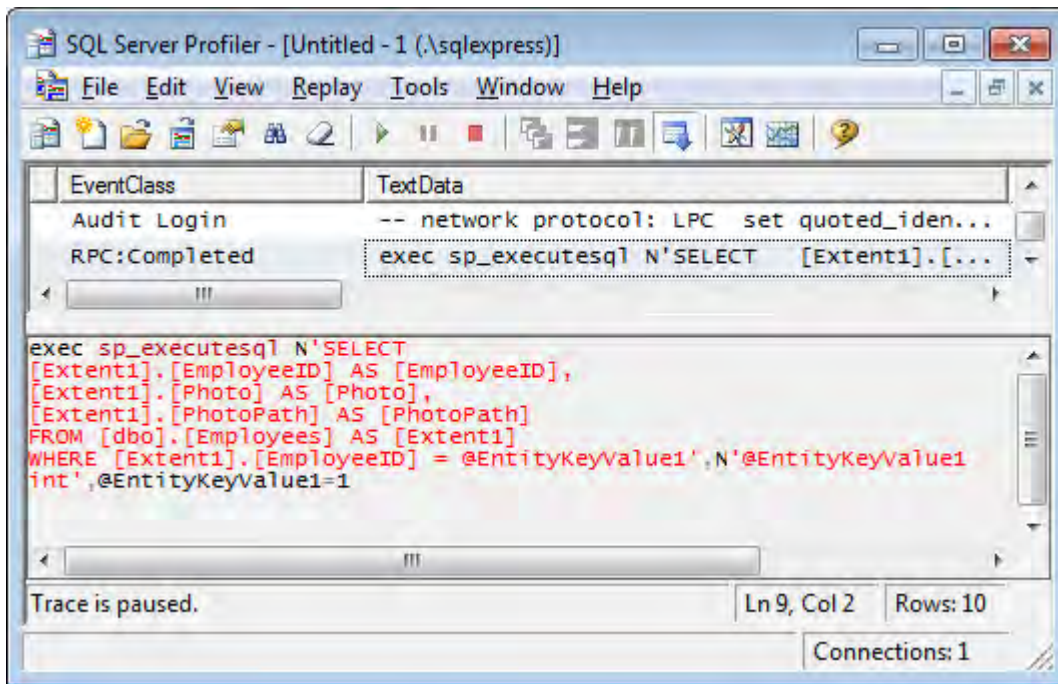
using (var context = new NorthwindContext())
{
    Employee employee = context.Employees.First();
    byte[] photo = employee.EmployeePhoto.Photo;
}

```

The following screen shot from the SQL Profiler shows the query that has been submitted to SQL Server as the result of reading the first employee object:



Accessing the EmployeePhoto navigation property of the employee object on the next line causes EF to submit a second query to the SQL Server to lazy (implicit) load the EmployeePhoto (By default, EF fetches associated objects and collections lazily whenever you access them):




Where to Use this Mapping?

I recommend using Table Splitting only for mapping of the legacy databases, actually that's the reason we start this post from an existing database like Northwind. For [green-field development](#) scenarios consider using [shared primary key association](#) instead. There are several reasons why you may want to split the Employee table to two tables when designing a new physical data model for your application. In fact, it is very common for most applications to require a core collection of data attributes of any given entity, and then a specific subset of the noncore data attributes. For example, the core columns of the Employee table would include the columns required to store their name, address, and phone numbers; whereas noncore columns would include the Photo column. Because Employee.Photo is large, and required only by a few applications, you would want to consider splitting it off into its own table. This would help to improve retrieval access times for applications that select all columns from the Employee table yet do not require the photo. This also works pretty well for EF since it doesn't support lazy loading at the scalar property or complex type level.

Summary

In this post we learned about mapping a one-to-one association with table splitting. It enabled us to have lazy loading for the EmployeePhoto entity, something that we would have missed, had we mapped it with a complex type. We saw that on the database side it looks like a complex type mapping but on the object model it is not a complex type since we mapped EmployeePhoto as an Entity with an object identifier (EmployeeID). In fact, it's a special kind of a shared primary key association where both the principal and dependent entities are mapped to one single table. This somehow exotic one-to-one association mapping should be reserved only for the mapping of existing legacy databases.

Published Sunday, April 24, 2011 9:17 PM by [mortezam](#) 
Filed under: [C#](#), [Code First](#), [Entity Framework 4.1](#)

Comments

[# Associations in EF 4.1 Code First: Part 4 – Table Splitting](#)

Sunday, April 24, 2011 11:53 PM by [progg.ru](#)

Thank you for submitting this cool story - Trackback from progg.ru

[# re: Associations in EF 4.1 Code First: Part 4 – Table Splitting](#)

Tuesday, April 26, 2011 1:37 AM by Vahid hassani

Great post,

thanks

[# re: Associations in EF 4.1 Code First: Part 4 – Table Splitting](#)

Friday, May 06, 2011 10:52 PM by [Assaf S.](#)

I have used this method before but found it a bit annoying as there is no designer support.

I've ended up creating a view to return the columns I need. I then created an entity for the view.

It is not the best solutions as references are not created for the view entity, but at least I don't have to worry when I update my entities from database.

Assaf

[# re: Associations in EF 4.1 Code First: Part 4 – Table Splitting](#)

Monday, May 09, 2011 8:29 AM by LambdaCruiser

I fail to insert a new Employee with the above model and mapping.

My code is simple:

```
using (var context = new NorthwindContext())
{
    Employee e1 = new Employee();
    e1.LastName = "Some last name";
    e1.FirstName = "Some name";
    e1.EmployeePhoto = new EmployeePhoto();
    context.Employees.Add(e1);
    context.SaveChanges();
}
```

The Inner Exception message is "Cannot insert explicit value for identity column in table 'Employees' when IDENTITY_INSERT is set to OFF."

How to fix this?

[# re: Associations in EF 4.1 Code First: Part 4 – Table Splitting](#)

Tuesday, May 10, 2011 6:52 AM by LambdaCruiser

I've found out that the code works if I let EF create a new database.

But I still get the exception when running the code against original Northwind DB. So it's some weird issue with the database I guess.

EF 4.1: Mapear uma tabela para várias entidadesTuesday, May 10, 2011 6:30 PM by [EF 4.1: Mapear uma tabela para várias entidades](#)

Pingback from EF 4.1: Mapear uma tabela para várias entidades

re: Associations in EF 4.1 Code First: Part 4 – Table SplittingTuesday, May 10, 2011 11:22 PM by [mortezam](#)

@LambdaCruiser: Great question! To understand the reason for this somehow weird behavior, let's first take a look at the SQL statement that being generated and submitted to SQL Server by EF as a result of your code that tries to add a new employee object:

```
exec sp_executesql N'insert [dbo].[Employees]([EmployeeID], [LastName], [FirstName], [Title], [TitleOfCourtesy], [BirthDate], [HireDate], [Address], [City], [Region], [PostalCode], [Country], [HomePhone], [Extension], [Notes], [ReportsTo], [Photo], [PhotoPath]) values (@0, @1, @2, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null) ',N'@0 int,@1 nvarchar(max) ,@2 nvarchar(max) ',@0=0,@1=N'Some last name',@2=N'Some name'
```

As you can see above, EF used your specified EmployeeId (which in this case is zero because you haven't specified it) when inserting a new Employee record to the database. However, the EmployeeId is actually an Identity column in Northwind and we can't insert into a column containing an identity (at least not until IDENTITY_INSERT being set to ON) and basically that's the reason for the exception that you are getting.

So now an even bigger question comes up: Why EF includes a value for the EmployeeId when inserting a new Employee record into the database? The answer to this question can be seen from your second experiment when letting Code First creating a new database from your object model instead of using the Northwind. If you take a closer look at the Employee table in your new database, you'll see that unlike Northwind, EmployeeId is not marked as an identity, it's just an integer primary key. What all these mean to us is that we always have to make sure that we are providing unique keys when inserting a new record in an entity splitting scenario because the identity is turned off (or at least supposed to be turned off) on the table's primary key.

Back to your question, unfortunately there is not much you can do when dealing with a legacy database like Northwind other than going ahead and manually switching off identity on the EmployeeId column and then like I said, it's your responsibility to provide valid unique keys when adding a new Employee object. Something like the following code will work (again, after you switching off identity on EmployeeId):

```
using (var context = new NorthwindContext())
{
    Employee e1 = new Employee()
    {
        EmployeeID = 1,
        LastName = "Some last name",
        FirstName = "Some name",
        EmployeePhoto = new EmployeePhoto()
        {
            EmployeeID = 1
        }
    }
};
context.Employees.Add(e1);
```

```
context.SaveChanges();
}
```

There is a catch here though: Like you see in the above code, you need to set the primary key on both Employee and EmployeePhoto entities or it won't generate the proper SQL statement (well, strangely enough, it will still work if you only set it on the EmployeePhoto but not the other way around). Hope this helps and thanks again for your question. I'll add a new section to the post to clarify this matter.

re: Associations in EF 4.1 Code First: Part 4 – Table Splitting

Thursday, May 12, 2011 6:57 AM by LambdaCruiser

Thank you for a very detailed answer.

Indeed, EF infers that the EmployeeId is `_not_` an Identity in this mapping. I think the answer to this behaviour is in Part 3 of this post series.

"a very important consequence of the principal/dependent decision for one-to-one associations: the dependent primary key will become non-Identity by default."

I changed part of the mapping from:

```
modelBuilder.Entity<Employee>()
    .HasRequired(e => e.EmployeePhoto)
    .WithRequiredDependent();
```

to

```
modelBuilder.Entity<Employee>()
    .HasRequired(e => e.EmployeePhoto)
    .WithRequiredPrincipal();
```

Now the inserts seem to work as expected, the Id is treated like an Identity.

re: Associations in EF 4.1 Code First: Part 4 – Table Splitting

Thursday, May 19, 2011 11:49 PM by dencio

@mortezaam: Thanks for the great post.

@LambdaCruiser: Thanks for pointing that out.

re: Associations in EF 4.1 Code First: Part 4 – Table Splitting

Saturday, May 28, 2011 4:06 PM by [mortezaam](#)

@LambdaCruiser: You are absolutely right. Let me clarify why your fluent API code creates the correct schema in terms of making the PK to be an identity column. First, let's revisit your fluent API code again:

```
modelBuilder.Entity<Employee>().HasRequired(e => e.EmployeePhoto).WithRequiredPrincipal();
```

From the documentation for the [WithRequiredPrincipal](#) method:

"WithRequiredPrincipal method Configures the relationship to be required:required with a navigation property on the other side of the relationship. The entity type being configured will be the principal in the relationship. The entity type that the relationship targets will be the dependent and contain a foreign key to the principal."

The important point is that the entity type being configured is always the one we start our fluent API code with, which in this example is the Employee entity. The entity type that the relationship targets is always the one we specify by the first fluent API method (after the Entity method) which is HasRequired in this

example and it targets EmployeePhoto entity. Therefore, we can infer that Employee is the principal and EmployeePhoto is the dependent in this fluent API code which is exactly what we are looking for. As a result, Code First makes the EmployeeId an identity in the resulting DB schema. Thank you :)

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet

Monday, June 27, 2011 12:20 PM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet](#)

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet - Bilim-Teknoloji | Positive Pozitive.Net

Monday, June 27, 2011 3:46 PM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet - Bilim-Teknoloji | Positive Pozitive.Net](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet - Bilim-Teknoloji | Positive Pozitive.Net](#)

ASP.NET, ASP.NET MVC, .NET and NuGet | 哈哈808,开心呵呵网

Sunday, July 03, 2011 9:21 PM by [ASP.NET, ASP.NET MVC, .NET and NuGet | 哈哈808,开心呵呵网](#)

Pingback from [ASP.NET, ASP.NET MVC, .NET and NuGet | 哈哈808,开心呵呵网](#)

Enlaces: ASP.NET, ASP.NET MVC, .NET y NuGet « Thinking in .NET

Monday, July 11, 2011 2:56 PM by [Enlaces: ASP.NET, ASP.NET MVC, .NET y NuGet « Thinking in .NET](#)

Pingback from [Enlaces: ASP.NET, ASP.NET MVC, .NET y NuGet « Thinking in .NET](#)

Links: ASP.NET, ASP.NET MVC, .NET and NuGet | Tu???'n's blog

Friday, July 15, 2011 7:58 AM by [Links: ASP.NET, ASP.NET MVC, .NET and NuGet | Tu???'n's blog](#)

Pingback from [Links: ASP.NET, ASP.NET MVC, .NET and NuGet | Tu???'n's blog](#)

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet - Technology | Zeytin.Net

Monday, July 18, 2011 4:06 PM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet - Technology | Zeytin.Net](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet - Technology | Zeytin.Net](#)

re: Associations in EF 4.1 Code First: Part 4 – Table Splitting

Monday, August 01, 2011 1:41 AM by [richardbhong](#)

Thanks for this great post. Very detailed and guide. I just subscribe to your RSS> Looking forward for more. Cheers!!!

re: Associations in EF 4.1 Code First: Part 4 – Table Splitting

Monday, August 08, 2011 5:19 PM by [akanmuratcimen](#)

how can we use same properties in different models?

re: Associations in EF 4.1 Code First: Part 4 – Table Splitting

Monday, September 19, 2011 7:57 PM by Mark Phillips

Thanks for the post.

Is there a way to set the key in EmployeePhoto using the Fluent API instead of the DataAnnotations [Key]

attribute?

Thanks

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog

Tuesday, September 20, 2011 1:00 AM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog](#)

re: Associations in EF 4.1 Code First: Part 4 – Table Splitting

Tuesday, September 20, 2011 10:44 PM by [morteza](#)

@Mark Phillips: Yes, fluent API offers [HasKey\(\)](#) method to configure the primary key property(s) for an entity like the following code:

```
modelBuilder.Entity<EmployeePhoto>().HasKey(e => e.EmployeeID);
```

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog

Wednesday, September 28, 2011 1:00 AM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog](#)

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet | Freedom Developers

Thursday, September 29, 2011 2:48 AM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet | Freedom Developers](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet | Freedom Developers](#)

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog

Friday, October 21, 2011 1:13 PM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog](#)

re: Associations in EF 4.1 Code First: Part 4 – Table Splitting

Friday, October 28, 2011 1:47 AM by [ddredstar@hotmail.com](#)

How about the EmployeePhoto is optional for Employee? I have try HasOptional().WithRequired(), but it seems not works. Could you help to instruct me how to implement this case.

re: Associations in EF 4.1 Code First: Part 4 – Table Splitting

Thursday, November 03, 2011 10:11 PM by [morteza](#)

@ddredstar@hotmail.com: Because you are mapping both entities to the same table, EF expects you to configure a required one-to-one relationship between them and the fluent API code that I showed is the *only* way to achieve Table Splitting in code first be it required or optional. That said, you can still workaround this limitation by allowing NULL in the related entity columns (like I did for EmployeePhoto.Photo and EmployeePhoto.PhotoPath). Then when working with the object model, always use NULLs to indicate an optional association. For example, the following code shows how you can save an Employee without a Photo:

```
Employee employee = new Employee()
```

```
{
    EmployeePhoto = new EmployeePhoto()
};
context.Employees.Add(employee);
context.SaveChanges();
```

re: Associations in EF 4.1 Code First: Part 4 – Table Splitting

Tuesday, November 15, 2011 5:01 PM by [maximusmd](#)

Is this method the same as TPT? I am so confused. This EF thing is so big and so complicated :(

re: Associations in EF Code First: Part 4 – Table Splitting

Tuesday, December 13, 2011 10:12 AM by parleer

I tried implementing this pattern to separate a large table into multiple discrete types but it did not work. I received an error "The Entity types 'CampaginFeedback' and 'CampaignSurvey' cannot share table 'Campaign' because they are not in the same type hierarchy or do not have a valid one to one foreign key relationship with matching primary keys between them."

My classes look like this (simplified here):

```
public class Campaign {
    [Key]
    public int CampaignId {get;set;}
    public string Name {get;set;}
}
public class CampaignSurvey {
    [Key]
    public int CampaignId {get;set;}
    public string Question {get;set;}
    public string Answer {get;set;}
}
public class CampaignFeedback {
    [Key]
    public int CampaignId {get;set;}
    public string Feedback {get;set;}
}
```

What am I doing wrong?

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet ??? ASP.NET Chronicles

Saturday, December 24, 2011 8:20 PM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet ??? ASP.NET Chronicles](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet ??? ASP.NET Chronicles](#)

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet ??? ASP.NET Chronicles

Saturday, December 24, 2011 8:36 PM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet ??? ASP.NET Chronicles](#)

[ASP.NET Chronicles](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet ??? ASP.NET Chronicles](#)

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet ??? ASP.NET Chronicles

Saturday, December 24, 2011 8:47 PM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet ??? ASP.NET Chronicles](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet ??? ASP.NET Chronicles](#)

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog

Tuesday, December 27, 2011 1:00 AM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet – HD Software Co. Blog](#)

June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet | nalli.net

Saturday, January 07, 2012 11:13 PM by [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet | nalli.net](#)

Pingback from [June 26th Links: ASP.NET, ASP.NET MVC, .NET and NuGet | nalli.net](#)

[Terms of Use](#)

Associations in EF Code First: Part 5 – One-to-One Foreign Key Associations

This is the fourth post in a series that explains entity association mappings with EF Code First. I've covered the following posts so far:

- [Part 1 – Introduction and Basic Concepts](#)
- [Part 2 – Complex Types](#)
- [Part 3 – Shared Primary Key Associations](#)
- [Part 4 – Table Splitting](#)
- Part 5 – One-to-One Foreign Key Associations
- [Part 6 – Many-valued Associations](#)

In the [third part](#) of this series we saw the limitations of shared primary key association and argued that one-to-one associations are relatively rare and in many schemas, a one-to-one association is represented with a foreign key field. In this post, we are going to discuss how this is done by learning about one-to-one foreign key associations.

Introducing the Revised Model

In this revised version, each User always have two addresses: one billing address and another one delivery address. The following diagram demonstrates the domain model:



One-to-One Foreign Key Association

Instead of [sharing a primary key](#), two rows can have a foreign key relationship. One table has a foreign key that references the primary key of the associated table (The source and target of this foreign key constraint can even be the same table, representing a self-referencing relationship.). [An additional constraint enforces this relationship as a real one to one](#). In our example, if we declare that a particular address can be referenced by at most one User, we declare that a particular address can be referenced by at most one User. If we also declare that the BillingAddressId column is unique, we declare that a particular address can be referenced by at most one User.

isn't as strong as the guarantee from a shared primary key association, which allows a particular ad one user, period. With several foreign key columns (which is the case in our domain model since we have `DeliveryAddress`), we can reference the same address target row several times. But in any case, tw for the same purpose.

The Object Model

Let's start by creating an object model for our domain:

```
public class User
{
    public int UserId { get; set; }
    public string Name { get; set; }
    public int BillingAddressId { get; set; }
    public int DeliveryAddressId { get; set; }

    public Address BillingAddress { get; set; }
    public Address DeliveryAddress { get; set; }
}

public class Address
{
    public int AddressId { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public string ZipCode { get; set; }
}

public class Context : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Address> Addresses { get; set; }
}
```

As you can see, `User` class has introduced two new scalar properties as `BillingAddressId` and `DeliveryAddressId` navigation properties (`BillingAddress` and `DeliveryAddress`).

Configuring Foreign Keys With Fluent API

`BillingAddressId` and `DeliveryAddressId` are foreign key scalar properties representing the actual foreign keys that are established on. However, Code First will not recognize them as the foreign keys for the associations aligned with the [conventions](#) that it has to infer foreign keys. Therefore, we need to use fluent API (Code First) to know about the foreign key properties. The following fluent API code shows how:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>()
        .HasRequired(a => a.BillingAddress)
        .WithMany()
        .HasForeignKey(u => u.BillingAddressId);
}
```



```

    modelBuilder.Entity<User>()
        .HasRequired(a => a.DeliveryAddress)
        .WithMany()
        .HasForeignKey(u => u.DeliveryAddressId);
}

```

Alternatively, we can use Data Annotations to achieve this. EF 4.1 introduced a new attribute in [System.ComponentModel.DataAnnotations](#) namespace called [ForeignKeyAttribute](#). We can place it the property that represents the foreign key of the relationship:

```

public class User
{
    public int UserId { get; set; }
    public string Name { get; set; }
    public int BillingAddressId { get; set; }
    public int DeliveryAddressId { get; set; }

    [ForeignKey("BillingAddressId")]
    public Address BillingAddress { get; set; }

    [ForeignKey("DeliveryAddressId")]
    public Address DeliveryAddress { get; set; }
}

```

That said, we won't use this data annotation and will go with the fluent API way for a reason that you

Creating a SQL Server Schema

The object model seems to be ready to give us the desired SQL schema, however, if we try to create it will get an [InvalidOperationException](#) with this message:

The database creation succeeded, but the creation of the database objects did not. See InnerException details.

The inner exception is a [SqlException](#) containing this message:

Introducing FOREIGN KEY constraint 'User_DeliveryAddress' on table 'Users' may cause cycles or multiple cascade paths. Specify ON DELETE NO ACTION or ON UPDATE NO ACTION, or modify other FOREIGN constraints. Could not create constraint. See previous errors.

As you can tell from the type of the inner exception ([SqlException](#)), it has nothing to do with EF or C# purely by SQL Server when Code First was trying to create a database based on our object model.

What's a Multiple Cascade Path Anyway?

A *Multiple Cascade Path* happens when a cascade path goes from column col1 in table A to table B to table C. For example in our case Code First attempted to turn on cascade delete for both BillingAddress and DeliveryAddress columns in the Users table. In fact, Code First was trying to use [Declarative Referential Integrity \(DRI\)](#) the problem is that SQL Server is not fully [ANSI SQL-92](#) compliant when it comes to the cascading updates or deletes in a multiple cascade path scenario.

A [KB article](#) also explains why we received this error:

"In SQL Server, a table cannot appear more than one time in a list of all the cascading referential actions on a DELETE or an UPDATE statement. For example, the tree of cascading referential actions must only appear once in the cascading referential actions tree".

And it exactly applies to our example: The User table appeared twice in a list of cascading referential actions on the Addresses table. Basically, SQL Server does simple counting of cascade paths and, rather than actually exist, it assumes the worst and refuses to create the referential actions (cascades). Therefore you may or may not get this exception.

Overriding The Code First Convention To Resolve the Problem

As you saw, Code First automatically turns on cascade delete on a *required* one-to-many association. However, in order to resolve the exception that we got from SQL Server, we have no choice other than to change the behavior detected by convention. Basically we need to switch cascade delete off on at least one of the associations. There is no way to accomplish this other than using fluent API. Let's switch it off on DeliveryAddress

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>()
        .HasRequired(a => a.BillingAddress)
        .WithMany()
        .HasForeignKey(u => u.BillingAddressId);

    modelBuilder.Entity<User>()
        .HasRequired(a => a.DeliveryAddress)
        .WithMany()
        .HasForeignKey(u => u.DeliveryAddressId).WillCascadeOnDelete(false);
}
```

One-to-One Foreign Key Associations in EF Code First

As you may have noticed, both associations in the fluent API code has been configured as a *many-to-one* association, which is what we have expected. The reason is simple: Code First (and EF in general) does not natively support one-to-one association, EF does not support any association scenario that involves unique constraints at all. Fortunately, we can work around this by making the *many* side of the association (User) have a property that is a reference to an instance of another entity (Address) and use a *one-to-one* relationship. EF (of course) still thinks that the relationship is many-to-one. This is a workaround for one-to-one associations that comes with two consequences: First, EF won't create any additional constraint for us to enforce the one-to-one relationship, we need to manually create it ourselves. The second limitation that this lack of support impose to us is key associations cannot be bidirectional (e.g. we cannot define a property for the User on the Address entity).

Create a Unique Constraint To Enforce the Relationship as a One to One

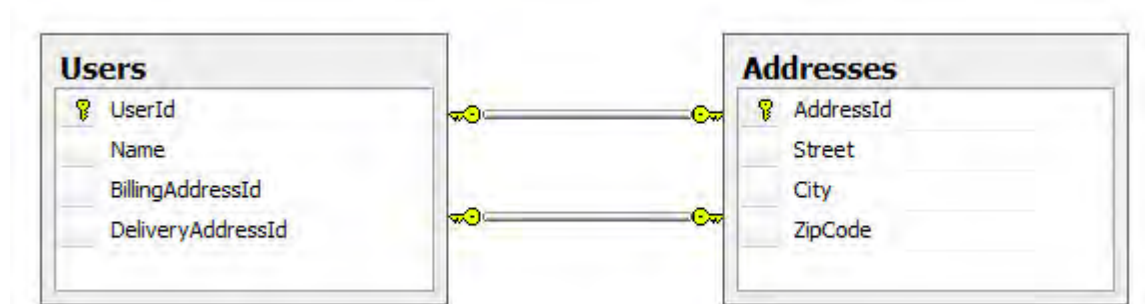
We can manually create unique constraints on the foreign keys in the database after Code First creates the database and prefer to create your database in one shot then there is a way to have Code First create the constraints during the creation process. For that we can take advantage of the new EF 4.1 [ExecuteSqlCommand](#) method to execute SQL commands to be executed against the database. The best place to invoke ExecuteSqlCommand is in the [Seed](#) method that has been overridden in a custom initializer class:

```
protected override void Seed(Context context)
{
    context.Database.ExecuteSqlCommand("ALTER TABLE Users ADD CONSTRAINT uc_BillingId UNIQUE (BillingAddressId)");
    context.Database.ExecuteSqlCommand("ALTER TABLE Users ADD CONSTRAINT uc_DeliveryId UNIQUE (DeliveryAddressId)");
}
```

This code adds unique constraints to the BillingAddressId and DeliveryAddressId columns in the Users table.

SQL Schema

The object model is ready now and will result in the following database schema:



It is also worth mentioning that we can still enforce cascade deletes for the Delivery Address relationship. We can write a [Delete Trigger](#) on the primary table that either deletes the rows in the dependent table or sets the foreign keys to NULL (In our case the foreign keys are Non-Nullable so it has to delete the dependent rows).

Source Code

Click [here](#) to download the source code for the one-to-one foreign key association sample that we have.

Summary

In this post we learned about one-to-one foreign key associations as a better way to create one-to-one relationships in EF. We also learned about the limitations such as the need for manual creation of unique constraints and also the fact that this type of relationship is bidirectional, all due to the lack of unique constraint support in EF. The good news is that the ADO.NET 4.1 has support for unique constraints in EF but support for unique constraints requires changes to the whole EF stack in a major release of EF (EF 4.1 is merely layered on top of the current .NET 4.0 functionality) and until then here is going to be the way to implement one-to-one foreign key associations in EF Code First.

Published Sunday, May 01, 2011 6:35 PM by [morteza](#) ★★★★★
 Filed under: [C#](#), [Code First](#), [Entity Framework 4.1](#)

Comments

re: [Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations](#)

Monday, May 02, 2011 2:07 AM by Prajakta

Nice article.

But not able insert records.

Getting error that u should provide values of parent table which is not null.

Can you please tell me why this is happening.?

[# re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations](#)

Monday, May 02, 2011 9:14 AM by [morte zam](#)

@Prajakta: Could you please show your code that causes the exception?

[# re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations](#)

Wednesday, May 04, 2011 6:32 PM by Frank

Excellent work. Wondering if you could help me.

I have a class User. User class is 1:1 to a Campaign class. User also has a 1:many on UserImages. When I add a new UserImage to my user. `UserImages.Add(new UserImage())` and look at the UserImages table, a proper UserId gets inserted, but the CampaignId defaults to 0 on the UserImages table (default int value). The actual User instance user has the correct CampaignId of 1 and not 0. Would have thought it would be setting it as well. UserImage class has a property CampaignId.

Been really struggling with this one. Any direction would be appreciated.

[# re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations](#)

Thursday, May 05, 2011 11:16 AM by [morte zam](#)

@Frank: Thanks. Could you please show your object model as well as the client code that adds a new UserImage to the database?

[# re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations](#)

Thursday, May 05, 2011 11:54 AM by Frank

POCOs:

```
public class User
{
    public Guid UserId { get; set; } //have to use Guid as it maps to aspnet db
    public int CampaignId { get; set; }
    public virtual ICollection<UserImage> { get; set; }
}
```

```
public class UserImage
{
    public int UserImageId { get; set; }
    public Guid UserId { get; set; }
    public int CampaignId { get; set; }
}
```

```
public class Campaign
{
    public int CampaignId { get; set; }
    public virtual ICollection<User> Users { get; set; }
}
```

User Mapping:

```

HasMany(u => u.UserImages)
.WithOptional()
.WillCascadeOnDelete();

```

```

HasMany(u => u.UserCampaigns)
.WithOptional()
.WillCascadeOnDelete();

```

Campaign Mapping:

```

HasKey(c => c.CampaignId)
.HasMany(c => c.Users)
.WithMany(u => u.Campaigns)
.Map(m => m.MapLeftKey("UserId"));

```

No mapping for UserImage

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Thursday, May 05, 2011 12:14 PM by Frank

Morteza,

So the user object has a campaigns collection and in my case, it will contain one campaign in the collection. The UserImage has a property called CampaignId. Would I have to set that manually? If so, any other way I can remodel my classes so the CampaignId gets set automatically when adding a UserImage?

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Thursday, May 05, 2011 9:51 PM by [morteza](#)

@Frank: Your fluent API code doesn't match your object model, but I still got the idea. Yes, adding a new UserImage to the UserImages collection wouldn't cause UserImage.CampaignId to be populated since the association between User and UserImage has nothing to do with the association between UserImage and Campaign classes. In fact, I don't see any reason to have yet another association between UserImage and Campaign entities by defining a CampaignId property on UserImage class because once you have a UserImage object you can simply access the Campaign information by accessing it on the related User (something like userImage.User.UserCampaign). Therefore, I would design the object model slightly different:

```

public class User
{
    public Guid UserId { get; set; }
    public int? CampaignId { get; set; }
    public virtual Campaign UserCampaign { get; set; }
    public virtual ICollection<UserImage> UserImages { get; set; }
}

public class UserImage
{
    public int UserImageId { get; set; }
    public Guid UserId { get; set; }
    public User User { get; set; }
}

public class Campaign
{
    public int CampaignId { get; set; }
}

```

```

}

public class Context : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<UserImage> UserImages { get; set; }
    public DbSet<Campaign> Campaigns { get; set; }
}

```

And then you can work with the object model like the following code:

```

using (var context = new Context())
{
    UserImage userImage = new UserImage()
    {
        User = new User()
        {
            UserCampaign = new Campaign()
        }
    };

    context.UserImages.Add(userImage);
    context.SaveChanges();
}

```

You can also use the method I showed in this post to enforce a one to one relationship between User and UserCampaign entities. Hope this helps.

[# re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations](#)

Thursday, May 05, 2011 11:45 PM by Frank

Thank you for taking the time. I got you and it makes sense assuming you only have 1:1 relationship. What I was trying to say (unsuccessfully) is that there could be possibilities of multiple campaigns. So it's actually many-to-many between user and campaign. So I think I'm left to setting the CampaignId manually. I even tried creating a dynamic property called ActiveCampaign which always returns the current campaign but it's useless and is not forcing the CampaignId to be set.

I welcome any other thoughts.

[# re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations](#)

Tuesday, May 10, 2011 7:02 PM by [asrfarinha](#)

Hello,

I'm trying to use this approach to achieve a one-to-one association. In my case there's an association between a User and a Team, and I need a navigation property in each of them.

I bumped into a problem when adding data.

This is what the models look like:

```

public class Team
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int OwnerID { get; set; }

    public virtual User Owner { get; set; }
}

```

```

}

public class User
{
    public int ID { get; set; }
    public string UserName { get; set; }
    public int TeamID { get; set; }

    public virtual Team Team { get; set; }
}

```

I added these bits to the DbContext:

```

modelBuilder.Entity<User>()
    .HasRequired(u => u.Team)
    .WithMany()
    .HasForeignKey(u => u.TeamID);

modelBuilder.Entity<Team>()
    .HasRequired(t => t.Owner)
    .WithMany()
    .HasForeignKey(t => t.OwnerID)
    .WillCascadeOnDelete(false);

```

And now when adding data like this:

```

u = new User();
u.UserName = "farinha";
t = new Team("Flour Power");
u.Team = t;
t.Owner = u;

```

```

context.Users.Add(u);
context.Teams.Add(t);
context.SaveChanges();

```

or even like this:

```

u = new User();
u.UserName = "farinha";
u.Team = new Team("Flour Power");

context.Users.Add(u);
context.SaveChanges();

```

I'm getting the following error:

Unable to determine a valid ordering for dependent operations. Dependencies may exist due to foreign key constraints, model requirements, or store-generated values.

Any idea of how to solve this? Or should I be adding the data in some different way?

Thanks in advance

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Wednesday, May 11, 2011 10:28 AM by [morteza](#)

@asrfarinha: Of course it won't work. Think about it, you are trying to add a new user which has a team, and the team itself is also new and this new team has the very same user as its owner. When you invoke SaveChanges(), EF tries to add these two new objects, now if it tries to add the User first, then it would need a TeamId to send along with other values, if it tries to add the Team object first in order to obtain a TeamId, then it would need an OwnerId which has to be obtained from the User record. Therefore, having two foreign keys that referencing each other table's primary keys created a mutual dependency that would stop you from inserting related records in one single unit of work (i.e. a call to SaveChanges method).

Although there are ways to solve this issue and proceed with the inserts but you should be aware that you haven't really created a one-to-one FK association here. In fact, you've created two unidirectional one-to-many associations between User and Team which are totally unrelated and have nothing to do with each other. Basically if you need to have a bidirectional one-to-one association between your entities, then you should consider creating a [shared primary key association](#) instead since like I explained in the post, one-to-one foreign key associations cannot be bidirectional. Hope this helps.

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Wednesday, May 11, 2011 3:46 PM by [asrfarinha](#)

Thanks for the reply. I believe I managed to solve this by making a few changes:

stackoverflow.com/.../entity-framework-saving-data-in-one-to-one-associations

These one-to-one associations are somewhat weird and uncommon, and that's why I'm a bit lost as to how to implement this one. According to the sample of the model I described, do you reckon a shared primary key association would be a better option? Thanks.

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Thursday, May 12, 2011 1:03 PM by [mortezam](#)

@asrfarinha: Like I said, there are ways to get over this exception and save the dependent objects like the one that has been proposed on Stackoverflow but the main problem in essence that you can't have a bidirectional association when creating a one-to-one FK relationship is still out there. Unfortunately you don't have much of a choice here, if you need a one-to-one bidirectional association then shared primary key is the only type that EF currently supports and it is your only option.

That being said, if your domain model allows at least one of the navigation properties to be read-only then there is a way to make your one-to-one FK association bidirectional. The trick is to replace the missing navigation property with a custom query. For example, let's assume that you need to read/write the Owner information of a Team object (hence the Owner navigation property) but on the User side you only need to read the Team information of a User object:

```
public class Team
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int OwnerID { get; set; }

    public virtual User Owner { get; set; }
}

public class User
{
    public int ID { get; set; }
    public string UserName { get; set; }

    [NotMapped]
    public Team Team { get; set; }
}

public class Context : DbContext
{
```



```

public Context()
{
    ((ObjectContextAdapter) this).ObjectContext.ObjectMaterialized += OnObjectMaterialized;
}

public DbSet<Team> Teams { get; set; }
public DbSet<User> Users { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Team>()
        .HasRequired(t => t.Owner)
        .WithMany()
        .HasForeignKey(t => t.OwnerID);
}

private void OnObjectMaterialized(object sender, ObjectMaterializedEventArgs e)
{
    if (e.Entity is User)
    {
        User user = (User) e.Entity;
        user.Team = this.Teams.Single(t => t.OwnerID == user.ID);
    }
}
}

```

As you can see above, every time you retrieve a User, the Team property gets populated at the time of object materialization. Hope you find this useful.

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Thursday, May 12, 2011 4:10 PM by [asrfarinha](#)

Uff... I'm getting even more confused as to what option should I follow.

In practice, both the User and the Team entities will be created at the same time, never deleted, and a Team won't ever change user. There will be exactly one Team per User and exactly one User per team. I'll need to have a way to navigate to the Team from the User and vice-versa.

I'm almost leaning towards having it all in the same table as the same entity, but it doesn't make a lot of sense from a OO point of view, if you're thinking about each object as a "real-world" Entity.

And thinking forwards, maybe a User will be allowed to have more than one Team in the future (maybe). So having them in separate tables would be good if I ever decide to change.

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Thursday, May 12, 2011 8:00 PM by [mortezaam](#)

@asrfarinha: If that's the case then don't create a one-to-one shared primary key. Your best bet would be to create a one-to-one FK association and make it bidirectional like the way I showed above. The only change you need to do is to have a real navigation property on User object (User.Team) and make the Owner property on Team object to be a read only query based navigation property since you said a Team would never change an Owner. If in the future the requirement for having more than one Team for a User comes up, then all you need to do is to drop the unique constraint on OwnerID and change User.Team to be of type ICollection<User> and your database schema would perfectly support that.

Associations in EF 4.1 Code First: Part 6 ??? Many-valued Associations

Tuesday, May 17, 2011 8:20 PM by [Associations in EF 4.1 Code First: Part 6 ??? Many-valued Associations](#)

Pingback from [Associations in EF 4.1 Code First: Part 6 ??? Many-valued Associations](#)

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Wednesday, May 18, 2011 1:35 PM by Dan

Great to see you finish this second series on the 4.1 release. Extremely helpful!

I'm wondering how you might enforce a 1 to 0..1 relationship on the DB though? Consider a uni-directional relationship between Person and User, where a Person has public int? UserId and public User User (you cannot navigate from User to Person).

EF maps this as a one-to-many, where technically in the DB a User can have 0..n People. In this case putting a unique index constraint on Person.UserId won't work, as the DB will throw a DbException as soon as you try and insert a second Person row with null UserId.

Triggers? Or is there something easier?

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Friday, May 20, 2011 12:35 AM by [mortezam](#)

@Dan: Great question! First of all, I should mention that this is a limitation of SQL Server as according to the ANSI standards SQL:92, SQL:1999, and SQL:2003, a Unique constraint should disallow duplicate non-NULL values, but allow multiple NULL values. In SQL Server however, a single NULL is allowed but multiple NULLs are not. There are a couple of workarounds to this problem but if you are using SQL Server 2008, then your best bet is to define a *Unique Filtered Index* based on a predicate that excludes NULLs. Consider the following as the object model for your scenario:

```
public class User
{
    public int UserId { get; set; }
    public string Username { get; set; }
}

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

    public User User { get; set; }
}

public class Context : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Person> Persons { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>()
            .HasOptional(p => p.User)
            .WithMany()
            .HasForeignKey(u => u.UserId);
    }
}
```

```

}
}

```

And here is how we can create a filtered index to enforce an *optional* one-to-one relationship between User and Person:

```

protected override void Seed(Context context)
{
    context.Database.ExecuteSqlCommand("CREATE UNIQUE NONCLUSTERED INDEX
idx_UserId_NotNULL ON People(UserId) WHERE UserId IS NOT NULL");
}

```

Now you can have as many Persons as you want with a NULL UserId while it still disallows Persons with a same UserId. Thanks for your comment and hope this helps :)

Code First ??????????????

Tuesday, May 24, 2011 11:27 AM by [Code First ??????????????](#)

Pingback from [Code First ??????????????](#)

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Friday, May 27, 2011 6:49 AM by Fred

Morteza, I don't understand section "One-to-One Foreign Key Associations in EF Code First" where to make a 1:1 you must instead model a n:1 and then set a unique constraint via T-SQL.

See these posts: stackoverflow.com/.../primary-foreign-key-in-entity-framework AND stackoverflow.com/.../cascade-delete-rule-in-ef-4-1-code-first-when-using-shared-primary-key-associatio

They show there is an easier way than that described here. Can you please clarify, as your blog is teh first place I stop for info! :)

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Saturday, May 28, 2011 1:07 PM by [mortezam](#)

@Fred: Those Stackoverflow threads discuss about [shared primary key associations](#) which I fully explained in the third part of this series. Like I described in that post, shared primary keys have a few limitations (like difficulty in saving related objects or the impossibility for having multiple dependent navigation properties) which don't make them uncommon but relatively rare and in many schemas, a one-to-one association is represented with a foreign key field and a unique constraint which is the motivation behind the mapping described in this post. Please take a look at the part 3 of this series if you haven't already. Hope this series help you choose the best mapping type for your one to one relationship scenarios :)

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Saturday, May 28, 2011 4:59 PM by Fred

OK I see now. The the "shared" 1:1 scenario is where you have PK of one table related to PK of another table.

This scenario is also 1:1 but PK of one table relates to FK of another table. And the approach is to model n:1 and then convert it to 1:1 in the db, using a unique constraint.

This is what I don't get: I can model it as 1:1 and it still works for me? Unless I am doing something wrong. I

can do this:

```
1) .HasRequired(q => q.SomeEntity)
   .WithRequiredPrincipal()           <-- note I didn't use .WithMany()
   .Map(m => m.MapKey("SomeOtherEntityID"))
   .WillCascadeOnDelete(true);
```

2) set unique constraint in db

So I model a 1:1, and I exactly that, and I also get cascade deletes without using triggers. I trust your code, so I don't understand why mine is working? I am using EF4.1.

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Sunday, May 29, 2011 7:35 PM by [morteza](#)

@Fred: Your fluent API code precisely creates an *independent one-to-one association*. In EF we usually create a 1:1 association by making the PK of the dependent entity to be also a FK of the principal entity, something we called a shared primary key association which you saw in the third part of this series. But let's say you don't like this and want to create a 1:1 association on a column other than the PK (and yet want to keep the association as a 1:1). EF would let you do that with one restriction which is you can't expose the FK property in your object model and that's exactly what you did with your fluent API code. This of course works, but you have to be aware of 2 caveats:

1. If you look at your database you'll see that while your object model showing a 1:1 association, the resulting relationship on the database is not, it's a typical 1:* relationship and if you reverse engineer the generated database into an EDMX file, you'll see that the relationship will be even picked up as a 1:* association by EF! Long story short, you still have to create a unique constraint on the FK (SomeOtherEntityID in your example) yourself, if you want to ensure your database consistency.

2. Like I mentioned, your code essentially creates an independent association as opposed to the FK association I have created in this article. Independent associations are the only type of relationship available in the first release of the EF and have been obsoleted ever since FK associations introduced in EF 4.0. The reason for that is because changing relationships between entities and concurrency checks was really difficult especially in N-Tier application scenarios when using independent associations. Therefore, the recommendation is to use this new foreign key association feature whenever possible.

If you are using your fluent API code to have a 1:1 bidirectional association, as that's the only benefit I can see with your mapping, then have a look at my answer to Asrfarinha above where I show a trick to make a one-to-one FK association bidirectional. Hope this helps.

Entity Framework saving data in one-to-one associations - Programmers Goodies

Thursday, September 15, 2011 4:56 PM by [Entity Framework saving data in one-to-one associations - Programmers Goodies](#)

Pingback from [Entity Framework saving data in one-to-one associations - Programmers Goodies](#)

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Friday, October 14, 2011 2:18 PM by Michel C

All of this is confusing. I guess its because EF doesnt really support 1:1 FK relations.

In my scenario i have a main class that has many related 1:1 classes.

A 1-1 B

A 1-1 C

A 1-1 D

C 1-* E

what i need is to be able to have A.B .. and B.A .. so that i can do B.A.C.SomeProp += someval etc.

At this point, i'm not 100% sure which direction i should take.

Oh, and i'm working off an existing database and getting relations done properly is a nightmare when it should be simple. What i don't understand is why FK isnt enough. (I'm also thinking of using lists and having a property that takes the first element of the list for the many side which i want as a 1 side)

There also seems to be some importance into how the entities are created

```
A = new A() { B = new B() }
```

```
context.AList.Add(A)
```

may work but

```
B = new B() { A = new A() }
```

```
context.BList.Add(B)
```

could fail.

So there seems to be a logical way to set members that makes sense depending on how the relations are set which is not explained anywhere. At least i havent found somewhere explaining it.

Is this something you plan on explaining at some point ?

re: Associations in EF 4.1 Code First: Part 5 – One-to-One Foreign Key Associations

Sunday, October 16, 2011 1:12 PM by [morteza](#)

@Michel C: You can use the trick I showed in this post to create 1:1 FK relationships between the entities in your model, hence avoiding "collection type navigation properties treated like a single object". About saving a graph of related objects, it doesn't really matter which object you pick to save your graph with, as long as the relations are set properly so I don't see any reason why *context.BList.Add(B)* could ever fail. If you got an exception while trying to save the objects then can you please be more specific?

re: Associations in EF Code First: Part 5 – One-to-One Foreign Key Associations

Friday, December 02, 2011 7:55 AM by [Gunnar](#)

This helped me tackle the fluent api configuration for my model. Thank you very much!

[Terms of Use](#)

Associations in EF Code First: Part 6 – Many-valued Associations

This is the sixth and last post in a series that explains entity association mappings with EF Code First association types so far:

- [Part 1 – Introduction and Basic Concepts](#)
- [Part 2 – Complex Types](#)
- [Part 3 – Shared Primary Key Associations](#)
- [Part 4 – Table Splitting](#)
- [Part 5 – One-to-One Foreign Key Associations](#)
- Part 6 – Many-valued Associations

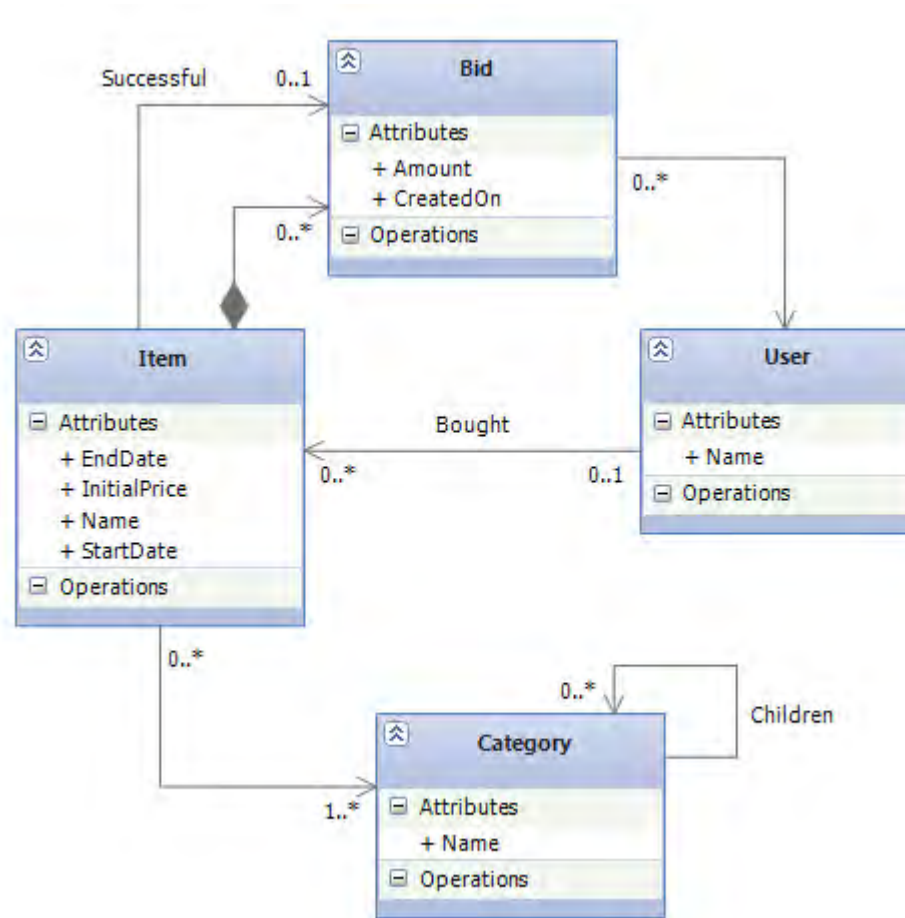
Support for many-valued associations is an absolutely basic feature of an ORM solution like Entity Framework. Surprisingly, we've managed to get this far without needing to talk much about these types of associations—surprisingly, there is not much to say on the topic—these associations are so easy to use in EF that I spend a lot of effort explaining it. To get an overview, we first consider a domain model containing different types of associations and will provide necessary explanations around each of them. Since this is the last post in the series, I will show you two tricks at the end of this post that you might find them useful in your EF Code First development.

Many-valued entity associations

A many-valued entity association is by definition a collection of entity references. One-to-many associations are an important kind of entity association that involves a collection. We go so far as to discourage the use of one-to-many association styles when a simple bidirectional many-to-one/one-to-many will do the job. A many-to-many association always be represented as two many-to-one associations to an intervening class. This model is usually more extensible, so we tend not to use many-to-many associations in applications.

Introducing the OnlineAuction Domain Model

The model we introducing here is related to an online auction system. OnlineAuction site auctions for different items. Auctions proceed according to the “English auction” model: users continue to place bids on a period for that item expires, and the highest bidder wins. A high-level overview of the domain model is shown in the following class diagram:



Each item may be auctioned only once, so we have a single auction item entity named Item. Bid is : Item.

The Object Model

The following shows the POCO classes that form the object model for this domain:

```

public class User
{
    public int UserId { get; set; }
    public string Name { get; set; }

    public virtual ICollection<Item> BoughtItems { get; set; }
}

public class Item
{
    public int ItemId { get; set; }
    public string Name { get; set; }
    public double InitialPrice { get; set; }
    public DateTime StartDate { get; set; }
    public DateTime EndDate { get; set; }
    public int? BuyerId { get; set; }
}
  
```



```

    public int? SuccessfulBidId { get; set; }

    public virtual User Buyer { get; set; }
    public virtual Bid SuccessfulBid { get; set; }
    public virtual ICollection<Bid> Bids { get; set; }
    public virtual ICollection<Category> Categories { get; set; }
}

public class Bid
{
    public int BidId { get; set; }
    public double Amount { get; set; }
    public DateTime CreatedOn { get; set; }
    public int ItemId { get; set; }
    public int BidderId { get; set; }

    public virtual Item Item { get; set; }
    public virtual User Bidder { get; set; }
}

public class Category
{
    public int CategoryId { get; set; }
    public string Name { get; set; }
    public int? ParentCategoryId { get; set; }

    public virtual Category ParentCategory { get; set; }
    public virtual ICollection<Category> ChildCategories { get; set; }
    public virtual ICollection<Item> Items { get; set; }
}

```

The Simplest Possible Association

The association from Bid to Item (and vice versa) is an example of the simplest possible kind of entity association. It has two properties in two classes. One is a collection of references, and the other a single reference. This is called a *bidirectional one-to-many association*. The property ItemId in the Bid class is a foreign key to the Item entity, something that we call a [Foreign Key Association](#) in EF 4. We defined the type of the ItemId property as int which can't be null because we can't have a bid without an item—a constraint will be generated in the database to reflect this. We use [HasRequired](#) method in fluent API to create this type of association:

```

class BidConfiguration : EntityTypeConfiguration<Bid>
{
    internal BidConfiguration()
    {
        this.HasRequired(b => b.Item)
            .WithMany(i => i.Bids)
            .HasForeignKey(b => b.ItemId);
    }
}

```

An Optional One-to-Many Association Between User and Item Entities

Each item in the auction may be bought by a User, or might not be sold at all. Note that the foreign key in the Item class is of type `Nullable<int>` which can be NULL as the association is in fact to-zero-or-one. The following method to create this association between User and Item (using this method, the foreign key must be nullable, otherwise EF Code First throws an exception):

```
class ItemConfiguration : EntityTypeConfiguration<Item>
{
    internal ItemConfiguration()
    {
        this.HasOptional(i => i.Buyer)
            .WithMany(u => u.BoughtItems)
            .HasForeignKey(i => i.BuyerId);
    }
}
```

A Parent/Child Relationship

In the object model, the association between User and Item is fairly loose. We'd use this mapping if the entities had their own lifecycle and were created and removed in unrelated business processes. Certainly, a much stronger relationship exists between an item and its bids; some entities are bound together so that their lifecycles aren't truly independent. It seems reasonable that deletion of an item implies deletion of all bids for the item. A particular bid instance belongs to one item instance for its entire lifetime. In this case, cascading deletions makes sense. In fact, this is what the filled out diamond in the above UML diagram means. If you enable cascading delete, the association between Item and Bid is called a *parent/child relationship*, and that's exactly what EF Code First does by default with the `HasRequired` method.

In a parent/child relationship, the parent entity is responsible for the lifecycle of its associated child entities. This has the same semantic as a composition using EF [complex types](#), but in this case only entities are involved. The advantage of using a parent/child relationship is that the child may be loaded individually or referenced from another entity. A bid, for example, may be loaded and manipulated without retrieving the owning item instance without storing the owning item at the same time. Furthermore, you reference the same Bid instance from multiple Item instances. Of course, the single SuccessfulBid (take another look at the Item class in the object model above). Obviously, it can't be shared.

Many-to-Many Associations

The association between Category and Item is a many-to-many association, as can be seen in the UML diagram. In a many-to-many association mapping hides the intermediate association table from the application, so it's not an unwanted entity in your domain model. That said, in a real system, you may not have a many-to-many association since my experience is that there is almost always other information that must be attached to each instance of the associated instances (such as the date and time when an item was added to a category) and that this information is represented via an intermediate association class (In EF, you can map the association with `HasManyToMany` and map two one-to-many associations for either side.).

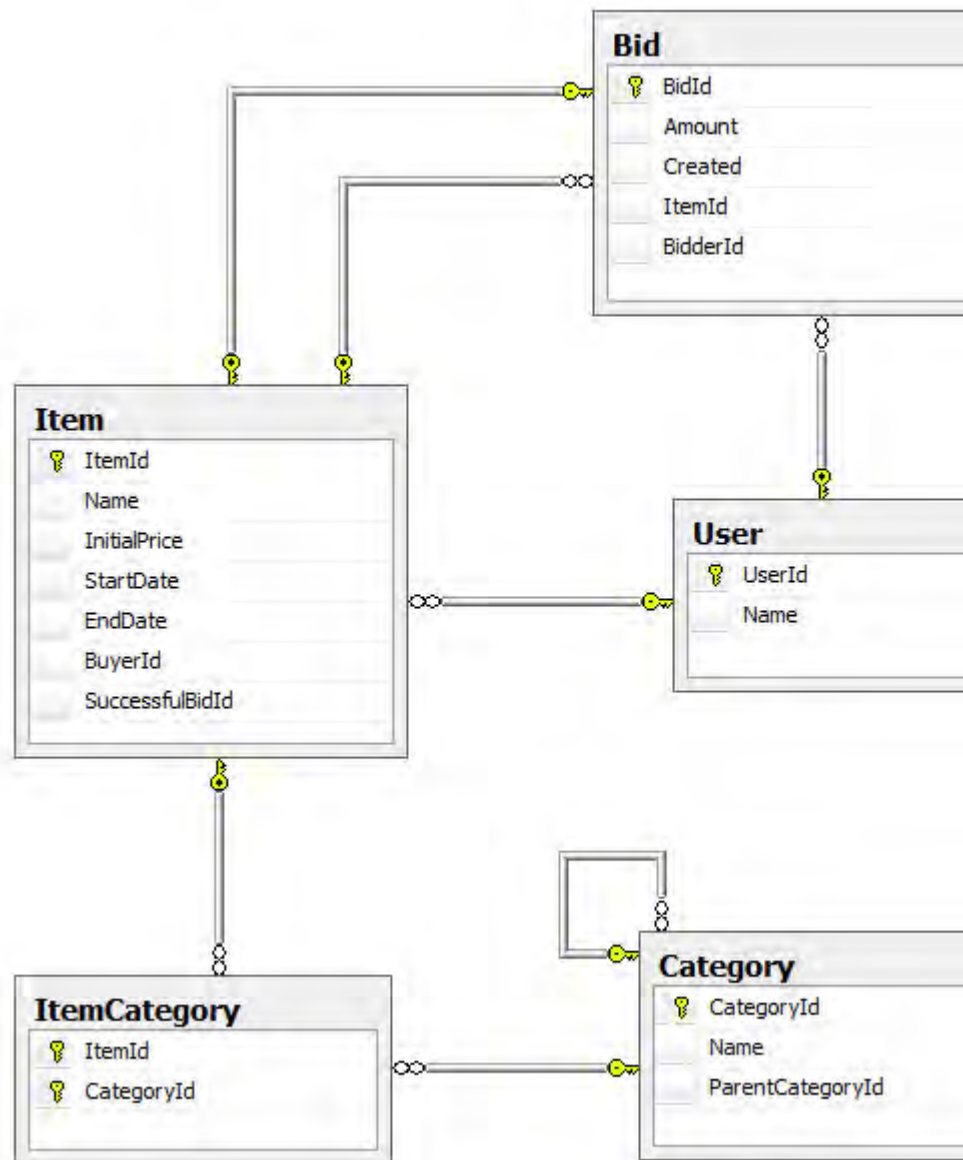
In a many-to-many relationship, the join table (or link table, as some developers call it) has two columns from the Category and Item tables. The primary key is a composite of both columns. In EF Code First,

associations mappings can be customized with a fluent API code like this:

```
class ItemConfiguration : EntityTypeConfiguration<Item>
{
    internal ItemConfiguration()
    {
        this.HasMany(i => i.Categories)
            .WithMany(c => c.Items)
            .Map(mc =>
            {
                mc.MapLeftKey("ItemId");
                mc.MapRightKey("CategoryId");
                mc.ToTable("ItemCategory");
            });
    }
}
```

SQL Schema

The following shows the SQL schema that Code First creates from our object model:



Get the Code First Generated SQL DDL

A common process, if you're starting with a new application and new database, is to generate DDL automatically during development; At the same time (or later, during testing), a professional DBA ve the SQL DDL and creates the final database schema. You can export the DDL into a text file and ha [CreateDatabaseScript](#) on `ObjectContext` class generates a data definition language (DDL) script the objects (tables, primary keys, foreign keys) for the metadata in the the store schema definition langu next section, you'll see where this metadata come from):

```
using (var context = new Context())
{
    string script = ((IObjectContextAdapter)context).ObjectContext.CreateDatabas
}
```

You can then use one of the classes in the .Net File IO API like [StreamWriter](#) to write the script on tl

```

[BuyerId] [int] null,
[SuccessfulBidId] [int] null,
primary key ([ItemId])
);
create table [dbo].[ItemCategory] (
[ItemId] [int] not null,
[CategoryId] [int] not null,
primary key ([ItemId], [CategoryId])
);
create table [dbo].[User] (
[UserId] [int] not null identity,
[Name] [nvarchar](max) null,
primary key ([UserId])
);
alter table [dbo].[Bid] add constraint [Bid_Bidder] foreign key ([BidderId])
references [dbo].[User]([UserId]) on delete cascade;
alter table [dbo].[Bid] add constraint [Bid_Item] foreign key ([ItemId]) refe
alter table [dbo].[Category] add constraint [Category_ParentCategory] foreign
alter table [dbo].[Item] add constraint [Item_Buyer] foreign key ([BuyerId])
alter table [dbo].[ItemCategory] add constraint [Item_Categories_Source] fore

```

Note how Code First enables cascade deletes for the parent/child relationship between Item and Bid.

Get the Runtime EDM

One of the benefits of Code First development is that we don't need to deal with the Edmx file, how that the concept of EDM doesn't exist at all. In fact, at runtime, when the context is used for the first time, it derives the EDM (CSDL, MSL, and SSDL) from our object model and this EDM is even [cached in the](#) instance of `DbCompiledModel`. Having access to this generated EDM is beneficial in many cases. As you can add it to our solution and use it as a class diagram for our domain model. More importantly, we can use it for debugging when there is a need to look at the model that Code First creates internally. This EDM is a conceptual schema definition language (CSDL) something that drives the EF runtime behavior. The [WriteEdmx](#) Method from the [EdmxWriter](#) class like the following code:

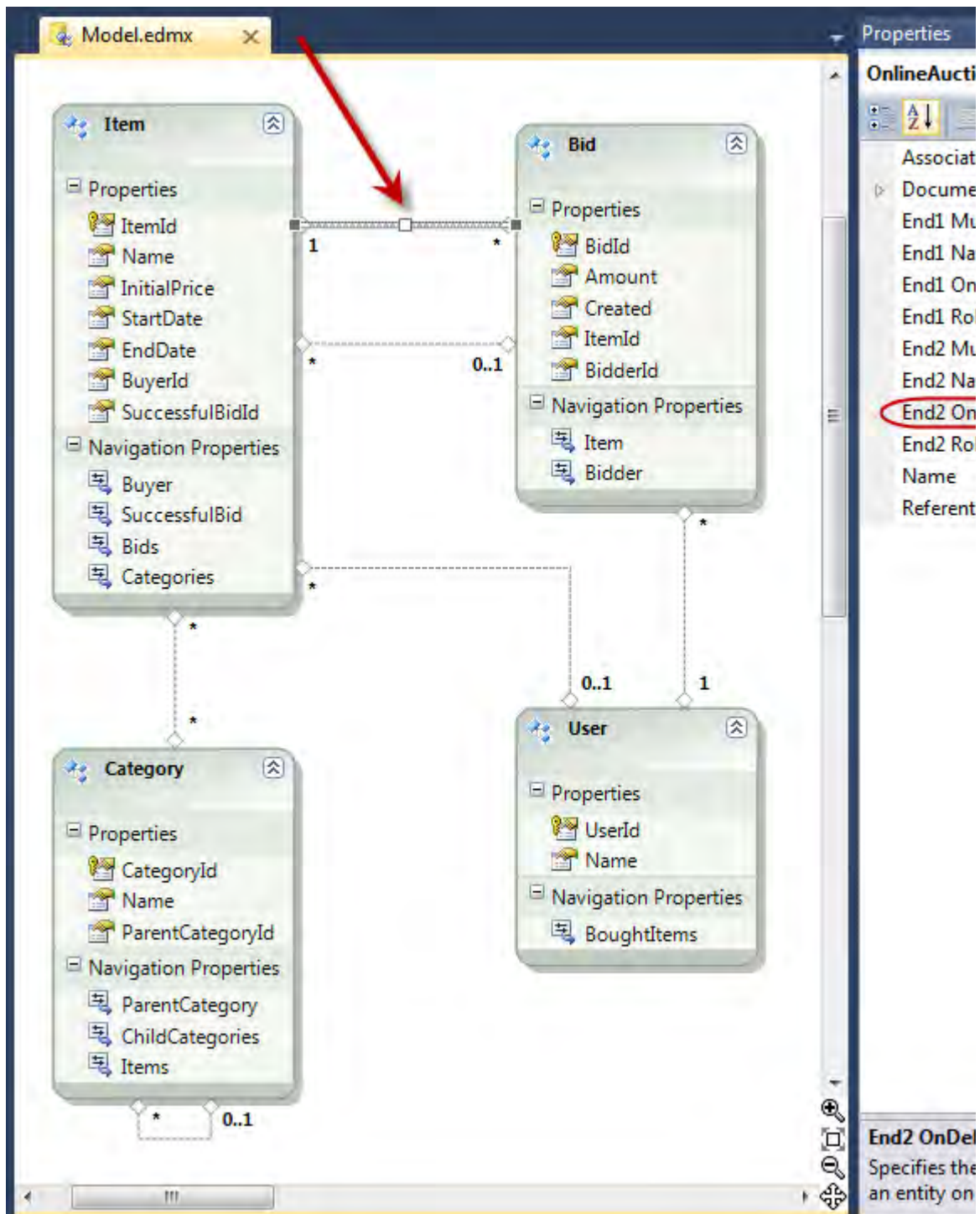
```

using (var context = new Context())
{
    XmlWriterSettings settings = new XmlWriterSettings();
    settings.Indent = true;

    using (XmlWriter writer = XmlWriter.Create(@"Model.edmx", settings))
    {
        EdmxWriter.WriteEdmx(context, writer);
    }
}

```

After running this code, simply right click on your project and select *Add Existing Item...* and then br Model.edmx file to the project. Once you added the file, double click on it and visual studio will perfe file in the designer:




Also note how cascade delete is also enabled in the CSDL for the parent/child association between

Source Code

Click [here](#) to download the source code for the OnlineAuction site that we have seen in this post.

Summary

In this series, we focused on the structural aspect of the object/relational paradigm mismatch and di main ORM problems relating to associations. We explored the programming model for persistent cl Code First fluent API for fine-grained classes and associations. Many of the techniques we've show concepts of object/relational mapping and I am hoping that you'll find them useful in your Code First

Published Tuesday, May 17, 2011 5:16 AM by [morteza](#) 
Filed under: [C#](#), [Code First](#), [Entity Framework 4.1](#)

Comments

[# re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations](#)

Tuesday, May 17, 2011 2:20 AM by [Martin H. Normark](#)

Hi

Regarding the ItemCategory relationship. Say you wanted to add the possibility to avoid displaying child categories, only in specific relationships.

In the database you'd have a bit column in the ItemCategory table that could be named DisplayChildCategories.

How do you include this column in the relationship?

And where would it be mapped, which class?

[# re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations](#)

Tuesday, May 17, 2011 11:09 AM by [Horizon Net](#)

Great posts. Thank you for sharing.

[# re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations](#)

Tuesday, May 17, 2011 4:15 PM by [haitao](#)

I have been eagerly waiting for Part 6 for a long time. Thanks for the great post.

[# re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations](#)

Wednesday, May 18, 2011 7:03 AM by [Dencio](#)

Thank you. Very informative, very helpful post.

[# re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations](#)

Wednesday, May 18, 2011 8:30 AM by [DJRodriguez](#)

Hello,

Thanks for all your posts, they have helped me to understand a lot about code first!

Could you help me understand what is the best way to map something like this to an existing database/tables?

Model:

```

public class Item
{
    public int ItemId { get; set; }
    public string Name { get; set; }
    public virtual IDictionary<int, Attribute> Attributes { get; set; }
}

public class Attribute
{
    public int Id { get; set; }
    public string Name { get; set; }
    public virtual IDictionary<int, ExpectedValue> ExpectedValues { get; set; }
}

public class ExpectedValue
{
    public int Id { get; set; }
    public string Name { get; set; }
}

```

Attribute Table:

```

id int
Name nvarchar(64)
Description nvarchar(128)

```

ExpectedValue Table:

```

id int
Name nvarchar(64)
Value int
MonitoredItemAttributeMapID int

```

MonitoredItemAttributeMap table (Many to Many Mapping table):

```

id int
MonitoredItemID int
AttributeID int

```

Thank you for taking the time to look at this and any solution is helpful!

Cheers,
DJ

[# What does multiplicity 1, 0..1, * mean? | Code First :: Entity Framework](#)

Thursday, May 19, 2011 8:42 PM by [What does multiplicity "1", "0..1", "*" mean? | Code First :: Entity Framework](#)

Pingback from [What does multiplicity 1, 0..1, * mean? | Code First :: Entity Framework](#)

[# re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations](#)

Saturday, May 21, 2011 4:48 PM by [morteza](#)

@Martin H. Normark: Like I described in the post, many-to-many associations cannot have a payload, and if that's the case then you always have to break it down to two many-to-one association to an intervening class. For example, the following is going to be the object model for the scenario you described:

```

public class Item
{
    public int ItemId { get; set; }
    public virtual ICollection<ItemCategory> ItemCategories { get; set; }
}

```



```

}

public class Category
{
    public int CategoryId { get; set; }
    public virtual ICollection<ItemCategory> ItemCategories { get; set; }
}

public class ItemCategory
{
    public int ItemCategoryId { get; set; }
    public bool DisplayChildCategories { get; set; }
    public int CategoryId { get; set; }
    public int ItemId { get; set; }

    public virtual Item Item { get; set; }
    public virtual Category Category { get; set; }
}

Hope this helps.

```

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Saturday, May 21, 2011 6:02 PM by [morteza](#)

@DJRodriguez: Like I described in the post a many-to-many association cannot have a payload and in your case, even the Id as a primary key on the MonitoredItemAttributeMap table is considered to be a payload and therefore you can't create a many-to-many association between the Item and Attribute classes. In this case, like I described in the post, you have to break it down to two many-to-one association to an intervening class. I named this intermediate association class as MonitoredItemAttributeMap and created the following object model for your existing database:

```

public class Item
{
    public int Id { get; set; }
    public string Name { get; set; }

    public virtual ICollection<MonitoredItemAttributeMap> ItemAttributes { get; set; }
}

public class Attribute
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }

    public virtual ICollection<MonitoredItemAttributeMap> ItemAttributes { get; set; }
}

public class ExpectedValue
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Value { get; set; }
}

```

```

public int MonitoredItemAttributeMapID { get; set; }

public MonitoredItemAttributeMap ItemAttribute { get; set; }
}

public class MonitoredItemAttributeMap
{
    public int Id { get; set; }
    public int MonitoredItemID { get; set; }
    public int AttributeID { get; set; }

    public Item Item { get; set; }
    public Attribute Attribute { get; set; }
}

public class Context : DbContext
{
    public DbSet<Item> Items { get; set; }
    public DbSet<Attribute> Attributes { get; set; }
    public DbSet<ExpectedValue> ExpectedValues { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<MonitoredItemAttributeMap>()
            .HasRequired(m => m.Item)
            .WithMany(i => i.ItemAttributes)
            .HasForeignKey(m => m.MonitoredItemID);
    }
}

```

Thanks for your comment and I hope you find this helpful :)

[# re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations](#)

Tuesday, May 24, 2011 3:28 AM by Mohammad Tajari

Keep up your good work!

Baba khafan. Baba inkare :)

[# re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations](#)

Wednesday, May 25, 2011 12:50 PM by Scott

All very good articles. Thank you!

[# Code First ??????????????](#)

Friday, May 27, 2011 11:22 AM by [Code First ??????????????](#)

Pingback from [Code First ??????????????](#)

[# re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations](#)

Thursday, June 09, 2011 2:43 AM by [Sasan](#)

merci aghaye manafi babate in post

Associations in EF 4.1 Code First: Part 6 – Many-valued Associations - Enterprise .Net

Thursday, July 07, 2011 3:40 AM by progg.ru

Thank you for submitting this cool story - Trackback from progg.ru

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Tuesday, July 12, 2011 11:20 AM by Ricardo Peres

When will other types of collections be supported?

At least, IEnumerable<T>... sometimes, we don't want a collection to be changed.

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Saturday, August 27, 2011 6:26 PM by Bilal

Awesome post! To me its the best article on EF :)

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Tuesday, August 30, 2011 10:52 AM by julien

Insert doesn't work if i want to insert a Category with Categories children :(

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Tuesday, August 30, 2011 3:36 PM by Alessandro Antonio de Brito

Hi! Great post!

But, I'm still having some doubts about one-to-many mapping.

I have to map the one-to-many relationship like this: (p.s.: I simplified the model to show my doubt.)

1) Table structure:

```
TB_PERSON
(
  ID_PERSON INT PRIMARY KEY IDENTITY,
  DC_PERSON VARCHAR(100) NOT NULL
)

TB_ADDRESS
(
  ID_ADDRESS INT PRIMARY KEY IDENTITY,
  DC_ADDRESS VARCHAR(500) NOT NULL,
  ID_PERSON INT NOT NULL REFERENCES TB_PERSON(ID_PERSON)
)
```

2) Class Structure

```
public class Address
{
  public int Id { get; set; }
  public string Description { get; set; }
  virtual Person PersonA { get; set; }
}

public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }
  public virtual ICollection<Address> Addresses { get; set; }
}
```

3) Configuration files:

PersonConfiguration

```
{
  HasKey(p => p.Id);
  Property(p =>
p.Id).HasColumnName("ID_PERSON").HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity).I
  Property(p => p.Name).HasColumnName("DC_ADDRESS").IsRequired();
}
```

AddressConfiguration

```
{
  HasKey(p => p.Id);
  Property(p =>
p.Id).HasColumnName("ID_ADDRESS").HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity)
  Property(p => p.Description).HasColumnName("DC_PERSON").IsRequired();
}
```

Questions:

a) Where and How I have to configure the relationship between "Person" and "Address" classes ? I've tried all combinations without success. I saw some examples that put a foreign key property in the class to navigate. Is this the only way so solve this problem?

b) Is it really necessary to declare "PersonA" property in the "Address" class? Is it an obligation on the Code-First approach?

Thanks in advance!

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Thursday, September 01, 2011 10:40 PM by [mortezam](#)

@Alessandro Antonio de Brito: The following object model is all you need:

```
public class Address
{
  public int Id { get; set; }
  public string Description { get; set; }
  public virtual int PersonId { get; set; }
}

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

public class Context : DbContext
{
  public DbSet<Person> Persons { get; set; }
  public DbSet<Address> Addresses { get; set; }

  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Person>().Property(p => p.Id).HasColumnName("ID_PERSON");
    modelBuilder.Entity<Person>().Property(p =>
p.Name).HasColumnName("DC_PERSON").IsRequired();
    modelBuilder.Entity<Person>().ToTable("TB_PERSON");
  }
}
```

```

        modelBuilder.Entity<Address>().Property(p => p.Id).HasColumnName("ID_ADDRESS");
        modelBuilder.Entity<Address>().Property(p =>
p.Description).HasColumnName("DC_ADDRESS").IsRequired();
        modelBuilder.Entity<Address>().Property(p => p.PersonId).HasColumnName("ID_PERSON");
    }
}

```

As you can see above, you don't really need to configure the association between Person and Address entities as it will be picked up by convention and Code First will automatically configure it for you. Having a foreign key like PersonId on the Address class is not required but is recommended. Having the foreign key along with the navigation properties will create a *foreign key association* as opposed to an *independent association* that have been introduced in the first version of EF.

Note that it is not necessary to define a PersonA property on the Address class. You should do that only if you want to make your association bidirectional and looks like you don't have such a requirement in your domain model so I set up a unidirectional association between the two entities. Hope this helps.

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Wednesday, September 14, 2011 6:35 AM by [craigfreeman74](#)

When did we start changing the associations/configurations from:

```

modelBuilder.Entity<Item>()
    .HasOptional(i => i.Buyer)
    .WithMany(u => u.BoughtItems)
    .HasForeignKey(i => i.BuyerId);

```

To:

```

class ItemConfiguration : EntityTypeConfiguration<Item>
{
    internal ItemConfiguration()
    {
        this.HasOptional(i => i.Buyer)
            .WithMany(u => u.BoughtItems)
            .HasForeignKey(i => i.BuyerId);
    }
}

```

Are these the same thing?

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Wednesday, September 14, 2011 9:55 PM by [morteza](#)

@craigfreeman74: Yes, they are exactly the same thing and have the same effect. Having the fluent API code in a separate class that inherits from `EntityTypeConfiguration<T>` allows a cleaner way of writing fluent API code and is recommended especially when you have too many entities in your domain that you want to write fluent API code for.

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Tuesday, September 20, 2011 1:02 PM by Mark Phillips

Excellent Series!!! Thank you.

Will you be doing anything in relation to sprocs and views?

Thanks again.

Mark

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Tuesday, September 20, 2011 10:54 PM by [mortezaam](#)

Mark Phillips: You're very welcome. I don't really have any plans to write about stored procedures or views but if you have any questions, feel free to post them here. I might be able to help :)

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Wednesday, September 21, 2011 5:55 AM by Farid

Hi

How the data is populated into the the mapping table. eg categoryItem

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Friday, September 23, 2011 12:43 PM by [mortezaam](#)

@Farid: The join table (e.g. ItemCategory) is populated when you define new relationships between the related objects (e.g. Item and Category) in your application. For example, the following code will result in a submission of an INSERT INTO command by EF to add a new record to the ItemCategory table:

```
Category category = new Category();
Item item = new Item();

category.Items.Add(item);

context.Categories.Add(category);
context.SaveChanges();
```

And the following code will result in a DELETE command to remove the inserted record:

```
category.Items.Remove(item);
context.SaveChanges();
```

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Tuesday, November 01, 2011 4:19 PM by Jonpa

Hi!

I'm trying to delete an entity in the same way as in your response to Farid, that is:

```
category.Items.Remove(item)
```

but in my case

```
entity.Entities.Remove(entity2)
```

The problem is when i call SaveChanges() i get the following error: A relationship from the 'Entity_Entities' AssociationSet is in the 'Deleted' state. Given multiplicity constraints, a corresponding 'Entity_Entities_Target' must also in the 'Deleted' state.

My model looks like this:

```
public class Entity
```

```
{
  ...
  public virtual ICollection<Entity2> Entities {...}
}
```

Im using Entity as an aggregate root, that is, my context only exposes DBSet<Entity> so the only way to handle Entity2 is through Entity.

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Thursday, November 03, 2011 9:18 PM by [morte zam](#)

@Jonpa: The code you showed should (and will) work for a many-many association like the one I introduced in this post. Can you please post your object model along with the fluent API codes involving Entity and Entity2? I particularly like to see the Entity2 class definition.

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Friday, November 04, 2011 7:05 AM by Jonpa

The relationship between Entity and Entity2 is an one to many relationship. I think the problem lies in that my foreign key in Entity2 is non nullable (as I want to be).

Well here's the code (note that in my production code, the entities have better name :))

```
public class Entity : EntityBase
{
  private ICollection<Entity2> _entities;

  public string Name { get; set; }

  public virtual ICollection<Entity2> Entities
  {
    get { return (_entities ?? (_entities = new HashSet<Entity2>())); }
  }
}

public class Entity2 : EntityBase
{
  public string Name { get; set;}
}

public abstract class EntityBase
{
  public Guid Id { get; set; }
  public DateTime Created { get; set; }
  public byte[] Version { get; set; }

  public EntityBase()
  {
    Id = Guid.NewGuid();
    Created = DateTime.Now;
  }
}
```

And my fluent API code:

```
public class EntityBaseConfiguration<T> : EntityTypeConfiguration<T> where T : EntityBase
{
  public EntityBaseConfiguration()
  {
    HasKey(e => e.Id);
    Property(e => e.Version).IsRowVersion();
  }
}
```

```

class EntityConfiguration : EntityBaseConfiguration<Entity>
{
    public EntityConfiguration()
    {
        Map(e => { e.ToTable("Entiteter"); e.MapInheritedProperties(); });
        Property(e => e.Name).IsRequired().HasMaxLength(100);
        HasMany(e => e.Entities).WithRequired().WillCascadeOnDelete();
    }
}

class EntityConfiguration2 : EntityBaseConfiguration<Entity2>
{
    public EntityConfiguration2()
    {
        Map(e => { e.ToTable("Entiteter2"); e.MapInheritedProperties(); });
        Property(e => e.Name).IsRequired().HasMaxLength(100);
    }
}

```

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Monday, November 07, 2011 11:06 AM by [mortezaam](#)

@Jonpa: Knew it cannot be a many-to-many association since the exception you were getting was something that usually comes out of a required one-to-many association. Yes, you thought is correct, your code (`entity.Entities.Remove(entity2)`) would cause Entity's foreign key in Entity2's table to become Null but your fluent API code (`HasMany(e => e.Entities).WithRequired()`) made that FK column Not Nullable and that's where the problem happens. To fix this problem I suggest you first add a FK property to your Entity2 class like the following:

```

public class Entity2 : EntityBase
{
    public string Name { get; set;}
    public Guid? EntityId { get; set; }
}

```

And then change your fluent API code in EntityConfiguration where you setup the one-to-many association to this:

```

HasMany(e => e.Entities)
    .WithOptional()
    .HasForeignKey(e => e.EntityId)
    .WillCascadeOnDelete();

```

And you should be good to go. Hope this helps.

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Monday, November 07, 2011 1:09 PM by Jonpa

Thanks for the reply.

I could do the changes you suggest but then my domain model could (and temporarily will) find itself in a "illegal" state. Lets say I have the classic Order -> OrderLines relation. I dont want to allow that an OrderLine could be stored without an associated Order. But with the suggested solution, allowing the OrderId foreign key in OrderLine to nullable, this is actually a valid state.

Im not saying that this is a big problem because there are ways to achive this anyway but still... I prefer to have my domain model as close to the problem domain as possible and then have a database model to support this. I also want my domain model to bee totally persistent ignorant if it is possible (which it almost never is due to technical problems like lazy loading and such in the OR mapper).

Anyway, thanks for the reply and lets hope that future releases of Entity Framework will support the "correct" way ;)of modelling this.

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Monday, November 07, 2011 10:18 PM by [morteza](#)

@Jonpa: OK, I thought having an Entity2 without an Entity makes sense based on your domain model but if that's not the case then don't do it since like you mentioned, your model should exactly reflect your business domain. Based on what you described, the correct way of mapping this association would be as follows:

```
public class Entity : EntityBase
{
    public string Name { get; set; }
    public virtual ICollection<Entity2> Entities { get; set; }
}

public class Entity2 : EntityBase
{
    public string Name { get; set;}
    public Guid EntityId { get; set; }
    public Entity Entity { get; set; }
}
```

And the fluent API code to configure the associaion:

```
HasMany(e => e.Entities)
    .WithRequired(e => e.Entity)
    .HasForeignKey(e => e.EntityId);
```

Note how I turn the association into a bidirectional one so that we can take advantage of it when removing an Entity2 from the Entity. Entities collection:

```
// Assuming entity and entity2 are loading from database and that they are related:
entity.Entities.Remove(entity2);
entity2.Entity = new Entity();
context.SaveChanges();
```

This way, you can have your required relationship in place without getting any exception when changing the associations between your objects.

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Tuesday, November 08, 2011 12:06 PM by Jonpa

Ok, but if I want to remove the Entity2 object, or rather delete it. I dont want to assign a new Entity object to the Entity2 object, I only want to delete it. Then I will get the same exception again or am I missing something? Like in the Order -> OrderLine example. How will I design my model to be able to remove an OrderLine from an Order?

/Jonpa

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Thursday, November 10, 2011 8:31 PM by [morteza](#)

@Jonpa: Then just delete the Entity2 object. For example, in the case of an OrderLine object:

```
context.OrderLines.Remove(orderLine);
```

Please note that you don't need to be worried about the existing association between the Order and OrderLine objects, when you delete the OrderLine object, EF automatically updates the OrderLines collection on the related Order. The bottom line is that an OrderLine object can never exist without an Order, you have to either delete it or replace its Order property with another Order be it new or existing.

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Saturday, November 12, 2011 7:37 AM by Jonpa

My context only exposes Entity (or Order on the Order, OrderLine example). So the only way to "mingle" with Entity2 (OrderLine) is through Entity. Note that this is exactly what I want. I don't want my context to expose Entity2 because Entity is my aggregate root. So is there a way to achieve this?

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Sunday, November 13, 2011 2:23 PM by Jonpa

I don't think that is possible to achieve what I want. In this post blogs.msdn.com/.../deleting-foreign-key-relationships-in-ef4.aspx he talks about deleting foreign key relationships and my example is listed in case 1.

Like I said, I really hope Microsoft will make it possible to do what I want in the future. I solved the problem like this if anyone is interested. I override the SaveChanges() in my DbContext class and I check every entity of type OrderLine, in the ChangeTracker, to see if any Order has a reference to it. If the OrderLine doesn't belong to any Order I manually set the State of the entity to Deleted. When I call base.SaveChanges() the OrderLine entities will be deleted.

Thanks for your time and I really like your posts about EF code first so keep up the good work!

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Friday, November 18, 2011 11:12 AM by [mortezaam](#)

@Jonpa: An aggregate root is a concept that has to be applied on your Repository classes and not on the definition of the DbContext class. By removing the non-aggregate root DbSet from your DbContext (e.g. DbSet<OrderLine>) you make simple tasks very hard like the way you had to delete the orphan OrderLines by overriding the SaveChanges method. What you are looking for is implemented in some other ORM frameworks though. For example, in NHibernate, associations have a setting called *cascade* which you can set to *delete-orphan*. As a result, NHibernate deletes any persistent entity instance that has been removed (dereferenced) from the association (e.g. any persistent OrderLine should be deleted if it's removed from the OrderLines collection of a persistent Order.). I am also hoping that EF implements these features on the associations but to be honest, I don't see that happening in the near future.

re: Associations in EF 4.1 Code First: Part 6 – Many-valued Associations

Friday, November 18, 2011 2:50 PM by Jonpa

@mortezaam I am using repositories and that's why I'm having the problem. I don't have a repository for OrderLine (because Order is the aggregate root) so the only way to delete the OrderLine, after it has been removed from the collection in Order, is to override SaveChanges.

I don't want to leave the responsibility to delete the OrderLine to the client and I don't want my domain model to know how it is persisted. So like I said the only way, as I see it, is to override SaveChanges.

I have implemented my repositories pretty much like in this example app microsoftnlayerapp.codeplex.com. This app shows how one could implement many of the patterns and practices that Evans talks about in his DDD book.

re: Associations in EF Code First: Part 6 – Many-valued Associations

Wednesday, November 30, 2011 9:57 AM by [arkhanwu](#)

thanks for this great series, very very helpful~ looking forward to your new posts!

re: Associations in EF Code First: Part 6 – Many-valued Associations

Monday, December 12, 2011 1:45 PM by [AroglDarthu](#)

Never mind my previous comment... The setup worked when I put it in your sample application. Turned out the entity was detached. Setting it to Modified and calling SaveChanges, caused it not save the navigational properties. Perfectly logical ;-)

Thanks,

Twan

re: Associations in EF Code First: Part 6 – Many-valued Associations

Friday, December 16, 2011 11:58 PM by Jim Shaw

Thanks for your articles. It was extremely helpfull in understanding the 'clear as mudd' WithRequiredDependent, etc.. and how foreign key relationships are handled.

Although I still have a couple of issues with parent child relationships (in EF that is, my kids never listen to me anyways, lol)

In most parent-child relationships, in a one to many (as a Course has many chapters which have many sections...). The child entity maintaining a fk relationship to its parent is reasonable (a section may wish to know its parent chapter so it can display its title properly as "<chapter #>.<section #> title".

However, I have a couple of situations where an entity class has potentially different parental relationships. That is, a child entity can be a child of more than one type (class) of parent. Each instance will only be associated with one parent instance, but two different instances of the child may have different parental types... eg

```
class TextBlock { int Id, string Text, string Style... }
```

```
class BlockGroup { int Id, ICollection<TextBlock> GroupList, TextBlock Title,... }
```

```
class BulletedList { int Id, ICollection<TextBlock> MyList, TextBlock Title,... }
```

```
class Chapter { Id, Number, TextBlock Title,... }
```

As you can see TextBlock can be a child of BlockGroup OR a child of BulletedList OR a reference to the Chapter's title. For my example the instance will never be part of both and if the parent is deleted the child will also be deleted. The Ef 4.1 model adds two FK columns on my TextBlock table, one for BlockGroup relationship and one for the BulletedList relationship. I have modeled all the other relationships to TextBlock as a FK relationship in the parent with cascade delete ste to true.

I can live with the model given me, as EF 4.1 saves me a lot of CRUD operations and makes life a lot easier. However, in my case it would be nicer if the TextBlock table had no knowledge of its parent (which in the case of my class design is true) and the relationship would be handled in a link table.

My reasoning is that in some cases the TextBlock instance is just a block of text, in others it is one entry in a list and so needs to have its order entry preserved, somehow. But its order within the list is not a property of the class itself. Since the parent maintains an ordered list of the items, there is no need to maintain the order within the entity itself (other than what is required to persist and restore the data to and from the DB).

So what I would like is another way to map the parent's list (ICollection<TextBlock>) to a link table, which would maintain the reference to the parent, an index and a reference to the child, and also allow for normal cascading of deletes...

```
table BlockGroupListTextBlocks {
    int Id
    int BlockGroupId (FK to [BlockGroup][Id])
    int TextBlockId (FK to [TextBlock][Id])
}
```

```

    int index
}

```

which would be mapped to the BlockGroup's 'GroupList' property...

Thank you for your time...

re: Associations in EF Code First: Part 6 – Many-valued Associations

Sunday, December 18, 2011 6:49 PM by [morteza](#)

@Jim Shaw: I think you are in the right path. An optional entity association, be it one-to-one or one-to-many, is best represented in an SQL database with a join table. We always try to avoid nullable columns in a relational database schema since information that is unknown degrades the quality of the data you store. That said, EF unfortunately does not have a good support for this type of mapping which means you need to create new entities to represent the join tables for your optional one-to-many relationships. For example, the associations between TextBlock and BlockGroup entities could be represented with the following object model:

```

public class TextBlock
{
    public int TextBlockId { get; set; }
    public string Text { get; set; }
    public string Style { get; set; }
}

public class BlockGroup
{
    public int BlockGroupId { get; set; }

    public TextBlock Title { get; set; }
    public ICollection<BlockGroupTextBlock> GroupList { get; set; }
}

public class BlockGroupTextBlock
{
    public int Id { get; set; }
    public int Index { get; set; }
    public int BlockGroupId { get; set; }

    public TextBlock TextBlock { get; set; }
}

public class Context : DbContext
{
    public DbSet<TextBlock> TextBlocks { get; set; }
    public DbSet<BlockGroup> BlockGroups { get; set; }
    public DbSet<BlockGroupTextBlock> BlockGroupTextBlocks { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<BlockGroup>()
            .HasMany(b => b.GroupList)
            .WithOptional()
            .HasForeignKey(b => b.BlockGroupId);
    }
}

```

```

modelBuilder.Entity<BlockGroup>()
    .HasOptional(b => b.Title)
    .WithRequired();

modelBuilder.Entity<BlockGroupTextBlock>()
    .HasRequired(b => b.TextBlock)
    .WithOptional()
    .WillCascadeOnDelete();
}
}

```

Hope this helps.

re: Associations in EF Code First: Part 6 – Many-valued Associations

Wednesday, December 28, 2011 5:35 PM by [Ran Davidovitz](#)

Lets say (i know its bad but i just want it while i filling data - to make it fast), to directly write to the ItemCategory table, can i add a new class for the ItemCategory and add it to the context as DBSET or will it try to create another ItemCategory table ?

re: Associations in EF Code First: Part 6 – Many-valued Associations

Thursday, January 05, 2012 4:04 PM by [mortezaam](#)

@Ran Davidovitz: Yes, you can define an ItemCategory class and it won't create a new one for you. The catch is that you will have to change the navigation properties on both sides as well. Something like the following will do the trick:

```

public class Item
{
    public int ItemId { get; set; }
    public virtual ICollection<ItemCategory> ItemCategories { get; set; }
}

public class Category
{
    public int CategoryId { get; set; }
    public virtual ICollection<ItemCategory> ItemCategories { get; set; }
}

public class ItemCategory
{
    public int ItemId { get; set; }
    public int CategoryId { get; set; }
}

public class Context : DbContext
{
    public DbSet<Item> Items { get; set; }
    public DbSet<Category> Categories { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {

```

```

modelBuilder.Entity<ItemCategory>()
    .HasKey(ic => new { ic.CategoryId, ic.ItemId });
}
}

```

That being said, I think you should NOT do this. Not sure how it helps you with data entry, but for that you can write a custom initializer class and then override its Seed method where you can then use the *ExecuteSqlCommand* to execute raw SQL statements for your data entry. Hope this helps.

re: Associations in EF Code First: Part 6 – Many-valued Associations

Sunday, January 15, 2012 10:23 AM by JohnBee

Thanks for your articles, i learned a lot from them. I'm doing my first steps in EF, and still dont know many things about it. Can you give me a hint how to solve my problem?

I have classes:

```

[Table("Firma")]
public class Firma
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int FirmalD { get; set; }

    [MaxLength(150), Required]
    public string Nazwa { get; set; }

    [MaxLength(50), Required]
    [DisplayName("Nazwa skrócona")]
    public string NazwaKrotka { get; set; }

    public int? Telefon { get; set; }
    public int? Fax { get; set; }

    [MaxLength(150)]
    public string Ulica { get; set; }

    //[MaxLength(5)]
    public int Kod { get; set; }

    [MaxLength(150)]
    public string Miasto { get; set; }

    [MaxLength(150), Required]
    [DisplayName("E-mail")]
    public string Email { get; set; }

    public bool Aktywny { get; set; }

    public virtual ICollection<FirmaOddzialHistoria> Oddzialy { get; set; }
}

[Table("FirmaOddzialHistoria")]
public class FirmaOddzialHistoria
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int FirmaOddzialHistorialD { get; set; }

    public virtual Firma Firma { get; set; }
    public virtual FirmaOddzial FirmaOddzial { get; set; }
    public DateTime DataStart { get; set; }
    public DateTime DataKoniec { get; set; }
}

```

```
[Table("FirmyOddzialy")]
public class FirmaOddzial
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int OddzialId { get; set; }

    public int? Telefon { get; set; }
    public int? Fax { get; set; }

    [MaxLength(150)]
    public string Ulica { get; set; }

    [MaxLength(5)]
    public string Kod { get; set; }

    [MaxLength(150)]
    public string Miasto { get; set; }

    [MaxLength(150), Required]
    public string Email { get; set; }

    public bool Aktywny { get; set; }

    public virtual ICollection<FirmaOddzialHistoria> Firmy { get; set; }
    public virtual ICollection<UzytkownikHistoria> Uzytkownicy { get; set; }
    public Limit Limit { get; set; }
}
```

and i want do something like this (this code don't work):

```
repository.FirmyOddzialy.Where(p => p.Firmy.DataStart<=DateTime.Now &&
p.Firmy.DataStop>=DateTime.Now && p.Firmy.Firma.FirmaId==1)
```

Best regards

re: Associations in EF Code First: Part 6 – Many-valued Associations

Tuesday, January 17, 2012 2:23 PM by [mortezam](#)

@JohneBee: Something like this will do the trick:

```
var query = (from o in repository.FirmyOddzialy
             from f in o.Firmy
             where f.DataStart <= DateTime.Now && f.DataStop >= DateTime.Now && f.Firma.FirmaId
             == 1
             select o);
```

[Terms of Use](#)

Inheritance with EF Code First: Part 1 – Table per Hierarchy (TPH)

A simple strategy for mapping classes to database tables might be “one table for every entity persists” approach sounds simple enough and, indeed, works well until we encounter inheritance. Inheritance is a visible structural mismatch between the object-oriented and relational worlds because object-oriented model both “*is a*” and “*has a*” relationships. SQL-based models provide only “has a” relationships because SQL database management systems don’t support type inheritance—and even when it’s available, it’s proprietary or incomplete.

There are three different approaches to representing an inheritance hierarchy:

- **Table per Hierarchy (TPH):** Enable polymorphism by denormalizing the SQL schema, and use a discriminator column that holds type information.
- **Table per Type (TPT):** Represent “is a” (inheritance) relationships as “has a” (foreign key) relationships.
- **Table per Concrete class (TPC):** Discard polymorphism and inheritance relationships completely in the SQL schema.

I will explain each of these strategies in a series of posts and this one is dedicated to TPH. In this series I will deeply dig into each of these strategies and will learn about “why” to choose them as well as “how” to implement them. Hopefully it will give you a better idea about which strategy to choose in a particular scenario.

Inheritance Mapping with Entity Framework Code First

All of the inheritance mapping strategies that we discuss in this series will be implemented by EF Code First. The CTP5 build of the new EF Code First library has been [released](#) by ADO.NET team earlier this month. Code-First enables a pretty powerful code-centric development workflow for working with data. I’m a big fan of the EF Code First approach, and I’m pretty excited about a lot of productivity and power that it brings. With inheritance mapping, not only Code First fully supports all the strategies but also gives you ultimate power to work with domain models that involves inheritance. The fluent API for inheritance mapping in CTP5 has improved a lot and now it’s more intuitive and concise in compare to CTP4.

A Note For Those Who Follow Other Entity Framework Approaches

If you are following EF’s “Database First” or “Model First” approaches, I still recommend to read this although the implementation is Code First specific but the explanations around each of the strategies are applied to all approaches be it Code First or others.

A Note For Those Who are New to Entity Framework and Code-First

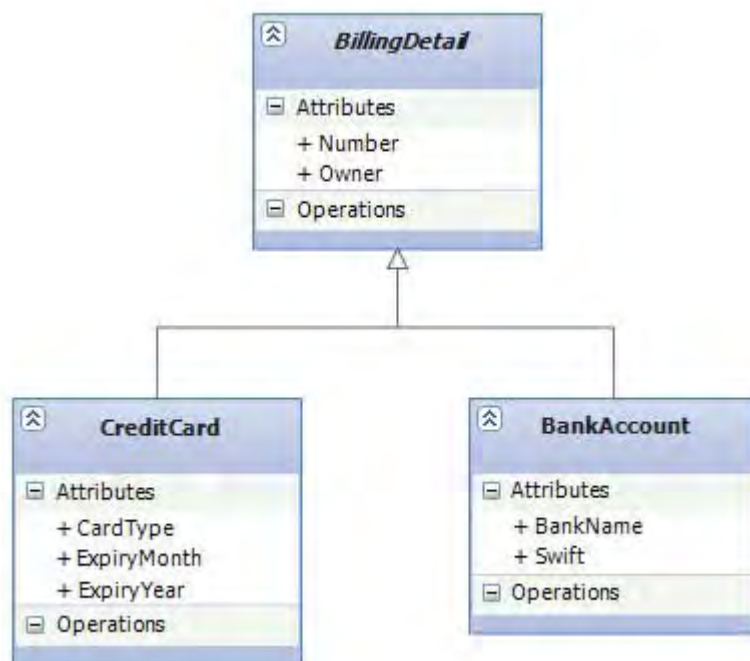
If you choose to learn EF you've chosen well. If you choose to learn EF with Code First you've done get started, you can find a great walkthrough by Scott Guthrie [here](#) and another one by ADO.NET team post, I assume you already setup your machine to do Code First development and also that you are familiar with Code First fundamentals and basic concepts. You might also want to check out my other posts on EF like [Complex Types](#) and [Shared Primary Key Associations](#).

A Top Down Development Scenario

These posts take a *top-down approach*; it assumes that you're starting with a domain model and try to create a new SQL schema. Therefore, we start with an existing domain model, implement it in C# and then let EF create the database schema for us. However, the mapping strategies described are just as relevant when working bottom up, starting with existing database tables. I'll show some tricks along the way that help with nonperfect table layouts.

The Domain Model

In our domain model, we have a *BillingDetail* base class which is abstract (note the italic font on the diagram below). We do allow various billing types and represent them as subclasses of *BillingDetail*. Currently, we support *CreditCard* and *BankAccount*.



Implement the Object Model with Code First

As always, we start with the POCO classes. Note that in our *DbContext*, I only define one *DbSet* for which is *BillingDetail*. Code First will find the other classes in the hierarchy based on *Reachability C*

```

public abstract class BillingDetail
{
    public int BillingDetailId { get; set; }
    public string Owner { get; set; }
    public string Number { get; set; }
}
  
```

```

}

public class BankAccount : BillingDetail
{
    public string BankName { get; set; }
    public string Swift { get; set; }
}

public class CreditCard : BillingDetail
{
    public int CardType { get; set; }
    public string ExpiryMonth { get; set; }
    public string ExpiryYear { get; set; }
}

public class InheritanceMappingContext : DbContext
{
    public DbSet<BillingDetail> BillingDetails { get; set; }
}

```

This object model is all that is needed to enable inheritance with Code First. If you put this in your app, you would be able to immediately start working with the database and do CRUD operations. Before going about how EF Code First maps this object model to the database, we need to learn about one of the types of inheritance mapping: polymorphic and non-polymorphic queries.

Polymorphic Queries

LINQ to Entities and EntitySQL, as object-oriented query languages, both support *polymorphic queries* for instances of a class and all instances of its subclasses, respectively. For example, consider the following query:

```

IQueryable<BillingDetail> linqQuery = from b in context.BillingDetails select b;
List<BillingDetail> billingDetails = linqQuery.ToList();

```

Or the same query in EntitySQL:

```

string eSqlQuery = @"SELECT VALUE b FROM BillingDetails AS b";
ObjectContext objectContext = ((IObjecContextAdapter)context).ObjectContext;
ObjectQuery<BillingDetail> objectQuery = objectContext.CreateQuery<BillingDetail>();
List<BillingDetail> billingDetails = objectQuery.ToList();

```

`linqQuery` and `eSqlQuery` are both polymorphic and return a list of objects of the type `BillingDetail` abstract class but the actual concrete objects in the list are of the subtypes of `BillingDetail`: `CreditCard` and `BankAccount`.

Non-polymorphic Queries

All LINQ to Entities and EntitySQL queries are polymorphic which return not only instances of the specified class to which it refers, but all subclasses of that class as well. On the other hand, *Non-polymorphic queries* whose polymorphism is restricted and only returns instances of a particular subclass. In LINQ this can be specified by using `OfType<T>()` Method. For example, the following query returns only `BankAccount`:

```
IQueryable<BankAccount> query = from b in context.BillingDetails ofType<BankAccount>
                                select b;
```

EntitySQL has [OFTYPE](#) operator that does the same thing:

```
string eSqlQuery = @"SELECT VALUE b FROM OFTYPE(BillingDetails, Model.BankAccount)
```

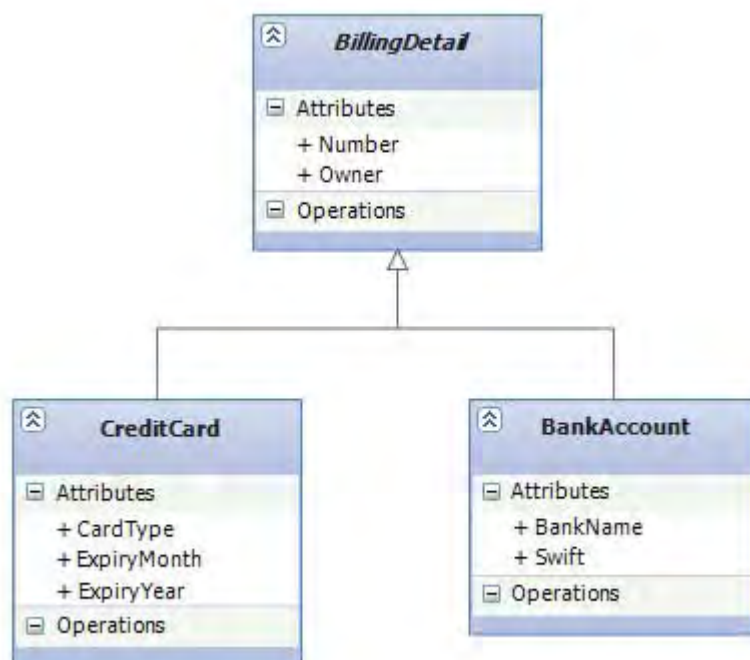
In fact, the above query with OFTYPE operator is a short form of the following query expression that uses [IS OF](#) operators:

```
string eSqlQuery = @"SELECT VALUE TREAT(b as Model.BankAccount)
                    FROM BillingDetails AS b
                    WHERE b IS OF(Model.BankAccount)";
```

(Note that in the above query, *Model.BankAccount* is the fully qualified name for BankAccount class change "Model" with your own namespace name.)

Table per Hierarchy (TPH)

An entire class hierarchy can be mapped to a single table. This table includes columns for all proper classes in the hierarchy. The concrete subclass represented by a particular row is identified by the discriminator column. You don't have to do anything special in Code First to enable TPH. It's the default mapping strategy:





| BillingDetails | | |
|----------------|---------------|----------|
| | Column Name | Nullable |
| 🔑 | BillingId | No |
| | Discriminator | No |
| | Owner | Yes |
| | Number | Yes |
| | BankName | Yes |
| | Swift | Yes |
| | CardType | Yes |
| | ExpiryMonth | Yes |
| | ExpiryYear | Yes |
| | | |

This mapping strategy is a winner in terms of both performance and simplicity. It's the best-performing strategy to represent polymorphism—both polymorphic and nonpolymorphic queries perform well—and it's even easy to implement by hand. Ad-hoc reporting is possible without complex joins or unions. Schema evolution is straightforward.

Discriminator Column

As you can see in the DB schema above, Code First has to add a special column to distinguish between classes: the discriminator. This isn't a property of the persistent class in our object model; it's used internally by Code First. By default, the column name is "Discriminator", and its type is string. The values default to the persistent class names—in this case, "BankAccount" or "CreditCard". EF Code First automatically selects the discriminator values when it retrieves the discriminator values.

TPH Requires Properties in SubClasses to be Nullable in the Database

TPH has one major problem: Columns for properties declared by subclasses will be nullable in the database. For example, Code First created an (INT, NULL) column to map CardType property in CreditCard class. In a typical mapping scenario, Code First always creates an (INT, NOT NULL) column in the database for a property in persistent class. But in this case, since BankAccount instance won't have a CardType property, the CardType field must be NULL for that row so Code First creates an (INT, NULL) instead. If your subclasses define several non-nullable properties, the loss of NOT NULL constraints may be a serious problem for your view of data integrity.

TPH Violates the Third Normal Form


Another important issue is normalization. We've created [functional dependencies](#) between nonkey columns, violating the [third normal form](#). Basically, the value of Discriminator column determines the corresponding columns that belong to the subclasses (e.g. BankName) but Discriminator is *not* part of the primary key table. As always, denormalization for performance can be misleading, because it sacrifices long-term maintainability, and the integrity of data for immediate gains that may be also achieved by proper optimization of SQL execution plans (in other words, ask your DBA).

Generated SQL Query

Let's take a look at the SQL statements that EF Code First sends to the database when we write queries for Entities or EntitySQL. For example, the polymorphic query for BillingDetails that you saw, generates the following SQL statement:

```
SELECT
  [Extent1].[Discriminator] AS [Discriminator],
  [Extent1].[BillingDetailId] AS [BillingDetailId],
  [Extent1].[Owner] AS [Owner],
  [Extent1].[Number] AS [Number],
  [Extent1].[BankName] AS [BankName],
  [Extent1].[Swift] AS [Swift],
  [Extent1].[CardType] AS [CardType],
  [Extent1].[ExpiryMonth] AS [ExpiryMonth],
  [Extent1].[ExpiryYear] AS [ExpiryYear]
FROM [dbo].[BillingDetails] AS [Extent1]
WHERE [Extent1].[Discriminator] IN ('BankAccount', 'CreditCard')
```

Or the non-polymorphic query for the BankAccount subclass generates this SQL statement:

Published Friday, December 24, 2010 4:47 AM by [morteza](#) 

Filed under: [Entity Framework](#), [C#](#), [Code First](#), [CTP5](#), [.NET](#)

```
SELECT
  [Extent1].[BillingDetailId] AS [BillingDetailId],
  [Extent1].[Owner] AS [Owner],
  [Extent1].[Number] AS [Number],
  [Extent1].[BankName] AS [BankName],
  [Extent1].[Swift] AS [Swift]
FROM [dbo].[BillingDetails] AS [Extent1]
WHERE [Extent1].[Discriminator] = 'BankAccount'
```

Comments

re: Inheritance Mapping Strategies with Entity Framework Code First CTP5 Part 1: Table per Hierarchy (TPH)

Friday, December 24, 2010 1:35 AM by [webdiyer](#)

nice feature, thanks

re: Inheritance Mapping Strategies with Entity Framework Code First CTP5 Part 1: Table per Hierarchy (TPH)

Friday, December 24, 2010 5:03 AM by Ericpoon

Nice post, but some words on the right side maybe over the content div, so I cannot read it through.

re: Inheritance Mapping Strategies with Entity Framework Code First CTP5 Part 1: Table per Hierarchy (TPH)

Friday, December 24, 2010 5:42 AM by David

Great post, thanks. Using this method how can I access the discriminator value from my object if I want to use it in a projection? Like `context.BillingDetails.Select(x => new { Number = x.Number, DiscriminatorValue = /* how do I get the discriminator value? */ });`

re: Inheritance Mapping Strategies with Entity Framework Code First CTP5 Part 1: Table per Hierarchy (TPH)

Friday, December 24, 2010 4:44 PM by [mortezam](#)

@Ericpoon: I've done some modifications so now it should be fully viewable on all screens with a resolution higher than 1024x768. Please check it out and let me know if you still have difficulties in viewing the blog post. Thank you for letting me know about this, really appreciate it :)

re: Inheritance Mapping Strategies with Entity Framework Code First CTP5 Part 1: Table per Hierarchy (TPH)

Friday, December 24, 2010 6:39 PM by [mortezam](#)

@David: If you want to read the discriminator value to filter the rows then you should instead write a non-polymorphic query like the one you saw in the post: `context.BillingDetails.OfType<BankAccount>()`. But if you are interested in the discriminator value itself — for example let's say for an Ad-hoc report — then you should be aware that the discriminator column is used internally by Code First and you cannot read/write its values from an inheritance mapping standpoint. To achieve this, the best way that I can think of is to use the new `SqlQuery` method on `DbContext.Database` which allows raw SQL queries to be executed against the database. But first we need to create a new non-persistent class since `SqlQuery` method is generic and needs a type to materialize the query results into (we cannot reuse `BillingDetail` class since `Discriminator` is not exposed in there). I named this new type as `BillingReport`:

```
[NotMapped]
public class BillingReport
{
    public int BillingId { get; set; }
    public string Number { get; set; }
    public string Owner { get; set; }
    public string Discriminator { get; set; }
    public string BankName { get; set; }
    public string Swift { get; set; }
    public int? CardType { get; set; }
    public string ExpiryMonth { get; set; }
    public string ExpiryYear { get; set; }
}
```

```
}
```

Now we can give this type to the `SqlQuery` method along with our custom SQL query:

```
List<BillingReport> reports = context.Database.SqlQuery<BillingReport>("SELECT * FROM BillingDetails").ToList();
```

Running this code will give us all the rows in the `BillingDetails` table with all the columns including the `Discriminator`. Hope this helps and thanks for your great question, I might add this trick to the post as well :)

[# re: Inheritance Mapping Strategies with Entity Framework Code First CTP5 Part 1: Table per Hierarchy \(TPH\)](#)

Sunday, December 26, 2010 1:22 AM by Venkat

Hello, which tool do you use for drawing UML diagrams ?

[# re: Inheritance Mapping Strategies with Entity Framework Code First CTP5 Part 1: Table per Hierarchy \(TPH\)](#)

Sunday, December 26, 2010 11:31 AM by [morte zam](#)

@Venkat: I use Visual Studio 2010 Architecture and Modeling features which come with the Ultimate edition. I've added a Modeling Project to my solution and created UML diagrams in there.

[# re: Inheritance Mapping Strategies with Entity Framework Code First CTP5: Part 1 – Table per Hierarchy \(TPH\)](#)

Tuesday, January 04, 2011 10:51 AM by [Batslhor](#)

I think you should start write some small project using ASP MVC, and summarize your posts in it.

[# re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy \(TPH\)](#)

Thursday, January 06, 2011 5:46 PM by [morte zam](#)

@Batslhor: That's a very good idea. I will definitely create a sample project like the one you suggested once I finish the Associations in EF Code First series. Thanks :)

[# re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy \(TPH\)](#)

Tuesday, January 11, 2011 5:54 AM by Sébastien F.

First of all, thank you for those post, it's really a great help when trying things with EF CF !

Do you know if there is a way, via the Fluent Api, to ignore a field on a subclass ?

I tried that : `Ignore(o => ((MyDerivedClass)o).MyFieldToIgnore)` but I get a "System.InvalidOperationException: The expression 'o => Convert(o).MyFieldToIgnore' is not a valid property expression. It must be of the form 'e => e.Property'.

[# re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy \(TPH\)](#)

Tuesday, January 11, 2011 9:26 PM by [morte zam](#)

@Sébastien F.: Yes, for that you need to call `Ignore` method and provide it with the specific property on the subclass that you want to ignore. For example, here is the fluent API code to ignore `CardType` in `CreditCard` subclass:


```
modelBuilder.Entity<CreditCard>().Ignore(c => c.CardType);
```

Alternatively, you may want to use Data Annotations for this matter. Placing a *NotMapped* attribute on the property will have the same result as the previous code:

```
public class CreditCard : BillingDetail
{
    [NotMapped]
    public int CardType { get; set; }
    public string ExpiryMonth { get; set; }
    public string ExpiryYear { get; set; }
}
```

Also thanks for your comment, I am glad you found it useful :)

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Wednesday, January 12, 2011 8:32 AM by [Cosmin Onea](#)

Can one of the subclasses have a required relationship?

E.g. Imagine CreditCard.CardType was a foreign key to CreditCardTypes table.

How would you map that?

In my case if I use HasRequired like

```
modelBuilder.Entity<CreditCard>().HasRequired(ct => ct.CardType);
```

I get

"Two entities with different keys are mapped to the same row. Ensure these two mapping fragments do not map two groups of entities with different keys to two overlapping groups of rows.

Problem in mapping fragments starting at lines 6, 49:Entity

Types CodeFirstNamespace.BankAccount, CodeFirstNamespace.CreditCard are being mapped to the same rows in table BillingDetails. Mapping conditions can be used to distinguish the rows that these types are mapped to."

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Thursday, January 13, 2011 10:18 AM by [morteza](#)

@Cosmin Onea: Yes, this scenario is perfectly possible with Code First. To create this association, first we add a new CreditCardType class to our data model:

```
public class CreditCardType
{
    public int CreditCardTypeId { get; set; }
    public string CardDescription { get; set; }
}
```

Then, we create a many-to-one association from CreditCard to CreditCardType by adding a new navigation property to our CreditCard subclass:

```
public class CreditCard : BillingDetail
```

```

{
    public int CardTypeId { get; set; }
    public string ExpiryMonth { get; set; }
    public string ExpiryYear { get; set; }
    public CreditCardType CreditCardType { get; set; }
}

```

(As you can see, I renamed CardType to CardTypeId so that it will be better distinguished from the new CreditCardType navigation property).

The last step would be to let Code First know about the foreign key for this association which is going to be our CardTypeId property:

```

public class InheritanceMappingContext : DbContext
{
    public DbSet<BillingDetail> BillingDetails { get; set; }
    public DbSet<CreditCardType> CreditCardTypes { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<CreditCard>().HasRequired(c =>
c.CreditCardType).WithMany().HasForeignKey(c => c.CardTypeId);
    }
}

```

Running this code will turn CardTypeId column to a foreign key referencing CreditCardTypes table. The interesting point is that even though we use *HasRequired()* method to configure this association (and also the fact that CardTypeId is not nullable) but still CardTypeId will be mapped to a (INT, NULL) column since we are in a TPH scenario which means it still needs to be null for BankAccount records in BillingDetails table. Hope this helps :)

[# re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy \(TPH\)](#)

Sunday, February 06, 2011 11:03 PM by [TearlessLight](#)

Awesome! Finally, after a week of searching, this is the first non-sensical explanation of what and how that stupid discriminator column does. I mean, I know what it is for but no one had any explanation of how it operates. Every example you see out there (besides this one) talks about how you can choose your own and set conditions on the mappings but none (NONE) of them work and are not supported but this explanation makes sense. You don't need to have those manual conditions set.

[# re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy \(TPH\)](#)

Monday, February 21, 2011 6:47 AM by [Nekketsu](#)

Very nice posts!!!

Is it possible to specify the type of discriminator column (Varchar(10), Varchar(25), etc)?

Thank you!

[# re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy \(TPH\)](#)

Monday, February 21, 2011 8:11 PM by [mortezam](#)

@[Nekketsu](#): Thanks for your comment. Currently there is no way to explicitly specify the data type of

the discriminator column in Code First. It is assumed from the type of the constant assigned to it in the fluent API code (like the example you saw in this post that the type inferred to be an int.). In fact, this is something that the EF team are looking into to see if they can enable it for the RTM.

For now the only way that I can think of is to change the discriminator column type directly by using the new *SqlCommand* method which allows raw SQL commands to be executed against the database. The best place to invoke *SqlCommand* method for this matter is inside a *Seed* method that has been overridden in a custom *Initializer* class:

```
protected override void Seed(EntityMappingContext context)
{
    context.Database.SqlCommand("ALTER TABLE BillingDetails ALTER COLUMN Discriminator
    NVARCHAR(20)");
}
```

Hope this helps :)

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Tuesday, February 22, 2011 1:47 PM by [TerminalFrost](#)

Just wanted to say thanks! This answered a lot of questions I had not only about EF but OO inheritance in relational databases as well.

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Wednesday, March 16, 2011 12:31 PM by [Jan C. de Graaf](#)

First, thanks for the article.

I have an issue with a TPH implementation using another model. Translated to your sample/structure the following code gives me an exception:

```
var creditCard = context.CreditCards.First();
var creditCardNowInDb = context.Entry(creditCard).GetDatabaseValues().ToObject();
```

The exception thrown is:

'CardType' is not a member of type 'CodeFirstNamespace.BillingDetail' in the currently loaded schemas. Near escaped identifier, line xx, column xx.

Any ideas?

Thanks, Jan C. de Graaf

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Thursday, March 17, 2011 3:02 PM by [mortezam](#)

@Jan C. de Graaf: This seems to be a bug that existed in CTP5 which has not been fixed in EF 4.1 RC yet. I saw your question on MSDN forum so we have to wait for EF team to confirm this bug. Meanwhile, as a workaround, you can use the *Reload* method (`context.Entry(creditCard).Reload()`) and then clone the *creditCard* object yourself, if that was the intention behind using *GetDatabaseValues* method. Thanks :)

Inheritance with Entity Framework 4.1 « Nathan's blog

Friday, April 08, 2011 9:41 AM by [Inheritance with Entity Framework 4.1 « Nathan's blog](#)

Pingback from [Inheritance with Entity Framework 4.1 « Nathan's blog](#)

Inheritance with Entity Framework 4.1 « Nathan's blog

Friday, April 08, 2011 10:12 AM by [Inheritance with Entity Framework 4.1 « Nathan's blog](#)

Pingback from [Inheritance with Entity Framework 4.1 « Nathan's blog](#)

Inheritance with Entity Framework 4.1 « Nathan's blog

Friday, April 08, 2011 11:21 AM by [Inheritance with Entity Framework 4.1 « Nathan's blog](#)

Pingback from [Inheritance with Entity Framework 4.1 « Nathan's blog](#)

Entity Framework 4.1 Code First????????????????????? – jixingseng « Hot Trends

Sunday, May 08, 2011 5:08 AM by [Entity Framework 4.1 Code First????????????????????? – jixingseng « Hot Trends](#)

Pingback from [Entity Framework 4.1 Code First????????????????????? – jixingseng « Hot Trends](#)

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Tuesday, May 10, 2011 3:22 AM by Aamir Ali

For accessing Discriminator column i added one Discriminator(notmapped) column to BillingDetail class but when i try to access, it always gives me null value why is it so??

Do i always have to make a new notmapped class to access the discriminator column is there no other way to do it???

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Wednesday, May 11, 2011 10:56 AM by [mortezam](#)

@Aamir Ali: Please read my answer to David's comment above. And yes, you always have to create a separate class to read the discriminator column value since like I said the discriminator column is internally used by EF and you cannot access it from the classes that participate in the inheritance hierarchy.

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Wednesday, May 11, 2011 1:48 PM by [bteal](#)

Thanks for the post. I'm using TPH on a small project that has a legacy schema and I'm running into a problem using a custom discriminator column. I'm using the fluent API to map the subclasses to the discriminator and everything works fine when reading data. On an add operation EF is throwing an exception saying that my discriminator column cannot be null. Is this something that EF should handle automatically under the covers based on the subclass type I'm trying to add or do I have to do something explicitly?

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Thursday, May 12, 2011 10:20 AM by [mortezam](#)

@bteal: Yes, EF internally writes the discriminator column value based on the object type and you don't need to do anything special in this regard. Could you please post your object model as well as the code that throws when you try to add an object to the hierarchy?

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Wednesday, May 18, 2011 11:37 AM by Anderson Fortaleza

Mr. Manavi, thank you for this most excellent and useful article!

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Friday, May 27, 2011 4:10 PM by horsinaround

Thank you for your article! There is so little about this topic out there at this time.

With a discriminator column as part of a complex primary key in a legacy table, is it even possible to use TPH in Code First? I am reading this: "Basically, the value of Discriminator column determines the corresponding values of the columns that belong to the subclasses (e.g. BankName) but Discriminator is not part of the primary key for the table." to mean No to my question. I am having a difficult time getting to work. Thanks in advance for a reply.

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Saturday, May 28, 2011 3:05 PM by [morteza](#)

@horsinaround: Actually by that line I was more explaining why a TPH mapping violates the third normal form but the answer to your question is still a no, unfortunately. The reason for that is because the discriminator column is internally used by EF and cannot be exposed as a property in the subclasses so you can't register it as a composite PK along with the other PK coming from the base class. If you don't expose it and configure it as the discriminator column for the TPH mapping then it wouldn't be part of the PK, so either ways we are out of luck to map this legacy table to an inheritance hierarchy.

Entity Framework 4.1 Code First learning path (2)

Friday, June 24, 2011 7:49 PM by [Entity Framework 4.1 Code First learning path \(2\)](#)

Pingback from Entity Framework 4.1 Code First learning path (2)

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Monday, July 11, 2011 9:13 PM by Venkatesh (heman_1978@hotmail.com)

Hi,

I have created the same example using a custom discriminator column and is resulting in an issue. I have created a question in MSDN forum. The link to the same is social.msdn.microsoft.com/.../d09fb12a-4751-4292-853a-a99759984cd6

could you please look into the query and let me know what mistake I made.

Thanks and regards

Venkatesh. S

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Tuesday, August 02, 2011 12:22 PM by drammer

Hi,

I am having trouble mapping an entity that derives from a class like CreditCard. I am doing something like the following:

BillingDetail (abstract / own table)

- BillingDetailType (discriminator) (e.g. BA, CC)

CreditCard (abstract / own table) inherits from BillingDetail

- CardType (discriminator) (e.g. Visa, MasterCard)

VisaCard (concrete / CreditCard table) inherits from CreditCard

- Level int

MasterCard (concrete / CreditCard table) inherits from CreditCard

- Level int

- Group int

I can't formulate the correct mapping.

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Sunday, August 07, 2011 3:20 PM by [morteza](#)

@drammer: What I understand from your desired schema is that in your first level of hierarchy, you want to use TPT to map the CreditCard class and also TPH to map other BillingDetail's subtypes such as BankAccount. This is NOT possible. You cannot mix inheritance hierarchies at a particular level. If you choose to have a separate table for the CreditCard entity then all other subclasses at the same level (e.g. BankAccount) also need to be mapped to a separate table as well. Mixing inheritance hierarchies on different levels is possible though. In your case, you can use TPT to map the first level of your hierarchy (i.e. CreditCard and BankAccount) and TPH to map the second level (i.e. VisaCard and MasterCard). The following object model shows how this can be done:

```
public abstract class BillingDetail
{
    public int BillingDetailId { get; set; }
    public string Number { get; set; }
}

public class BankAccount : BillingDetail
{
    public string BankName { get; set; }
}

public abstract class CreditCard : BillingDetail
{
    public int Level { get; set; }
}

public class VisaCard : CreditCard
{
    [Required]
    public string CardName { get; set; }
}

public class MasterCard : CreditCard
{
    public int Group { get; set; }
}

public class Context : DbContext
{
    public DbSet<BillingDetail> BillingDetails { get; set; }
    public DbSet<CreditCard> CreditCards { get; set; }
}
```

```

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<BillingDetail>().ToTable("BillingDetail");
    modelBuilder.Entity<BankAccount>().ToTable("BankAccount");

    modelBuilder.Entity<CreditCard>()
        .Map<VisaCard>(m => m.Requires(vc => vc.CardName).HasValue())
        .Map<MasterCard>(m => m.Requires(mc => mc.Group).HasValue())
        .ToTable("CreditCard");
}
}

```

This mapping gives you almost your desired schema except that I couldn't find a way to use a discriminator column like CardType on CreditCard table. Instead, I designate one column on each of the subclasses to be not-null for a row that represents an instance of that subclass. So for example a VisaCard row on the CreditCard table will always have a value for CardName and NULL for Group. Hope this helps.

EF 4.1 Mapping Inheritance on a Many-to-Many relationship - Programmers Goodies

Tuesday, August 16, 2011 12:52 AM by [EF 4.1 Mapping Inheritance on a Many-to-Many relationship - Programmers Goodies](#)

Pingback from [EF 4.1 Mapping Inheritance on a Many-to-Many relationship - Programmers Goodies](#)

How to map class hierarchy (base class and inherited classes) to a database - Programmers Goodies

Thursday, August 25, 2011 10:10 PM by [How to map class hierarchy \(base class and inherited classes\) to a database - Programmers Goodies](#)

Pingback from [How to map class hierarchy \(base class and inherited classes\) to a database - Programmers Goodies](#)

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Wednesday, September 21, 2011 9:57 AM by Mark Phillips

Thank you. very helpful

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Monday, October 03, 2011 4:39 PM by Tim

Hi, the current version of EF (SP2) does not support passing in a value in the "HasValue" method. Let's say I have a table "ContactPoints", and there's a "ContactPointTypeID" column which will be used as the discriminator. How would I map that into different subclasses of EmailAddress, PhoneNumber, and PostalAddress?

Ex. (does not work):

builder

```

.Entity<ContactPoint>()
    .HasKey(p => p.ID)
    .Map<EmailAddress>(m =>
{

```

```

    m.MapInheritedProperties();

    m.Requires(cp => cp.PointTypeID).HasValue(/* There is no way to pass in a value! There's no
parameters! */);

    })

    .ToTable("ContactPoint", "contact");

```

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Thursday, October 06, 2011 10:08 PM by [morte zam](#)

@Tim: The following object model will create a TPH mapping for your domain:

```

public abstract class ContactPoint
{
    public int Id { get; set; }
}

public class EmailAddress : ContactPoint
{
    public string Email { get; set; }
}

public class PhoneNumber : ContactPoint
{
    public string Phone { get; set; }
}

public class PostalAddress : ContactPoint
{
    public string Postal { get; set; }
}

public class Context : DbContext
{
    public DbSet<ContactPoint> ContactPoints { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<ContactPoint>()
            .Map<EmailAddress>(m => m.Requires("ContactPointTypeID").HasValue("EA"))
            .Map<PhoneNumber>(m => m.Requires("ContactPointTypeID").HasValue("PN"))
            .Map<PostalAddress>(m => m.Requires("ContactPointTypeID").HasValue("PA"))
            .ToTable("ContactPoint", "Contact");
    }
}

```

Please note that the Requires method overload that you've used is NOT meant to be used when you have a discriminator column. It's only for when you want to designate the subtype based on the nullability of one column in the mapped table. See my answer to "drammer" above for an example. Hope this helps.

Entity framework code first creates ???discriminator??? column - Programmers Goodies

Friday, October 07, 2011 6:40 AM by [Entity framework code first creates ???discriminator??? column - Programmers Goodies](#)

Pingback from Entity framework code first creates ???discriminator??? column - Programmers Goodies

Entity Framework Code First 0 to 1 mapping - Programmers Goodies

Tuesday, October 11, 2011 11:00 AM by [Entity Framework Code First 0 to 1 mapping - Programmers Goodies](#)

Pingback from Entity Framework Code First 0 to 1 mapping - Programmers Goodies

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Wednesday, October 26, 2011 2:27 PM by Jonny

Hi!

Great post (as all of your others regarding EF 4.1 Code First).

I've got one problem, I have implemented the "State pattern" and the state object uses "table per hierarchy". I have an Order entity that contains a State entity, and the State entity contains the corresponding Order entity.

When I create my Order it gets the state Created and when I save it, it has the right state. When I read my Order back (through a repository) and manipulate it to change its state and save it again and then read it back once again it still has the same state (Created).

If I don't save the created Order until it has a new state, that state will be saved correctly. So the problem is that once the Order is saved I can't change the state.

Sorry for my poor English but I hope you understand my problem!?

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Thursday, October 27, 2011 1:58 PM by Jonny

Nevermind my question, I found the problem.

re: Inheritance with EF Code First CTP5: Part 1 – Table per Hierarchy (TPH)

Sunday, November 06, 2011 5:17 PM by rekna

Consider following TPH scenario :

A (base class)

AB (inherited class) FK referenceId => reference to B.Id

AC (inherited class) FK referenceId => reference to B.Id

AD (inherited class) no reference to B

I can't get this configured. It looks like it is thinking there should be 2 FK's referenceId and referenceId2, but I want it to map to the same referenceId (there should be only one column referenceId in the table mapped to A)

P.S. I already have a table (I'm not generating it from code).

re: Inheritance with EF Code First: Part 1 – Table per Hierarchy (TPH)

Friday, November 18, 2011 11:50 AM by [morteza](#)

@rekna: This is not possible. Best thing you can do is to move the B reference to the base class (class A), like the following object model:

```
class A
```

```
{
    public int Id { get; set; }
    public int BId { get; set; }
    public B B { get; set; }
}

class B
{
    public int BId { get; set; }
}

class AB : A { }

class AC : A { }

class AD : A { }
```

[# Inheritance with EF Code First: Part 1 ??? Table per Hierarchy \(TPH\) | My Blog](#)

Wednesday, January 18, 2012 10:17 PM by [Inheritance with EF Code First: Part 1 ??? Table per Hierarchy \(TPH\) | My Blog](#)

Pingback from [Inheritance with EF Code First: Part 1 ??? Table per Hierarchy \(TPH\) | My Blog](#)

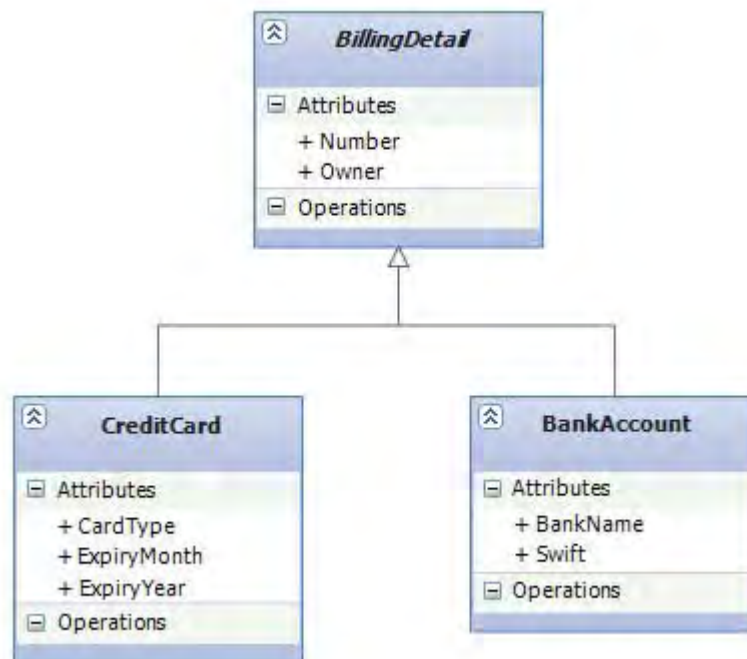
[Terms of Use](#)

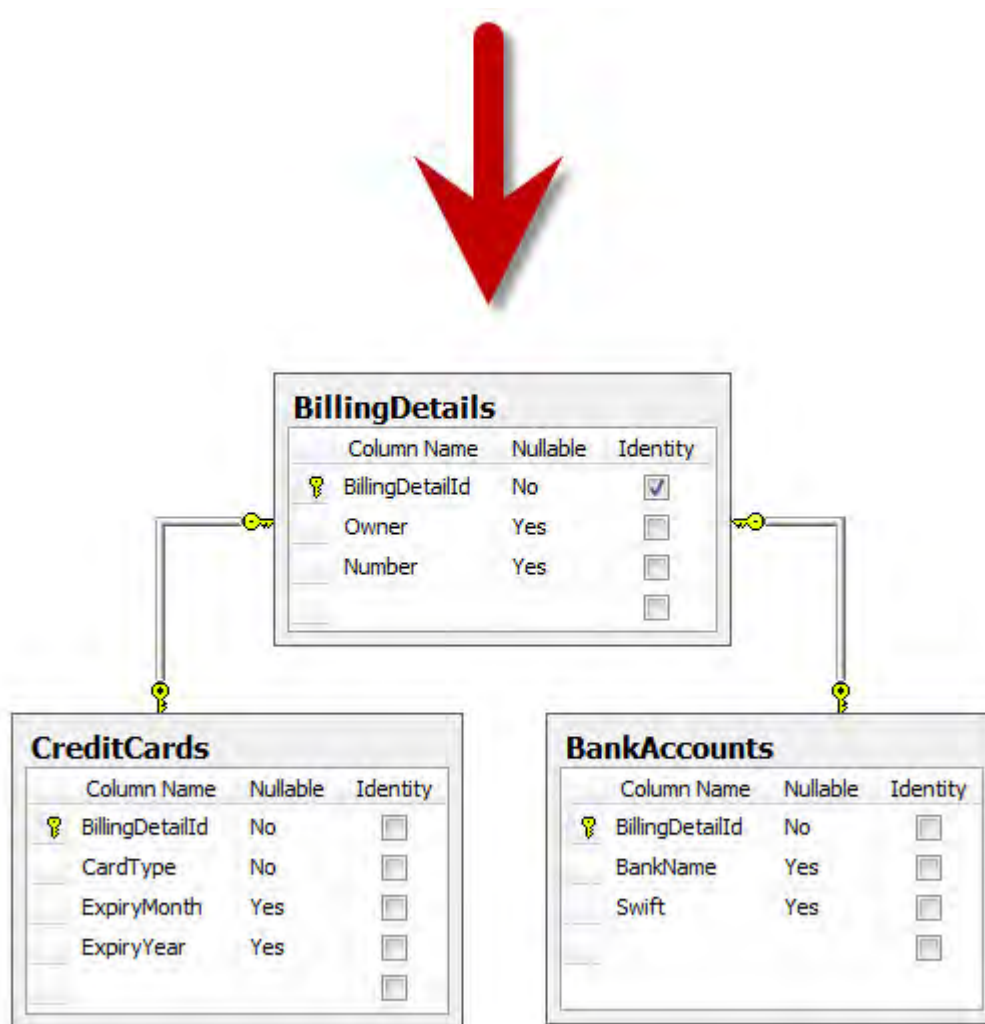
Inheritance with EF Code First: Part 2 – Table per Type (TPT)

In the previous [blog post](#) you saw that there are three different approaches to representing an inheritance relationship in EF Code First. We explained Table per Hierarchy (TPH) as the default mapping strategy in EF Code First. We argued that TPH may be too serious for our design since it results in denormalized schemas that can become a pain to run. In today's blog post we are going to learn about Table per Type (TPT) as another inheritance mapping strategy. We will see that TPT doesn't expose us to this problem.

Table per Type (TPT)

Table per Type is about representing inheritance relationships as relational foreign key associations. Each class that declares persistent properties—including abstract classes—has its own table. The table for subclass contains its own primary key for each noninherited property (each property declared by the subclass itself) along with a primary key that references the base class table. This approach is shown in the following figure:





For example, if an instance of the CreditCard subclass is made persistent, the values of properties (base class) are persisted to a new row of the BillingDetails table. Only the values of properties declared in the subclass (CreditCard) are persisted to a new row of the CreditCards table. The two rows are linked together by the BillingDetailId value. Later, the subclass instance may be retrieved from the database by joining the subclass table to the BillingDetails table.

TPT Advantages

The primary advantage of this strategy is that the SQL schema is normalized. In addition, schema evolution (modifying the base class or adding a new subclass is just a matter of modify/add one table). Integrity constraints are also straightforward (note how CardType in CreditCards table is now a non-nullable column).

Implement TPT in EF Code First

We can create a TPT mapping simply by placing Table attribute on the subclasses to specify the mapping. The Table attribute is a new data annotation and has been added to [System.ComponentModel.DataAnnotations](#).

```
public abstract class BillingDetail
{
    public int BillingDetailId { get; set; }
    public string Owner { get; set; }
}
```

```

    public string Number { get; set; }
}

[Table("BankAccounts")]
public class BankAccount : BillingDetail
{
    public string BankName { get; set; }
    public string Swift { get; set; }
}

[Table("CreditCards")]
public class CreditCard : BillingDetail
{
    public int CardType { get; set; }
    public string ExpiryMonth { get; set; }
    public string ExpiryYear { get; set; }
}

public class InheritanceMappingContext : DbContext
{
    public DbSet<BillingDetail> BillingDetails { get; set; }
}

```

If you prefer **fluent API**, then you can create a TPT mapping by using `ToTable()` method:

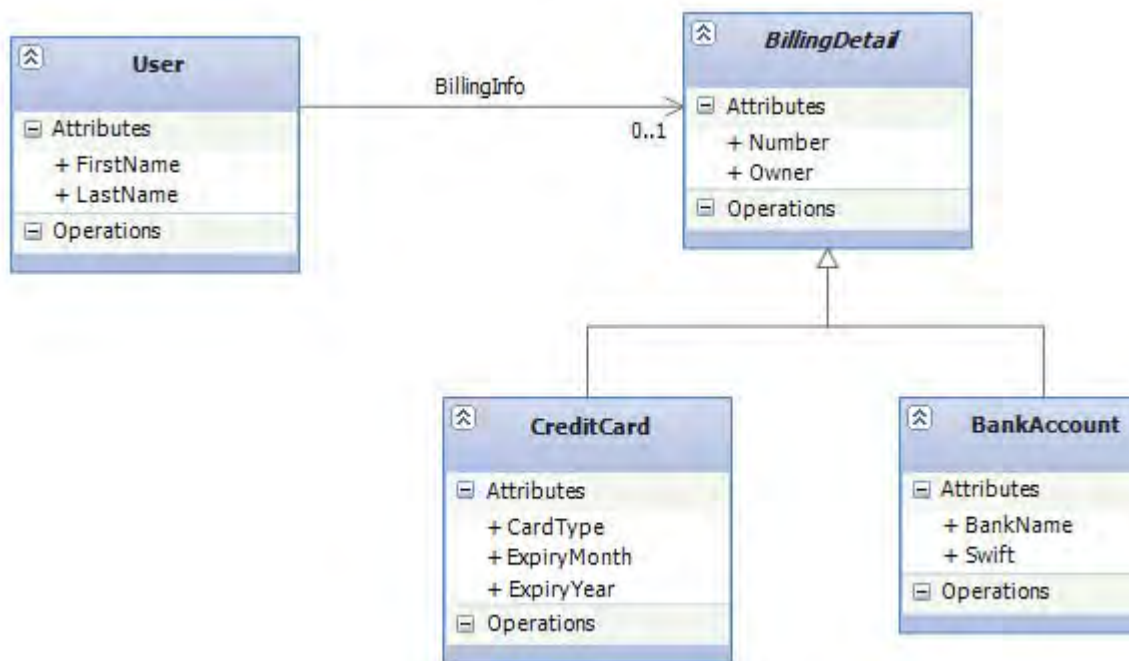
```

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<BankAccount>().ToTable("BankAccounts");
    modelBuilder.Entity<CreditCard>().ToTable("CreditCards");
}

```

Polymorphic Associations

A *polymorphic association* is an association to a base class, hence to all classes in the hierarchy with a concrete class at runtime. For example, consider the `BillingInfo` property of `User` in the following for one particular `BillingDetail` object, which at runtime can be any concrete instance of that class.



In fact, because `BillingDetail` is abstract, the association must refer to an instance of one of its subclasses—`BankAccount`—at runtime.

Implement Polymorphic Associations with EF Code First

We don't have to do anything special to enable polymorphic associations in EF Code First; The use association to some `BillingDetails`, which can be `CreditCard` or `BankAccount` so we just create this a naturally polymorphic:

```

public class User
{
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int BillingDetailId { get; set; }

    public virtual BillingDetail BillingInfo { get; set; }
}
  
```

In other words, as you can see above, a polymorphic association is an association that may refer in class that was explicitly specified as the type of the navigation property (e.g. `User.BillingInfo`).

The following code demonstrates the creation of an association to an instance of the `CreditCard` sub

```

using (var context = new InheritanceMappingContext())
{
    CreditCard creditCard = new CreditCard()
    {
        Number = "987654321",
        CardType = 1
    }
}
  
```

```

};
User user = new User()
{
    UserId      = 1,
    BillingInfo = creditCard
};
context.Users.Add(user);
context.SaveChanges();
}

```

Now, if we navigate the association in a second context, EF Code First automatically retrieves the C

```

using (var context = new InheritanceMappingContext())
{
    User user = context.Users.Find(1);
    Debug.Assert(user.BillingInfo is CreditCard);
}

```

Polymorphic Associations with TPT

Another important advantage of TPT is the ability to handle polymorphic associations. In the database association to a particular base class will be represented as a foreign key referencing the table of the (e.g. Users table has a foreign key that references BillingDetails table.)

Generated SQL For Queries

Let's take an example of a simple non-polymorphic query that returns a list of all the BankAccounts:

```
var query = from b in context.BillingDetails.OfType<BankAccount>() select b;
```

Executing this query (by invoking `ToList()` method) results in the following SQL statements being :
 the bottom, you can also see the result of executing the generated query in SQL Server Managemen

```

SELECT
  'OXOX' AS [C1],
  [Extent1].[BillingDetailId] AS [BillingDetailId],
  [Extent2].[Owner] AS [Owner],
  [Extent2].[Number] AS [Number],
  [Extent1].[BankName] AS [BankName],
  [Extent1].[Swift] AS [Swift]
FROM [dbo].[BankAccounts] AS [Extent1]
INNER JOIN [dbo].[BillingDetails] AS [Extent2]
ON [Extent1].[BillingDetailId] = [Extent2].[BillingDetailId]

```

Results

| | C1 | BillingDetailId | Owner | Number | BankName | Swift |
|---|------|-----------------|---------|-----------|----------|-------------|
| 1 | OXOX | 1 | Morteza | 123456789 | CIBC | CORPINBB303 |

Now, let's take an example of a very simple polymorphic query that requests all the BillingDetails with BankAccount and CreditCard types:

```
var query = from b in context.BillingDetails select b;
```

This LINQ query seems even more simple than the previous one but the resulting SQL query is not expect:


```

SELECT
CASE WHEN ([UnionAll1].[C3] = 1) THEN 'OX0X' ELSE 'OX1X' END AS [C1]
[UnionAll1].[BillingDetailId] AS [C2],
[Extent3].[Owner] AS [Owner],
[Extent3].[Number] AS [Number],
CASE WHEN ([UnionAll1].[C3] = 1) THEN [UnionAll1].[C1] END AS [C3],
CASE WHEN ([UnionAll1].[C3] = 1) THEN [UnionAll1].[C2] END AS [C4],
CASE WHEN ([UnionAll1].[C3] = 1) THEN
    CAST(NULL AS int) ELSE [UnionAll1].[CardType] END AS [C5],
CASE WHEN ([UnionAll1].[C3] = 1) THEN
    CAST(NULL AS varchar(1)) ELSE [UnionAll1].[ExpiryMonth] END AS [
CASE WHEN ([UnionAll1].[C3] = 1) THEN
    CAST(NULL AS varchar(1)) ELSE [UnionAll1].[ExpiryYear] END AS [C
FROM
    (SELECT
    [Extent1].[BillingDetailId] AS [BillingDetailId],
    CAST(NULL AS varchar(1)) AS [C1],
    CAST(NULL AS varchar(1)) AS [C2],
    [Extent1].[CardType] AS [CardType],
    [Extent1].[ExpiryMonth] AS [ExpiryMonth],
    [Extent1].[ExpiryYear] AS [ExpiryYear],
    cast(0 as bit) AS [C3]
    FROM [dbo].[CreditCards] AS [Extent1]
    UNION ALL
    SELECT
    [Extent2].[BillingDetailId] AS [BillingDetailId],
    [Extent2].[BankName] AS [BankName],
    [Extent2].[Swift] AS [Swift],
    CAST(NULL AS int) AS [C1],
    CAST(NULL AS varchar(1)) AS [C2],
    CAST(NULL AS varchar(1)) AS [C3],
    cast(1 as bit) AS [C4]
    FROM [dbo].[BankAccounts] AS [Extent2])
AS [UnionAll1]
INNER JOIN [dbo].[BillingDetails] AS [Extent3]
ON [UnionAll1].[BillingDetailId] = [Extent3].[BillingDetailId]

```

Results

| | C1 | C2 | Owner | Number | C3 | C4 | C5 | C6 | C7 |
|---|------|----|---------|-----------|------|-------------|------|------|------|
| 1 | OX1X | 2 | Morteza | 987654321 | NULL | NULL | 1 | 10 | 12 |
| 2 | OX0X | 1 | Morteza | 123456789 | CIBC | CORPINBB303 | NULL | NULL | NULL |

As you can see, EF Code First relies on an [INNER JOIN](#) to detect the existence (or absence) of row CreditCards and BankAccounts so it can determine the concrete subclass for a particular row of the the SQL [CASE](#) statements that you see in the beginning of the query is just to ensure columns that particular row have NULL values in the returning flattened table. (e.g. BankName for a row that repr

TPT Considerations


Even though this mapping strategy is deceptively simple, the experience shows that performance is complex for complex class hierarchies because queries always require a join across many tables. In addition, it is difficult to implement by hand— even ad-hoc reporting is more complex. This is an important consideration when using handwritten SQL in your application (For ad hoc reporting, database views provide a way to offset this strategy. A view may be used to transform the table-per-type model into the much simpler table-per-

Summary

In this post we learned about Table per Type as the second inheritance mapping in our series. So far we have discussed require extra consideration with regard to the SQL schema (e.g. in TPT, foreign keys are changes with the Table per Concrete Type (TPC) that we will discuss in the next post.

References

- [ADO.NET team blog](#)
- [Java Persistence with Hibernate book](#)

Published Tuesday, December 28, 2010 11:41 PM by [morteza](#) 

Filed under: [Entity Framework](#), [C#](#), [Code First](#), [CTP5](#), [.NET](#)

Comments

re: Inheritance Mapping Strategies with Entity Framework Code First CTP5: Part 2 – Table per Type (TPT)

Wednesday, December 29, 2010 3:21 PM by [Yakup Ipek](#)

Very nice explained article.Thanks

re: Inheritance Mapping with EF Code First CTP5: Part 2 – Table per Type (TPT)

Wednesday, January 05, 2011 8:14 AM by [Juliën](#)

Hi Morteza,

Excellent posts on the matter.

However, having read your posts on EF CTP5 mapping associations and several posts on the MSDN Forums I'm having difficulty with a "simple" mapping approach you mention. I hope you are willing to shed some light on it.

It regards a TPT project with EF Code First, CTP5. With this, there is a simple concept where every object (model) is derived from a BaseEntity. This works fine, except the 1-to-1 mapping strategy.

The question is quite straightforward: is the concept below possible?

```
[Table("BaseEntity")]
public abstract class BaseEntity
{
    [Key, DatabaseGenerated(DatabaseGenerationOption.Identity)]
    public Guid Id { get; set; }
    public Guid Owner { get; set; }
    public DateTime DateAdd { get; set; }
    public DateTime DateMod { get; set; }
}
```

```
[Table("Customer")]
public class Customer : BaseEntity
{
    public string DisplayName { get; set; }
    public Address DeliveryAddress { get; set; }
    public Address BillingAddress { get; set; }
}

[Table("Address")]
public class Address : BaseEntity
{
    public string Streetname { get; set; }
    public string HouseNumber { get; set; }
    public string City { get; set; }
}
```

Example - DbContext

My DbContext would look something like:

```
public class MagicContext : DbContext
{
    public DbSet<BaseEntity> Objects { get; set; }
    public MagicContext(string connectionString) : base(connectionString) {}
    protected override void OnModelCreating(System.Data.Entity.ModelConfiguration.ModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<Customer>()
            .HasOptional(c => c.BillingAddress)
            .WithRequired();

        // ??? How to map Address, if needed
    }
}
```

In a nutshell:

1. A Customer can have an optional –not required- DeliveryAddress and/or BillingAddress
2. The properties DeliveryAddress as well as BillingAddress can refer to the same – or separate – records in the “Address” table

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Friday, January 14, 2011 12:50 PM by [mortezam](#)

@Juliën: Yes, it is absolutely possible but we need to do some modifications. You have an interesting object model since you used inheritance to keep common properties in a base entity. However, the strategy that you've chosen (TPT) to map this inheritance is not the right one for this scenario. The best strategy to use in this type of scenarios is [Table per Concrete Type \(TPC\)](#) where inheritance is used for the top level of the class hierarchy and polymorphism isn't really required. So first we need to change it to TPC by the following fluent API code:

```
modelBuilder.Entity<Customer>().Map(m =>
{
    m.MapInheritedProperties();
    m.ToTable("Customer");
});

modelBuilder.Entity<Address>().Map(m =>
{
```

```

    m.MapInheritedProperties();
    m.ToTable("Address");
});

```

Next, what you want to accomplish in terms of having multiple addresses for a customer is not possible with a *shared primary key association*. Like I described in [this post](#) as the second limitation for this type of association, if we need to have more than one address for a Customer entity (e.g. Billing Address and Delivery Address) then this mapping style wouldn't be adequate. This scenario would be best achieved by creating a *One-to-One Foreign Key Association* for each of the Addresses. However, CTP5 (and EF in general) does not natively support one-to-one FK associations (It's likely to be supported in the next RTM though), so we will create one-to-many associations between customer and address and then will manually create a unique constraint on foreign keys in Customer table to preserve data integrity in the database (e.g. two users cannot have the same address for their billing). To do that, first we need to change the Customer entity to introduce our new navigation properties as well as their corresponding foreign keys:

```

public class Customer : BaseEntity
{
    public string DisplayName { get; set; }

    public Guid? BillingAddressId { get; set; }
    public virtual Address BillingAddress { get; set; }

    public Guid? DeliveryAddressId { get; set; }
    public virtual Address DeliveryAddress { get; set; }
}

```

Next we configure the associations by fluent API:

```

modelBuilder.Entity<Customer>().HasOptional(c => c.BillingAddress).WithMany().HasForeignKey(c =>
c.BillingAddressId).WillCascadeOnDelete(false);
modelBuilder.Entity<Customer>().HasOptional(c => c.DeliveryAddress).WithMany().HasForeignKey(c
=> c.DeliveryAddressId).WillCascadeOnDelete();

```

To create the unique constraints on foreign keys (to make sure that each address has been used by one customer only), I took advantage of the new CTP5's *SqlCommand()* method on *DbContext.Database* which allows raw SQL commands to be executed against the database. This can be done in the *Seed()* method in a custom Initializer class:

```

context.Database.SqlCommand("ALTER TABLE Customer ADD CONSTRAINT uc_DeliveryAddressId
UNIQUE(DeliveryAddressId)");
context.Database.SqlCommand("ALTER TABLE Customer ADD CONSTRAINT uc_BillingAddressId
UNIQUE(BillingAddressId)");

```

And that's all we need to do to make it work. I created a sample project and put all these together so that you can run and see it for yourself which can be downloaded from [here](#).

Thanks for your comment by the way, I'm glad it helped :)

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Wednesday, March 09, 2011 11:28 AM by Mickael

Hello,

Firstable I want to thank you for great posts you made about EF Code First CTP5.

Allow me to ask you a question about Table Per Type (TPT) strategy.

For example if I have a class hierarchy with a class named "SuperClass" and other class named "SubClass", the TPT strategy will create database table "SuperClass" and table "SubClass". How can I force the foreign key created to associate the two tables to have delete rule as Cascade?

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Wednesday, March 09, 2011 5:46 PM by [mortezam](#)

@Mickael: As of CTP5, there is no way to switch on cascade delete between the related tables in the hierarchy. In fact, you don't really need it since Code First will always delete both records in the Subclass and SuperClass tables once you remove a SubClass object. If you are interested to have cascade deletes turned on to preserve the database referential integrity, then you have to manually switch it on after Code First creates the database. I realize this isn't a great answer and I hope we would be able to do this by using fluent API in the RTM. Thanks :)

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Thursday, March 10, 2011 5:45 AM by Mickael

Thanks Morteza for the fast answer. Yesterday I recheck all my domain code and found that I was making some mistakes. I also disabled `OneToManyCascadeDeleteConvention` and manually delete child instances. That solved my problem.

Like you said in your last comment, deleting a record of a subclass automatically deletes the record of its superclass.

Thanks again.

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Friday, March 18, 2011 10:13 AM by von

I have a problem with a class inheriting from an abstract class. If I set a property as `HasOptional` the mapping does not work. However, if I change it to `HasRequired` then everything works. This is the error message I get:

(13,10) : error 3032: Problem in mapping fragments starting at lines 13, 35:EntityTypes CodeFirstNamespace.Employee are being mapped to the same rows in table Employee. Mapping conditions can be used to distinguish the rows that these types are mapped to.

Here's the code:

```
public abstract class Person
{
    public Guid Id { get; set; }
}

public class Employee : Person
{
    public DateTime? DateOfBirth { get; set; }
    public virtual Department Department { get; set; }
}

public class Department
{
    public Guid Id { get; set; }
    public string Name { get; set; }
}
```

```

}

public class MyContext : DbContext
{
    public DbSet<Employee> Employees { get; set; }
    public DbSet<Department> Departments { get; set; }

    protected override void OnModelCreating(System.Data.Entity.ModelConfiguration.ModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<Person>().ToTable("Person");
        modelBuilder.Entity<Employee>().ToTable("Employee");
        modelBuilder.Entity<Employee>().HasRequired(e => e.Department);
    }
}

```

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Friday, March 18, 2011 11:27 AM by [mortezam](#)

@von: This was a bug in CTP5 which has been fixed in EF 4.1 RC. You can find the latest EF release from [here](#). Once you install it, your code will perfectly create the desired schema. The only modification you need in your code is to change ModelBuilder to *DbModelBuilder*, which is the new name they choose for ModelBuilder class to align with the other core classes. Hope this helps :)

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Saturday, March 19, 2011 6:28 AM by von

Thanks Morteza! I'll give it a try.

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Sunday, March 20, 2011 4:57 AM by von

I just like to say my issue has been resolved. Microsoft has done a good job on that one. Thanks to you again Morteza.

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Sunday, March 20, 2011 9:01 AM by Von

Hi Again,

I've posted my issue here (blogs.msdn.com/.../ef-4-1-release-candidate-available.aspx). But let me post here as well as I've found your blog to be of more help. But please take note that the issue is with RC1, the following code is working in CTP5. So the problem is, if I have more than one property of the same entity type in a single class, EF will throw an error "error 0111: There is no property with name 'REFERENCED_ENTITY_Id1' defined in type referred by Role 'REFERENCING_ENTITY'". Here is the code:

```

public abstract class Note
{
    public virtual Guid Id { get; set; }
    public virtual string Description { get; set; }
}

public class Note1 : Note
{
    public virtual Employee Employee { get; set; }
}

public class Note2 : Note
{
    public virtual Employee Employee { get; set; }
}

```



```

}

public abstract class Person
{
    public Guid Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person
{
    public DateTime? DateOfBirth { get; set; }
}

public class MyContext : DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>().ToTable("Person");
        modelBuilder.Entity<Employee>().ToTable("Employee");
        modelBuilder.Entity<Note>().ToTable("Note");
        modelBuilder.Entity<Note1>().ToTable("Note1");
        modelBuilder.Entity<Note1>().HasRequired(x => x.Employee);
        modelBuilder.Entity<Note2>().ToTable("Note2");
        modelBuilder.Entity<Note2>().HasRequired(x => x.Employee);
    }
}

```

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Sunday, March 20, 2011 10:24 AM by [mortezaam](#)

@von: This seems to be a bug in EF 4.1 RC. You should be able to create a schema from this object model as you did in CTP5. One workaround would be to rename the Employee navigation property on Note1 and Note2 subclasses:

```

public class Note1 : Note
{
    public virtual Employee Employee1 { get; set; }
}

public class Note2 : Note
{
    public virtual Employee Employee2 { get; set; }
}

```

While this will resolve the issue, a better solution would be to move up the Employee property to the Note base class which not only creates the desired schema but also offers a better OO design. The generated schema would be a bit different though since in this way the Note table will hold the EmployeeId FK instead of Note1 and Note2 tables holding it:

```

public abstract class Note
{
    public virtual Guid Id { get; set; }
    public virtual string Description { get; set; }
    public virtual Employee Employee { get; set; }
}

```

```
public class Note1 : Note { }
```

```
public class Note2 : Note { }
```

Hope this helps :)

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Sunday, March 20, 2011 11:02 AM by Von

Not an elegant solution but I'll try the first suggestion. The second suggestion will not for me because the Note class is being used by other classes - other than Employee. Thanks again for your quick reply!

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Sunday, March 20, 2011 12:49 PM by [mortezaam](#)

@von: I realize that my first suggested workaround is not a great solution, in fact, I was about to suggest another solution which was to change your *independent associations* to *foreign key associations* by exposing the FK properties (EmployeeId) in Note1 and Note2 subclasses like the following:

```
public class Note1 : Note
{
    public Guid EmployeeId { get; set; }
    public virtual Employee Employee { get; set; }
}
```

```
public class Note2 : Note
{
    public Guid EmployeeId { get; set; }
    public virtual Employee Employee { get; set; }
}
```

However, I didn't bring this up due to another bug in EF 4.1 RC and that is if you run this object model, you wouldn't get an exception anymore but Code First creates the FK constraint on EmployeeId only on one of the subclass tables (Note1 in this case) and fails to create it on the other one. You may want to save this for the final RTM since foreign key associations are always recommended regardless.

More to the point, from what I can see, probably the best solution here is to replace inheritance with aggregation since you can well find that the inheritance isn't adding any value to your object model except that it creates your desired DB schema which is well achievable in other ways:

```
public class Note
{
    public virtual Guid Id { get; set; }
    public virtual string Description { get; set; }
}
```

```
public class Note1
{
    public virtual Guid Id { get; set; }
    public Guid EmployeeId { get; set; }

    public virtual Employee Employee { get; set; }
}
```



```

    public virtual Note Note { get; set; }
}

public class Note2
{
    public virtual Guid Id { get; set; }
    public Guid EmployeeId { get; set; }

    public virtual Employee Employee { get; set; }
    public virtual Note Note { get; set; }
}

public abstract class Person
{
    public Guid Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person
{
    public DateTime? DateOfBirth { get; set; }
}

public class Context : DbContext
{
    public DbSet<Person> Persons { get; set; }
    public DbSet<Note> Notes { get; set; }
    public DbSet<Note1> Notes1 { get; set; }
    public DbSet<Note2> Notes2 { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>().ToTable("Person");
        modelBuilder.Entity<Employee>().ToTable("Employee");

        modelBuilder.Entity<Note1>().HasRequired(x => x.Note).WithRequiredDependent();
        modelBuilder.Entity<Note2>().HasRequired(x => x.Note).WithRequiredDependent();
    }
}

```

Here I created two [Shared Primary Key Associations](#) between Note and Note1 and Note and Note2 which at the end creates the very same database schema, of course without any exception :)

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Sunday, March 20, 2011 8:35 PM by Von

I've tried your first suggestion having " public Guid EmployeeId { get; set; } ". And actually it was in my CTP5 code as CTP5 has a bug with FK, in that having additional Id property (eg EmployeeId for Employee entity) fixes it. The problem is, if I do that EF produces an additional EmployeeId column in the table.

Now for the second solution, it will break a lot of what I already wrote for the UI and business logic rules. If I

use the second solution, to be able to get to the Description field I need to do this.

*employee is an instance of Employee

employee.Note1.Note.Description

where in the UI there are already codes like this:

employee.Note1.Description

I'll see if the first solution works.

So this is what I am looking to have:

```
public abstract class Note
{
    public Guid Id { get; set; }
    public string Description { get; set; }
}

public class Note1 : Note
{
    public virtual Employee Employee { get; set; }
}

public class Note2 : Note
{
    public virtual Employee Employee { get; set; }
}

public class Employee : Person
{
    public virtual Note1 SomeNotes { get; set; }
    public virtual Note2 SomeOtherNotes { get; set; }
}
```

I'll update you of what I'll find out. Thanks for your help!

[# re: Inheritance with EF Code First CTP5: Part 2 – Table per Type \(TPT\)](#)

Sunday, March 20, 2011 9:10 PM by Von

Okay so the code below did the trick. I think the problem before was, I declared EmployeeId as virtual in Note1 and Note2 and that instructed EF to create a column for it.

```
public abstract class Person
{
    public Guid Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person
{
    public DateTime? DateOfBirth { get; set; }
    public virtual IList<Note1> SomeNotes { get; set; }
    public virtual IList<Note2> SomeOtherNotes { get; set; }
}

public abstract class Note
{
    public virtual Guid Id { get; set; }
    public virtual string Description { get; set; }
}

public class Note1 : Note
{
    public Guid EmployeeId { get; set; }
}
```

```

    public virtual Employee Employee { get; set; }
}

public class Note2 : Note
{
    public Guid EmployeeId { get; set; }
    public virtual Employee Employee { get; set; }
}

public class MyContext : DbContext
{
    public DbSet<Person> People { get; set; }
    public DbSet<Employee> Employees { get; set; }
    public DbSet<Note> Notes { get; set; }
    public DbSet<Note1> Notes1 { get; set; }
    public DbSet<Note2> Notes2 { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>().ToTable("Person");
        modelBuilder.Entity<Employee>().ToTable("Employee");
        modelBuilder.Entity<Employee>().HasMany(e => e.SomeNotes);
        modelBuilder.Entity<Employee>().HasMany(e => e.SomeOtherNotes);
        modelBuilder.Entity<Note>().ToTable("Note");
        modelBuilder.Entity<Note1>().ToTable("Note1");

        // If I put the .WithMany() here, EF complains about:
        // There is no property with name 'Employee_Id1' defined in type referred by Role 'Note'.
        // And if I remove SomeOtherNotes and leave SomeNotes the error will go away.

        modelBuilder.Entity<Note1>().HasRequired(x => x.Employee);
        modelBuilder.Entity<Note2>().ToTable("Note2");
        modelBuilder.Entity<Note2>().HasRequired(x => x.Employee);
    }
}

```

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Wednesday, March 23, 2011 10:15 AM by David

Is there any way to add Include() clauses for TPT (or TPH) class hierarchies to a query?

I have 4 classes ServiceSpec_PhysicalAudio, ServiceSpec_PhysicalSDVideo, ServiceSpec_PhysicalHDVideo, ServiceSpec_Digital that inherit from a ServiceSpec base class.

I naively tried to do this with the following code, but it (obviously) didn't work. (I could only imagine what the SQL statement would look like if it did!).

```

_dbContext.Destinations
.Include(destination => destination.ServiceSpecs)
.Include(destination => destination.ServiceSpecs.OfType<ServiceSpec_PhysicalAudio>
().Select(serviceSpec => serviceSpec.Format))
.Include(destination => destination.ServiceSpecs.OfType<ServiceSpec_PhysicalSDVideo>
().Select(serviceSpec => serviceSpec.Format))
.Include(destination => destination.ServiceSpecs.OfType<ServiceSpec_PhysicalSDVideo>
().Select(serviceSpec => serviceSpec.Standard))
.Include(destination => destination.ServiceSpecs.OfType<ServiceSpec_PhysicalHDVideo>
().Select(serviceSpec => serviceSpec.Format))
.Include(destination => destination.ServiceSpecs.OfType<ServiceSpec_PhysicalHDVideo>
().Select(serviceSpec => serviceSpec.Standard))
.Include(destination => destination.ServiceSpecs.OfType<ServiceSpec_PhysicalHDVideo>
().Select(serviceSpec => serviceSpec.FrameRate))
.Include(destination => destination.ServiceSpecs.OfType<ServiceSpec_Digital>().Select(serviceSpec =>
serviceSpec.Service));

```

Is there any way to force the loading of references and collections of subtypes of a polymorphic type during query execution? If not, what's the best workaround? Load them in `_ObjectContext_ObjectMaterialized()`?

In general, in many/most cases I want to load my full object graph when I load an entity. I wish EF had a context-level switch that forced full load, or an alternate opt-out - instead of opt-in (`Include()`) - model to control object graph loading.

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Thursday, March 24, 2011 12:10 PM by [morteza](#)

@David: You should be aware that the lambda expression in the new `Include` method overload is merely a property selector and you can't have any sort of filtering logic in it. In fact, you cannot have any filtering/ordering logic when eager loading navigation properties with `Include` method in general. There are 2 solutions that you can apply in this scenario; you can either use anonymous projections:

```
var query = (from d in context.Destinations where d.DestinationId == 1
            select new
            {
                Destination = d,
                ServiceSpec_PhysicalAudio = d.ServiceSpecs.OfType<ServiceSpec_PhysicalAudio>(),
                ServiceSpec_PhysicalSDVideo = d.ServiceSpecs.OfType<ServiceSpec_PhysicalSDVideo>(),
                ServiceSpec_PhysicalHDVideo = d.ServiceSpecs.OfType<ServiceSpec_PhysicalHDVideo>(),
            })
            .Select(d => d.Destination);
```

Or you can use the new `Query` method defined on `DbCollectionEntry` class which is not eager loading anymore which means you will have 2 round trips to the database:

```
Destination destination = context.Destinations.Single(d => d.DestinationId == 1);
context.Entry(destination)
    .Collection(d => d.ServiceSpecs)
    .Query()
    .Where(s => s is ServiceSpec_PhysicalAudio
              || s is ServiceSpec_PhysicalAudio
              || s is ServiceSpec_PhysicalSDVideo
              || s is ServiceSpec_PhysicalHDVideo)
    .Load();
```

Hope this helps.

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Monday, March 28, 2011 1:57 PM by [Vincent-Philippe Lauzon](#)

Excellent blog post!

It's the only place I found information about table inheritance working with the RC API.

Thanks a lot for the content!

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Thursday, March 31, 2011 10:36 AM by Tony

Has the performance issue with creating the crazy number of joins and unions been solved yet?

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Thursday, April 07, 2011 10:26 AM by [morteza](#)

@Tony: Like I described in the post, it's not an issue, it's just how TPT works by nature in essence that queries always require a join across many tables. Even other ORM frameworks generate pretty much the same SQL query when it comes to polymorphic associations. What's new in EF 4.1 RC is that now you can mix inheritance mapping strategies in your hierarchy to come up with the best performance possible.

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Wednesday, April 13, 2011 12:05 AM by von

I just noticed a strange thing again with the RC version. The following issue was working with CTP5.

If I have the configurations below I would get an Order table with the following columns:

- Id
- DateOrdered
- OrderedById
- ApprovedById
- Person_Id (this one is an extra column created by EF)

If I remove the ApprovedBy fields in the Order class. The Order table will be like this:

- Id
- DateOrdered
- OrderedById

The extra Person_Id went away.

```
public class Order
{
    public Guid Id { get; set; }
    public DateTime DateOrdered { get; set; }
    public Person OrderedBy { get; set; }
    public Guid OrderedById { get; set; }
    public Person ApprovedBy { get; set; }
    public Guid? ApprovedById { get; set; }
}

public class Person
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Order> Orders { get; set; }
}

public class MyContext : DbContext
{
    public DbSet<Order> Orders { get; set; }
    public DbSet<Person> People { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Order>().ToTable("Orders");
        modelBuilder.Entity<Order>().HasRequired(o => o.OrderedBy);
        modelBuilder.Entity<Order>().HasOptional(o => o.ApprovedBy);
        modelBuilder.Entity<Person>().ToTable("People");
        modelBuilder.Entity<Person>().HasMany(p => p.Orders);
    }
}
```

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Wednesday, April 13, 2011 11:50 AM by [morteza](#)

@von: This one is a bit tricky. When you have both the `ApprovedBy` and `OrderedBy` properties as well as the `Orders` on the `Person` entity, you virtually defining three associations between `Order` and `Person` entities:

1. `(Person) 1 – * (Order)` via `Order.ApprovedBy` and `ApprovedById`
2. `(Person) 1 – * (Order)` via `Order.OrderedBy` and `OrderedById`
3. `(Person) 1 – * (Order)` via `Person.Orders` (an *Independent Association* inferred by Code First, hence the `Person_Id` column on `Orders` table)

The reason for that is because you did not specify `Person.Orders` to be the *InverseProperty* for any of the other two associations so Code First creates a third one for you. However, when you delete the first association by removing the `ApprovedBy` property, you left with one association via `OrderedBy`. Now based on conventions, Code First will infer `Person.Orders` as the other end for this association instead of creating a new one since there is only one association exists and Code First can safely assume `Person.Orders` is the navigation property for this association on the `Person` end, something that was not possible in the previous scenario. Hope this helps.

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Wednesday, May 18, 2011 10:52 AM by von

This is not directly related to the title. But I found this place to be very helpful so let me ask here.

I have this:

```
public class Person
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Order> Orders { get; set; }
}
```

If I do: `var person = context.People.Find(1)`; the `Orders` collection will be lazy loaded which is what I want. However, when it's time to iterate through the `Orders` I want a behavior such as "SELECT ALL `Orders`". But a "SELECT " is happening for EACH of the item in the `Orders` collection. Here is my code:

```
IEnumerable<Order> orders = from o in person.Orders
```

```
select o;
```

If I do the following the same behavior happen: `IEnumerable<Order> orders = person.Orders`;

So what can I do so that all `Orders` will be queried at one time? I know I can do this: `var orders = context.Orders.Find(o=>o.OrderedBy.Id == person.Id)`. I was thinking if there is a better way to do it.

* please look at my previous comments for complete reference on Entities and Context (if needed)

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Saturday, May 21, 2011 4:23 PM by [morteza](#)

@von: Your code is perfectly fine and it always cause one and only one SQL query being submitted to the database to bring back all the related orders. I think you have something else going on in there, my wild guess is that you are iterating through the `Orders` collection, let's say in a for each loop (which cause EF to lazy load the `Orders`) and then access an unrelated navigation property like `ApprovedBy` in the loop which of course will cause another SQL query to lazy load the `Person` entity if it's not already in the cache. If that's the case then you can be explicit and eager load the target navigation property to avoid this lazy loading behavior. Please let me know if you need help on that. Thanks :)

User entity, which essentially change the association to be a foreign key association, like the following code:

```
public class User
{
    public int UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int BillingDetailId { get; set; }

    public virtual BillingDetail BillingInfo { get; set; }
}
```

re: Inheritance with EF Code First CTP5: Part 2 – Table per Type (TPT)

Wednesday, November 16, 2011 10:33 AM by [maximusmd](#)

Shouldn't you have a BillingDetailId declared in the User class? How are you obtaining the BillingInfo ?

I thought you had to have a foreign key in the User entity. I am confused.... thanks

re: Inheritance with EF Code First: Part 2 – Table per Type (TPT)

Wednesday, November 16, 2011 11:51 AM by [mortezaam](#)

@maximusmd: Your thought is correct; the User entity does have a foreign key to the BillingDetail class, however, explicitly defining that foreign key as a property on the User is just a matter of having a foreign key association instead of an independent association. Having said that, I added the foreign key property to the User class to avoid any confusion. Hope it helps :)

re: Inheritance with EF Code First: Part 2 – Table per Type (TPT)

Friday, December 23, 2011 8:54 AM by [edmondthieffry](#)

Thank you for the clear article.

But I still have a question. I use the technique above to implement subtables/classes that inherit from an abstract(!) base table/class. When I try now to build an MVC view I can't figure out how to design the view model that serves as the base for my view. This view needs to be able to display data from the base class (billingdetail) as well from the derived classes depending on the type of base class chosen in the view (e.g. radiobuttons to choose between bankaccount and creditcard).

I suppose the viewpage would inherit of model X with X defining the TPT designed derived classes. But if I want to display common attributes that are defined in the abstract class, I'm unable to refer directly to the base class properties; I have to duplicate my markup depending on the derived class e.g. bankaccount.owner and creditcard.owner.

If I have a lot of common properties: is there a way not to duplicate and have the view markup refer to the abstract base class properties directly? Imagine having 5 different subclasses: the common data is displayed on top of the view (screen), the specific data follows below that: you really do not want to have to write 5 times the same but slightly different markup for the common data...

Thx for any help.

re: Inheritance with EF Code First: Part 2 – Table per Type (TPT)

Wednesday, January 18, 2012 4:09 PM by [mortezaam](#)

@edmondthieffry: You don't really need to duplicate your common markup related to the base class view in each and every sub class view. You can simply create a Partial View for your abstract base class

(e.g. `_BillingDetailPartial.cshtml`) and then ask the `HtmlHelper` object to render it in each sub class view like the following code:

```
@model Models.CreditCard
@Html.Partial("_BillingDetailPartial", Model)
@*Your markup for Credit Card specific properties...*@
```

[# Inheritance with EF Code First: Part 1 ??? Table per Hierarchy \(TPH\) | My Blog](#)

Wednesday, January 18, 2012 10:20 PM by [Inheritance with EF Code First: Part 1 ??? Table per Hierarchy \(TPH\) | My Blog](#)

Pingback from [Inheritance with EF Code First: Part 1 ??? Table per Hierarchy \(TPH\) | My Blog](#)

[Terms of Use](#)

Inheritance with EF Code First: Part 3 – Table per Concrete Type (TPC)

This is the third (and last) post in a series that explains different approaches to map an inheritance I First. I've described these strategies in previous posts:

- [Part 1 – Table per Hierarchy \(TPH\)](#)
- [Part 2 – Table per Type \(TPT\)](#)

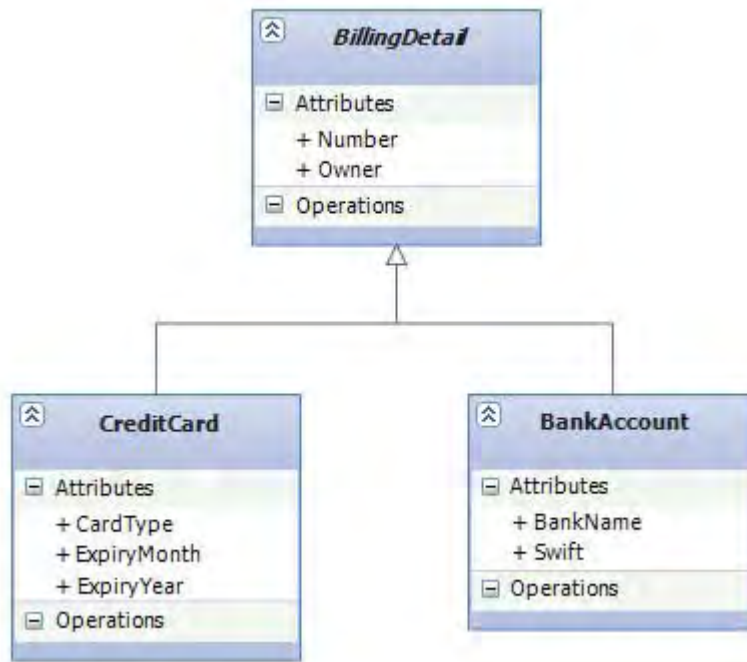
In today's blog post I am going to discuss Table per Concrete Type (TPC) which completes the inhe supported by EF Code First. At the end of this post I will provide some guidelines to choose an inhe based on what we've learned in this series.

TPC and Entity Framework in the Past

Table per Concrete type is somehow the simplest approach suggested, yet using TPC with EF is on has not been covered very well so far and I've seen in some resources that it was even discouragel just because Entity Data Model Designer in VS2010 doesn't support TPC (even though the EF runti means if you are following EF's Database-First or Model-First approaches then configuring TPC req XML in the EDMX file which is not considered to be a fun practice. Well, no more. You'll see that wit is perfectly possible with fluent API just like other strategies and you don't need to avoid TPC due tc support as you would probably do in other EF approaches.

Table per Concrete Type (TPC)

In Table per Concrete type (aka Table per Concrete class) we use exactly one table for each (nonat properties of a class, including inherited properties, can be mapped to columns of this table, as sho



| Column Name | Identity |
|-----------------|-------------------------------------|
| BillingDetailId | <input checked="" type="checkbox"/> |
| Owner | <input type="checkbox"/> |
| Number | <input type="checkbox"/> |
| BankName | <input type="checkbox"/> |
| Swift | <input type="checkbox"/> |

| Column Name | Identity |
|-----------------|-------------------------------------|
| BillingDetailId | <input checked="" type="checkbox"/> |
| Owner | <input type="checkbox"/> |
| Number | <input type="checkbox"/> |
| CardType | <input type="checkbox"/> |
| ExpiryMonth | <input type="checkbox"/> |
| ExpiryYear | <input type="checkbox"/> |

As you can see, the SQL schema is not aware of the inheritance; effectively, we've mapped two unexpressive class structure. If the base class was concrete, then an additional table would be needed class. I have to emphasize that there is no relationship between the database tables, except for the similar columns.

TPC Implementation in Code First

Just like the [TPT implementation](#), we need to specify a separate table for each of the subclasses. We first that we want all of the inherited properties to be mapped as part of this table. In CTP5, there is EntityMappingConfiguration class called MapInheritedProperties that exactly does this for u object model as well as the fluent API to create a TPC mapping:

```
public abstract class BillingDetail
{
    public int BillingDetailId { get; set; }
    public string Owner { get; set; }
    public string Number { get; set; }
}

public class BankAccount : BillingDetail
{
    public string BankName { get; set; }
    public string Swift { get; set; }
}

public class CreditCard : BillingDetail
{
    public int CardType { get; set; }
    public string ExpiryMonth { get; set; }
    public string ExpiryYear { get; set; }
}

public class InheritanceMappingContext : DbContext
{
    public DbSet<BillingDetail> BillingDetails { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<BankAccount>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("BankAccounts");
        });

        modelBuilder.Entity<CreditCard>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("CreditCards");
        });
    }
}
```

}

The Importance of EntityMappingConfiguration Class

As a side note, it worth mentioning that EntityMappingConfiguration class turns out to be a mapping in Code First. Here is an snapshot of this class:

```
namespace System.Data.Entity.ModelConfiguration.Configuration.Mapping
{
    public class EntityMappingConfiguration<TEntityType> where TEntityType :
    {
        public ValueConditionConfiguration Requires(string discriminator);
        public void ToTable(string tableName);
        public void MapInheritedProperties();
    }
}
```

As you have seen so far, we used its Requires method to [customize TPH](#). We also used its ToTable method to [create a TPT](#) and now we are using its MapInheritedProperties along with ToTable method to create

TPC Configuration is Not Done Yet!

We are not quite done with our TPC configuration and there is more into this story even though the database has perfectly created a TPC mapping for us in the database. To see why, let's start working with our objects. The following code creates two new objects of BankAccount and CreditCard types and *tries* to add them to the database.

```
using (var context = new InheritanceMappingContext())
{
    BankAccount bankAccount = new BankAccount();
    CreditCard creditCard = new CreditCard() { CardType = 1 };

    context.BillingDetails.Add(bankAccount);
    context.BillingDetails.Add(creditCard);

    context.SaveChanges();
}
```

Running this code throws an [InvalidOperationException](#) with this message:

The changes to the database were committed successfully, but an error occurred while updating the database. The ObjectContext might be in an inconsistent state. Inner exception message: AcceptChanges failed because the object's key values conflict with another object in the ObjectStateManager. Make sure key values are unique before calling AcceptChanges.

The reason we got this exception is because DbContext.SaveChanges() internally invokes SaveChanges() on the internalObjectContext. ObjectContext's SaveChanges method on its turn by default calls AcceptChanges() to perform the database modifications. AcceptChanges method merely iterates over all entries in the context and invokes AcceptChanges on each of them. Since the entities are in Added state, AcceptChanges creates a temporary EntityKey with a regular EntityKey based on the primary key values (i.e. BillingDetailId)

database and that's where the problem occurs since both the entities have been assigned the same key by the database (i.e. on both `BillingDetailId = 1`) and the problem is that `ObjectStateManager` has the same type (i.e. `BillingDetail`) with the same `EntityKey` value hence it throws. If you take a closer look at the schema above, you'll see why the database generated the same values for the primary keys: the `BankAccounts` and `CreditCards` table has been marked as identity.

How to Solve The Identity Problem in TPC

As you saw, using SQL Server's int identity columns doesn't work very well together with TPC since all entity keys when inserting in subclasses tables with all having the same identity seed. Therefore, a mechanism other than `Sequence` (where each table has its own initial seed value) will be needed, or a mechanism other than `Sequence` should be used. Some other RDBMSes have other mechanisms allowing a sequence (identity) to be used on multiple tables, and something similar can be achieved with GUID keys in SQL Server. While using GUID keys with different starting seeds will solve the problem but yet another solution would be to completely switch off the key property. As a result, we need to take the responsibility of providing unique keys when inserting. We will go with this solution since it works regardless of which database engine is used.

Switching Off Identity in Code First

We can switch off identity simply by placing `DatabaseGenerated` attribute on the primary key property and setting `DatabaseGenerationOption.None` to its constructor. `DatabaseGenerated` attribute is a new data annotation added to [System.ComponentModel.DataAnnotations](#) namespace in CTP5:

```
public abstract class BillingDetail
{
    [DatabaseGenerated(DatabaseGenerationOption.None)]
    public int BillingDetailId { get; set; }
    public string Owner { get; set; }
    public string Number { get; set; }
}
```

As always, we can achieve the same result by using fluent API, if you prefer that:

```
modelBuilder.Entity<BillingDetail>()
    .Property(p => p.BillingDetailId)
    .HasDatabaseGenerationOption(DatabaseGenerationOption.None);
```

Working With The Object Model

Our TPC mapping is ready and we can try adding new records to the database. But, like I said, now we need to provide unique keys when creating new objects:

```
using (var context = new InheritanceMappingContext())
{
    BankAccount bankAccount = new BankAccount()
    {
        BillingDetailId = 1
    };
    CreditCard creditCard = new CreditCard()
    {
```

```
        BillingDetailId = 2,  
        CardType = 1  
    };  
  
    context.BillingDetails.Add(bankAccount);  
    context.BillingDetails.Add(creditCard);  
  
    context.SaveChanges();  
}
```

Polymorphic Associations with TPC is Problematic

The main problem with this approach is that it doesn't support [Polymorphic Associations](#) very well. / associations are represented as foreign key relationships and in TPC, the subclasses are all mapped to their base class (abstract `BillingDetail` in our example) cannot be represented as a polymorphic association to their base class (abstract `BillingDetail` in our example) cannot be represented as a key relationship. For example, consider the domain model we introduced [here](#) where `User` has a polymorphic association to `BillingDetail`. This would be problematic in our TPC Schema, because if `User` has a many-to-one relationship to `BillingDetail`, the `Users` table would need a single foreign key column, which would have to refer to both tables. This isn't possible with regular foreign key constraints.

Schema Evolution with TPC is Complex

A further conceptual problem with this mapping strategy is that several different columns, of different semantics. This makes schema evolution more complex. For example, a change to a base class changes to multiple columns. It also makes it much more difficult to implement database integrity constraints on subclasses.

Generated SQL

Let's examine SQL output for polymorphic queries in TPC mapping. For example, consider this polymorphic query on `BillingDetails` and the resulting SQL statements that being executed in the database:

```
var query = from b in context.BillingDetails select b;
```



```

SELECT
CASE WHEN ([UnionAll1].[C4] = 1) THEN 'OXOX' ELSE 'OX1X' END AS
[UnionAll1].[BillingDetailId] AS [C2],
[UnionAll1].[Owner] AS [C3],
[UnionAll1].[Number] AS [C4],
CASE WHEN ([UnionAll1].[C4] = 1) THEN [UnionAll1].[BankName] END
CASE WHEN ([UnionAll1].[C4] = 1) THEN [UnionAll1].[Swift] END AS
CASE WHEN ([UnionAll1].[C4] = 1) THEN
CAST(NULL AS int) ELSE [UnionAll1].[C1] END AS [C7],
CASE WHEN ([UnionAll1].[C4] = 1) THEN
CAST(NULL AS varchar(1)) ELSE [UnionAll1].[C2] END AS [C8],
CASE WHEN ([UnionAll1].[C4] = 1) THEN
CAST(NULL AS varchar(1)) ELSE [UnionAll1].[C3] END AS [C9]
FROM
(SELECT
[Extent1].[BillingDetailId] AS [BillingDetailId],
[Extent1].[Owner] AS [Owner],
[Extent1].[Number] AS [Number],
[Extent1].[BankName] AS [BankName],
[Extent1].[Swift] AS [Swift],
CAST(NULL AS int) AS [C1],
CAST(NULL AS varchar(1)) AS [C2],
CAST(NULL AS varchar(1)) AS [C3],
cast(1 as bit) AS [C4]
FROM [dbo].[BankAccounts] AS [Extent1]
UNION ALL
SELECT
[Extent2].[BillingDetailId] AS [BillingDetailId],
[Extent2].[Owner] AS [Owner],
[Extent2].[Number] AS [Number],
CAST(NULL AS varchar(1)) AS [C1],
CAST(NULL AS varchar(1)) AS [C2],
[Extent2].[CardType] AS [CardType],
[Extent2].[ExpiryMonth] AS [ExpiryMonth],
[Extent2].[ExpiryYear] AS [ExpiryYear],
cast(0 as bit) AS [C3]
FROM [dbo].[CreditCards] AS [Extent2])
AS [UnionAll1]
    
```

Results

| | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 |
|---|------|----|---------|-----------|------|-------------|------|------|------|
| 1 | OXOX | 1 | Morteza | 987654321 | CIBC | CORPIBB3034 | NULL | NULL | NULL |
| 2 | OX1X | 2 | Morteza | 987654321 | NULL | NULL | 1 | 10 | 12 |

Just like the [SQL query generated](#) by TPT mapping, the [CASE](#) statements that you see in the begin to ensure columns that are irrelevant for a particular row have NULL values in the returning flattened for a row that represents a CreditCard type).

TPC's SQL Queries are Union Based

As you can see in the above screenshot, the first SELECT uses a FROM-clause subquery (which is rectangle) to retrieve all instances of BillingDetails from all concrete class tables. The tables are correlated by a join operator, and a literal (in this case, 0 and 1) is inserted into the intermediate result; (look at the lines reads this to instantiate the correct class given the data from a particular row. A union requires that combined, project over the same columns; hence, EF has to pad and fill up nonexistent columns will really perform well since here we can let the database optimizer find the best execution plan to compare tables. There is also no Joins involved so it has a better performance than the SQL queries generated required between the base and subclasses tables.

Choosing Strategy Guidelines

Before we get into this discussion, I want to emphasize that there is no one single "best strategy fits you saw, each of the approaches have their own advantages and drawbacks. Here are some rules of best strategy in a particular scenario:

- If you don't require polymorphic associations or queries, lean toward **TPC**—in other words, if for BillingDetails and you have no class that has an association to BillingDetail base class. If at the [top level of your class hierarchy](#), where polymorphism isn't usually required, and when more class in the future is unlikely.
- If you do require polymorphic associations or queries, and subclasses declare relatively few properties (the main difference between subclasses is in their behavior), lean toward **TPH**. Your goal is to minimize nullable columns and to convince yourself (and your DBA) that a denormalized schema won't last long run.
- If you do require polymorphic associations or queries, and subclasses declare many properties (mainly by the data they hold), lean toward **TPT**. Or, depending on the width and depth of your data, consider the possible cost of joins versus unions, use **TPC**.

By default, choose TPH only for simple problems. For more complex cases (or when you're overruling and insisting on the importance of nullability constraints and normalization), you should consider the TPT point, ask yourself whether it may not be better to remodel inheritance as delegation in the object model (of making composition as powerful for reuse as inheritance). Complex inheritance is often best avoided unrelated to persistence or ORM. EF acts as a buffer between the domain and relational models, but can ignore persistence concerns when designing your classes.


Summary

In this series, we focused on one of the main structural aspect of the object/relational paradigm mismatch and discussed how EF solve this problem as an ORM solution. We learned about the three well-known strategies and their implementations in EF Code First. Hopefully it gives you a better insight about the hierarchies as well as choosing the best strategy for your particular scenario.

Happy New Year and Happy Code-Firsting!

References

- [ADO.NET team blog](#)
- [Java Persistence with Hibernate book](#)

Published Monday, January 03, 2011 12:32 AM by [morteza](#) 
Filed under: [Entity Framework](#), [C#](#), [Code First](#), [CTP5](#), [.NET](#)

Comments

re: Inheritance Mapping Strategies with Entity Framework Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Tuesday, January 04, 2011 8:03 AM by Paul

I really enjoyed this series, thanks.

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Wednesday, January 05, 2011 1:51 PM by Dimitris Foukas

Excellent blog series - I am recommending this to my colleagues!

I wonder if the limitation of no polymorphic associations with TPC is specific to CTP5 or is inherent to code first...

AFAIK EF core, on which code first is based upon, supports polymorphic associations to base classes in TPC mapping scenarios.

Nevertheless my simple code experiment confirms your diagnosis!

See also the following forum post where I tried to remedy this shortcoming:

social.msdn.microsoft.com/.../27418a0d-897f-442d-ba65-5e8a50700082

Thanks,

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Thursday, January 06, 2011 10:10 AM by [morteza](#)

@Dimitris Foukas: Great question! Let me clarify it. When talking about polymorphic associations in TPC, we need to consider 2 different scenarios in terms of multiplicities. For example in our example, we are not able to create a *one-to-many* association from User to BillingDetail because then both subclass tables would need a foreign key reference to the User table and EF does not natively support this scenario. So this is not possible:

```
public class User
{
    public int UserId { get; set; }
    public virtual ICollection<BillingDetail> BillingDetails { get; set; }
}

public abstract class BillingDetail
{
    public int BillingDetailId { get; set; }
    public virtual User User { get; set; }
}
```

}

That said, the only way to create a polymorphic association in a TPC mapping like this is to create a *many-to-one* association from User to BillingDetail which means the Users table would need a single foreign key column which would have to refer both concrete subclass tables:

```
public class User
{
    public int UserId { get; set; }
    public virtual BillingDetail BillingInfo { get; set; }
}

public abstract class BillingDetail
{
    public int BillingDetailId { get; set; }
    public virtual ICollection<User> Users { get; set; }
}
```

However, this mapping throws an exception due to a bug in CTP5. In fact, I wanted to illustrate this many-to-one association as the way we can implement polymorphic associations in TPC but then I noticed this bug and decided not to do it until it goes away in the RTM. That's why the title reads as "*Polymorphic Associations with TPC are Problematic*" and not impossible.

In a nutshell, a *one-to-many* association is not possible but *many-to-one* associations *will* work. In other words, in TPC you always need to keep the foreign key outside of the inheritance hierarchy. Hope this helps, thanks :)

[# re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type \(TPC\)](#)

Thursday, January 06, 2011 6:00 PM by Dimitris

Crystal clear!

You should really evolve this blog series into a Code first cookbook - there is a window of opportunity there IMO...

[# re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type \(TPC\)](#)

Friday, January 07, 2011 5:47 PM by Jan Klima

All your posts in this blog so far are simply great, thanks...

[# Наследование в EF Code First CTP5: часть 3 – таблица для каждого типа \(TPC\)](#)

Monday, January 10, 2011 7:50 AM by progg.ru

Thank you for submitting this cool story - Trackback from progg.ru

[# re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type \(TPC\)](#)

Tuesday, January 11, 2011 9:37 PM by mortezam

@Dimitris: That is actually a very good idea especially because none of the current EF books cover Code First development. I will think about it, thanks!

[# re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type \(TPC\)](#)

Wednesday, January 12, 2011 11:56 PM by Christof

Hi,

I don't get it to work with a second level of inheritance

```

    A
  / \
 AA  AB
 / \
ABA  ABB

```

A and AB are abstract, the rest concrete.

```

public abstract class A
{
    public A()
    {
        Id = Guid.NewGuid();
    }
    [DatabaseGenerated(DatabaseGenerationOption.None)]
    public Guid Id { get; set; }
}

public class AA : A
{
    public string aa { get; set; }
}

public abstract class AB : A
{
    public string ab { get; set; }
}

public class ABA : AB
{
    public string aba { get; set; }
}

public class ABB : AB
{
    public string abb { get; set; }
}

```

I get an NullReferenceException the moment I add something to the DbSet<A> collection in the context

Christof

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Thursday, January 13, 2011 10:35 PM by [mortezam](#)

@Christof: This is a bug in CTP5, your multi-level hierarchy is designed to work in Code First and hopefully EF team will make sure it is fixed for the RTM. Until then, one possible workaround would be to use TPT mapping instead of TPC.

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Wednesday, February 16, 2011 4:43 AM by Guest

Help me please create context class for this model:

```

public abstract class Publication
{
    public int Id { get; set; }
}

```

```

    public ICollection<CoAuthor> CoAuthors { get; set; }
}

public class JournalArticle : Publication
{
    public int Id { get; set; }
    public string ArticleName { get; set; }
}

public class CoAuthor
{
    public int Id { get; set; }
    public int EmployeeId { get; set; }
}

```

I have:

```

public class ModelContext : DbContext
{
    public DbSet<Publication> Publications { get; set; }
    public DbSet<CoAuthor> CoAuthors { get; set; }
    protected override void OnModelCreating(ModelBuilder builder)
    {
        builder.Entity<JournalArticle>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("JournalArticle", "Science");
        });
        builder.Entity<CoAuthor>().ToTable("CoAuthor", "Science");
    }
}

```

But I give error "Schema specified is not valid. Errors: (35,6) : error 0040: Type Publication is not defined in namespace CodeFirstDatabaseSchema (Alias=Self). (64,8) : error 0100: The referenced EntitySet Publication for End Publication could not be found in the containing EntityContainer."

Thank you

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Wednesday, February 16, 2011 10:37 AM by [morteza](#)

@Guest: This is a bug in CTP5. Basically polymorphic associations like the one you are trying to setup between Publication and CoAuthor throws an exception when using TPC strategy. Please read my answer to *Dimitris Foukas* above where I discussed this issue in detail. The workaround for now is to avoid polymorphic associations when using TPC or choosing another strategy like [Table Per Type \(TPT\)](#) for this scenario. Hope this helps.

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Monday, February 28, 2011 2:05 PM by minux

I wonder if EF could support a scenario where additional properties are saved serialized in a xml column. This is usefull when additional properties are not directly queried.

If no directly, would it be possible with a type that has a complex type property

```

class A {
    public int Id {get;set;}
    public string Name {get;set;}
    public Bag CustomProperties {get;set;} // option where A is not an abstract class
}

```

[ComplexType]

```
public abstract class Bag { }
public class B1 : Bag { // could be : A
    public string Prop1 {get;set}
}
public class B2 : Bag { // could be : A
    public string Prop2 {get;set;}
}
new A { Name = "foo", CustomProperties = new B2 { Prop2 = "bar" } }
```

or

```
new B2 { Name = "foo", Prop2 = "bar" }
```

could be persisted as

```
('foo', 'B2', '<root><Prop2>bar</Prop2></root>')
```

Any clue ? I know it's a scenario not very often supported but it has some advantage.

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Tuesday, March 22, 2011 5:36 AM by [ali akbar](#)

Hi morteza

I am from iran ,

I have question about TPC inheritancy

I have three table Category,Product,DisContProduct

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

public class Category
{
    public int CategoryId { get; set; }
    public string Name { get; set; }
    public ICollection Products { get; set; }
}

public class DiscontionuedProduct : Product
{
    public DateTime DiscontinuedOn { get; set; }
}
```

after run

```
/*Table Per Concrete Type*/
```

```
modelBuilder.Entity<Category>();
```

```
modelBuilder.Entity<Product>().ToTable("Products");
```

```
modelBuilder.Entity<DiscontionuedProduct>().Map(mc => { mc.MapInheritedProperties();
}).ToTable("DiscontionuedProducts");
```

```
//Table Products Id,Name,CategoryCategoryId
```

```
//Table DiscontionuedProducts Id,Name,DiscontinuedOn but Dont Create CategoryCategoryId??? why
```

Why ?????

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Tuesday, March 22, 2011 11:39 AM by [morteza](#)

@ali akbar: It seems that you are still using CTP5 assembly, if so, then please do upgrade it to [EF 4.1 RC](#) which is the latest EF release. After that have a look at my discussion with *Dimitris Foukas* above which is around the same question. Basically you cannot create a polymorphic many-to-one association from Product to Category when using TPC mapping. There are a couple of ways that you can work out this limitation though:

1. You can make Product class *abstract* and keep the association the same way it is.
2. You can change the association to be a one-to-many from Product to Category by defining a Product navigation property in Category class which essentially means keeping the FK column out of the inheritance hierarchy.
3. And Finally you can use [TPT strategy](#) instead which is the recommended strategy when polymorphic associations are involved.

Having said all that, Your Product Category object model issue would be best resolved by the third solution but you may want to consider other ways for your different future scenarios. Hope this helps :)

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Friday, March 25, 2011 2:57 PM by Sanjay

Hi,

I am trying to do TPC (Table Per Concrete Type) mapping in EF4 - CTP 5. I have three level of hierarchy : type C inherits from type B, which in turn inherits from type A. All classes are concrete and there is one table per type in database. When I try to map it, I am getting error 'Invalid column name 'Discriminator''. Any ideas, what I might be missing? (I have two level inheritance TPC working, I think 3 level inheritance is causing this problem

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Saturday, March 26, 2011 7:41 PM by [morteza](#)

@Sanjay: This is a bug in CTP5 and the best workaround for this would be upgrading to [EF 4.1 RC](#) :) Please let me know if you have any difficulty in mapping your inheritance hierarchy with EF 4.1. Thanks.

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Monday, March 28, 2011 5:23 AM by [ali akbar](#)

thanks for answer to my question.

i have some another problem

1)i want to update one of properties of entity (i wont to load first & change property & update

i want Update One Filed)how id do it by entityframework

2)how can id set the entity that[one property is readonly] (i dont update filed like createDate prop)

thanks .

sorry for bad english writing

TPH Inheritance mapping in EF 4.1 with Code First & Nathan's blog

Sunday, April 10, 2011 10:53 AM by [TPH Inheritance mapping in EF 4.1 with Code First « Nathan's blog](#)

Pingback from TPH Inheritance mapping in EF 4.1 with Code First « Nathan's blog

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Tuesday, April 19, 2011 3:07 PM by [Lee Dumond](#)

Nice tutorial, very clear and easy to follow. However, when I attempt to recreate your sample, I get the following error:

Invalid object name 'dbo.BillingDetails'.

It seems to be wanting a BillingDetails table in the database, but of course there isn't supposed to be one.

Did something change since the RTW release? I'm using EF 4.1.10331.0 here.

Entity Framework 4.1: Inheritance (7) « Vincent-Philippe Lauzon's blog

Tuesday, April 19, 2011 8:36 PM by [Entity Framework 4.1: Inheritance \(7\) « Vincent-Philippe Lauzon's blog](#)

Pingback from Entity Framework 4.1: Inheritance (7) « Vincent-Philippe Lauzon's blog

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Wednesday, April 20, 2011 9:40 PM by [morteza](#)

@Lee Dumond: It's a bug in EF 4.1 RTW and I've just [confirmed](#) this with the EF team. Here is the comment from *Diego Vega*, a program manager on the EF team regarding this issue:

“What you are describing seems to match an issue we found very late in EF 4.1 RTW cycle and we decided to postpone because our evaluation indicated that it had a very low impact: while this unnecessary table is created for an abstract base type, the table does not participate in any mapping, therefore it shouldn't appear in any query produced by EF. For cases in which you are using Code First to generate the database, the table will be added to the database schema but will never be used. For cases in which you are using the Code First API to map to an existing database it shouldn't have an impact either, since no queries generated by EF will ever fail as a result of the table not being there.”

And you are correct, the RC and CTP5 build of the Code First library was working just fine in essence that they never create a table for an abstract base class in the inheritance hierarchy like BillingDetail when creating a TPC mapping.

By the way, Diego also mentioned that he is not aware of a workaround that just removes the table without changing the shape of the model. Therefore, it means that the workaround for now is to just manually remove the BillingDetails table after Code First generates the database for you. If you are using the Code First API to map to an existing database, then just don't create a table for the BillingDetail class in your database and Code First will still work with it without any problem. Please let me know if you have any questions or if you see any other behavior or higher impact. Thanks very much for bringing this important issue to my attention :)

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Thursday, April 21, 2011 1:10 PM by [Lee Dumond](#)

Yes... the issue is that I am using the CodeFirst API against an existing database, so there is no database generation happening here, and no BillingDetails table in the existing database at all.

The error is complaining about exactly that -- that there is no BillingDetails table in the existing database. So, it appears that the workaround you suggest (to not create a BillingDetails table) would not apply.

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Thursday, April 21, 2011 1:38 PM by [morteza](#)

@Lee Dumond: Thanks for the update; however, I couldn't reproduce the exception that you are getting, any chance that you could send your sample that throws when the table is not there to my email bmanavi@gmail.com?

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Friday, April 22, 2011 5:25 PM by [Lee Dumond](#)

I may have deleted it. Let me see if I can recreate it again...

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Sunday, April 24, 2011 8:51 PM by [Lee Dumond](#)

I see you answered my inquiry over at MSDN. Turns out you and Diego were correct after all. Thanks for all your help.

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Friday, April 29, 2011 2:31 PM by Rafael - [@rsantosdev](#)

Hi my friend!

First of all I would like to congratulate you for your great post series about EF!

I read your post about TPC and was wondering why you not implement a DbSet for concrete classes and not for the abstract class?!

I did a test and seems to work good to me.

```
namespace TablePerConcreteClass
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var context = new InheritanceMappingContext())
            {
                var bankAccount = new BankAccount { BankName = "Banco do Brasil" };
                var creditCard = new CreditCard { CardType = 1 };

                context.BankAccounts.Add(bankAccount);
                context.CreditCards.Add(creditCard);

                context.SaveChanges();
            }
        }
    }

    public abstract class BillingDetail
    {
        public int Id { get; set; }
        public string Owner { get; set; }
        public string Number { get; set; }
    }

    public class BankAccount : BillingDetail
    {
        public string BankName { get; set; }
        public string Swift { get; set; }
    }

    public class CreditCard : BillingDetail
```

```

{
    public int CardType { get; set; }
    public string ExpiryMonth { get; set; }
    public string ExpiryYear { get; set; }
}

public class InheritanceMappingContext : DbContext
{
    public DbSet<BankAccount> BankAccounts { get; set; }
    public DbSet<CreditCard> CreditCards { get; set; }
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<BankAccount>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("BankAccounts");
        });

        modelBuilder.Entity<CreditCard>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("CreditCards");
        });
    }
}

```

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Friday, April 29, 2011 6:58 PM by [morteza](#)

@Rafael: The only downside of defining DbSetes for the subclasses instead of the base class is that you won't be able to run polymorphic queries against your object model anymore, something that would have achieved by a query like context.BillingDetails.ToList(). That being said, if you are using TPC for the top level of your class hierarchy, where polymorphism isn't usually required then defining DbSetes with subclasses is the way to go. In fact, this way you don't even need to drop down to fluent API to specify a TPC mapping, Code First will use TPC to map your classes by default. So in a nutshell, it'll work both ways and which one to pick depends on your use case. Again if the subclasses are unrelated and merely share a set of common properties (e.g. CreatedBy, UpdatedDate, etc.) then the code you posted is absolutely fine and recommended. Thanks and hope this helps.

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Tuesday, June 14, 2011 9:56 AM by Nico Denzl

Excellent blog post series. Thanks

polymorphic association and splitting tables - Programmers Goodies

Thursday, July 14, 2011 12:41 AM by [polymorphic association and splitting tables - Programmers Goodies](#)

Pingback from [polymorphic association and splitting tables - Programmers Goodies](#)

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Monday, August 01, 2011 4:50 PM by Britto

Hi Morteza,

This is very nice article and have been helpful for solving lot of issues. However, my requirements are little more complex. I have an additional level in the Hierarchy.

```

public abstract class EntityBase {
    [DatabaseGenerated(System.ComponentModel.DataAnnotations.DatabaseGeneratedOption.Identity)]
    public virtual long Id { get; set; }
    public virtual string CreatedBy { get; set; }
    public virtual DateTime? CreatedOn { get; set; }
    public virtual string ModifiedBy { get; set; }
    public virtual DateTime? LastModifiedOn { get; set; }
}

public class Person : EntityBase
{
    public virtual string FirstName { get; set; }
    public virtual string LastName { get; set; }
}

public partial class Employee : Person
{
    public string department { get; set; }
    public DateTime? JoiningDate { get; set; }
}

```

I have TPC mapping between [EntityBase and Person] and [Person and Employee]. However, having TPC between Person and Employee duplicates the columns from Person in Employee. Is there a way to avoid this duplication? Am i using the right Mapping strategy?

Another problem is adding an Employee to existing Person. How do I relate the Person and Employee? Do i need a Navigation Property?

Please advice.

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Sunday, August 07, 2011 11:38 AM by [mortezam](#)

@Britto: If you don't want the duplication of the base class columns in the child class table then TPC is not for you in this scenario. You've correctly chosen TPC to map the inheritance between EntityBase and Person entities though; this is the top level of your class hierarchy, where polymorphism isn't really required. I would suggest keeping this TPC and mixing it with a [TPT](#) strategy to map the inheritance between Person and Employee entities to avoid the duplication. The following shows all the fluent API code required to make this happen:

```

public class Context : DbContext
{
    public DbSet<Person> Persons { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>().ToTable("Person");
        modelBuilder.Entity<Employee>().ToTable("Employee");
    }
}

```

(Note how I implicitly create a TPC mapping by defining a DbSet<Person> instead of DbSet<EntityBase>).

Regarding your second question, the only way to change the type of an existing Person to an Employee is to use a store procedure (or a dynamic SQL command) that can be called explicitly from the code. Take a look at [this thread](#) where I answered a similar question a while ago. That said, if you have such a

requirement (a Person can become an Employee at some point) then you might want to reconsider your decision for using Inheritance in the first place (objects usually don't change type). You can collapse your hierarchy and remodel it as a delegation. In other words, you can define a navigation property like Person on Employee class and map it with [one-to-one foreign key associations](#) or [shared primary key associations](#). Hope this helps.

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Wednesday, September 21, 2011 6:08 PM by Mark Phillips

Morteza,

First of all thanks for taking the time to create this series on Code First Inheritance.

I was reviewing the db schema from the TPC tutorial. It appears that EF generates a table for the BillingDetails even though no data is ever inserted into it when a CreditCard or BankAccount object is saved. This seems really strange. Is there some way to avoid this?

Thanks again,

Mark

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Friday, September 23, 2011 10:20 PM by [mortezam](#)

@Mark Phillips: That is a bug in EF 4.1 RTW that unfortunately hasn't been fixed yet (EF 4.1 Update 1 still creates this extra table). Please read my answer to "Lee Dumond" above where I explained this bug in detail. Hope this helps.

Entity Framework 4.1 Code First: Get all Entities with a specific base class - Programmers Goodies

Thursday, September 29, 2011 4:20 PM by [Entity Framework 4.1 Code First: Get all Entities with a specific base class - Programmers Goodies](#)

Pingback from Entity Framework 4.1 Code First: Get all Entities with a specific base class - Programmers Goodies

NHibernate for Entity Framework Developers: Part 1 – Writing and Mapping Classes

Sunday, October 02, 2011 1:28 AM by [Enterprise .Net](#)

The approach to managing persistent data has been a key design decision in every software project we

NHibernate for Entity Framework Developers: Part 1 ??? Writing and Mapping Classes

Sunday, October 02, 2011 11:20 AM by [NHibernate for Entity Framework Developers: Part 1 ??? Writing and Mapping Classes](#)

Pingback from NHibernate for Entity Framework Developers: Part 1 ??? Writing and Mapping Classes

EF Code First – NEWSEQUENTIALID() « Janka J??nos Szakmai Blogja

Monday, October 03, 2011 12:29 AM by [EF Code First – NEWSEQUENTIALID\(\) « Janka J??nos Szakmai Blogja](#)

Pingback from EF Code First – NEWSEQUENTIALID() « Janka J??nos Szakmai Blogja

EF Code First – NEWSEQUENTIALID()

Monday, October 03, 2011 1:20 AM by [Janka János szakmai blogja](#)

Az EF Code First sajnos alapértelmezésbe newid()- et állít be a GUID típusú kulcsoknak, ha azok el vannak

re: Inheritance with EF Code First CTP5: Part 3 – Table per Concrete Type (TPC)

Thursday, November 10, 2011 10:15 AM by Mehdi

It was indeed useful.

[Terms of Use](#)