Отношения между моделями в Fluent API

Последнее обновление: 31.10.2015

Связь один-к нулю или - к одному (One-to–Zero-or-One)

При такой связи для одной модели наличие другой необязательно. Например:

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

        public Company Company { get; set; }
}

public class Company
{
    public int Id { get; set; }
    public string Name { get; set; }

    public Phone BestSeller { get; set; }
}
```

Смартфон обязательно имеет производителя, но производитель может не иметь наиболее продаваемого телефона. То есть в данном случае связь один-к нулю или ко многим. В Fluent API она устанавливается следующим образом:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Phone>()
        .HasRequired(c => c.Company)
        .WithOptional(c => c.BestSeller);

        base.OnModelCreating(modelBuilder);
}
```

Метод **HasRequired()** указывает, что для сущности Phone обязательно должно быть указано навигационное свойство Company. А метод **WithOptional()**, наоборот, устанавливает необязательную связь между объектом предыдущего выражения - Company и его свойством BestSeller.

Связь один-к-одному (One-to-One)

В данной конфигурации уже оба объекта связи должны иметь ссылку друг на друга:

```
modelBuilder.Entity<Phone>()
    .HasRequired(c => c.Company)
    .WithRequiredPrincipal(c => c.BestSeller);
//или так
//modelBuilder.Entity<Company>()
```

```
//    .HasRequired(c => c.BestSeller)
//    .WithRequiredPrincipal(c => c.Company);
```

Метод **WithRequiredPrincipal()** настраивает обязательную связь и устанавливает одну из сущностей в качестве основной. Так, в данном случае основной сущность устанавливается модель Phone: `WithRequiredPrincipal(c => c.BestSeller)`. А таблица, на которую отображается модель Company, будет содержать внешний ключ к таблице Phones.

Связь многие-ко-многим (many-tomany)

Пусть у нас есть ситуация, когда любая из моделей содержит список объектов другой модели. Например, компания может производить несколько телефонов, а над одним телефоном могут работать несколько компаний:

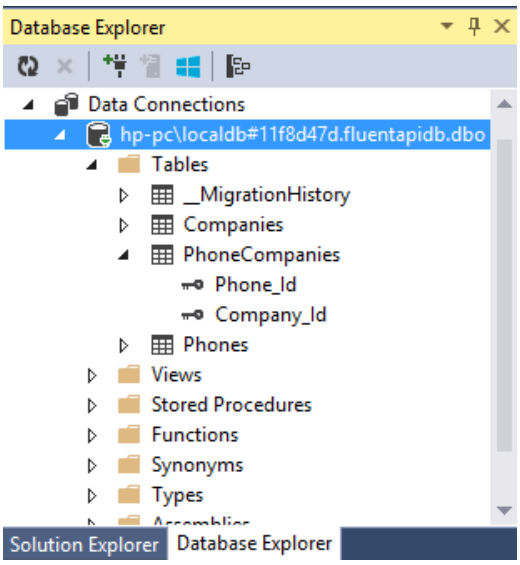
```
public class Phone
{
    public int Id { get; set; }
    public string Name { get; set; }

        public ICollection<Company> Companies { get; set; }

    public Phone()
    {
        Companies = new List<Company>();
    }
}

public class Company
{
    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Phone> Phones { get; set; }

    public Company()
    {
        Phones = new List<Phone>();
    }
}
```

Тогда настройка связи между ними будет выглядеть следующим образом:

```
modelBuilder.Entity<Phone>()
    .HasMany(p => p.Companies)
    .WithMany(c => c.Phones);
```

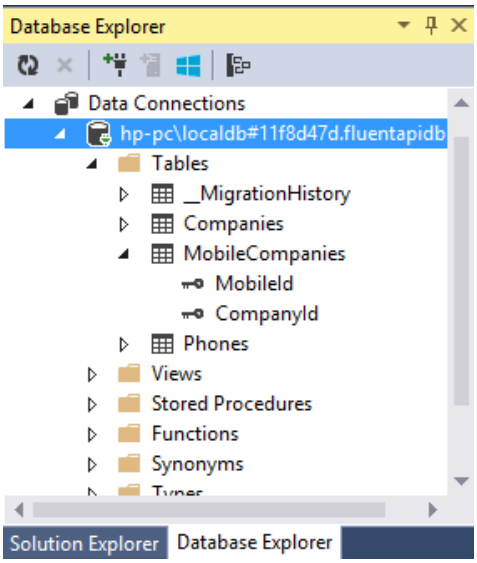
HasMany() устанавливает множественную связь между объектом Phone и объектами Company. А метод **WithMany()** добавляет обратную множественную связь между объектом Company и объектами Phone.

В результате при работе с базой данных будет сформирована третья таблица-посредник между двумя сущностями:



Но нас может не устраивать подобное название таблицы и ее столбцов, и мы можем стандартное поведение переопределить следующим образом:

```
modelBuilder.Entity<Phone>()
    .HasMany(p => p.Companies)
    .WithMany(c => c.Phones)
        .Map(m =>
    {
        m.ToTable("MobileCompanies");
        m.MapLeftKey("MobileId");
        m.MapRightKey("CompanyId");
    }); ;
```



Связь один-ко-многим (One-to-Many)

При связи один-ко-многим одна модель может ссылаться на множество объектов другой модели. Например, одна компания производит множество телефонов:

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

    public Company Company { get; set; }
}

public class Company
```

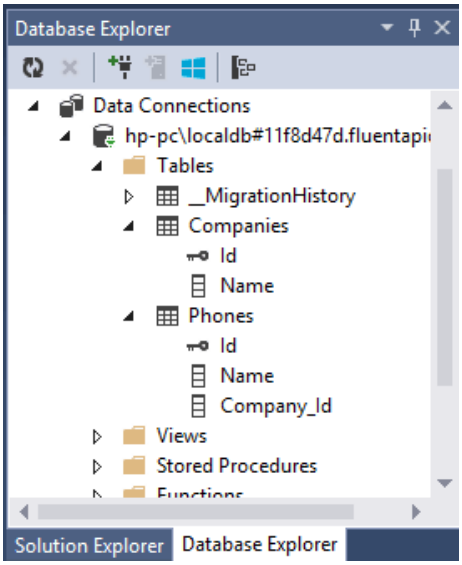
```
{
    public int Id { get; set; }
    public string Name { get; set; }
        public ICollection<Phone> Phones { get; set; }
        public Company()
    {
        Phones = new List<Phone>();
    }
}
```

А сама связь через Fluent API будет выглядеть так:

```
modelBuilder.Entity<Company>()
    .HasMany(p => p.Phones)
    .WithRequired(p=>p.Company);
```

Метод **HasMany()** устанавливает множественную связь между объектом Company и объектами Phone, а метод **WithRequired()** требует обязательной установки свойства Company у класса Phone.

После генерации таблиц таблица для моделей Phone будет содержать столбец-внешний ключ Company_Id для связи с таблицей Companies:



Настройка внешнего ключа

Возможно, нас не устраивает такое название столбца и внешнего ключа, которое дается EF по умолчанию. С помощью метода **HasForeignKey()** мы можем переопределить действие по умолчанию. Для этого определим свойство, которое будет представлять внешний ключ:

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

    public Company Company { get; set; }
    public int Manufacturer { get; set; }
}
```

Теперь настроим свойство Manufacturer в качестве внешнего ключа к таблице Companies:

```
modelBuilder.Entity<Company>()
    .HasMany(p => p.Phones)
    .WithRequired(p=>p.Company)
    .HasForeignKey(s=>s.Manufacturer);
```

Отключение каскадного удаления

По умолчанию, если свойство-внешний ключ зависимой сущности не представляет собой тип Nullable, то в таблице для этой сущности устанавливается каскадное удаление, по удалению главной сущности. То есть в предыдущем примере свойство Manufacturer, которое выполняет роль внешнего ключа, имеет тип int. Поэтому при генерации таблицы будет действовать правило `ON DELETE CASCADE`.

Если бы у нас свойство Manufacturer представляло бы тип `int?`, а вместо метода `WithRequired` использовался бы метод **WithOptional()** (`modelBuilder.Entity<Company>().HasMany(p => p.Phones).WithOptional(p=>p.Company)`), который не требует наличия внешнего ключа, то каскадное удаление бы не добавлялось в таблицу.

И чтобы через Fluent API отключить каскадное удаление, надо использовать метод **WillCascadeOnDelete(false)**:

```
modelBuilder.Entity<Company>()
    .HasMany(p => p.Phones)
    .WithRequired(p=>p.Company)
    .HasForeignKey(s=>s.Manufacturer)
    .WillCascadeOnDelete(false);
```

Соответственно, если используется true, то каскадное удаление, наоборот, включается: `WillCascadeOnDelete(true)`

А также можно использовать дополнительные методы для отключения каскадного удаления при отдельных видах отношений:

```
using System.Data.Entity.ModelConfiguration.Conventions;
.....................................................
modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>();
```

Назад **Содержание** **Вперед**