Fluent API и аннотации

Последнее обновление: 31.10.2015

Если мы используем подход Code First, то классы моделей сопоставляются с таблицами с помощью ряда правил в Entity Framework. Но иногда необходимо изменить и переопределить логику этих правил. Для этого используется Fluent API и аннотации данных.

Fluent API

Fluent API по большому счету представляет набор методов, которые определяются сопоставление между классами и их свойствами и таблицами и их столбцами. Как правило, функционал Fluent API задействуется при переопределении метода **OnModelCreating**:

```
class FluentContext : DbContext
{
    public FluentContext() :base("DefaultConnection")
    {}

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

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // использование Fluent API
                base.OnModelCreating(modelBuilder);
    }
}
```

В качестве экспериментальной модели возьмем следующую модель:

```
public class Phone
{
    public int Ident { get; set; }
    public string Name { get; set; }
    public int Discount { get; set; }
    public int Price { get; set; }
}
```

Сопоставление класса с таблицей

По умолчанию EF сопоставляет модель с одноименной таблицей, но мы можем переопределить это поведение с помощью метода **ToTable()**:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Phone>().ToTable("Mobiles");
    base.OnModelCreating(modelBuilder);
}
```

Теперь все объекты Phone будут храниться в таблице Mobiles. Но мы также с ними сможем работать через свойство db.Phones.

Если по какой-то сущности нам не надо создавать таблицу, то мы можем ее проигнорировать с помощью метода **Ignore()**:

```
modelBuilder.Ignore<Company>();
```

Переопределение первичного ключа

По умолчанию в Entity Framework первичный ключ должен представлять свойство модели с именем Id или [Имя_класса]Id, например, PhoneId. Чтобы переопределить первичный ключ через Fluent API, надо использовать метод **HasKey()**:

```
modelBuilder.Entity<Phone>().HasKey(p => p.Ident);
```

В данном случае первичным ключом будет свойство Ident класса Phone.

Чтобы настроить составной первичный ключ, мы можем указать два свойства:

```
modelBuilder.Entity<Phone>().HasKey(p => new { p.Ident, p.Name });
```

Сопоставление свойств

Чтобы сопоставить свойство с определенным столбцом, используется метод **HasColumnName()**:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).HasColumnName("PhoneName");
```

В данном случае свойство Name будет сопоставляться со столбцом PhoneName.

Если мы не хотим, чтобы с каким-то свойством вообще шло сопоставление, то мы можем его исключить с помощью метода **Ignore()**:

```
modelBuilder.Entity<Phone>().Ignore(p => p.Discount);
```

Теперь свойство Discount класса Phone не будет сопоставляться ни с никаким столбцом из таблицы в бд.

Столбцы в таблице в БД могут допускать значение NULL, которое указывает, что значение не определено. По умолчанию все столбцы при Code First, если не применяются аннотации данных, за исключением идентификатора допускают значение NULL. Но мы можем указать с помощью метода **IsRequired()**, что значение для этого столбца и свойства требуется обязательно:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).IsRequired();
```

Если нам, наоборот, надо указать, чтобы столбец мог принимать значения NULL, то мы можем использовать метод **IsOptional()**:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).IsOptional();
```

Настройка строк

Для строк мы модем указать максимальную длину с помощью метода **HasMaxLength()**. Например, длина не более 50 символов:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).HasMaxLength(50);
```

Также для строк можно определить, будут ли они храниться в кодировке Unicode:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).IsUnicode(false);
```

Параметр `false` указывает, что строки будут храниться не в Unicode-кодировке.

Настройка чисел decimal

Если у нас есть свойство с типом `decimal`, то мы можем указать для него точность число цифр в числе и число цифр после запятой:

```
// допустим, свойство Price - decimal
modelBuilder.Entity<Phone>().Property(p => p.Price).HasPrecision(15,2);
```

Теперь число decimal может содержать до 15 цифр и 2 цифры после запятой. Если же мы не указываем, то действуют значения по умолчанию - 18 и 2.

Настройка типа столбцов

По умолчанию EF сам выбирает тип данных в бд, исходя из типа данных свойства. Но мы также можем явно указать, какой тип данных в БД должен использоваться для столбца с помощью метода **HasColumnType()**:

```
modelBuilder.Entity<Phone>().Property(p => p.Name).HasColumnType("varchar");
```

Сопоставление модели с несколькими таблицами

С помощью Fluent API мы можем поместить ряд свойств модели в одну таблицу, а другие свойства связать со столбцами из другой таблицы:

```
modelBuilder.Entity<Phone>().Map(m =>
{
    m.Properties(p => new { p.Ident, p.Name });
    m.ToTable("Mobiles");
})
.Map(m =>
{
    m.Properties(p => new { p.Ident, p.Price, p.Discount });
    m.ToTable("MobilesInfo");
});
```

Таким образом, данные для свойства Name будут храниться в таблице Mobiles, а данные для свойств Price и Discount - в таблице MobilesInfo. И столбец идентификатора будет общим.

Назад **Содержание** **Вперед**