Бесплатная электронная книга

УЧУСЬ

Entity Framework

Free unaffiliated eBook created from
Stack Overflow contributors.

#entity-framework

1
1: Entity Framework 2
2
2
Examples 2
Entity Framework C # ( ) 2
Entity Framework NuGet 4
Entity Framework? 7
2: .t4 9
Examples 9
9
XML- 9
3: Code First - Fluent API 11
11
Examples 11
11
: 11
: 11
: 13
13
13
14
(NOT NULL) 14
15
4: Entity Framework SQLite 17
17
Examples 17
Entity Framework SQLite 17
SQLite 17
18
App.config 19

19

SQLite ... 19

SQLite DbContext ... 20

5: Entity-Framework Postgresql ... 21

Examples ... 21

, Entity Framework 6.1.3 PostgresSql ... 21

6: ... 22

Examples ... 22

... 22

... 23

7: ... 26

... 26

Examples ... 26

... 26

... 27

... 27

... 28

... 28

... 28

... 28

8: ... 30

Examples ... 30

CreateDatabaseIfNotExists ... 30

DropCreateDatabaseIfModelChanges ... 30

DropCreateDatabaseAlways ... 30

... 30

MigrateDatabaseToLatestVersion ... 31

9: First DataAnnotations ... 32

... 32

Examples ... 32

[Key] ... 32

[] ... 33

[MaxLength] [MinLength] 33
[Range (min, max)] 34
[DatabaseGenerated] 35
[NotMapped] 36
[] 37
[] 38
[Index] 38
[ForeignKey (string)] 39
[StringLength (int)] 39
[Timestamp] 40
[ConcurrencyCheck] 41
[InverseProperty (string)] 41
[ComplexType] 42
10: 44
Examples 44
44
11: Entity ( ) 46
46
Examples 46
1- Entity Framework @ () 46
2- Entity Framework @ - 50
3- - @ (MVC) 53
4- Entity Framework @ 55
12: EF 59
Examples 59
AsNoTracking 59
59
, , 60
async 60
61
61
61

- 62
13: EntityFramework ( ) 64
Examples 64
64
65
14: 67
Examples 67
« » 67
15: 69
Examples 69
Database.BeginTransaction () 69
16: 70
70
Examples 70
70
70
71
17: 72
Examples 72
72
72
74
Sql () 75
76
«- » 77
Entity Framework 77
18: 79
Examples 79
79
19: 81
81

Examples 81
81
81
82
DecimalPropertyConvention 83
84
85
20: : , 87
87
Examples 87
87
88
21: Entity Framework Code : « » « 90
90
Examples 90
« » 90
« »: 91
-- 93
-- 94
« »: 95
« »: 96
22: Entity Framework Code : 99
99
Examples 99
99
103
104
23: 106
106
Examples 106
106
106

107

109

Около

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

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

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

глава 1: Начало работы с Entity Framework

замечания

Entity Framework (EF) - объектно-реляционный картограф (ORM), который позволяет разработчикам .NET работать с реляционными данными с использованием объектов, специфичных для домена. Это устраняет необходимость в большей части кода доступа к данным, который разработчикам обычно приходится писать.

Entity Framework позволяет создавать модель путем написания кода или использования полей и строк в EF Designer. Оба этих подхода могут использоваться для таргетинга на существующую базу данных или создания новой базы данных.

Entity Framework является основным ORM, который Microsoft предоставляет для .NET Framework и рекомендованной Microsoft технологии доступа к данным.

Версии

Версия	Дата выхода
1,0	2008-08-11
4,0	2010-04-12
4,1	2011-04-12
4.1 Обновление 1	2011-07-25
4.3.1	2012-02-29
5.0	2012-08-11
6,0	2013-10-17
6,1	2014-03-17
Core 1.0	2016-06-27

Примечания к выпуску: https://msdn.microsoft.com/ru-ru/data/jj574253.aspx

Examples

Использование Entity Framework из C # (сначала код)

Сначала код позволяет создавать ваши сущности (классы) без использования GUI-дизайнера или файла .edmx. *Сначала* он называется *Code* , потому что вы можете *сначала* создать свои модели, а *структура Entity* будет автоматически создавать базу данных в соответствии с сопоставлениями. Или вы также можете использовать этот подход с существующей базой данных, которая *сначала* называется *кодом с существующей базой данных.* Например, если вы хотите, чтобы таблица содержала список планет:

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

Теперь создайте свой контекст, который является мостом между вашими сущностями и базой данных. Дайте ему один или несколько `DbSet<>` :

```
using System.Data.Entity;

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

Мы можем использовать это, выполнив следующие действия:

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

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

В этом примере мы создаем новую `Planet` с свойством `Name` со значением `"Jupiter"` и свойство `AverageDistanceFromSun` со значением `778.5`

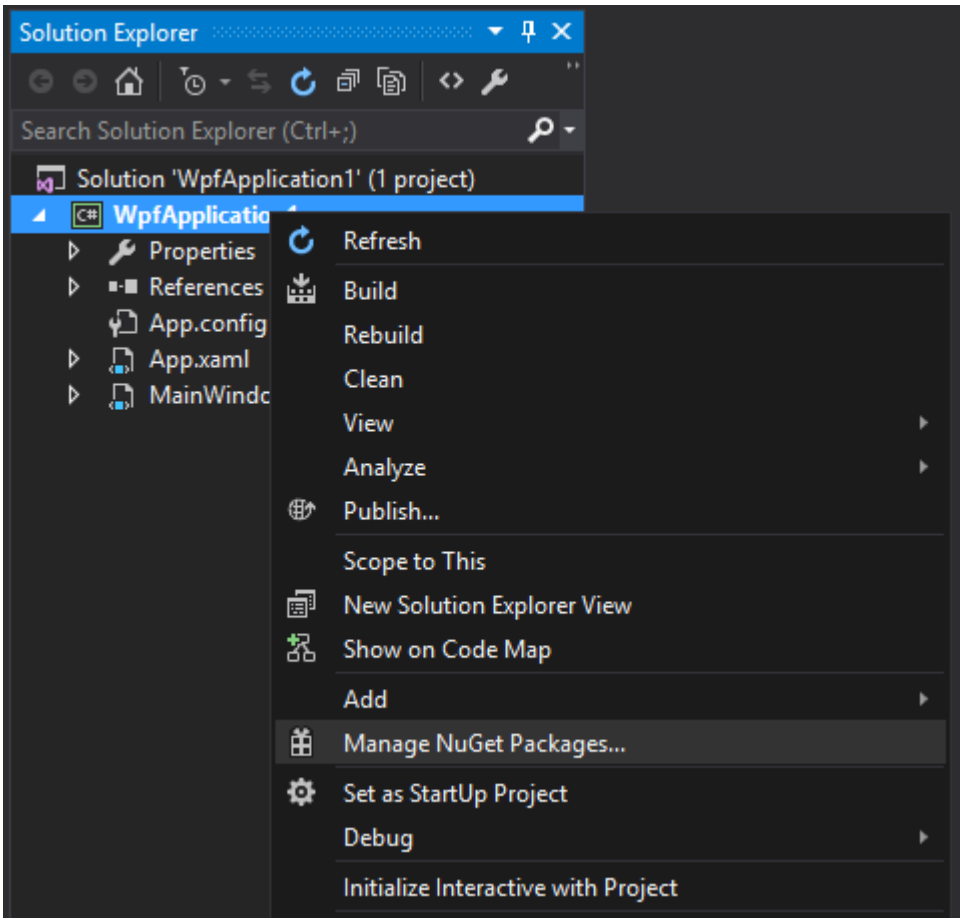
Затем мы можем добавить эту `Planet` в контекст с помощью `DbSet Add() DbSet` и зафиксировать наши изменения в базе данных с помощью метода `SaveChanges()` .

Или мы можем извлекать строки из базы данных:

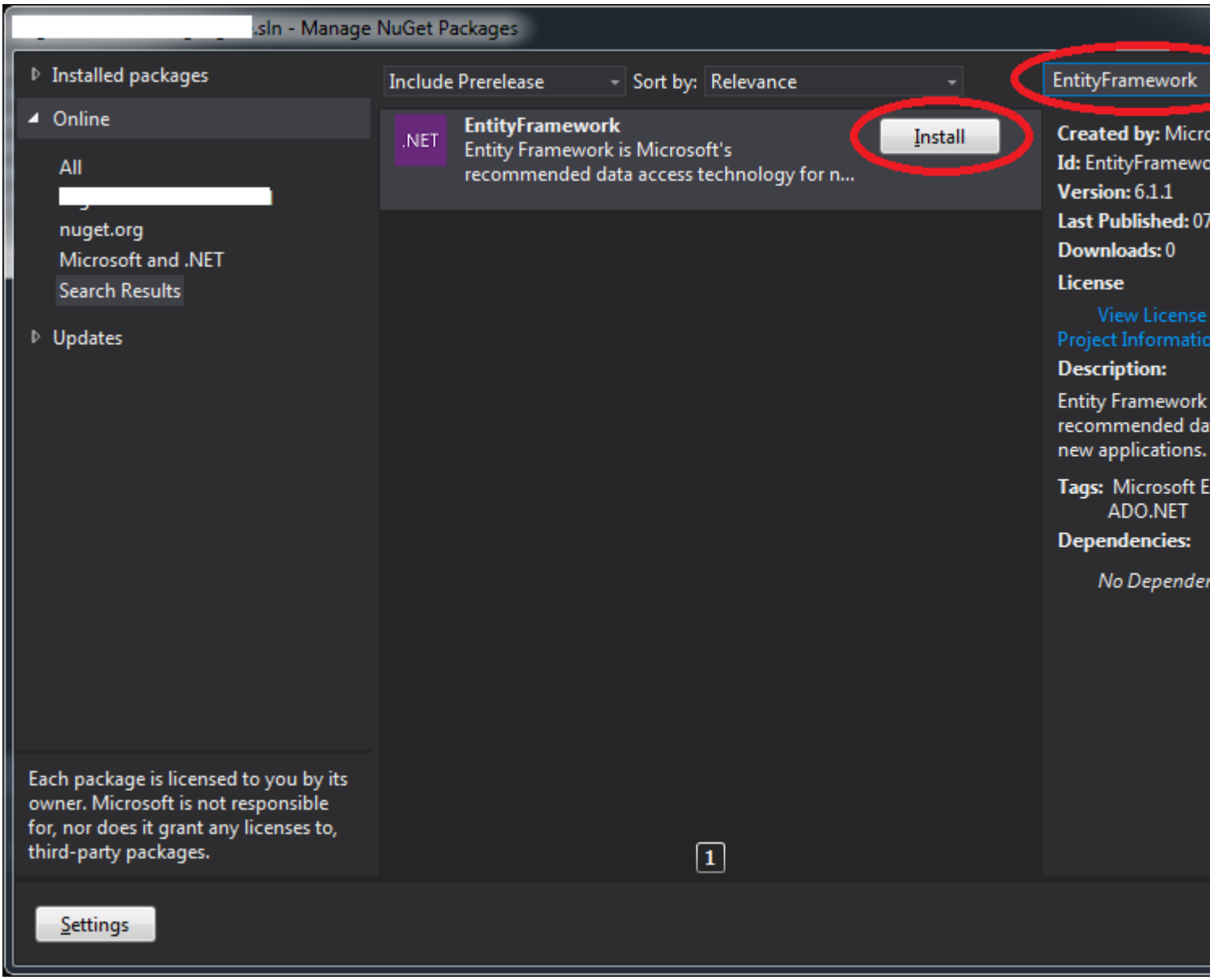
```
using(var context = new PlanetContext())
{
    var jupiter = context.Planets.Single(p => p.Name == "Jupiter");
    Console.WriteLine($"Jupiter is {jupiter.AverageDistanceFromSun} million km from the
sun.");
}
```

Установка пакета Entity Framework NuGet

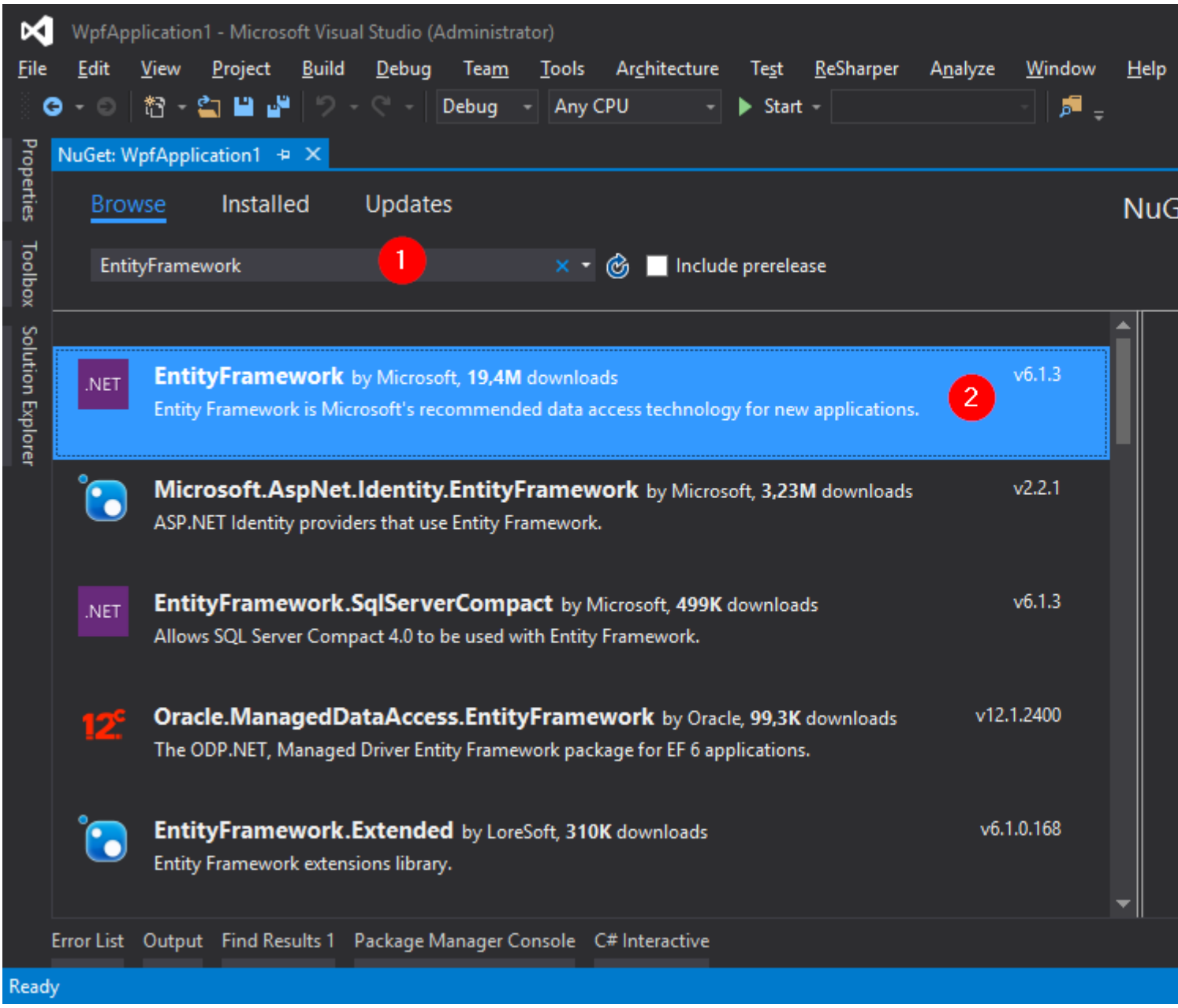
В своей Visual Studio откройте окно **Solution Explorer,** затем `щелкните правой кнопкой мыши` на своем проекте и выберите « *Управление пакетами NuGet»* в меню:



В открывшемся окне введите `EntityFramework` в поле поиска в правом верхнем углу.



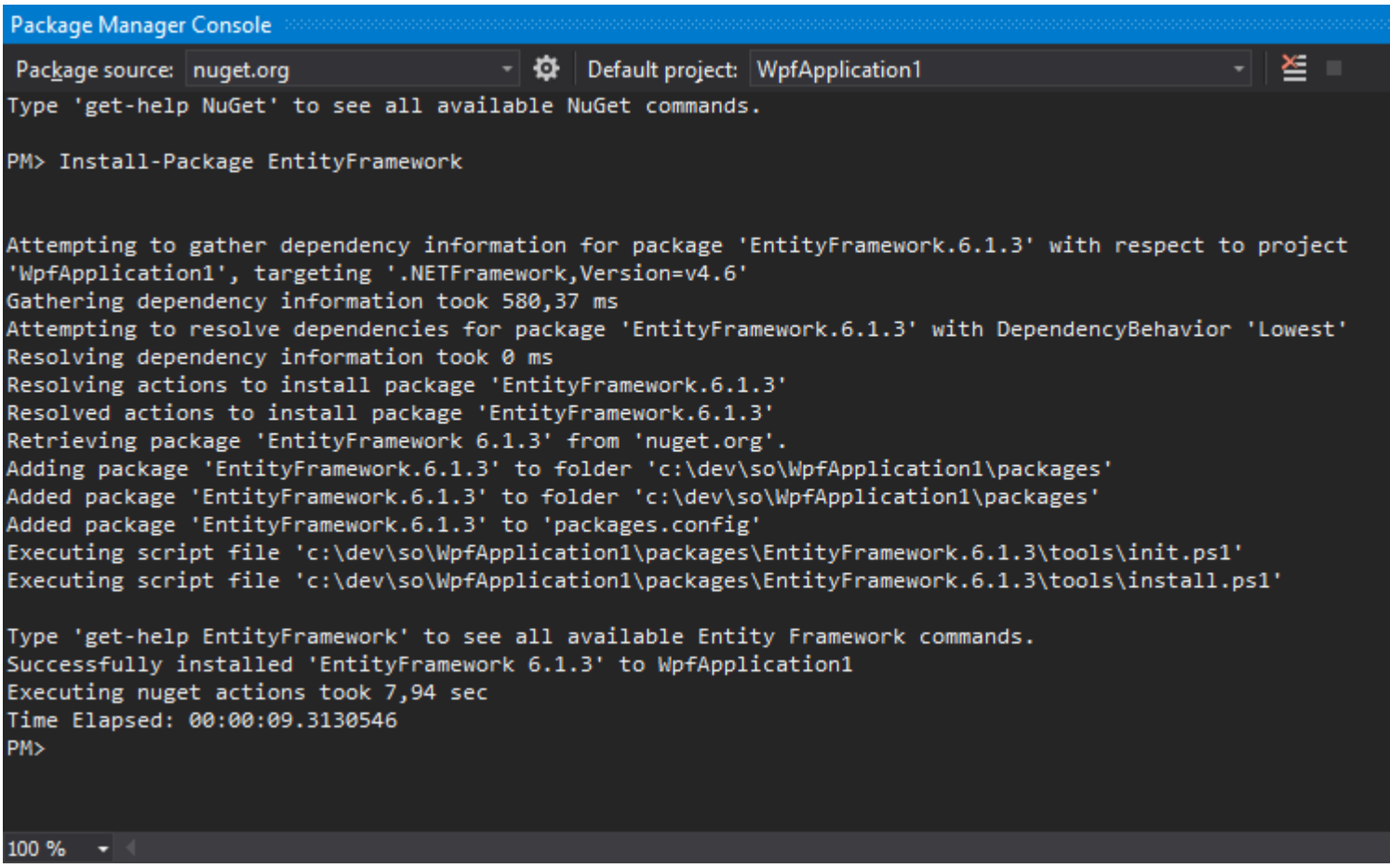
Если вы используете Visual Studio 2015, вы увидите что-то вроде этого:



Затем нажмите «Установить».

Мы также можем установить инфраструктуру сущности с помощью консоли диспетчера пакетов. Для этого вам сначала нужно открыть его с помощью *меню «Сервис» -> «Диспетчер пакетов NuGet» -> «Диспетчер пакетов»*, затем введите следующее:

```
Install-Package EntityFramework
```



Это установит Entity Framework и автоматически добавит ссылку на сборку в вашем проекте.

Что такое Entity Framework?

Написание и управление кодом ADO.Net для доступа к данным - утомительная и монотонная работа. Microsoft предоставила **инфраструктуру O / RM под названием « Entity Framework»** для автоматизации действий, связанных с базой данных для вашего приложения.

Структура Entity - это структура Object / Relational Mapping (O / RM). Это усовершенствование ADO.NET, которое дает разработчикам автоматизированный механизм для доступа и хранения данных в базе данных.

Что такое O / RM?

ORM - это инструмент для хранения данных из объектов домена в реляционную базу данных, такую как MS SQL Server, автоматическим способом без большого программирования. O / RM включает в себя три основные части:

1. Объекты класса домена
2. Объекты реляционной базы данных
3. Отображение информации о том, как объекты домена сопоставляются с объектами

реляционной базы данных (**ex** tables, views и хранимые процедуры)

ORM позволяет нам сохранить наш дизайн базы данных отдельно от нашего класса класса. Это делает приложение удобным и расширяемым. Он также автоматизирует стандартную операцию CRUD (создание, чтение, обновление и удаление), так что разработчику не нужно писать его вручную.

Прочитайте Начало работы с Entity Framework онлайн: https://riptutorial.com/ru/entity-framework/topic/815/начало-работы-с-entity-framework

глава 2: .t4 шаблонов в инфраструктуре сущностей

Examples

Динамическое добавление интерфейсов к модели

При работе с существующей моделью, которая довольно велика и часто регенерируется в тех случаях, когда требуется абстракция, было бы дорого обойтись вручную, чтобы обновить модель с интерфейсами. В таких случаях можно захотеть добавить динамическое поведение к генерации модели.

В следующем примере будет показано, как автоматически добавлять интерфейсы для классов с именами конкретных столбцов:

В вашей модели перейдите в файл `.tt` чтобы модифицировать метод `EntityClassOpening` следующим образом: это добавит интерфейс `IPolicyNumber` для объектов с `POLICY_NO` и `IUniqueId` в `UNIQUE_ID`

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

Это один конкретный случай, но он показывает способность изменять шаблоны `.tt` .

Добавление XML-документации в классы объектов

На всех сгенерированных классах моделей по умолчанию отсутствуют комментарии к документации. Если вы хотите использовать комментарии документации XML для всех сгенерированных классов сущностей, найдите эту часть внутри *[modelname]* .tt (имя *модели* - текущее имя файла EDMX):

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
```

```
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code); // used to write model namespace
#>
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
```

Вы можете добавить комментарии к документации XML перед линией `UsingDirectives` как показано в примере ниже:

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

Сгенерированный комментарий к документации должен содержать имя объекта, как указано ниже.

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

Прочитайте .t4 шаблонов в инфраструктуре сущностей онлайн: https://riptutorial.com/ru/entity-framework/topic/3964/-t4-шаблонов-в-инфраструктуре-сущностей

глава 3: Code First - Fluent API

замечания

Существует два общих способа указания HOW Entity Framework сопоставляет классы POCO с таблицами, столбцами базы данных и т. Д .: **аннотации данных** и **свободный API** .

Хотя аннотации данных являются простыми для чтения и понимания, у них отсутствуют определенные функции, такие как указание поведения «Cascade on Delete» для объекта. Fluent API, с другой стороны, немного сложнее в использовании, но обеспечивает гораздо более сложный набор функций.

Examples

Модели отображения

EntityFramewok Fluent API - это мощный и элегантный способ сопоставления ваших **кодовых** моделей домена с базой данных. Это также можно использовать с *кодом сначала с существующей базой данных* . У вас есть два варианта при использовании *Fluent API* : вы можете напрямую сопоставлять свои модели с методом *OnModelCreating* или создавать классы сопоставления, которые наследуются от *EntityTypeConfiguration,* а затем добавлять эти модели в *modelBuilder* по методу *OnModelCreating* . *Второй* вариант, который я предпочитаю, и я покажу пример этого.

Шаг первый: создайте модель.

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

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

Шаг второй: создать класс сопоставления

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

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

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

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

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

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

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

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

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

Объясним отображения:

- **HasKey** - определяет первичный ключ. Также можно использовать *составные первичные ключи* . Например: *this.HasKey (m => new {m.DepartmentId, m.PositionId})* .
- **Свойство** - позволяет нам настраивать свойства модели.
- **HasColumnType** - указать тип столбца уровня базы данных. Обратите внимание, что это может быть разным для разных баз данных, таких как Oracle и MS SQL.
- **HasDatabaseGeneratedOption** - указывает, вычисляется ли свойство на уровне базы данных. Числовые ПК имеют значение *DatabaseGeneratedOption.Identity* по умолчанию, вы должны указать *DatabaseGeneratedOption.None,* если вы не хотите, чтобы это было так.
- **HasMaxLength** - ограничивает длину строки.
- **IsRequired** - маркирует свойство как требуемое.

- **HasPrecision** - позволяет указать точность для десятичных знаков.
- **Игнорировать** - **Игнорировать** свойство полностью и не отображать его в базе данных. Мы проигнорировали FullName, потому что этот столбец не нужен в нашей таблице.
- **ToTable** - указать имя таблицы и имя схемы (необязательно) для модели.
- **HasColumnName** - связывает свойство с именем столбца. Это не требуется, если имена свойств и имена столбцов идентичны.

Шаг третий: добавьте класс сопоставления в конфигурации.

Нам нужно сказать EntityFramework, чтобы использовать наш класс mapper. Для этого нам нужно добавить его в *modelBuilder.Configurations* on *OnModelCreating* :

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

И это все. Мы все готовы идти.

Основной ключ

Используя метод .HasKey (), свойство может быть явно сконфигурировано как первичный ключ объекта.

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

public class PersonContext : DbContext
{
    // ..

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

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

Композитный первичный ключ

Используя метод .HasKey (), набор свойств может быть явно сконфигурирован как составной первичный ключ объекта.

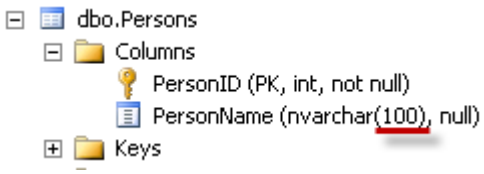
```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

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

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

Максимальная длина

Используя метод .HasMaxLength (), максимальное количество символов может быть настроено для свойства.

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

public class PersonContext : DbContext
{
    // ..

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

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

Полученный столбец с указанной длиной столбца:



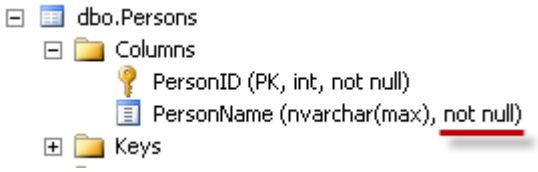
Обязательные свойства (NOT NULL)

Используя метод .IsRequired (), свойства могут быть указаны как обязательные, что означает, что столбец будет иметь ограничение NOT NULL.

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

public class PersonContext : DbContext
{
```

```
    // ..

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

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

Полученный столбец с ограничением NOT NULL:



Идентификация внешнего ключа

Когда свойство навигации существует на модели, Entity Framework автоматически создаст столбец внешнего ключа. Если требуется конкретное имя внешнего ключа, но не содержится в качестве свойства в модели, оно может быть установлено явно с использованием Fluent API. Используя метод `Map` при установлении отношений с внешним ключом, любое уникальное имя может использоваться для внешних ключей.

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

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

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

После определения отношения метод `Map` позволяет явно указать имя внешнего ключа, выполнив `MapKey` . В этом примере, что привело бы к имени столбца Employer_Id, теперь является EmployerId.

Прочитайте Code First - Fluent API онлайн: https://riptutorial.com/ru/entity-framework/topic/4530/code-first---fluent-api

глава 4: Entity Framework с SQLite

Вступление

SQLite - это автономная, безсерверная транзакционная база данных SQL. Он может использоваться в .NET-приложении, используя как свободно доступную библиотеку .NET SQLite, так и Entity Framework SQLite. В этом разделе рассказывается о настройке и использовании провайдера Entity Framework SQLite.

Examples

Настройка проекта для использования Entity Framework с поставщиком SQLite

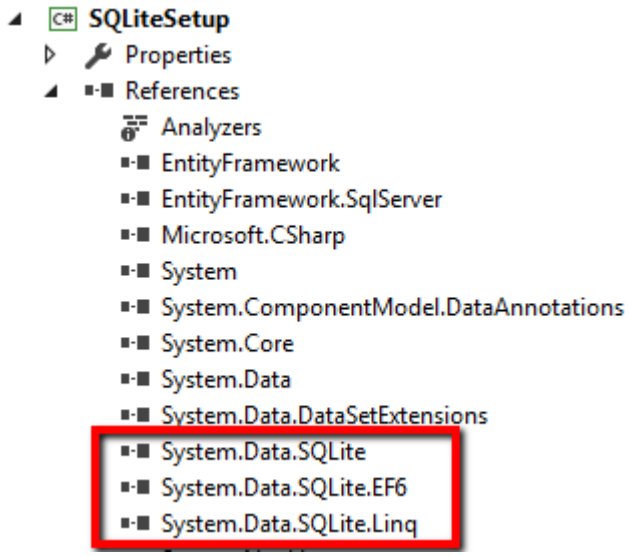
Библиотека Entity Framework поставляется только с поставщиком SQL Server. Для использования SQLite потребуются дополнительные зависимости и конфигурация. Все необходимые зависимости доступны на NuGet.

Установка управляемых библиотек SQLite

Все управляемые настройки могут быть установлены с помощью консоли управления пакетами NuGet. Запустите команду `Install-Package System.Data.SQLite` .

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

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

Как показано выше, при установке `System.Data.SQLite` все связанные с ним управляемые библиотеки устанавливаются вместе с ним. Сюда входит `System.Data.SQLite.EF6` , поставщик EF для SQLite. Проект также теперь ссылается на сборки, необходимые для использования поставщика SQLite.



Включая неуправляемую библиотеку

Управляемые библиотеки SQLite зависят от неуправляемой сборки с именем

`SQLite.Interop.dll` . Он входит в состав сборок пакетов, загружаемых с помощью пакета SQLite, и они автоматически копируются в ваш каталог сборки при создании проекта. Однако, поскольку он неуправляемый, он не будет включен в ваш список ссылок. Но обратите внимание: эта сборка наиболее распространена с приложением для сборки SQLite.

Примечание. Эта сборка зависит от бита, то есть вам нужно будет включить конкретную сборку для каждой битности, которую вы планируете поддерживать (x86 / x64).

Редактирование App.config проекта

Файл `app.config` потребует некоторых изменений, прежде чем SQLite можно будет использовать в качестве поставщика Entity Framework.

Необходимые исправления

При установке пакета файл `app.config` автоматически обновляется для включения необходимых записей для SQLite и SQLite EF. К сожалению, эти записи содержат некоторые ошибки. Они должны быть изменены до того, как они будут работать правильно.

Сначала найдите элемент `DbProviderFactories` в файле конфигурации. Он находится внутри элемента `system.data` и будет содержать следующее

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

Это может быть упрощено, чтобы содержать одну запись

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

При этом мы указали, что поставщики SQLite EF6 должны использовать фабрику SQLite.

Добавить строку подключения SQLite

Строки подключения могут быть добавлены в файл конфигурации в корневом элементе. Добавьте строку соединения для доступа к базе данных SQLite.

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

Здесь важно отметить `provider` . Он был установлен в `System.Data.SQLite.EF6` . Это говорит EF, что, когда мы используем эту строку соединения, мы хотим использовать SQLite. Указанный `data source` является лишь примером и будет зависеть от местоположения и имени вашей базы данных SQLite.

Ваш первый SQLite DbContext

После завершения всей установки и конфигурации вы можете начать использовать `DbContext` который будет работать в вашей базе данных SQLite.

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

Указав `name=TestContext` , я указываю, что строка соединения TestContext, расположенная в файле `app.config` должна использоваться для создания контекста. Эта строка подключения была настроена на использование SQLite, поэтому в этом контексте будет использоваться база данных SQLite.

Прочитайте Entity Framework с SQLite онлайн: https://riptutorial.com/ru/entity-framework/topic/9280/entity-framework-c-sqlite

глава 5: Entity-Framework с Postgresql

Examples

Предварительные шаги, необходимые для использования Entity Framework 6.1.3 с PostgresSql с использованием Npgsqlddexprovider

1) Взял резервную копию Machine.config из местоположений C: \ Windows \ Microsoft.NET \ Framework \ v4.0.30319 \ Config и C: \ Windows \ Microsoft.NET \ Framework64 \ v4.0.30319 \ Config

2) Скопируйте их в другое место и отредактируйте их как

a) найти и добавить в `<system.data> <DbProviderFactories>`

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

b) если они уже существуют над входом, проверьте его и обновите.

3. Замените исходные файлы на измененные.
4. запустить команду командной строки разработчика для VS2013 в качестве администратора.
5. если Npgsql уже установлен, используйте команду «gacutil -u Npgsql» для удаления, затем установите новую версию Npgsql 2.5.0 с помощью команды «gacutil -i [path of dll]»,
6. Сделайте выше для Mono.Security 4.0.0.0
7. Загрузите NpgsqlDdexProvider-2.2.0-VS2013.zip и запустите NpgsqlDdexProvider.vsix (закройте все экземпляры визуальной студии)
8. Найден EFTools6.1.3-beta1ForVS2013.msi и запустите его.
9. После создания нового проекта установите версию EntityFramework (6.1.3), NpgSql (2.5.0) и NpgSql.EntityFramework (2.5.0) из Manage Nuget Packages.10). Выполнено ... Добавить новую модель данных Entity в ваш проект MVc

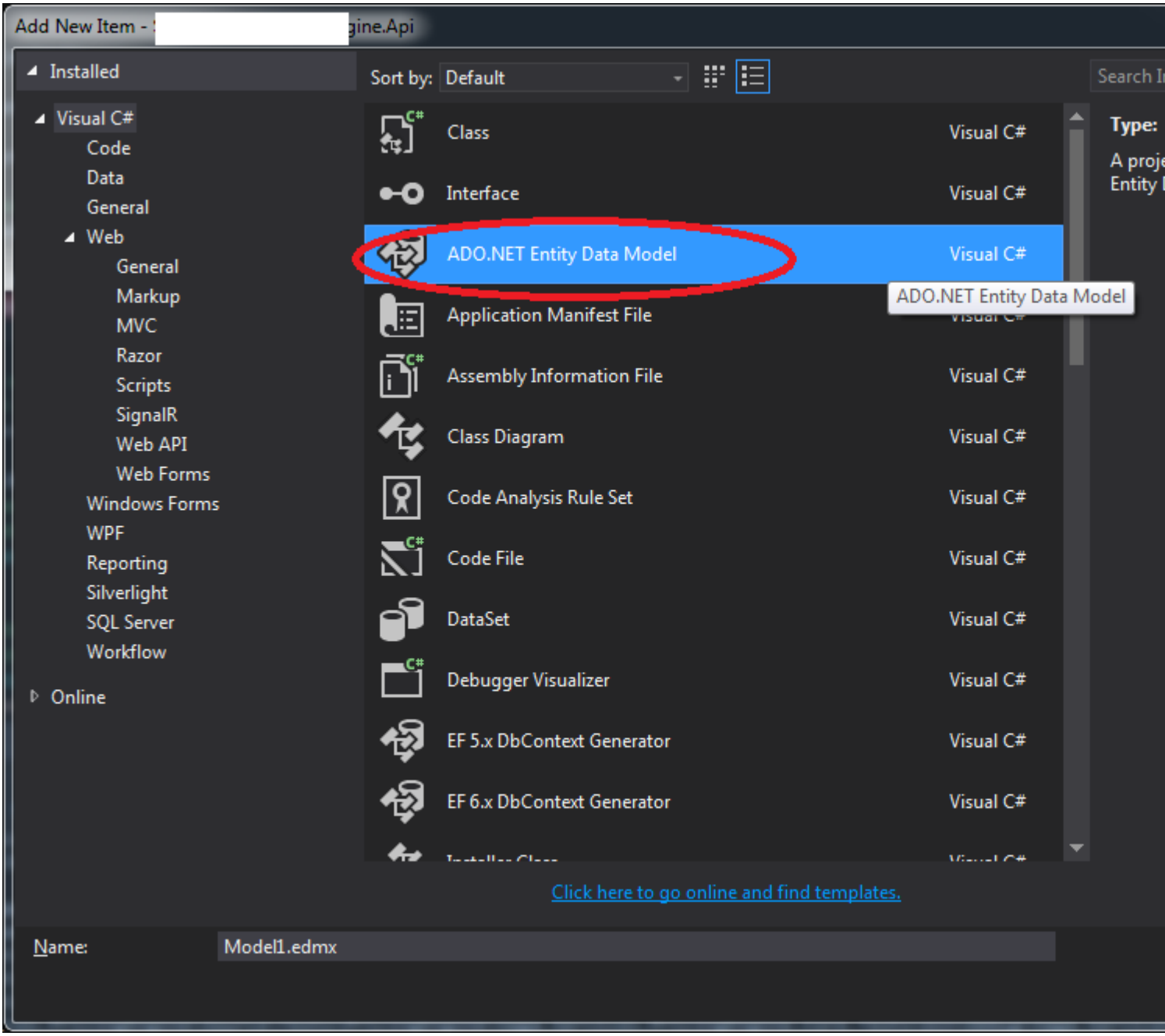
Прочитайте Entity-Framework с Postgresql онлайн: https://riptutorial.com/ru/entity-framework/topic/7647/entity-framework-c-postgresql

глава 6: Генерация первой модели базы данных

Examples

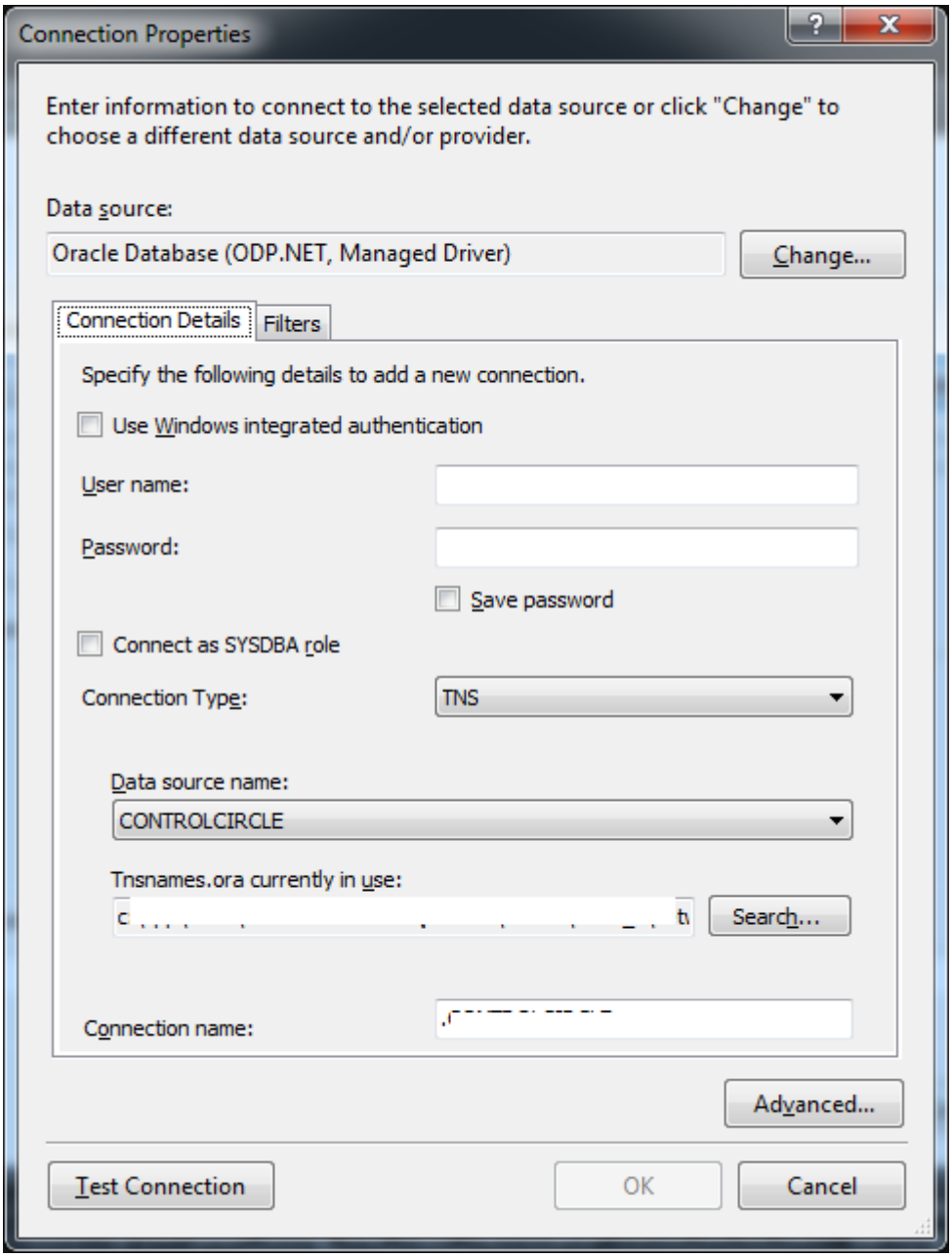
Создание модели из базы данных

В `Visual Studio` перейдите в свой `Solution Explorer` затем нажмите « `Project` вы добавите модель « `Правая» мышь` . Выберите `ADO.NET Entity Data Model`



Затем выберите « `Generate from database` и нажмите « `Next` в следующем окне « `New Connection...`

и укажите базу данных, из которой вы хотите сгенерировать модель (может быть `MSSQL` , `MySQL` или `Oracle`)



После этого нажмите « `Test Connection` чтобы узнать, правильно ли настроено подключение (не переходите, если он не работает здесь).

Нажмите « `Next` затем выберите нужные параметры (например, стиль для генерации имен сущностей или добавления внешних ключей).

Нажмите « `Next` раз, на этом этапе у вас должна быть модель, созданная из базы данных.

Добавление аннотаций данных к сгенерированной модели

В стратегии генерации кода T4, используемой Entity Framework 5 и выше, атрибуты аннотации данных по умолчанию не включены. Чтобы включить в аннотации данных поверх

некоторого свойства каждой модели восстановления, открыть файл шаблона входит в EDMX (с `.tt` расширением) , а затем добавить , `using` заявление по `UsingDirectives` способом , как показано ниже:

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

В качестве примера предположим, что шаблон должен включать `KeyAttribute` который указывает свойство первичного ключа. Чтобы автоматически вставить `KeyAttribute` во время регенерации модели, найдите часть кода, содержащего `codeStringGenerator.Property` как `codeStringGenerator.Property` ниже:

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

Затем вставьте if-условие для проверки свойства ключа следующим образом:

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

Применяя изменения выше, все созданные классы моделей будут иметь `KeyAttribute` первичного ключа после обновления модели из базы данных.

До

```
using System;

public class Example
```

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

После

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

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

Прочитайте Генерация первой модели базы данных онлайн: https://riptutorial.com/ru/entity-framework/topic/4414/генерация-первой-модели-базы-данных

глава 7: Загрузка связанных объектов

замечания

Если модели правильно связаны, вы можете легко загрузить связанные данные с помощью EntityFramework. У вас есть три варианта выбора: *ленивая загрузка* , *интенсивная загрузка* и *явная загрузка* .

Модели, используемые в примерах:

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

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

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

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

Examples

Ленивая загрузка

По умолчанию включена *ленивая загрузка* . Ложная загрузка достигается за счет создания производных прокси-классов и переопределения виртуальных навигационных прогнозов. Ленивая загрузка происходит, когда свойство открывается в первый раз.

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

Чтобы отключить Lazy для определенных свойств навигации, просто удалите ключевое слово virtual из объявления свойства:

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

Если вы хотите полностью отключить Lazy-загрузку, вам нужно изменить конфигурацию, например, в *конструкторе контекста* :

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

Примечание. Не забудьте *выключить* Lazy, если вы используете сериализацию. Поскольку сериализаторы получают доступ к каждой собственности, которую вы собираетесь загрузить из базы данных. Кроме того, вы можете запустить цикл между свойствами навигации.

Яркая загрузка

Высокая загруженность позволяет одновременно загружать все нужные вам объекты. Если вы предпочитаете, чтобы все ваши сущности работали в одном вызове базы данных, то *Eager loading* - это путь. Он также позволяет загружать несколько уровней.

У вас есть *две возможности* для загрузки связанных объектов, вы можете выбрать *строго типизированные* или *строковые* перегрузки метода *Include* .

Сильно напечатано.

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

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

Этот метод доступен с Entity Framework 4.1. Убедитесь, что у вас есть ссылка, `using System.Data.Entity;` задавать.

Перегрузка строк.

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

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

Явная загрузка

После поворота *Lazy* load вы можете лениво загружать объекты, явно вызывая метод *Load* для записей. *Ссылка* используется для загрузки свойств одиночной навигации, тогда как *коллекция* используется для получения коллекций.

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

Так как на *загрузке Eager* вы можете использовать перегрузки вышеперечисленных методов для загрузки entiteis по их именам:

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

Связанные с фильтром объекты.

Используя метод *Query,* мы можем фильтровать загруженные связанные объекты:

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

Запросы проекции

Если нужны связанные данные в денормализованном типе или, например, только подмножество столбцов, можно использовать проекционные запросы. Если нет причин использовать дополнительный тип, существует возможность вставить значения в анонимный тип .

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

Или с синтаксисом запроса:

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

Прочитайте Загрузка связанных объектов онлайн: https://riptutorial.com/ru/entity-framework/topic/4678/загрузка-связанных-объектов

глава 8: Инициализаторы баз данных

Examples

CreateDatabaseIfNotExists

Реализация `IDatabaseInitializer` который по умолчанию используется в EntityFramework. Как следует из названия, он создает базу данных, если она не существует. Однако, когда вы меняете модель, она выдает исключение.

Использование:

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

DropCreateDatabaseIfModelChanges

Эта реализация `IDatabaseInitializer` уменьшает и воссоздает базу данных, если модель автоматически изменяется.

Использование:

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

DropCreateDatabaseAlways

Эта реализация `IDatabaseInitializer` уменьшает и воссоздает базу данных каждый раз, когда ваш контекст используется в домене приложений приложений. Остерегайтесь потери данных из-за того, что база данных воссоздана.

Использование:

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

Пользовательский инициализатор базы данных

Вы можете создать свою собственную реализацию `IDatabaseInitializer` .

Пример реализации инициализатора, который будет переносить базу данных на 0, а затем полностью переносится на самую новую миграцию (например, при выполнении интеграционных тестов). Для этого вам понадобится тип `DbMigrationsConfiguration` .

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

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

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

MigrateDatabaseToLatestVersion

Реализация `IDatabaseInitializer` которая будет использовать First First Migrations для обновления базы данных до последней версии. Чтобы использовать этот инициализатор, вы также должны использовать тип `DbMigrationsConfiguration` .

Использование:

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

Прочитайте Инициализаторы баз данных онлайн: https://riptutorial.com/ru/entity-framework/topic/5526/инициализаторы-баз-данных

глава 9: Код First DataAnnotations

замечания

Entity Framework Code-First предоставляет набор атрибутов DataAnnotation, которые вы можете применить к своим классам и свойствам домена. Атрибуты DataAnnotation переопределяют стандартные соглашения Code-First.

1. **System.ComponentModel.DataAnnotations** включает атрибуты, которые влияют на нулеустойчивость или размер столбца.
2. **System.ComponentModel.DataAnnotations.Schema** namespace включает атрибуты, которые влияют на схему базы данных.

Примечание. DataAnnotations предоставляет только набор параметров конфигурации. Fluent API предоставляет полный набор параметров конфигурации, доступных в Code-First.

Examples

Атрибут [Key]

Ключ - это поле в таблице, которое однозначно идентифицирует каждую строку / запись в таблице базы данных.

Используйте этот атрибут, чтобы **переопределить стандартное соглашение Code-First** . Если применяется к свойству, он будет использоваться в качестве **столбца первичного ключа** для этого класса.

```
using System.ComponentModel.DataAnnotations;

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

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

Если требуется составной первичный ключ, атрибут [Key] можно также добавить к нескольким свойствам. Порядок столбцов в составном ключе должен быть указан в форме [**Ключ, Столбец (Заказ = x)]** .

```
using System.ComponentModel.DataAnnotations;

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

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

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

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

Без атрибута [Key] EntityFramework вернется к соглашению по умолчанию, которое должно использовать свойство класса в качестве первичного ключа с именем «Id» или «{ ClassName} Id».

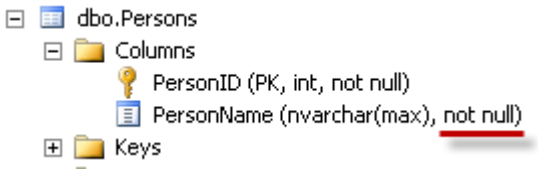
```
public class Person
{
    public int PersonID { get; set; }        // <- will be used as primary key

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

[Обязательный] атрибут

При применении к свойству класса домена база данных создаст столбец NOT NULL.

```
using System.ComponentModel.DataAnnotations;

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

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

Полученный столбец с ограничением NOT NULL:



Примечание. Его также можно использовать с asp.net-mvc в качестве атрибута проверки.

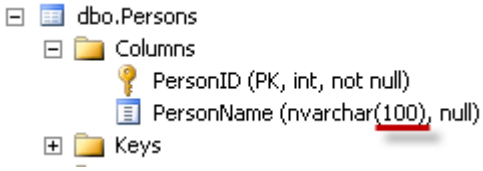
[MaxLength] и [MinLength]

Атрибут **[MaxLength (int)]** может применяться к свойству типа строки или массива класса домена. Entity Framework установит размер столбца на указанное значение.

```
using System.ComponentModel.DataAnnotations;

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

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

Полученный столбец с указанной длиной столбца:



[MinLength (int)] - атрибут проверки, он не влияет на структуру базы данных. Если мы попытаемся вставить / обновить Person с PersonName длиной менее 3 символов, это сообщение не будет выполнено. Мы получим `DbUpdateConcurrencyException` которое нам нужно будет обработать.

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

Оба атрибута **[MaxLength]** и **[MinLength]** также могут использоваться с asp.net-mvc в качестве атрибута проверки.

[Range (min, max)] атрибут

Указывает числовой минимальный и максимальный диапазон для свойства

```
using System.ComponentModel.DataAnnotations;

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

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

Если мы попытаемся вставить / обновить класс со значением вне диапазона, это сообщение не будет выполнено. Мы получим `DbUpdateConcurrencyException` которое нам нужно будет обработать.

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

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

Он также может использоваться с asp.net-mvc в качестве атрибута проверки.

Результат:

Атрибут [DatabaseGenerated]

Указывает, как база данных генерирует значения для свойства. Существует три возможных значения:

1. `None` указывает, что значения не генерируются базой данных.
2. `Identity` указывает, что столбец является столбцом идентификации , который обычно используется для целых первичных ключей.
3. `Computed` указывает, что база данных генерирует значение для столбца.

Если значение имеет значение, отличное от `None` , Entity Framework не будет вносить изменения, внесенные в свойство обратно в базу данных.

По умолчанию (на основе `StoreGeneratedIdentityKeyConvention`) свойство целочисленного ключа будет рассматриваться как столбец идентификатора. Чтобы переопределить это соглашение и заставить его обрабатывать как столбец не идентичности, вы можете использовать атрибут `DatabaseGenerated` со значением `None` .

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

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

public class Bar
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
```

```
    public int Id { get; set; } // non-identity column
}
```

Следующий SQL создает таблицу с вычисленным столбцом:

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

Чтобы создать объект для представления записей в приведенной выше таблице, вам нужно будет использовать атрибут `DatabaseGenerated` со значением `Computed` .

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

Атрибут [NotMapped]

По соглашению Code-First Entity Framework создает столбец для каждого общедоступного свойства поддерживаемого типа данных и имеет как getter, так и setter. **Аннотации [NotMapped]** должны применяться к любым свойствам, для которых нам **НЕ** нужен столбец в таблице базы данных.

Примером свойства, которое мы не хотим хранить в базе данных, является полное имя студента, основанное на их имени и фамилии. Это можно вычислить «на лету», и нет необходимости хранить его в базе данных.

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

Свойство «FullName» имеет только getter и setter, поэтому по умолчанию Entity Framework **НЕ** создаст для него столбец.

Другим примером свойства, которое мы, возможно, не хотим хранить в базе данных, является «Средняя оценка» учащегося. Мы не хотим получать AverageGrade по запросу; вместо этого у нас может быть рутина в другом месте, которая ее вычисляет.

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

«AverageGrade» должен быть помечен аннотацией **[NotMapped]** , иначе Entity Framework

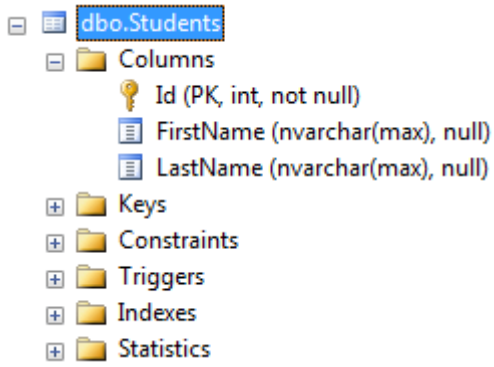
создаст для него столбец.

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

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

    public string FirstName { set; get; }

    public string LastName { set; get; }

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

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

Для вышеупомянутого `DbMigration.cs` мы увидим внутри `DbMigration.cs`

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

и в SQL Server Management Studio



Атрибут [Таблица]

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

Сообщает Entity Framework использовать конкретное имя таблицы вместо создания одного (то есть `Person` или `Persons`)

Мы также можем указать схему для таблицы, используя атрибут [Таблица]

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

Атрибут [Столбец]

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

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

Сообщает Entity Framework вместо использования имени столбца вместо имени свойства. Вы также можете указать тип данных базы данных и порядок столбца в таблице:

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

Атрибут [Index]

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

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

Создает индекс базы данных для столбца или набора столбцов.

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

Это создает индекс с определенным именем.

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

Это создает уникальный индекс.

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

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

Это создает составной индекс, используя 2 столбца. Для этого вы должны указать одно и

то же имя индекса и указать порядок столбцов.

Примечание . Атрибут Index был введен в Entity Framework 6.1. Если вы используете более раннюю версию, информация в этом разделе не применяется.

Атрибут [ForeignKey (string)]

Указывает имя пользовательского внешнего ключа, если требуется внешний ключ, не соответствующий соглашению EF.

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

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

Это также можно использовать, если у вас несколько отношений с одним типом объекта.

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

public class Customer
{
    ...

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

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

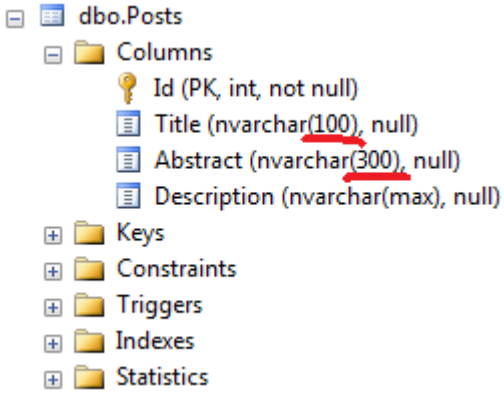
Без атрибутов `ForeignKey` EF может их перепутать и использовать значение `BillingAddressID` при извлечении `MailingAddress` , или просто может возникнуть другое имя для столбца на основе его собственных соглашений об именах (например, `Address_MailingAddress_Id`) и попытаться использовать это вместо этого (что приведет к ошибке, если вы используете это с существующей базой данных).

Атрибут [StringLength (int)]

```
using System.ComponentModel.DataAnnotations;

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

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

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

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

Определяет максимальную длину для строкового поля.

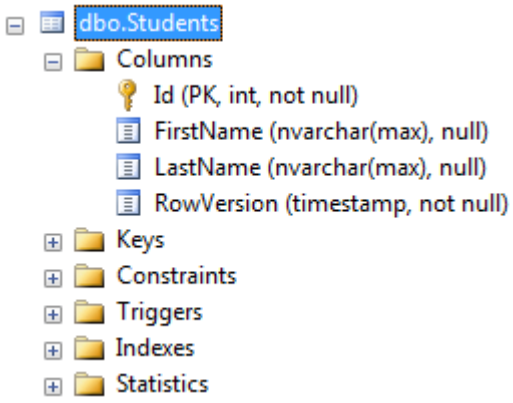


Примечание . Его также можно использовать с asp.net-mvc в качестве атрибута проверки.

Атрибут [Timestamp]

Атрибут [TimeStamp] может применяться только к одному свойству байтового массива в данном классе Entity. Entity Framework создаст в таблице базы данных столбец с нулевым значением в таблице базы данных для этого свойства. Entity Framework автоматически использует этот столбец TimeStamp при проверке параллелизма.

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

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

    public string FirstName { set; get; }

    public string LastName { set; get; }

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



Атрибут [ConcurrencyCheck]

Этот атрибут применяется к свойству класса. Вы можете использовать атрибут ConcurrencyCheck, если хотите использовать существующие столбцы для проверки параллелизма, а не отдельный столбец временной отметки для параллелизма.

```
using System.ComponentModel.DataAnnotations;

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

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

В приведенном выше примере атрибут ConcurrencyCheck применяется к свойству AuthorName класса Author. Итак, Code-First будет включать столбец AuthorName в команде update (where clause), чтобы проверить оптимистичный параллелизм.

Атрибут [InverseProperty (string)]

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

public class Department
{
    ...

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

public class Employee
{
    ...

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

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

InverseProperty может использоваться для идентификации *двухсторонних* отношений, когда между двумя сущностями существуют **несколько** *двухсторонних* отношений.

Он сообщает Entity Framework, какие свойства навигации должны соответствовать свойствам с другой стороны.

Entity Framework не знает, какая карта навигации имеет свойства с другой стороны, когда существуют несколько двунаправленных отношений между двумя объектами.

В качестве своего параметра ему нужно имя соответствующего свойства навигации в соответствующем классе.

Это также можно использовать для объектов, которые имеют отношение к другим объектам того же типа, образуя рекурсивное отношение.

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

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

    ...

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

Обратите также внимание на использование атрибута `ForeignKey` для указания столбца, который используется для внешнего ключа в таблице. В первом примере два свойства класса `Employee` могли иметь атрибут `ForeignKey` для определения имен столбцов.

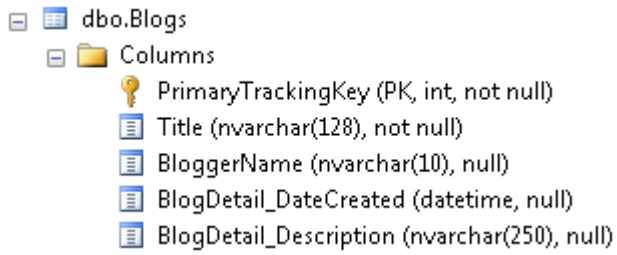
Атрибут [ComplexType]

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

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

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

public class Blog
{
    ...
```

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

Пометьте класс как сложный тип в Entity Framework.

Сложные типы (или *объекты ценности* в проекте, управляемом доменом) не могут отслеживаться самостоятельно, но отслеживаются как часть объекта. Вот почему BlogDetails в этом примере не имеет свойства ключа.



Они могут быть полезны при описании сущностей домена на нескольких классах и разбиении этих классов на полный объект.

Прочитайте Код First DataAnnotations онлайн: https://riptutorial.com/ru/entity-framework/topic/4161/код-first-dataannotations

глава 10: Комплексные типы

Examples

Первые комплексные типы кода

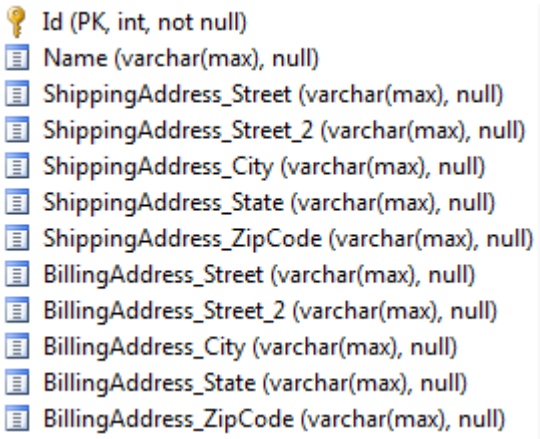
Сложный тип позволяет отображать выбранные поля таблицы базы данных в один тип, который является дочерним по отношению к основному типу.

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

Этот сложный тип затем может использоваться в нескольких типах сущностей. Его можно даже использовать более одного раза в одном типе сущности.

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

Этот тип сущности будет затем сохранен в таблице в базе данных, которая будет выглядеть примерно так.



Конечно, в этом случае предпочтительной будет ассоциация 1: n (Customer-Address), но пример показывает, как можно использовать сложные типы.

Прочитайте Комплексные типы онлайн: https://riptutorial.com/ru/entity-framework/topic/5527/комплексные-типы

глава 11: Лучшие практики для платформы Entity (простой и профессиональный)

Вступление

В этой статье мы рассмотрим простую и профессиональную практику использования Entity Framework.

Простой: потому что ему нужен только один класс (с одним интерфейсом)

Профессионал: поскольку он применяет принципы архитектуры SOLID

Я не хочу больше говорить ... давайте наслаждаться этим!

Examples

1- Entity Framework @ Уровень данных (основы)

В этой статье мы будем использовать простую базу данных под названием «Компания» с двумя таблицами:

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

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

1-1 Создать код платформы Entity Framework

В этом слое мы генерируем код Entity Framework (в библиотеке проектов) (см. Эту статью, как вы это можете сделать), тогда у вас будут следующие классы

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

1-2 Создать базовый интерфейс

Мы создадим один интерфейс для наших основных функций

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

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

```
    TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class;

    #endregion Tables and Views functions

    #region Transactions Functions

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

    #endregion Transactions Functions

    #region Database Procedures and Functions

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

    #endregion Database Procedures and Functions
}
```

1-3 Внедрение базового интерфейса

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

    protected DbContext _dbContext;

    #endregion Protected Members

    #region Constractors

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

        ConfigureContext();
    }

    #endregion Constractors

    #region IRepository Implementation

    #region Tables and Views functions

    /// <summary>
    /// Query all
    /// Set noTracking to true for selecting only (read-only queries)
    /// Set noTracking to false for insert, update, or delete after select
```

```
    /// </summary>
    public virtual IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult :
class
    {
        var entityDbSet = GetDbSet<TResult>();

        if (noTracking)
            return entityDbSet.AsNoTracking();

        return entityDbSet;
    }

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

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

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

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

        return entity;
    }

    #endregion Tables and Views functions

    #region Transactions Functions

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

```
    /// <summary>
    /// Asynchronously saves all changes made in this context to the underlying database.
    /// </summary>
    /// <param name="cancellationToken">A System.Threading.CancellationToken to observe while
waiting for the task to complete.</param>
    /// <returns>A task that represents the asynchronous save operation.  The task result
contains the number of objects written to the underlying database.</returns>
    public virtual Task<int> CommitAsync(CancellationToken cancellationToken =
default(CancellationToken))
    {
        return _dbContext.SaveChangesAsync(cancellationToken);
    }

    #endregion Transactions Functions

    #region Database Procedures and Functions

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

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

    #endregion Database Procedures and Functions

    #endregion IRepository Implementation

    #region IDisposable Implementation

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

    #endregion IDisposable Implementation

    #region Protected Functions

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

    #endregion Protected Functions

    #region Private Functions

    private DbSet<TEntity> GetDbSet<TEntity>() where TEntity : class
    {
```

```
        return _dbContext.Set<TEntity>();
    }

    #endregion Private Functions

}
```

2- Entity Framework @ Бизнес-уровень

В этом слое мы напишем бизнес приложения.

Для каждого экрана презентации рекомендуется создать бизнес-интерфейс и класс реализации, которые содержат все необходимые функции для экрана.

Ниже мы будем писать бизнес для экрана продукта в качестве примера

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



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

    private IDbRepository _dbRepository;

    #endregion Private Members

    #region Constructors

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

    #endregion Constructors
```

```
    #region IProductBusiness Function

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

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

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

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

        return await result.ToListAsync();
    }

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

        await _dbRepository.CommitAsync();

        return newProduct;
    }

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

```
        var newProduct = _dbRepository.Add(new Product() { ProductName = productName, Category
= newCategory });

        _dbRepository.Commit();

        return newProduct;
    }

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

        _dbRepository.Commit();

        return product;
    }

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

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

        _dbRepository.Commit();

        return product;
    }

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

        return _dbRepository.Commit();
    }

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

        return _dbRepository.Commit();
    }

    /// <summary>
    /// Assuming we have the following procedure in database
    /// PROCEDURE [dbo].[GetProductsCategory] @CategoryID INT, @OrderBy VARCHAR(50)
```

```
    /// </summary>
    public IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId)
    {
        return
_dbRepository.Execute<IEnumerable<GetProductsCategory_Result>>("GetProductsCategory",
categoryId, "ProductName DESC");
    }


    #endregion IProductBusiness Function
}
```

3- Использование уровня бизнес-уровня @ Уровень представления (MVC)

В этом примере мы будем использовать уровень Business на уровне Presentation. И мы будем использовать MVC в качестве примера слоя Presentation (но вы можете использовать любой другой слой Presentation).

Нам нужно сначала зарегистрировать IoC (мы будем использовать Unity, но вы можете использовать любой IoC), а затем напишите наш слой Presentation

3-1 Регистрация типов Unity в MVC

3-1-1 Добавьте «Unity bootstrapper для ASP.NET MVC» NuGet backage

3-1-2 Добавьте UnityWebActivator.Start (); в файле Global.asax.cs (функция Application_Start ())

3-1-3. Изменить функцию UnityConfig.RegisterTypes следующим образом.

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

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

    }
```

3-2 Использование уровня бизнес-уровня @ Уровень представления (MVC)

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

    IProductBusiness _productBusiness;

    #endregion Private Members
```

```
    #region Constractors

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

    #endregion Constractors

    #region Action Functions

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

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

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

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

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

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

4- Entity Framework @ Единичный тестовый уровень

В слое Test Test мы обычно тестируем функциональные возможности Business Layer. И для этого мы удалим зависимости Data Layer (Entity Framework).

И теперь возникает вопрос: как я могу удалить зависимости Entity Framework для модульного тестирования функций Business Layer?

И ответ прост: мы будем поддельной реализацией интерфейса IDbRepository, тогда мы сможем выполнить наш модульный тест

4-1 Реализация базового интерфейса (поддельная реализация)

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

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

    #endregion Protected Members

    #region Constractors

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

    #endregion Constractors

    #region IRepository Implementation

    #region Tables and Views functions

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

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

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

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

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

        return entity;
    }

    #endregion Tables and Views functions

    #region Transactions Functions


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

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

    #endregion Transactions Functions

    #region Database Procedures and Functions

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

        throw new NotImplementedException();
    }

    #endregion Database Procedures and Functions

    #endregion IRepository Implementation

    #region IDisposable Implementation

    public void Dispose()
    {

    }

    #endregion IDisposable Implementation

    #region Private Functions
```

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

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

    #endregion Private Functions
}
```

4-2 Запустите тестирование устройства

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

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

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

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

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

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

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

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

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

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

```
}
```

Прочитайте Лучшие практики для платформы Entity (простой и профессиональный) онлайн: https://riptutorial.com/ru/entity-framework/topic/8879/лучшие-практики-для-платформы-entity--простой-и-профессиональный-

глава 12: Методы оптимизации в EF

Examples

Использование AsNoTracking

Плохой пример:

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

return location;
```

Поскольку приведенный выше код просто возвращает объект без изменения или добавления его, мы можем избежать затрат на отслеживание.

Хороший пример:

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

return location;
```

Когда мы используем функцию `AsNoTracking()` мы явно указываем Entity Framework, что сущности не отслеживаются контекстом. Это может быть особенно полезно при извлечении больших объемов данных из вашего хранилища данных. Если вы хотите вносить изменения в не отслеживаемые объекты, однако, вы должны помнить о том, чтобы присоединить их перед вызовом `SaveChanges` .

Загрузка только необходимых данных

Одной из проблем, часто встречающихся в коде, является загрузка всех данных. Это значительно увеличит нагрузку на сервер.

Предположим, у меня есть модель под названием «location», в которой есть 10 полей, но не все поля требуются одновременно. Предположим, мне нужен только параметр « LocationName» этой модели.

Плохой пример

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

return location.Name;
```

Хороший пример

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

return location;
```

Код в «хорошем примере» будет извлекать только «LocationName» и ничего больше.

Обратите внимание, что поскольку в этом примере не `AsNoTracking()` ни одно сущность, `AsNoTracking()` не требуется. В любом случае ничего не нужно отслеживать.

Получение полей с анонимными типами

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

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

Как и в предыдущем примере, из базы данных будут извлекаться только поля « LocationName» и «LocationArea», анонимный тип может содержать столько значений, которые вы хотите.

Выполнять запросы в базе данных, если это возможно, а не в памяти.

Предположим, мы хотим подсчитать, сколько уездов есть в Техасе:

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

Запрос верный, но неэффективный. `States.Single(…)` загружает состояние из базы данных. Далее, `Counties` загружают все 254 окружения со всеми полями во втором запросе. `.Count()` затем выполняется *в памяти* в загруженной коллекции `Counties` .
Мы загрузили много данных, которые нам не нужны, и мы можем сделать лучше:

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

Здесь мы делаем только один запрос, который в SQL преобразуется в счет и соединение. Мы возвращаем только счет из базы данных - мы сохранили возвращаемые строки, поля и создание объектов.

Легко видеть, где выполняется запрос, глядя на тип коллекции: `IQueryable<T>` против `IEnumerable<T>` .

Выполнять несколько запросов async и параллельно

При использовании асинхронных запросов вы можете выполнять несколько запросов одновременно, но не в том же контексте. Если время выполнения одного запроса составляет 10 с, время для плохого примера будет 20 с, а время для хорошего примера - 10 с.

Плохой пример

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

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

Хороший пример

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



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

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

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

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

Отключить отслеживание изменений и создание прокси

Если вы просто хотите получать данные, но ничего не модифицируете, вы можете отключить отслеживание изменений и создание прокси. Это улучшит вашу производительность, а также предотвратит ленивую загрузку.

Плохой пример:

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

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

Хороший пример:

```
using(var context = new Context())
{
    context.Configuration.AutoDetectChangesEnabled = false;
    context.Configuration.ProxyCreationEnabled = false;

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

Особенно часто их отключать от конструктора вашего контекста, особенно если вы хотите, чтобы они были настроены на ваше решение:

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

    //snip
}
```

Работа с объектами-заглушками

Предположим, что у нас есть категории `Product` и `Category` во взаимосвязи «многие-ко-многим»:

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

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

Если мы хотим добавить `Category` в `Product` , мы должны загрузить продукт и добавить категорию в свои `Categories` , например:

Плохой пример:

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

(где `db` - подкласс `DbContext`).

Это создает одну запись в таблице соединений между `Product` и `Category` . Однако эта таблица содержит только два значения `Id` . Это пустая трата ресурсов для загрузки двух полных объектов, чтобы создать одну крошечную запись.

Более эффективным способом является использование *объектов-заглушек* , то есть объектов сущности, созданных в памяти, содержащих только минимальный минимум данных, обычно только значение `Id` . Вот как это выглядит:

Хороший пример:

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

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

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

Конечный результат тот же, но он избегает двух обращений к базе данных.

Предотвращение дублирования

Вы хотите проверить, существует ли ассоциация, достаточно дешевого запроса. Например:

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

Опять же, это не будет загружать полные объекты в память. Он эффективно запрашивает таблицу соединений и возвращает только логическое значение.

Прочитайте Методы оптимизации в EF онлайн: https://riptutorial.com/ru/entity-framework/topic/2714/методы-оптимизации-в-ef

глава 13: Наследование с помощью EntityFramework (сначала код)

Examples

Таблица на иерархию

Этот подход будет генерировать одну таблицу в базе данных для представления всей структуры наследования.

Пример:

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

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

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

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

Созданная таблица будет выглядеть следующим образом:

Таблица: Люди Поля: Идентификатор Имя Дата рождения Дискримитратор Входная плата JobDescription LastPurchaseDate TotalVisits

Если «Дискриминатор» будет содержать имя подкласса в наследовании, а « AdmissionDate», «JobDescription», «LastPurchaseDate», «TotalVisits», являются нулевыми.

преимущества

- Лучшая производительность, поскольку соединения не требуются, хотя для многих столбцов для базы данных может потребоваться много операций подкачки.
- Простой в использовании и создании

- Легко добавлять дополнительные подклассы и поля

Недостатки

- Нарушает 3-ей нормальную форму Википедия: Третья нормальная форма
- Создает множество полей с нулевым значением

Таблица на тип

Этот подход будет генерировать (n + 1) таблицы в базе данных для представления всей структуры наследования, где n - количество подклассов.

Как:

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

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

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

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

Созданная таблица будет выглядеть следующим образом:

Таблица: Люди Поля: Идентификатор Имя Дата рождения

Таблица: Поля сотрудников: PersonId AdmissionDate JobDescription

Таблица: Клиенты: Поля: PersonId LastPurchaseDate TotalVisits

Где «PersonId» на всех таблицах будет основным ключом и ограничением для People.Id

преимущества

- Нормализованные таблицы

- Легко добавлять столбцы и подклассы
- Нет нулевых столбцов

Недостатки

- Для получения данных требуется соединение
- Вывод подкласса более дорогой

Прочитайте Наследование с помощью EntityFramework (сначала код) онлайн: https://riptutorial.com/ru/entity-framework/topic/7715/наследование-с-помощью-entityframework--сначала-код-

глава 14: Ограничения модели

Examples

Отношение «один ко многим»

UserType принадлежит многим пользователям <-> У пользователей есть один UserType

Одностороннее навигационное свойство с требуемым

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

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

Одностороннее свойство навигации с дополнительным (внешний ключ должен быть `Nullable`)

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

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

Двухстороннее свойство навигации с (обязательно / необязательно изменяет свойство внешнего ключа по мере необходимости)

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

```
    public int UserTypeId {get; set;}
    public virtual UserType UserType {get; set;}
}
```

необходимые

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

Необязательный

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

Прочитайте Ограничения модели онлайн: https://riptutorial.com/ru/entity-framework/topic/4528/ограничения-модели

глава 15: операции

Examples

Database.BeginTransaction ()

Для одной транзакции можно выполнить несколько операций, чтобы изменения можно было отбросить, если какая-либо из операций завершилась неудачей.

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

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

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

Обратите внимание, что это может быть соглашение разработчиков о вызове `transaction.Rollback()` явно, потому что это делает код более понятным. Кроме того, *могут* существовать (менее известные) поставщики запросов для Entity Framework, которые не реализуют `Dipsose` правильно, что также потребует явного вызова `transaction.Rollback()` .

Прочитайте операции онлайн: https://riptutorial.com/ru/entity-framework/topic/4944/операции

глава 16: Отслеживание и отсутствие отслеживания

замечания

Отслеживание поведения контролирует, будет ли Entity Framework хранить информацию об экземпляре сущности в своем трекере изменения. Если объект отслеживается, любые изменения, обнаруженные в объекте, будут сохраняться в базе данных во время `SaveChanges()` .

Examples

Отслеживание запросов

- По умолчанию запросы, возвращающие типы **объектов, отслеживаются**
- Это означает, что вы можете вносить изменения в эти экземпляры сущности и сохранять эти изменения в `SaveChanges()`

Пример :

- Изменения в рейтинге `book` будут обнаружены и сохранены в базе данных во время `SaveChanges()` .

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

Запросы без отслеживания

- Запросы на отслеживание не полезны, когда результаты используются в сценарии `read-only` для `read-only`
- Они `quicker to execute` потому что нет необходимости настраивать информацию отслеживания изменений

Пример :

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

С EF Core 1.0 вы также можете изменить поведение отслеживания по умолчанию на уровне `context instance` .

Пример :

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

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

Отслеживание и прогнозы

- Даже если тип результата запроса не является типом сущности, если результат `contains entity` типы `contains entity` они по-прежнему будут `tracked by default`

Пример :

- В следующем запросе, который возвращает `anonymous type` , `will be tracked` экземпляры `Book` в результирующем наборе

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

- Если результат `does not` содержит каких - либо `entity` типа, то `no tracking` не выполняется

Пример :

- В следующем запросе, который возвращает `anonymous type` с некоторыми значениями из объекта (но `no instances` фактического типа `entity`), **отслеживание не** выполняется.

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

Прочитайте Отслеживание и отсутствие отслеживания онлайн: https://riptutorial.com/ru/entity-framework/topic/6836/отслеживание-и-отсутствие-отслеживания

глава 17: Первичные миграции кода сущностей

Examples

Включить миграцию

Чтобы включить первичные миграции кода в инфраструктуре сущности, используйте команду

```
Enable-Migrations
```

на *консоли диспетчера пакетов* .

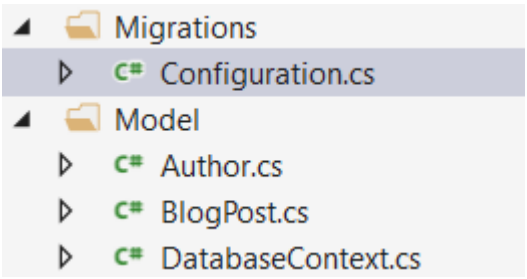
Вы должны иметь действительную реализацию `DbContext` содержащую объекты базы данных, управляемые EF. В этом примере контекст базы данных будет содержать объекты `BlogPost` и `Author` :

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

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

После выполнения команды должен появиться следующий вывод:

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

Кроме того, новая папка `Migrations` должна появиться с одним файлом `Configuration.cs` внутри:



Следующим шагом будет создание первого скрипта миграции базы данных, который создаст исходную базу данных (см. Следующий пример).

Добавьте первую миграцию

После того, как вы включили миграции (см. Этот пример), теперь вы можете создать первую миграцию, содержащую первоначальное создание всех таблиц базы данных, индексов и соединений.

Миграция может быть создана с помощью команды

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

Эта команда создаст новый класс, содержащий два метода `Up` и `Down` , которые используются для применения и удаления миграции.

Теперь примените команду, основанную на примере выше, чтобы создать миграцию с именем *Initial* :

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

Создается новая *временная метка* файла _Initial.cs (здесь показан только важный материал):

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

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

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

```
    DropTable("dbo.Authors");
}
```

Как вы можете видеть, в методе `Up()` создаются две таблицы `Authors` и `BlogPosts` и `BlogPosts` создаются поля. Кроме того, связь между двумя таблицами создается путем добавления поля `Author_AuthorId` . С другой стороны метод `Down()` пытается отменить действия миграции.

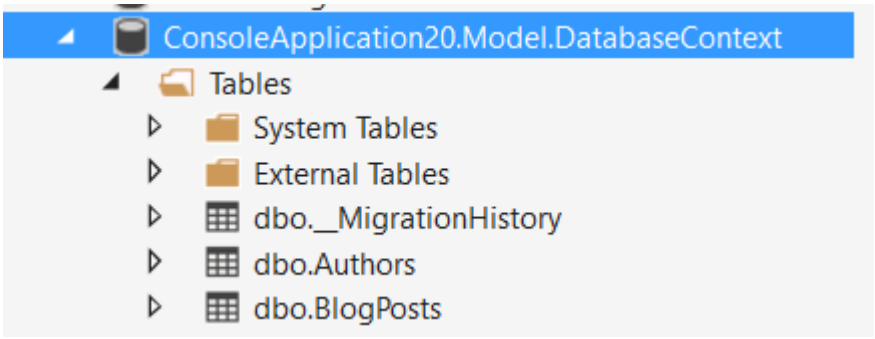
Если вы уверены в своей миграции, вы можете применить миграцию к базе данных с помощью команды:

```
Update-Database
```

Все ожидающие миграции (в этом случае *Initial* -migration) применяются к базе данных, а затем применяется метод семени (соответствующий пример)

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

Вы можете увидеть результаты действий в SQL Explorer:



Для команд `Add-Migration` и `Update-Database` доступны несколько параметров, которые можно использовать для настройки действий. Чтобы просмотреть все варианты, используйте

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

а также

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

Посещение данных во время миграции

После включения и создания миграций может потребоваться первоначальное заполнение или миграция данных в вашей базе данных. Существует несколько возможностей, но для простых перемещений вы можете использовать метод Seed () в файле Configuration, созданный после вызова `enable-migrations` .

Функция `Seed()` извлекает контекст базы данных, поскольку это единственный параметр, и вы можете выполнять операции EF внутри этой функции:

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

Вы можете выполнять все виды действий внутри `Seed()` . В случае сбоя полная транзакция (даже исправленные исправления) откат.

Примерная функция, которая создает данные только в том случае, если таблица пуста, может выглядеть так:

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

`AddOrUpdate()` функцией, предоставляемой разработчиками EF, является метод расширения `AddOrUpdate()` . Этот метод позволяет обновлять данные на основе первичного ключа или вставлять данные, если он еще не существует (пример берется из сгенерированного исходного кода Configuration.cs):

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

> Имейте в виду, что `Seed()` вызывается после применения **последнего** патча. Если вам нужно выполнить миграцию или данные семян во время патчей, необходимо использовать другие подходы.

Использование Sql () во время миграции

Например: вы собираетесь перенести существующий столбец из необязательного требуемого. В этом случае вам может потребоваться заполнить некоторые значения по умолчанию в вашей миграции для строк, где измененные поля фактически являются `NULL` . Если значение по умолчанию простое (например, «0»), вы можете использовать свойство `default` или `defaultSql` в определении столбца. Если это не так просто, вы можете использовать функцию `Sql()` функциях члена `Up()` или `Down()` ваших миграций.

Вот пример. Предполагая класс *Author,* который содержит адрес электронной почты как

часть набора данных. Теперь мы решили указать адрес электронной почты как обязательное поле. Чтобы перенести существующие столбцы, у бизнеса есть *умная* идея создания фиктивных адресов электронной почты, таких как `fullname@example.com` , где полное имя - полное имя авторов без пробелов. Добавление атрибута `[Required]` в поле « `Email` создало бы следующую миграцию:

```
public partial class AuthorsEmailRequired : DbMigration
{
    public override void Up()
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(nullable: false, maxLength: 512));
    }

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

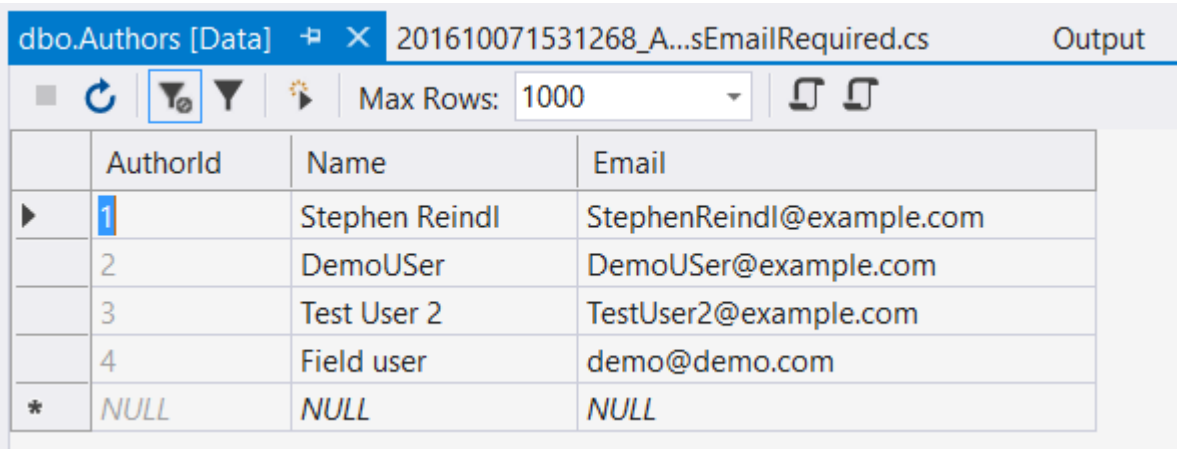
Это может привести к сбою в случае, если в базу данных находятся некоторые поля NULL:

> Невозможно вставить значение NULL в столбец «Электронная почта», таблица «App.Model.DatabaseContext.dbo.Authors»; столбец не допускает нулей. Ошибка UPDATE.

Добавьте следующее, как **до** того, `AlterColumn` команда `AlterColumn` поможет:

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

Вызов службы `update-database` преуспевает, и таблица выглядит так (пример показывает данные):



Другое использование

Вы можете использовать функцию `Sql()` для всех типов действий DML и DDL в вашей базе данных. Он выполняется как часть транзакции миграции; Если SQL сбой, полная миграция завершается с ошибкой и выполняется откат.

Выполнение «Обновить-База данных» в вашем коде

Приложениям, работающим в средах без разработки, часто требуется обновление базы данных. После использования команды `Add-Migration` для создания патчей базы данных необходимо запустить обновления в других средах, а затем и в тестовой среде.

Обычно возникают проблемы:

- нет Visual Studio, установленной в производственных средах, и
- никакие подключения не разрешены для подключения / среды клиентов в реальной жизни.

Обходной путь - это следующая последовательность кода, которая проверяет наличие обновлений и выполняет их по порядку. Пожалуйста, обеспечьте правильную обработку транзакций и исключений, чтобы гарантировать, что данные не будут потеряны в случае ошибок.

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

где `MyDbConfiguration` - это ваша настройка миграции где-нибудь в ваших источниках:

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

Первоначальная схема первичной структуры Entity Framework Шаг за шагом

1. Создайте консольное приложение.
2. Установите пакет EntityFramework nuget, запустив `Install-Package EntityFramework` в «Консоль диспетчера пакетов»
3. Добавьте строку подключения в файл app.config. Важно, чтобы в вашем соединении было включено имя `providerName="System.Data.SqlClient"` .
4. Создайте публичный класс, как хотите, что-то вроде « `Blog` »,
5. Создайте свой ContextClass, который наследуется от DbContext, что-то вроде « `BlogContext` »,
6. Определите свойство в вашем контексте типа DbSet, что-то вроде этого:

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

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

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

7. Важно передать имя соединения в конструкторе (здесь Your_Connection_Name)
8. В консоли диспетчера пакетов запустите команду `Enable-Migration` , это создаст папку переноса в вашем проекте
9. Запустите команду `Add-Migration Your_Arbitrary_Migraiton_Name` , это создаст класс миграции в папке миграции с двумя методами Up () и Down ()
10. Запустите команду « `Update-Database` , чтобы создать базу данных с таблицей блога

Прочитайте Первичные миграции кода сущностей онлайн: https://riptutorial.com/ru/entity-framework/topic/7157/первичные-миграции-кода-сущностей

глава 18: Первоначальный код сущности

Examples

Подключение к существующей базе данных

Для достижения простейшей задачи в Entity Framework - для подключения к существующей базе данных `ExampleDatabase` на вашем локальном экземпляре MSSQL вам нужно реализовать только два класса.

Сначала это класс сущности, который будет сопоставлен с нашей таблицей базы данных `dbo.People` .

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

Класс будет использовать соглашения Entity Framework и отобразить таблицу `dbo.People` которая, как ожидается, будет иметь первичный ключ `PersonId` и `PersonId` varchar (max) `FirstName` .

Во-вторых, это класс контекста, который происходит из `System.Data.Entity.DbContext` и который будет управлять объектами сущности во время выполнения, укомплектовывать их из базы данных, обрабатывать параллелизм и сохранять их обратно в базу данных.

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

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

Имейте в виду, что в конструкторе нашего контекста нам нужно установить инициализатор базы данных на нуль - мы не хотим, чтобы Entity Framework создавал базу данных, мы просто хотим получить к ней доступ.

Теперь вы можете манипулировать данными из этой таблицы, например, изменить `FirstName` первого лица в базе данных из консольного приложения следующим образом:

```
class Program
{
    static void Main(string[] args)
```

```
        {
            using (var ctx = new Context("DbConnectionString"))
            {
                var firstPerson = ctx.People.FirstOrDefault();
                if (firstPerson != null) {
                    firstPerson.FirstName = "John";
                    ctx.SaveChanges();
                }
            }
        }
    }
```

В приведенном выше коде мы создали экземпляр Context с аргументом « DbConnectionString». Это должно быть указано в нашем файле `app.config` следующим образом:

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

Прочитайте Первоначальный код сущности онлайн: https://riptutorial.com/ru/entity-framework/topic/5337/первоначальный-код-сущности

глава 19: Первые соглашения

замечания

Конвенция представляет собой набор правил по умолчанию для автоматической настройки концептуальной модели на основе определений классов домена при работе с Code-First. Соглашения Code-First определены в *пространстве* имен *System.Data.Entity.ModelConfiguration.Conventions* (EF 5 и EF 6).

Examples

Конвенция о первичном ключе

По умолчанию свойство является первичным ключом, если свойство класса называется « ID» (не чувствительным к регистру) или имя класса, за которым следует «ID». Если тип свойства первичного ключа является числовым или GUID, он будет настроен как столбец идентификатора. Простой пример:

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

Удаление конвенций

Вы можете удалить любое из соглашений, определенных в пространстве имен System.Data.Entity.ModelConfiguration.Conventions, путем переопределения метода `OnModelCreating` .

В следующем примере удаляется PluralizingTableNameConvention.

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

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

По умолчанию EF создаст таблицу БД с именем класса сущности, суффикс которой 's'. В этом примере, Code First настроен игнорировать PluralizingTableName соглашение так,

вместо `dbo.Products` таблицы `dbo.Product` таблица будет создана.

Тип обнаружения

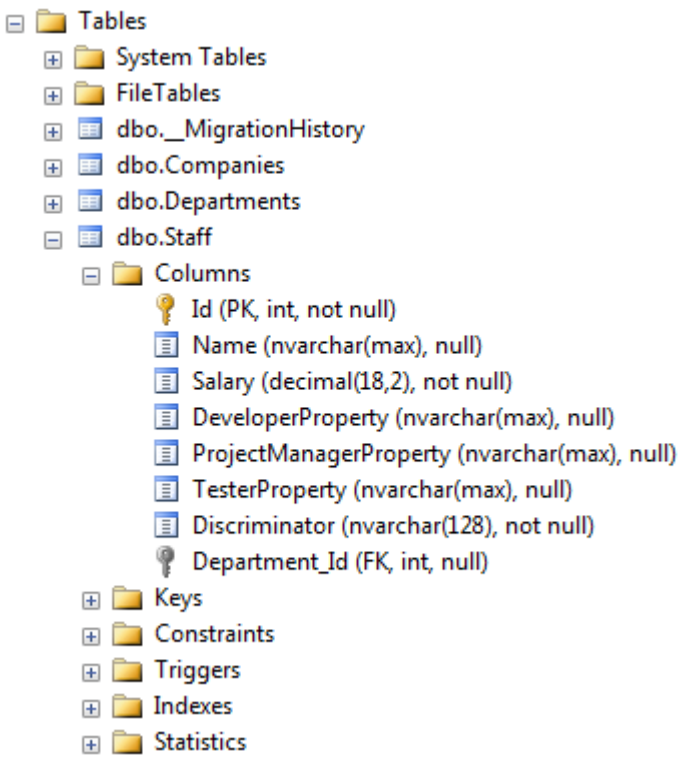
По умолчанию Code First включает в себя модель

1. Типы, определенные как свойство DbSet в классе контекста.
2. Типы ссылок, включенные в типы сущностей, даже если они определены в разных сборках.
3. Производные классы, даже если только базовый класс определяется как свойство DbSet

Вот пример, что мы добавляем `Company` только как `DbSet<Company>` в наш контекстный класс:

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

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

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

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

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

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

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

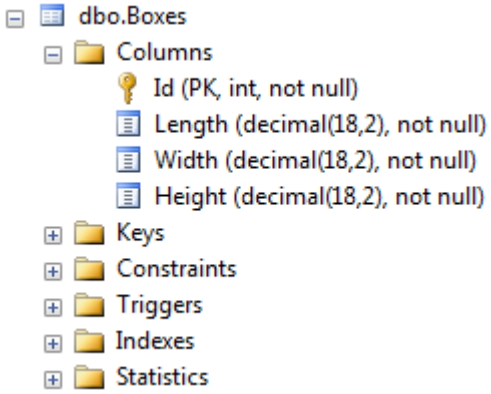
Мы видим, что все классы включены в модель



DecimalPropertyConvention

По умолчанию Entity Framework сопоставляет десятичные свойства десятичным (18,2) столбцам в таблицах базы данных.

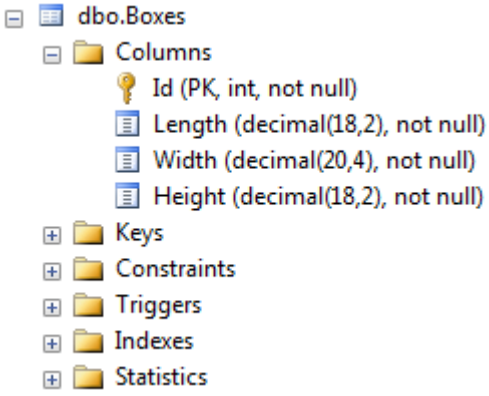
```
public class Box
{
    public int Id { set; get; }
    public decimal Length { set; get; }
    public decimal Width { set; get; }
    public decimal Height { set; get; }
}
```



Мы можем изменить точность десятичных свойств:

1. Использовать API Fluent:

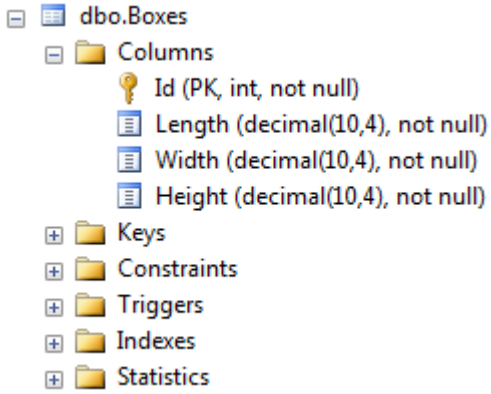
```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Box>().Property(b => b.Width).HasPrecision(20, 4);
}
```



Только свойство «Ширина» отображается десятичным (20, 4).

2. Замените соглашение:

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



Каждое десятичное свойство отображается в десятичные (10,4) столбцы.

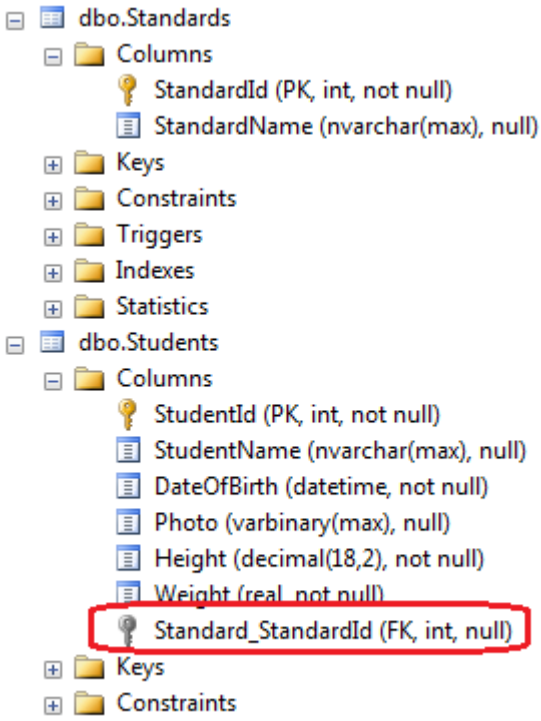
Конвенция об отношениях

Код Сначала выведите взаимосвязь между двумя объектами, использующими свойство навигации. Это свойство навигации может быть простым ссылочным типом или типом коллекции. Например, мы определили свойство стандартной навигации в классе Student и свойство навигации ICollection в стандартном классе. Таким образом, Code First автоматически создавал отношения «один ко многим» между таблицами стандартов и учеников, вставляя столбец внешнего ключа Standard_StandardId в таблицу «Студенты».

```
public class Student
```

```
{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime DateOfBirth { get; set; }

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

public class Standard
{

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

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

}
```

Вышеуказанные объекты создали следующее отношение, используя внешний ключ Standard_StandardId.



Конвенция о внешних ключах

Если класс A связан с классом B, а класс B имеет свойство с тем же именем и тип, что и первичный ключ A, тогда EF автоматически предполагает, что это свойство является внешним ключом.

```
public class Department
{
```

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

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

В этом случае DepartmentId является внешним ключом без явной спецификации.

Прочитайте Первые соглашения онлайн: https://riptutorial.com/ru/entity-framework/topic/2447/первые-соглашения

глава 20: Расширенные сценарии сопоставления: разбиение объектов, разбиение таблиц

Вступление

Как настроить модель EF для поддержки разделения объектов или разбиения таблиц.

Examples

Разделение сущностей

Итак, допустим, у вас есть класс сущности:

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

И затем предположим, что вы хотите сопоставить объект Person в двух таблицах: один с PersonId и Name, а другой с данными адреса. Конечно, вам также понадобится PersonId, чтобы определить, к кому принадлежит адрес. Таким образом, в основном вы хотите разделить объект на две (или даже более) части. Следовательно, имя, сущность разбивается. Вы можете сделать это, сопоставляя каждое из свойств с другой таблицей:

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

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

Это создаст две таблицы: People и PersonDetails. У человека есть два поля: PersonId и Name, PersonDetails имеет четыре столбца: PersonId, AddressLine, City и ZipCode. В People PersonId является основным ключом. В PersonDetails основным ключом также является PersonId, но он также является внешним ключом, ссылающимся на PersonId в таблице Person.

Если вы запросите People DbSet, EF сделает соединение в PersonIds, чтобы получить данные из обеих таблиц, чтобы заполнить сущности.

Вы также можете изменить название столбцов:

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

Это создаст одну и ту же структуру таблицы, но в таблице «Люди» вместо столбца «Имя» будет столбец «Имя пользователя», а в таблице PersonDetails вместо столбца PersonId будет создан ProprietorId.

Разделение таблиц

А теперь предположим, что вы хотите сделать противоположное от разделения объектов: вместо сопоставления одного объекта в две таблицы вам нужно сопоставить одну таблицу с двумя объектами. Это называется разбиением таблиц. Допустим, у вас есть одна таблица с пятью столбцами: PersonId, Name, AddressLine, City, ZipCode, где PersonId является первичным ключом. И тогда вы хотели бы создать EF-модель следующим образом:

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

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

```
  public Person Person { get; set; }
}
```

Одна вещь выпрыгивает прямо: в Address Address отсутствует адрес. Это связано с тем, что два объекта отображаются в одну и ту же таблицу, поэтому они должны иметь одинаковый первичный ключ. Если вы разделите таблицу, это то, с чем вам просто нужно иметь дело. Поэтому, помимо разделения таблиц, вам также необходимо настроить объект Address и указать первичный ключ. И вот как:

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

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

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

Прочитайте Расширенные сценарии сопоставления: разбиение объектов, разбиение таблиц онлайн: https://riptutorial.com/ru/entity-framework/topic/9362/расширенные-сценарии-сопоставления--разбиение-объектов--разбиение-таблиц

глава 21: Сопоставление отношений с Entity Framework Code Сначала: «один ко многим» и «многие ко многим»

Вступление

В этом разделе обсуждается, как вы можете сопоставить отношения «один ко многим» и «многие ко многим», используя First Entity Framework Code First.

Examples

Отображение «один ко многим»

Итак, допустим, у вас есть две разные сущности, что-то вроде этого:

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

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

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

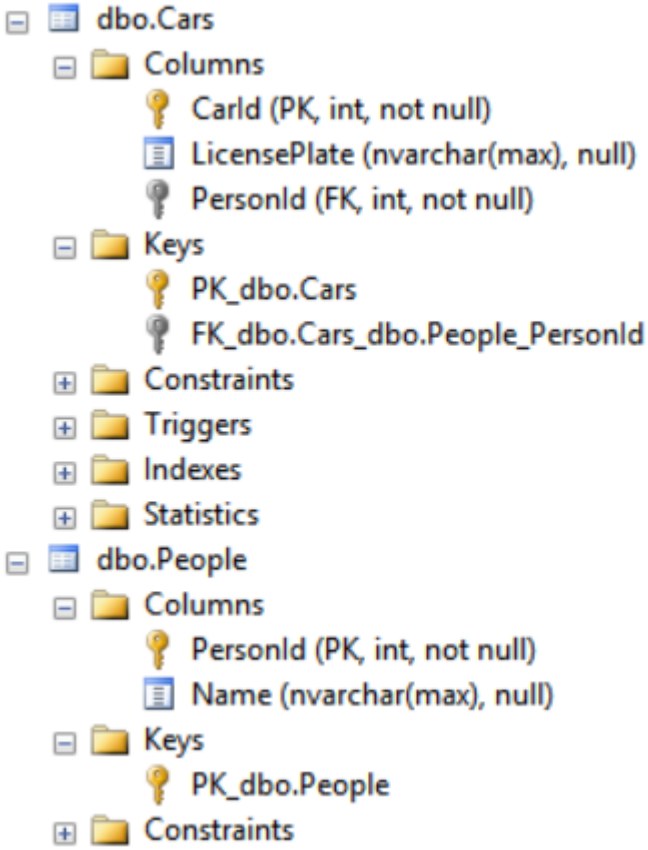
И вы хотите настроить отношения «один ко многим» между ними, то есть один человек может иметь ноль, один или несколько автомобилей, и один автомобиль принадлежит одному человеку точно. Каждое отношение является двунаправленным, поэтому, если у человека есть автомобиль, автомобиль принадлежит этому лицу.

Для этого просто измените классы моделей:

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

```
}

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

И это все :) У вас уже настроены ваши отношения. Конечно, в базе данных это внешние ключи.



Сопоставление «один ко многим»: против конвенции

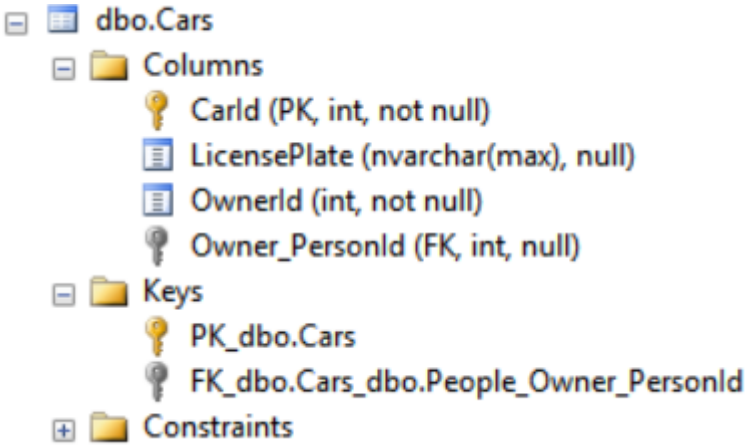
В последнем примере вы можете видеть, что EF определяет, какой столбец является внешним ключом и где он должен указывать. Как? Используя соглашения. Наличие свойства типа `Person` , которое называется `Person` с свойством `PersonId` приводит EF к выводу, что `PersonId` является внешним ключом и указывает первичный ключ таблицы, представленной типом `Person` .

Но что делать, если вы изменили *PersonId* на *OwnerId* и *Person* to *Owner* в типе **автомобиля** ?

```
public class Car
{
   public int CarId { get; set; }
```

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

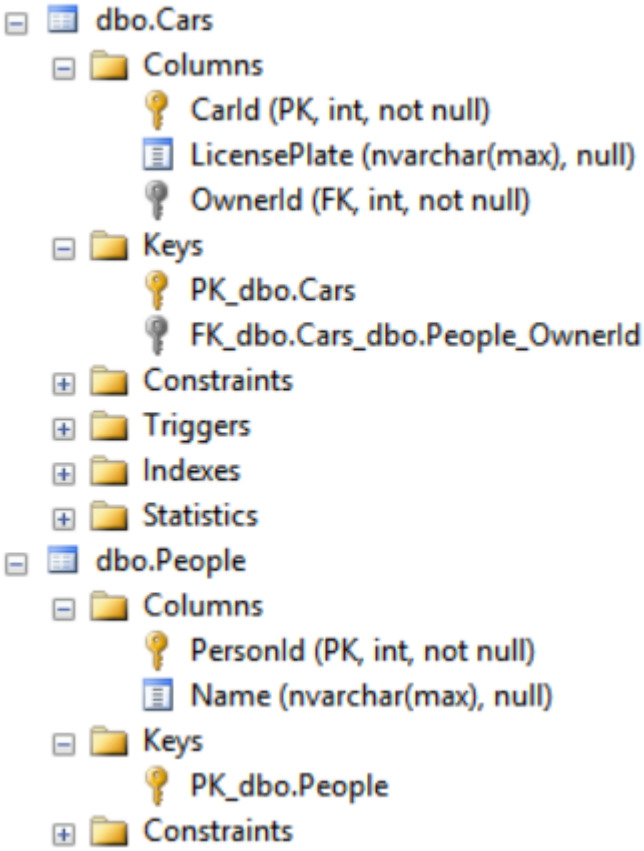
Ну, к сожалению, в этом случае соглашений недостаточно для создания правильной схемы БД:



Не волнуйтесь; вы можете помочь EF с некоторыми подсказками о ваших отношениях и ключах в модели. Просто настройте свой тип `Car` чтобы использовать свойство `OwnerId` как FK. Создайте конфигурацию типа сущности и примените ее в свой `OnModelCreating()` :

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

В основном это говорит о том, что `Car` имеет требуемое свойство, `Owner` (*HasRequired ()*) и в типе `Owner` , свойство `Cars` используется для возврата к *автообъектам (WithMany ()*). И, наконец, указывается свойство, представляющее внешний ключ (*HasForeignKey ()*). Это дает нам схему, которую мы хотим:



Вы также можете настроить отношения со стороны `Person` :

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

Идея такая же, только стороны разные (обратите внимание, как вы можете прочитать все это: «у этого человека много автомобилей, каждый автомобиль с требуемым владельцем»). Не имеет значения, настроите ли вы отношения со стороны « `Person` или « `Car` . Вы можете даже включить оба, но в этом случае будьте осторожны, чтобы указать те же отношения с обеих сторон!

Отображение нуля или один-ко-многим

В предыдущих примерах автомобиль не может существовать без человека. Что, если вы хотите, чтобы человек был необязательным со стороны автомобиля? Ну, это легко, зная, как делать «один ко многим». Просто измените `PersonId` в `Car` на значение NULL:

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

```
    public virtual Person Person { get; set; }
}
```

И затем используйте *HasOptional ()* (или *WithOptional ()* , в зависимости от того, с какой стороны вы выполните настройку):

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

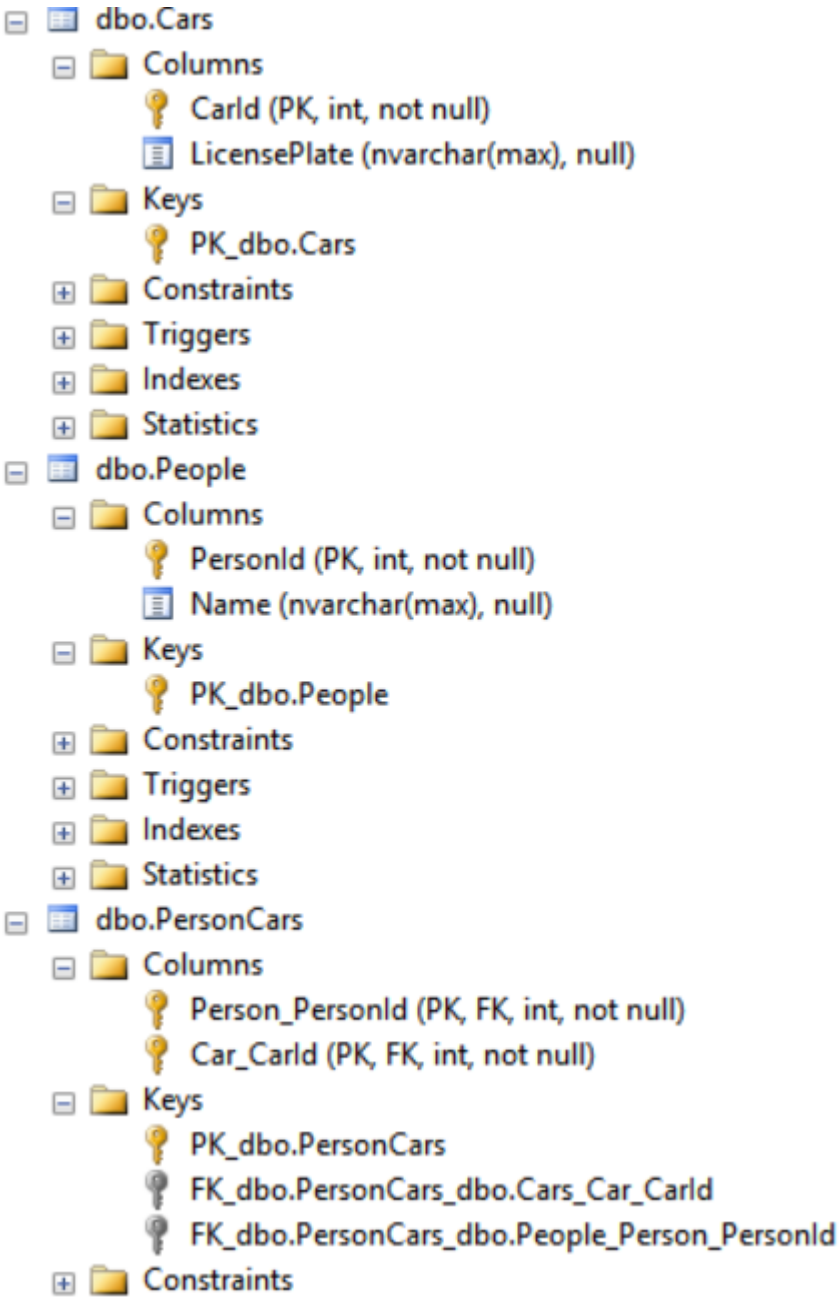
Многие-ко-многим

Перейдем к другому сценарию, где каждый человек может иметь несколько автомобилей, и каждый автомобиль может иметь несколько владельцев (но опять-таки, отношение двунаправлено). Это отношения «многие ко многим». Самый простой способ - позволить EF делать это с помощью условностей.

Просто измените модель следующим образом:

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

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

И схема:



Практически идеально. Как вы можете видеть, EF признал необходимость подключения таблицы, где вы можете отслеживать парные пары.

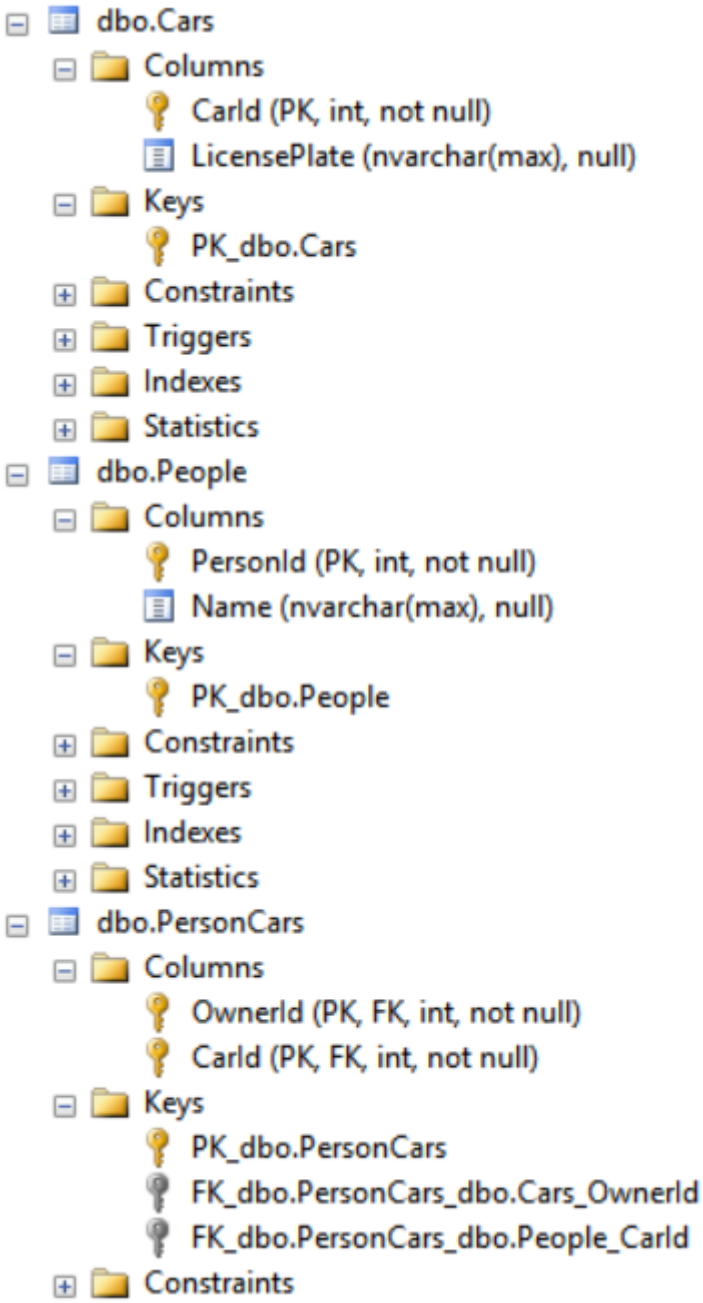
«Множество ко многим»: настройка таблицы соединений

Возможно, вы захотите переименовать поля в таблице соединений, чтобы быть немного более дружелюбными. Вы можете сделать это, используя обычные методы настройки (опять же, неважно, с какой стороны вы выполняете настройку):

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

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

Довольно легко читается: у этого автомобиля много владельцев (*HasMany ()*), причем каждый владелец имеет много автомобилей (*WithMany ()*). Сопоставьте это так, чтобы вы наложили левую клавишу на OwnerId (*MapLeftKey ()*), правую клавишу на CarId (*MapRightKey ()*) и все это на таблицу PersonCars (*ToTable ()*). И это дает вам именно эту схему:



«Множество ко многим»: настраиваемый элемент объединения

Я должен признать, что я не являюсь поклонником позволить EF вывести таблицу

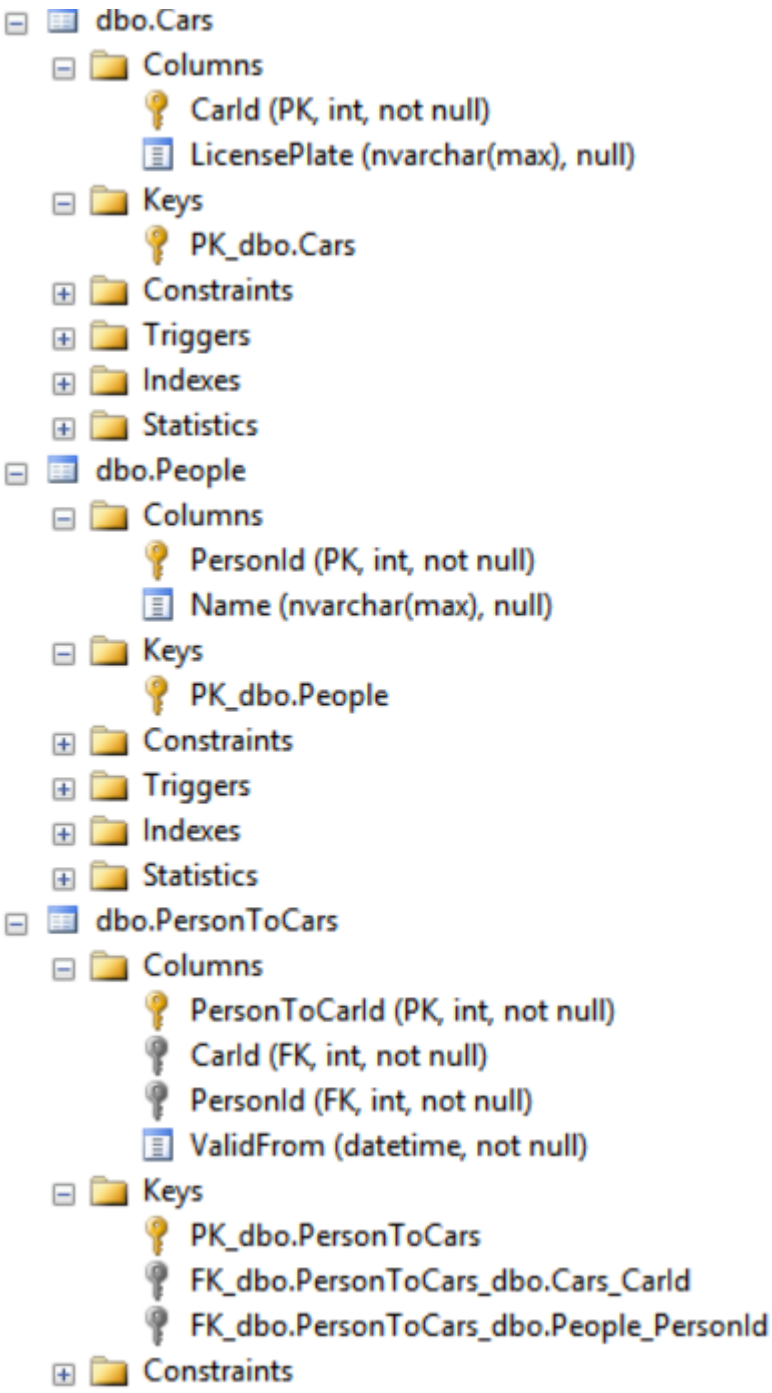
соединений с объединенной сущностью. Вы не можете отслеживать дополнительную информацию в ассоциации с персоналом (допустим, дата, с которой она действительна), потому что вы не можете изменить таблицу.

Кроме того, `CarId` в таблице соединений является частью первичного ключа, поэтому, если семья покупает новый автомобиль, вы должны сначала удалить старые ассоциации и добавить новые. EF скрывает это от вас, но это означает, что вам нужно выполнить эти две операции вместо простого обновления (не говоря уже о том, что частые вставки / удаления могут привести к фрагментации индекса - хорошо, что для этого есть легкое решение).

В этом случае вы можете создать объект объединения, который имеет ссылку как на один конкретный автомобиль, так и на одного конкретного человека. В основном вы смотрите на свою ассоциацию «многие-ко-многим» как комбинации двух ассоциаций «один-ко-многим»:

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

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

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

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

Это дает мне гораздо больший контроль, и это намного более гибко. Теперь я могу добавить пользовательские данные в ассоциацию, и каждая ассоциация имеет свой собственный первичный ключ, поэтому я могу обновить ссылку на автомобиль или владельца.



Обратите внимание, что это действительно просто комбинация двух отношений «один-ко-многим», поэтому вы можете использовать все параметры конфигурации, описанные в предыдущих примерах.

Прочитайте Сопоставление отношений с Entity Framework Code Сначала: «один ко многим» и «многие ко многим» онлайн: https://riptutorial.com/ru/entity-framework/topic/9413/сопоставление-отношений-с-entity-framework-code-сначала---один-ко-многим--и--многие-ко-многим-

глава 22: Сопоставление отношений с Entity Framework Code Сначала: индивидуально и вариации

Вступление

В этом разделе обсуждается, как сопоставить отношения типа один к одному с помощью Entity Framework.

Examples

Отображение от одного до нуля или одного

Итак, скажем еще раз, что у вас есть следующая модель:

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

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

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

И теперь вы хотите настроить его так, чтобы вы могли выразить следующую спецификацию: у одного человека может быть один или нулевой автомобиль, и каждый автомобиль принадлежит одному человеку точно (отношения двунаправленные, поэтому, если CarA принадлежит PersonA, то PersonA 'принадлежит «CarA».

Итак, немного изменим модель: добавьте свойства навигации и свойства внешнего ключа:

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

```
}

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

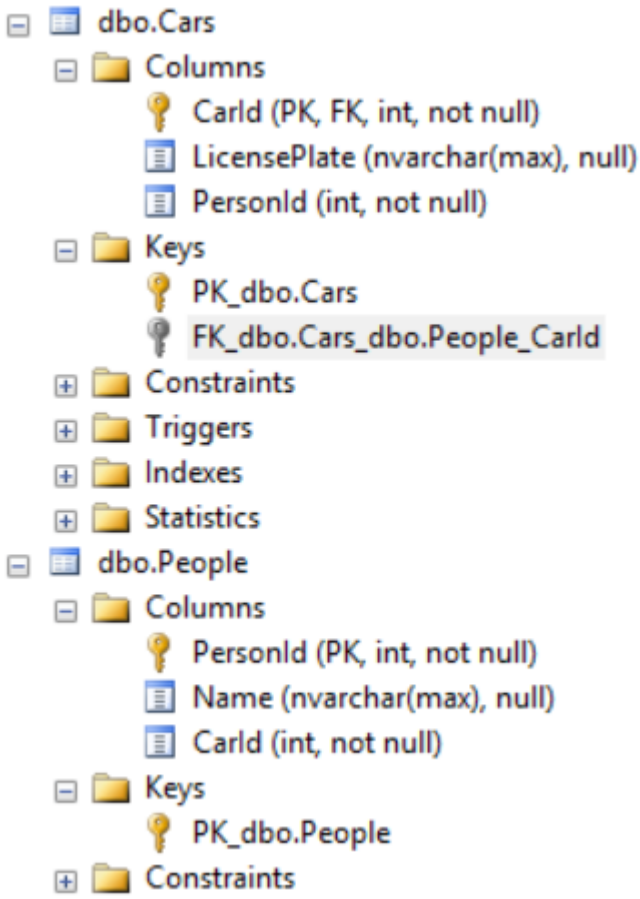
И конфигурация:

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

К этому времени это должно быть само собой разумеющимся. У автомобиля есть необходимый человек (*HasRequired ()*), с лицом, имеющим дополнительный автомобиль (*WithOptional ()*). Опять же, не имеет значения, с какой стороны вы настраиваете это отношение, просто будьте осторожны, когда вы используете правильную комбинацию Has / With и Required / Optional. Со стороны « `Person` это будет выглядеть так:

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

Теперь давайте рассмотрим схему db:



Посмотрите внимательно: вы можете видеть, что FK в `People` упоминает `Car` . Кроме того, FK в `Car` - это не `PersonId` , а `CarId` . Вот сценарий для FK:

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

Таким образом, это означает, что свойства ключа `CarId` и `PersonId` foregn, которые мы имеем в модели, в основном игнорируются. Они находятся в базе данных, но они не являются внешними ключами, как и следовало ожидать. Это связано с тем, что сопоставления «один к одному» не поддерживают добавление FK в вашу EF-модель. И это потому, что сопоставление «один к одному» довольно проблематично в реляционной базе данных.

Идея состоит в том, что каждый человек может иметь ровно одну машину, и этот автомобиль может принадлежать только этому человеку. Или могут быть записи людей, у которых нет автомобилей, связанных с ними.

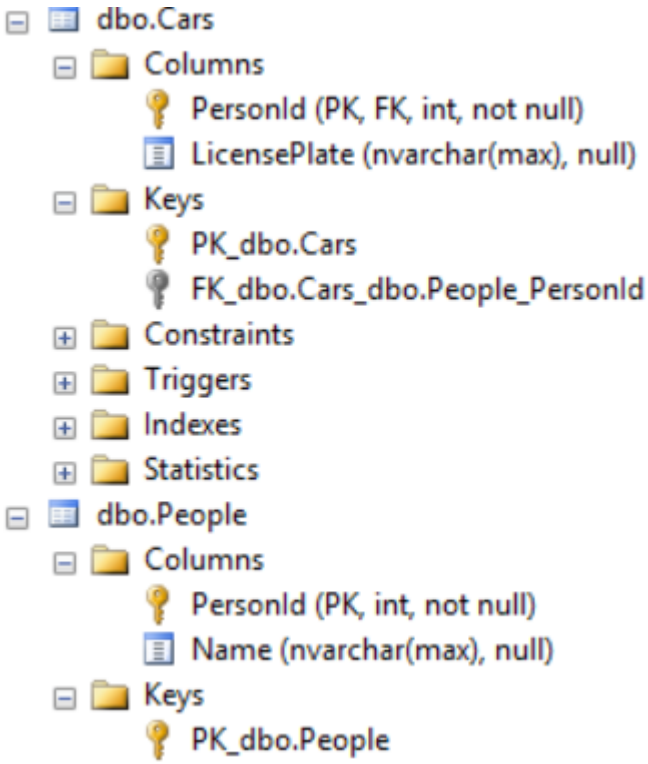
Итак, как это можно представить с помощью внешних ключей? Очевидно, что в `Car` может быть `PersonId` и `CarId` в `People` . Чтобы обеспечить, чтобы каждый человек мог иметь только один автомобиль, `PersonId` должен был быть уникальным в `Car` . Но если `PersonId` уникален в `People` , то как вы можете добавить две или более записи, где `PersonId` - `NULL` (более одного автомобиля, у которого нет владельцев)? Ответ: вы не можете (ну, на самом деле, вы можете создать отфильтрованный уникальный индекс в SQL Server 2008 и новее, но давайте забудем об этой техничности на мгновение, не говоря уже о других СУБД). Не

говоря уже о том, где вы указываете оба конца отношений ...

Единственный реальный способ обеспечить соблюдение этого правила, если таблицы `People` и `Car` имеют «тот же» первичный ключ (те же значения в подключенных записях). И для этого `CarId` in `Car` должен быть как ПК, так и FK для ПК людей. И это делает всю схему беспорядок. Когда я использую это, я скорее `PersonId` PK / FK в `Car` `PersonId` и настраиваю его следующим образом:

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

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

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

Не идеальный, но, может быть, немного лучше. Тем не менее, вы должны быть настороже при использовании этого решения, потому что это противоречит обычным соглашениям об именах, которые могут сбить вас с пути. Вот схема, сгенерированная из этой модели:



Таким образом, эта связь не применяется схемой базы данных, а сама Entity Framework. Вот почему вы должны быть очень осторожны, когда используете это, а не позволять кому-либо работать непосредственно с базой данных.

Отображение взаимно однозначного

Сопоставление «один к одному» (когда обе стороны требуются) также является сложной задачей.

Представим себе, как это можно представить с помощью внешних ключей. Опять же, `CarId` в `People` который относится к `CarId` в `Car` , и `PersonId` в автомобиле, который относится к `PersonId` в `People` .

Теперь, что произойдет, если вы хотите вставить автомобильную пластинку? Чтобы это было успешным, в этой автомобильной записи должен быть указан `PersonId` , потому что это необходимо. И для `PersonId` чтобы этот `PersonId` был действительным, соответствующая запись в `People` должна существовать. Хорошо, давайте продолжим и вставьте запись человека. Но для этого, чтобы добиться успеха, действительный `CarId` должен быть в записи человека, но этот автомобиль еще не вставлен! Этого не может быть, потому что мы должны сначала вставить запись упомянутого лица. Но мы не можем вставить запись упомянутого лица, потому что она относится к записи автомобиля, поэтому ее необходимо вставить сначала (внешний ключ-цессионал :)).

Таким образом, это не может быть представлено «логическим» способом. Опять же, вам нужно сбросить один из внешних ключей. Который вы бросаете, зависит от вас. Сторона, которая остается с внешним ключом, называется «зависимой», сторона, которая остается без внешнего ключа, называется «основной». И снова, чтобы гарантировать уникальность

в зависимом, ПК должен быть FK, поэтому добавление столбца FK и импорт его в вашу модель не поддерживаются.

Итак, вот конфигурация:

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

К настоящему времени вы действительно должны были получить логику этого :) Просто помните, что вы также можете выбрать другую сторону, просто будьте осторожны, чтобы использовать версии WithRequired Dependent / Principal (и вам все равно нужно настроить PK в автомобиле).

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

Если вы проверите схему DB, вы обнаружите, что она точно такая же, как и в случае решения «один к одному» или «нуль». Это потому, что опять же, это не выполняется схемой, а самой EF. Так что, будьте осторожны :)

Отображение одного или нуля к одному или нуля

И чтобы закончить, давайте вкратце рассмотрим случай, когда обе стороны являются необязательными.

К настоящему моменту вам должно быть очень скучно с этими примерами :), поэтому я не буду вдаваться в подробности и играть с идеей наличия двух FK-ов и потенциальных проблем и предупреждать вас об опасностях не применять эти правила в схемы, но только в самом EF.

Вот конфигурация, которую вам нужно применить:

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

```
}
```

Опять же, вы можете настроить и с другой стороны, просто будьте осторожны, чтобы использовать правильные методы :)

Прочитайте Сопоставление отношений с Entity Framework Code Сначала: индивидуально и вариации онлайн: https://riptutorial.com/ru/entity-framework/topic/9412/сопоставление-отношений-с-entity-framework-code-сначала--индивидуально-и-вариации

глава 23: Состояние управляющего объекта

замечания

Объекты в Entity Framework могут иметь различные состояния, которые перечислены в перечислении System.Data.Entity.EntityState . Этими состояниями являются:

```
Added
Deleted
Detached
Modified
Unchanged
```

Entity Framework работает с POCOs. Это означает, что объекты - это простые классы, которые не имеют свойств и методов для управления своим собственным состоянием. Состояние `ObjectStateManager` управляется самим контекстом в `ObjectStateManager` .

В этом разделе рассматриваются различные способы установки состояния объекта.

Examples

Состояние установки Добавлен один объект

`EntityState.Added` может быть установлен двумя полностью эквивалентными способами:

1. Установив состояние его записи в контексте:

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

2. Добавив его в `DbSet` контекста:

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

При вызове `SaveChanges` объект будет вставлен в базу данных. Когда у него есть столбец идентификатора (автоматически заданный, автоматически увеличивающий первичный ключ), то после `SaveChanges` свойство первичного ключа объекта будет содержать вновь сгенерированное значение, *даже если это свойство уже имеет значение* .

Состояние установки Добавлен граф объекта

Установка состояния *графа объекта* (набора связанных объектов) в `Added` отличается от установки отдельного объекта как `Added` (см. Этот пример).

В этом примере мы храним планеты и их луны:

Модель класса

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

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

контекст

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

Мы используем экземпляр этого контекста для добавления планет и их спутников:

пример

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

context.Planets.Add(mars);

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

Выход:

```
Added
Added
```

Мы видим здесь, что добавление `Planet` также добавляет состояние луны в `Added` .

При настройке состояния объекта на `Added` все объекты в своих свойствах навигации (свойства, которые «перемещаются» по отношению к другим объектам, например `Planet.Moons`) также помечены как `Added` , *если они уже не привязаны к контексту* .

Прочитайте Состояние управляющего объекта онлайн: https://riptutorial.com/ru/entity-framework/topic/5256/состояние-управляющего-объекта

кредиты

S. No	Главы	Contributors
1	Начало работы с Entity Framework	Adil Mammadov, Community, DavidG, Eldho, Jacob Linney, Martin4ndersen, Matas Vaitkevicius, Nasreddine, NovaDev, Parth Patel, Stephen Reindl, tmg
2	.t4 шаблонов в инфраструктуре сущностей	Matas Vaitkevicius, Tetsuya Yamamoto
3	Code First - Fluent API	Adil Mammadov, Daniel Lemke, Jason Tyler, tmg
4	Entity Framework с SQLite	Jason Tyler
5	Entity-Framework с Postgresql	skj123
6	Генерация первой модели базы данных	Matas Vaitkevicius, Tetsuya Yamamoto
7	Загрузка связанных объектов	Adil Mammadov, Florian Haider, Gert Arnold, hasan, Joshit, Matas Vaitkevicius, tmg
8	Инициализаторы баз данных	Jozef Lačný
9	Код First DataAnnotations	bubi, CptRobby, Daniel A. White, Daniel Lemke, DavidG, Diego, Gert Arnold, Jozef Lačný, Mark Shevchenko, Matas Vaitkevicius, Parth Patel, Piotrek, tmg, Tushar patel
10	Комплексные типы	CptRobby, Gert Arnold
11	Лучшие практики для платформы Entity (простой и профессиональный)	Braiam, Mina Matta
12	Методы оптимизации в EF	Amit Shahani, Anshul Nigam, DavidG, Gert Arnold, Jacob Linney, Kobi, lucavgobbi, Stephen Reindl, tmg, wertzui

13	Наследование с помощью EntityFramework (сначала код)	lucavgobbi
14	Ограничения модели	SOfanatic, Tushar patel
15	операции	CptRobby, DavidG, Gert Arnold
16	Отслеживание и отсутствие отслеживания	hasan, Sampath, Stephen Reindl, tmg
17	Первичные миграции кода сущностей	CGritton, hasan, Joshit, Mostafa, RamenChef, Stephen Reindl
18	Первоначальный код сущности	Balázs Nagy, Jozef Lačný
19	Первые соглашения	MacakM, Parth Patel, Sivanantham Padikkasu, Stephen Reindl, tmg
20	Расширенные сценарии сопоставления: разбиение объектов, разбиение таблиц	Akos Nagy
21	Сопоставление отношений с Entity Framework Code Сначала: «один ко многим» и «многие ко многим»	Akos Nagy
22	Сопоставление отношений с Entity Framework Code Сначала: индивидуально и вариации	Akos Nagy
23	Состояние	Gert Arnold

	управляющего объекта	