**EF – Entity Framework**

**Конвенция – принимаемое соглашение.**

Primary Key – первичный ключ

Foreign Key – внешний ключ

GoalKicker.com – Entity Framework Notes for Professionals

**О книге**  
Пожалуйста, не стесняйтесь свободно делиться этой книгой со всеми. Последнюю версию оригинальной книги можно скачать с сайта: https://goalkicker.com/EntityFrameworkBook  
Книга *Entity Framework записки для Профессионалов* является собранной с Stack Overflow Documentation, содержание написано замечательными людьми Stack Overflow.

Текстовое содержимое выпущено под лицензией Creative Commons BY-SA, см. главу «Авторы» в конце книги, кем созданы различные главы. Распространяется право на использование изображений, если не указано иное под конкретным из них. Это неофициальная, свободная книга создана в целях обучения и не связанна как либо с официальной группой или компанией Entity Framework, Stack Overflow и иными организациями. Все торговые марки и знаки являются собственностью соответствующих владельцев компаний. Информация представленная в книге не гарантирует точность и корректность, вы используете ее на свой риск.

Пожалуйста, отправляйте свои отзывы и исправления на адрес: [web@petercv.com](mailto:web@petercv.com)

Перевел: alx.korolenok@gmail.com

**Глава 1: Начиная работу с Entity**  
**Framework**

**Версии:**  
1.0 2008-08-11  
4.0 2010-04-12  
4.1 2011-04-12  
4.1 Update 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

Замечания о выпусках: <https://msdn.microsoft.com/en-ca/data/jj574253.aspx>

**Раздел 1.1: Что такое Entity Framework?**  
Написание и управление кода для доступа к данным с помощью ADO.Net является утомительной и монотонной работой.. Microsoft предоставляет фреймворк объектно-реляционного отображения (**O/RM framework)** называемый **"Entity Framework"** для автоматизации действий с базой данных для ваших приложений.

Entity framework это Объектно/Реляционное Отображение (Object/Relational Mapping (O/RM) framework).

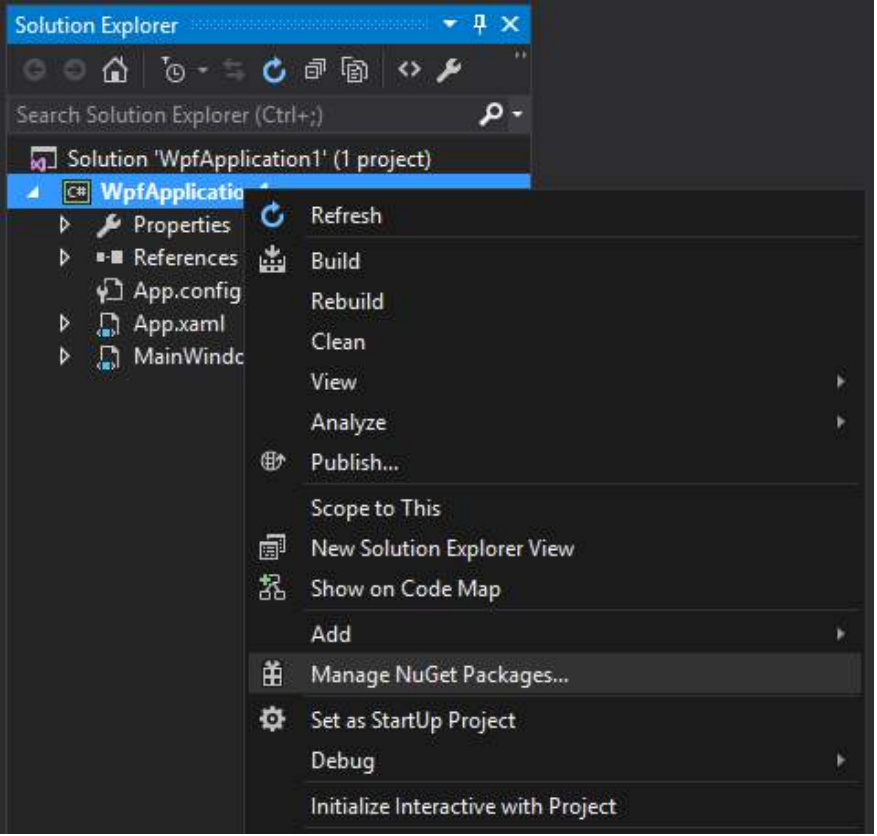
Это расширение ADO.NET, которое дает разработчикам автоматический механизм доступа и хранения данных в базах данных.

Что значит Объектно/Реляционное Отображение или **O/RM?**  
ORM это инструмент для хранения данных из объектов предметной области в реляционных базах данных, таких как MS SQL Server, в автоматическом режиме, без необходимости избыточного программирования.

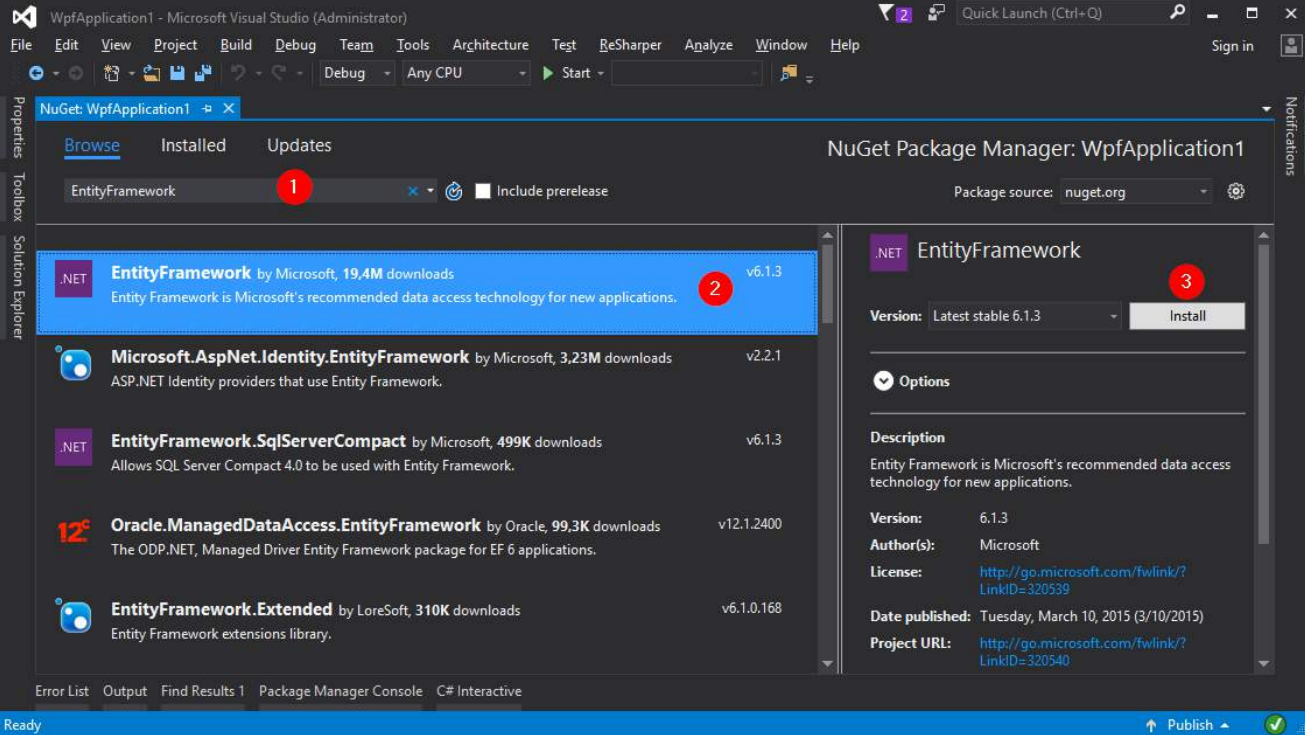
O/RM включает три главных части:  
1. Классы предметной области (сущности)   
2. Объекты реляционной базы данных  
3. Отображение информации о том как сущности отображаются на объекты базы данных (например таблицы, представления, хранимые процедуры).

ORM позволяет нам держать проект (дизайн) нашей базы данных отдельно от дизайна сущностей нашей предметной области. Это делает приложение более гибким к изменениям (ремонтопригодным) и расширяемым. Также это автоматизирует стандартные операции создания, чтения, обновления и удаления (CRUD), так что разработчику не нужно писать их самому.

**Раздел 1.2: Установка Entity Framework через менеджер пакетов NuGet**  
В Visual Studio откройте окно обозревателя решений (**Solution Explorer)** после чего нажмите правой кнопкой мыши по своему проекту. Появится контекстное меню, где нужно выбрать *Manage NuGet Packages*:



В появившемся окне, в поле поиска нужно набрать EntityFramework.



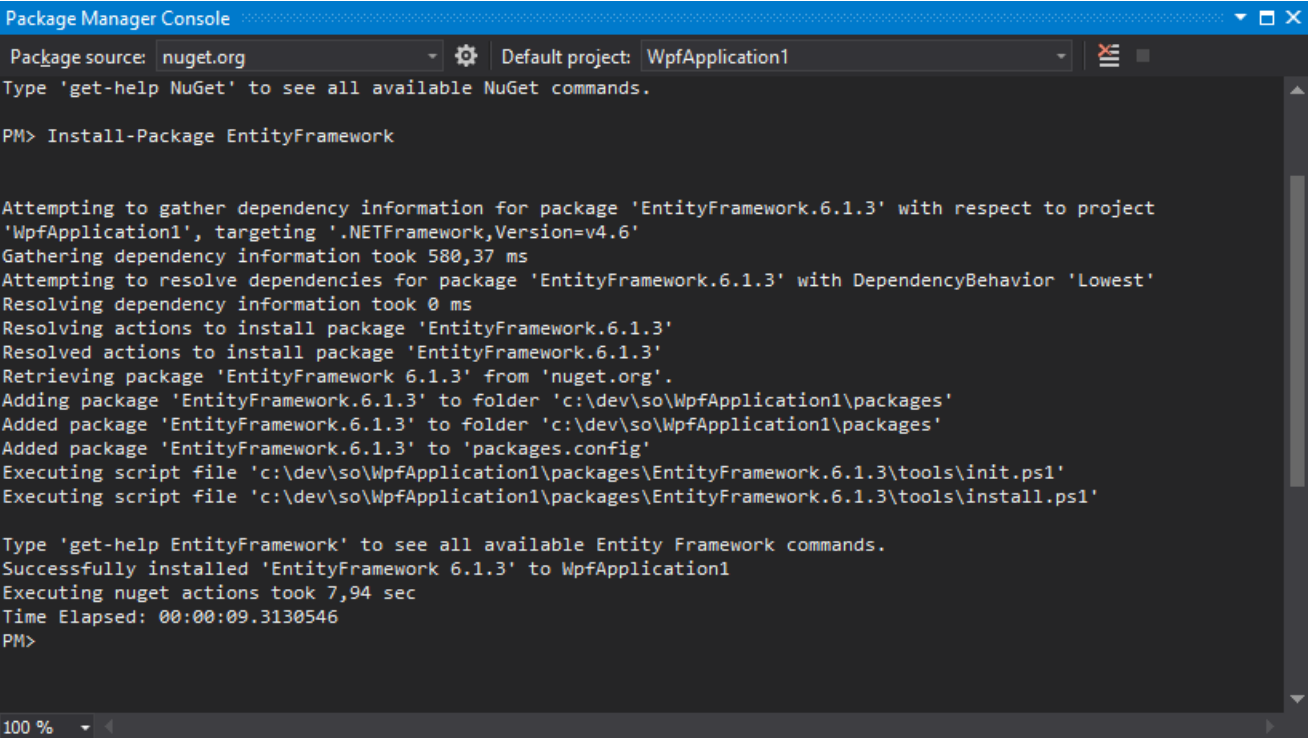
Далее нажмите кнопку установки (Install)

Также Entity Framework можно установить используя консоль менеджера пакетов (package manager console). Чтобы это сделать нужно открыть

*Tools menu -> NuGet Package Manager -> Package Manager Console*

и ввести команду:

Install-Package EntityFramework



Таким образом будет установлен EF и автоматически добавлена ссылка на сборку в вашем проекте.

**Раздел 1.3: Использование Entity Framework из C# (Code First)**  
Code first позволяет вам создавать ваши сущности (классы) без использование графического дизайнера или файла отображения данных .edmx. Этот подход называется «Сначала код» потому, что вы можете создать свои модели первыми и *Entity framework* создаст вам базу данных в соответствии с отображением автоматически. Или вы можете также использовать этот подход с существующей базой данных, который называется «Сначала код, с существующей БД» (*code first with existing database).*

Для примера, если вы хотите держать в таблице список планет:

public class Planet  
{  
 public string Name { get; set; } // Название  
 public decimal AverageDistanceFromSun { get; set; } // Среднее расстояние от солнца  
}

Теперь создайте ваш контекст данных, который является мостом между вашими классами сущностей и базой данных. Определите в этом контексте один или несколько наборов свойств, закрытый типом необходимой сущности. В данном случае **Planet**.

DbSet<> свойства:  
using System.Data.Entity;  
public class PlanetContext : DbContext  
{  
 public DbSet<Planet> Planets { get; set; }  
}

Мы может использовать этот контекст сделав следующее:

using(var context = new PlanetContext())  
{  
 var jupiter = new Planet  
 {  
 Name = "Jupiter",  
 verageDistanceFromSun = 778.5  
 };  
 context.Planets.Add(jupiter);  
 context.SaveChanges();  
}

В данном примере мы создаем новую планету со свойством Name, значение которого "Jupiter" и свойством AverageDistanceFromSun со значением 778.5

После чего мы можем добавить эту планету в наш контекст используя метод набора свойств (DbSet) Add() . Чтобы зафиксировать наши изменения в базе данных используем метод SaveChanges().

Или мы можем извлечь 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.");  
}

**Глава 2: конвенции Code First**  
**Раздел 2.1: конвенции удаления**

Вы можете удалить любую из конвенций определенных в пространстве имен System.Data.Entity.ModelConfiguration.Conventions  
путем переопределения метода OnModelCreating.

В следующем примере будет удалено соглашение об именовании множественных сущностей (PluralizingTableNameConvention).  
  
public class EshopContext : DbContext  
{  
 public DbSet<Product> Products { set; get; }  
 . . .  
 protected override void OnModelCreating(DbModelBuilder modelBuilder)  
 {  
 modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();  
 }  
}

По умолчанию EF будет создавать таблицы базы данных с именем класса после которого ставится суффикс 's' (обозначая множественное число). В данном примере, Code First сконфигурирован таким образом, что будет игнорировать данную конвенцию (PluralizingTableName), и вместо таблицы с именем **dbo.Products** будет создана **dbo.Product**.

**Раздел 2.2: конвенция первичного ключа**  
По умолчанию свойство, являющееся первичным ключом называется “ID” (не чувствительно к регистру), или по названию класса, за которым следует "ID". Если тип данных первичного ключа является числовым или GUID, то поле базы данных будет сконфигурировано как идентифицирующее (первичный ключ).

Простой пример:

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

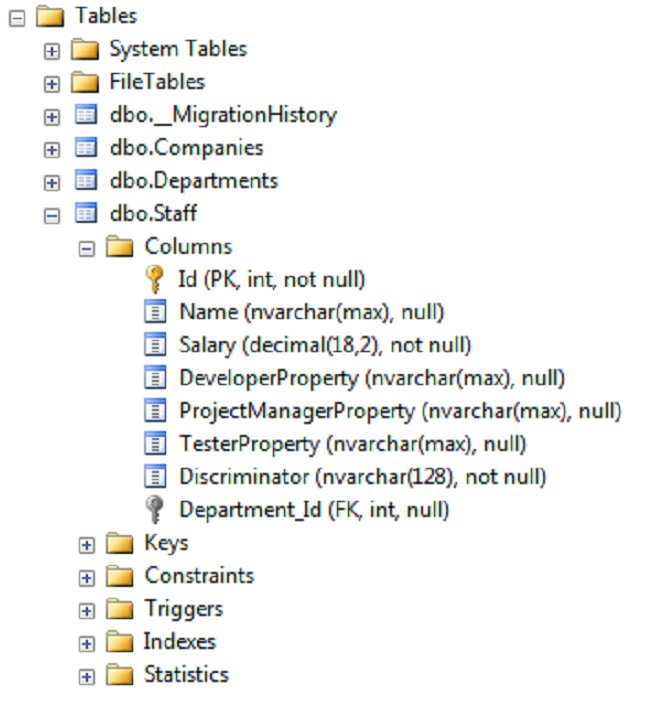
**Раздел 2.3: Раскрытие типа**  
По умолчанию Code First включает в модель следующее:  
1. Типы, определенные как наборы свойств (DbSet) в классе контекста.  
2. Ссылочные типы включенные в типы сущностей, даже если они определены в разных сборках.   
3. Унаследованные классы, даже если базовый класс определен как свойство DbSet.

В примере ниже мы добавим компанию как набор свойств в наш класс контекста:

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; }  
}

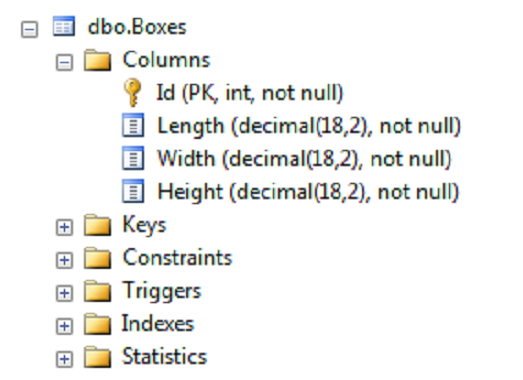
Как мы можем видеть, все наши классы включены в нашу модель:



**Раздел 2.4: конвенция десятичных свойств**

По умолчанию EF отображает свойства типа decimal в поля таблиц базы данных типа decimal(18,2).

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

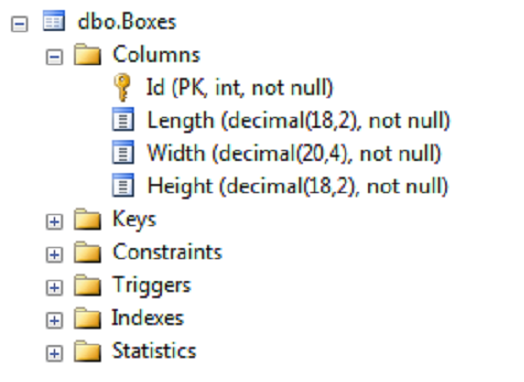


Мы можем менять точность таких (десятичных) свойств:

1. Используя Fluent API:

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

Только свойство "Width" будет отображено в decimal(20, 4).



1. Изменить конвенцию:

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

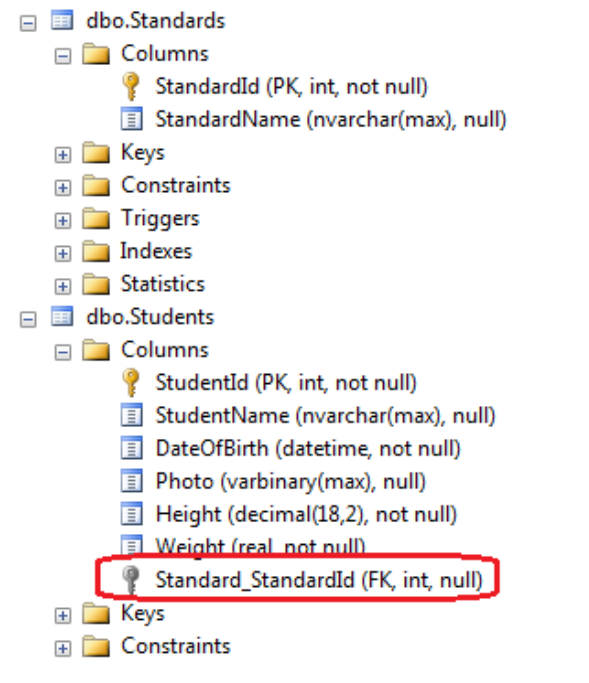
Каждое свойство типа decimal будет отображено в тип столбца decimal(10,4).

**Раздел 2.5: конвенция отношений**   
Code First делает вывод об отношениях между двумя сущностями используя навигационное свойство.

Это навигационное свойство может быть простым ссылочным типом или типом коллекции. К примеру, мы можем определить навигационное свойство **Standard** в классе Student и навигационное свойство типа ICollection в классе Standard. Так, Code First автоматически создаст отношение один-ко-многим между табицами Standards и Students. В БД, в таблице Students будет создан внешний ключ Standard\_StandardId.

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; }  
}

В вышеуказанных сущностях создано отношение использующее внешний ключ Standard\_StandardId.



**Раздел 2.6: конвенция внешнего ключа**   
Если класс A состоит в отношении с классом B и класс B имеет свойство с таким же именем и типом как первичный ключ класса A, тогда EF автоматически предполагает, что 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 данном случае DepartmentId является внешним ключом без явной спецификации.

Глава 3: Code First Аннотация данных

Аннотация данных (Data Annotation) – способ конфигурирования отображения через аннотирующие атрибуты.

# Пространства имен: System.ComponentModel.DataAnnotations, System.ComponentModel.DataAnnotations.Schema;

Раздел 3.1: Атрибут [Column]public class Person  
{  
 public int PersonID { get; set; }  
 [Column("NameOfPerson")]  
 public string PersonName { get; set; }  
}

Сообщает EF использовать конкретное имя столбца вместо используемого имени свойства. Вы также можете указать тип данных для столбца и номер по порядку столбца в таблице.  
  
[Column("NameOfPerson", TypeName = "varchar", Order = 1)]  
public string PersonName { get; set; }

Раздел 3.2: Атрибут [DatabaseGenerated]Уточняет как база данных генерирует значения для свойства. Возможны три варианта:

1. **None** указывает что значения не могут быть сгенерированы базой данных.  
2. **Identity** указывает идентифицирующий столбец, который обычно, используется для целочисленного первичного ключа.   
3. **Computed** указывает что база данных генерирует значение для столбца.

Если значение отличается от None, Entity Framework не зафиксирует в БД изменения произведенные над свойством.

По умолчанию (основываясь на конвенции StoreGeneratedIdentityKeyConvention) ключевое целочисленное свойство будет обработано как идентифицирующий столбец. Для переопределения данной конвенции и принудительной обработки как неидентифицирующего столбца вы должны 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 код SQL создает таблицу с вычисляемым полем:

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

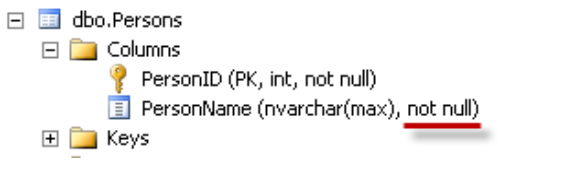
Чтобы создать сущность для представления строк вышеуказанной таблицы, вам понадобится использовать атрибут DatabaseGenerated со значением Computed.

[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; }  
}

**Раздел 3.3: [Required] attribute**  
Когда применяется свойство класса модели (сущности), БД будет создано NOT NULL.

using System.ComponentModel.DataAnnotations;  
public class Person  
{  
 public int PersonID { get; set; }  
 [Required]  
 public string PersonName { get; set; }  
}

В результате столбец будет создан с ограничением NOT NULL:

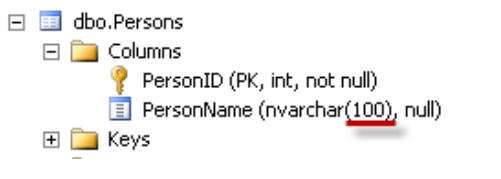


**Замечание:** Это также может быть использовано в asp.net-mvc как атрибут валидации (validation attribute).

**Раздел 3.4: Атрибуты [MaxLength] и [MinLength]**  
**[MaxLength(int)]** атрибут может быть применен к строке или свойству с типом массив. EF установит размер столбца по указанному значению значению.

using System.ComponentModel.DataAnnotations;  
public class Person  
{  
 public int PersonID { get; set; }  
 [MinLength(3), MaxLength(100)]  
 public string PersonName { get; set; }  
}

В результате будет создан столбец с указанной длиной.



**[MinLength(int)]** атрибут является валидационным, поэтому не влияет на структуру базы данных. Если мы попытаемся вставить/обновить имя (PersonName) сущности Person строкой менее 3 символов, то зафиксировать такие изменения не получится. Мы получим исключение 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]** могут также быть использованы в asp.net-mvc как атрибуты валидации.

**Раздел 3.5: 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; }  
}

Атрибут InverseProperty может быть использован для определения двусторонних отношений существующих между двумя сущностями.

Этот атрибут сообщает EF какие навигационные свойства должны соответствовать свойствам на другой стороне.

EF не знает какое навигационное свойство отображается с которым свойством с другой стороны, когда между двумя сущностями существуют множественные двунаправленные отношения.

Для этого требуется имя соответствующего класса в качестве параметра. Это также может быть использовано для сущностей которые находятся в отношениях с другими сущностями того же типа, формируя тем самым рекурсивные отношения.

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; }  
}

Примечательно также использование атрибута внешнего ключа (ForeignKey) для указания ограничения столбца той же самой таблицы.

В первом примере к вдум свойствам в классе Employee мог быть применен атрибут внешнего ключа ForeignKey, для определения имен столбцов.

**Раздел 3.6: 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; }  
}  
Mark the class as complex type in Entity Framework.  
Complex Types (Or *Value Objects* In Domain Driven Design) cannot be tracked on their own but they are tracked as  
part of an entity. This is why BlogDetails in the example does not have a key property.  
They can be useful when describing domain entities across multiple classes and layering those classes into a  
GoalKicker.com – Entity Framework Notes for Professionals 15  
complete entity.  
**Раздел 3.7: Атрибут [ForeignKey(string)]**  
Specifies custom foreign key name if a foreign key not following EF's convention is desired.  
public class Person  
{  
public int IdAddress { get; set; }  
[ForeignKey(nameof(IdAddress))]  
public virtual Address HomeAddress { get; set; }  
}  
This can also be used when you have multiple relationships to the same entity type.  
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; }  
}  
Without the ForeignKey attributes, EF might get them mixed up and use the value of BillingAddressID when  
fetching the MailingAddress, or it might just come up with a different name for the column based on its own  
naming conventions (like Address\_MailingAddress\_Id) and try to use that instead (which would result in an error if  
you're using this with an existing database).  
**Раздел 3.8: Атрибут [Range(min,max)]**  
Specifies a numeric minimum and maximum range for a property  
using System.ComponentModel.DataAnnotations;  
public partial class Enrollment  
{  
public int EnrollmentID { get; set; }  
[Range(0, 4)]  
public Nullable<decimal> Grade { get; set; }  
}  
If we try to insert/update a Grade with value out of range, this commit will fail. We’ll get a  
DbUpdateConcurrencyException that we'll need to handle.  
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*  
}  
}  
It can also be used with asp.net-mvc as a validation attribute.  
**Result:**  
**Раздел 3.9: Атрибут [NotMapped]**  
By Code-First convention, Entity Framework creates a column for every public property that is of a supported data  
type and has both a getter and a setter. **[NotMapped]** annotation must be applied to any properties that we do  
**NOT** want a column in a database table for.  
An example of a property that we might not want to store in the database is a student’s full name based on their  
first and last name. That can be calculated on the fly and there is no need to store it in the database.  
public string FullName => string.Format("{0} {1}", FirstName, LastName);  
The "FullName" property has only a getter and no setter, so by default, Entity Framework will **NOT** create a column  
for it.  
Another example of a property that we might not want to store in the database is a student’s "AverageGrade". We  
do not want to get the AverageGrade on-demand; instead we might have a routine elsewhere that calculates it.  
[NotMapped]  
public float AverageGrade { set; get; }  
The "AverageGrade" must be marked **[NotMapped]** annotation, else Entity Framework will create a column for it.  
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; }  
}  
For the above Entity we will see inside DbMigration.cs  
GoalKicker.com – Entity Framework Notes for Professionals 17  
CreateTable(  
"dbo.Students",  
c => new  
{  
Id = c.Int(nullable: false, identity: true),  
FirstName = c.String(),  
LastName = c.String(),  
})  
.PrimaryKey(t => t.Id);  
and in SQL Server Management Studio  
**Раздел 3.10: Атрибут [Table]**  
[Table("People")]  
public class Person  
{  
public int PersonID { get; set; }  
public string PersonName { get; set; }  
}  
Tells Entity Framework to use a specific table name instead of generating one (i.e. Person or Persons)  
We can also specify a schema for the table using [Table] attribute  
[Table("People", Schema = "domain")]  
**Раздел 3.11: Атрибут [Index]**  
public class Person  
{  
public int PersonID { get; set; }  
public string PersonName { get; set; }  
[Index]  
public int Age { get; set; }  
}  
Creates a database index for a column or set of columns.  
[Index("IX\_Person\_Age")]  
public int Age { get; set; }  
This creates an index with a specific name.  
GoalKicker.com – Entity Framework Notes for Professionals 18  
[Index(IsUnique = true)]  
public int Age { get; set; }  
This creates a unique index.  
[Index("IX\_Person\_NameAndAge", 1)]  
public int Age { get; set; }  
[Index("IX\_Person\_NameAndAge", 2)]  
public string PersonName { get; set; }  
This creates a composite index using 2 columns. To do this you must specify the same index name and provide a  
column order.  
**Note**: The Index attribute was introduced in Entity Framework 6.1. If you are using an earlier version the  
information in this section does not apply.  
**Раздел 3.12: Атрибут [Key]**  
Key is a field in a table which uniquely identifies each row/record in a database table.  
Use this attribute to **override the default Code-First convention**. If applied to a property, it will be used as the  
**primary key column** for this class.  
using System.ComponentModel.DataAnnotations;  
public class Person  
{  
[Key]  
public int PersonKey { get; set; } *// <- will be used as primary key*  
public string PersonName { get; set; }  
}  
If a composite primary key is required, the [Key] attribute can also be added to multiple properties. The order of the  
columns within the composite key must be provided in the form [**Key, Column(Order = 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; }  
}  
**Without the [Key] attribute**, EntityFramework will fall back to the default convention which is to use the property  
of the class as a primary key that is named "Id" or "{ClassName}Id".  
public class Person  
{  
public int PersonID { get; set; } *// <- will be used as primary key*  
  
GoalKicker.com – Entity Framework Notes for Professionals 19  
public string PersonName { get; set; }  
}  
**Раздел 3.13: 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; }  
}  
Defines a maximum length for a string field.  
**Note**: It can also be used with asp.net-mvc as a validation attribute.  
**Раздел 3.14: Атрибут [Timestamp]**  
**[TimeStamp]** attribute can be applied to only one byte array property in a given Entity class. Entity Framework will  
create a non-nullable timestamp column in the database table for that property. Entity Framework will  
automatically use this TimeStamp column in concurrency check.  
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; }  
}  
GoalKicker.com – Entity Framework Notes for Professionals 20  
**Раздел 3.15: Атрибут [ConcurrencyCheck]**  
This attribute is applied to the class property. You can use ConcurrencyCheck attribute when you want to use  
existing columns for concurrency check and not a separate timestamp column for concurrency.  
using System.ComponentModel.DataAnnotations;  
public class Author  
{  
public int AuthorId { get; set; }  
[ConcurrencyCheck]  
public string AuthorName { get; set; }  
}  
From above example, ConcurrencyCheck attribute is applied to AuthorName property of the Author class. So, CodeFirst will include AuthorName column in update command (where clause) to check for optimistic concurrency.