Entity Framework

Notes for Professionals

Ketabton.com

80+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official Entity Framework group(s) or company(s). All trademarks and registered trademarks are the property of their respective owners

(c) ketabton.com: The Digital Library

Contents

About 1

Chapter 1: Getting started with Entity Framework 2

- Section 1.1: Installing the Entity Framework NuGet Package 2
- Section 1.2: Using Entity Framework from C# (Code First) 4
- Section 1.3: What is Entity Framework? 5

Chapter 2: Code First Conventions 6

- Section 2.1: Removing Conventions 6
- Section 2.2: Primary Key Convention 6
- Section 2.3: Type Discovery 6
- Section 2.4: DecimalPropertyConvention 7
- Section 2.5: Relationship Convention 9
- Section 2.6: Foreign Key Convention 10

Chapter 3: Code First DataAnnotations 11

- Section 3.1: [Column] attribute 11
- Section 3.2: [DatabaseGenerated] attribute 11
- Section 3.3: [Required] attribute 12
- Section 3.4: [MaxLength] and [MinLength] attributes 12
- Section 3.5: [InverseProperty(string)] attribute 13
- Section 3.6: [ComplexType] attribute 14
- Section 3.7: [ForeignKey(string)] attribute 15
- Section 3.8: [Range(min,max)] attribute 15
- Section 3.9: [NotMapped] attribute 16
- Section 3.10: [Table] attribute 17
- Section 3.11: [Index] attribute 17
- Section 3.12: [Key] attribute 18
- Section 3.13: [StringLength(int)] attribute 19
- Section 3.14: [Timestamp] attribute 19
- Section 3.15: [ConcurrencyCheck] Attribute 20

Chapter 4: Entity Framework Code First 21

- Section 4.1: Connect to an existing database 21

Chapter 5: Entity framework Code First Migrations 23

- Section 5.1: Enable Migrations 23
- Section 5.2: Add your first migration 23
- Section 5.3: Doing "Update-Database" within your code 25
- Section 5.4: Seeding Data during migrations 25
- Section 5.5: Initial Entity Framework Code First Migration Step by Step 26
- Section 5.6: Using Sql() during migrations 27

Chapter 6: Inheritance with EntityFramework (Code First) 29

- Section 6.1: Table per hierarchy 29
- Section 6.2: Table per type 29

Chapter 7: Code First - Fluent API 31

- Section 7.1: Mapping models 31
- Section 7.2: Composite Primary Key 32
- Section 7.3: Maximum Length 33
- Section 7.4: Primary Key 33
- Section 7.5: Required properties (NOT NULL) 34

(c) ketabton.com: The Digital Library

Section 7.6: Explict Foreign Key naming 34
Chapter 8: Mapping relationship with Entity Framework Code First: One-to-one and variations 36
Section 8.1: Mapping one-to-zero or one 36
Section 8.2: Mapping one-to-one 39
Section 8.3: Mapping one or zero-to-one or zero 40
Chapter 9: Mapping relationship with Entity Framework Code First: One-to-many and Many-to-many 41
Section 9.1: Mapping one-to-many 41
Section 9.2: Mapping one-to-many: against the convention 42
Section 9.3: Mapping zero or one-to-many 43
Section 9.4: Many-to-many 44
Section 9.5: Many-to-many: customizing the join table 45
Section 9.6: Many-to-many: custom join entity 46
Chapter 10: Database first model generation 49
Section 10.1: Generating model from database 49
Section 10.2: Adding data annotations to the generated model 50
Chapter 11: Complex Types 52
Section 11.1: Code First Complex Types 52
Chapter 12: Database Initialisers 53
Section 12.1: CreateDatabaseIfNotExists 53
Section 12.2: DropCreateDatabaseIfModelChanges 53
Section 12.3: DropCreateDatabaseAlways 53
Section 12.4: Custom database initializer 53
Section 12.5: MigrateDatabaseToLatestVersion 54
Chapter 13: Tracking vs. No-Tracking 55
Section 13.1: No-tracking queries 55
Section 13.2: Tracking queries 55
Section 13.3: Tracking and projections 55
Chapter 14: Transactions 57
Section 14.1: Database.BeginTransaction() 57
Chapter 15: Managing entity state 58
Section 15.1: Setting state Added of a single entity 58
Section 15.2: Setting state Added of an object graph 58
Chapter 16: Loading related entities 60
Section 16.1: Eager loading 60
Section 16.2: Explicit loading 60
Section 16.3: Lazy loading 61
Section 16.4: Projection Queries 61
Chapter 17: Model Restraints 63
Section 17.1: One-to-many relationships 63
Chapter 18: Entity Framework with PostgreSQL 65
Section 18.1: Pre-Steps needed in order to use Entity Framework 6.1.3 with PostgresSql using Npgsqlddexprovider 65
Chapter 19: Entity Framework with SQLite 66
Section 19.1: Setting up a project to use Entity Framework with an SQLite provider 66
Chapter 20: .t4 templates in entity framework 69
Section 20.1: Dynamically adding Interfaces to model 69

(c) ketabton.com: The Digital Library

Section 20.2: Adding XML Documentation to Entity Classes 69
Chapter 21: Advanced mapping scenarios: entity splitting, table splitting 71
Section 21.1: Entity splitting 71
Section 21.2: Table splitting 72
Chapter 22: Best Practices For Entity Framework (Simple & Professional) 73
Section 22.1: 1- Entity Framework @ Data layer (Basics) 73
Section 22.2: 2- Entity Framework @ Business layer 76
Section 22.3: 3- Using Business layer @ Presentation layer (MVC) 79
Section 22.4: 4- Entity Framework @ Unit Test Layer 81
Chapter 23: Optimization Techniques in EF 85
Section 23.1: Using AsNoTracking 85
Section 23.2: Execute queries in the database when possible, not in memory 85
Section 23.3: Loading Only Required Data 85
Section 23.4: Execute multiple queries async and in parallel 86
Section 23.5: Working with stub entities 87
Section 23.6: Disable change tracking and proxy generation 88
Credits 89
You may also like 90

(c) ketabton.com: The Digital Library

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/EntityFrameworkBook

This *Entity Framework Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Entity Framework group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

(c) ketabton.com: The Digital Library

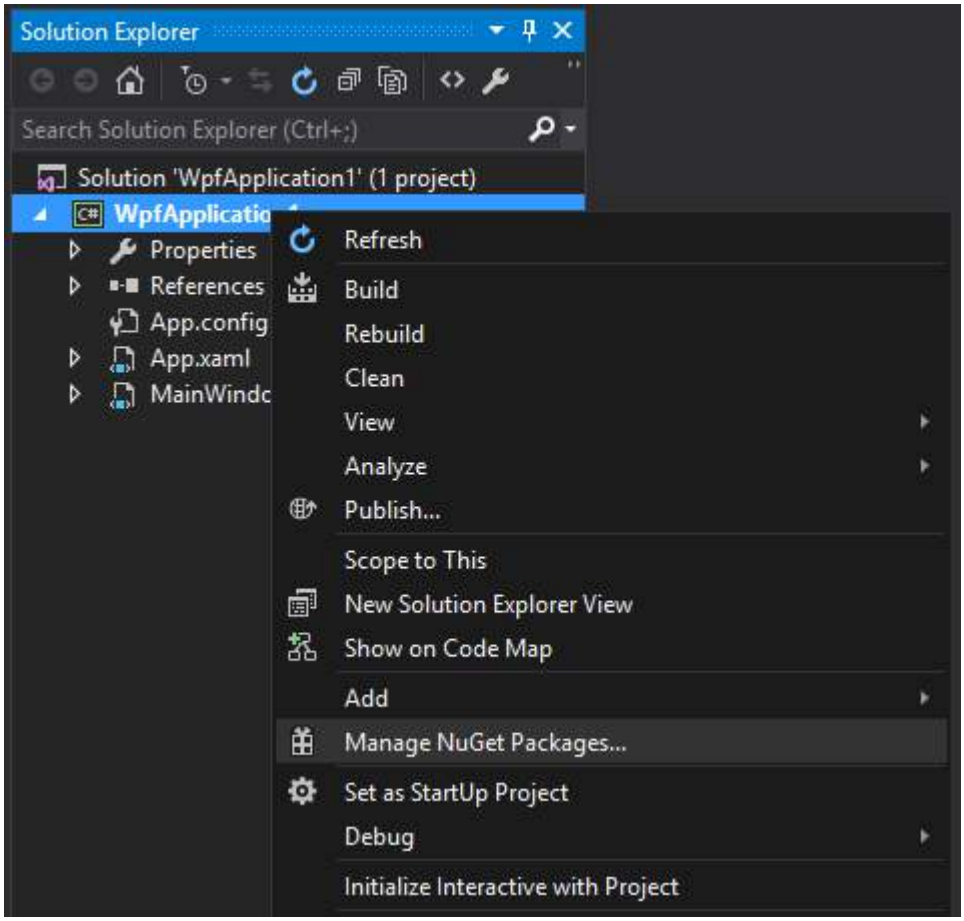
Chapter 1: Getting started with Entity Framework

Version	Release Date
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

Release Notes: https://msdn.microsoft.com/en-ca/data/jj574253.aspx

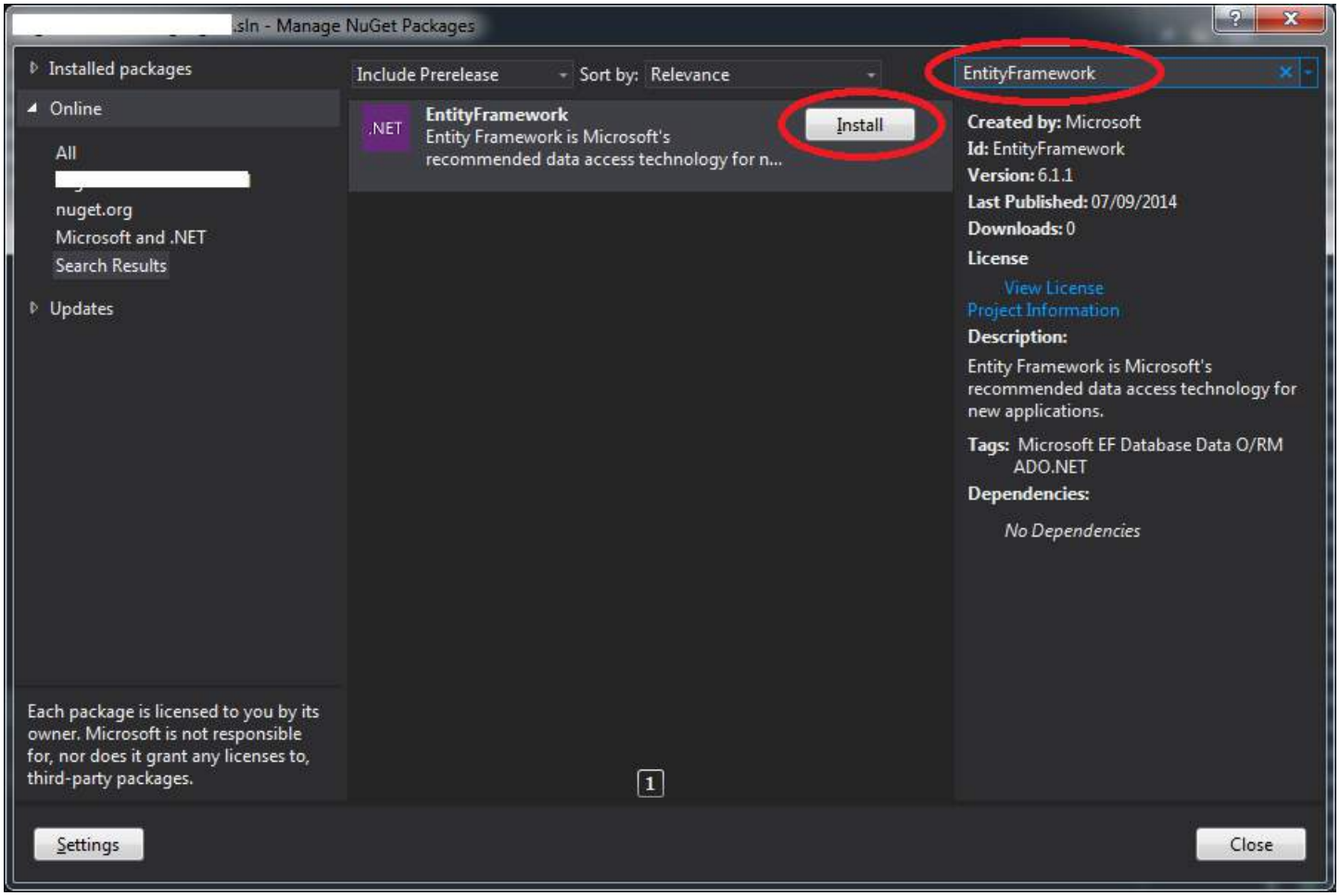
Section 1.1: Installing the Entity Framework NuGet Package

In your Visual Studio open the **Solution Explorer** window then `right click` on your project then choose *Manage NuGet Packages* from the menu:

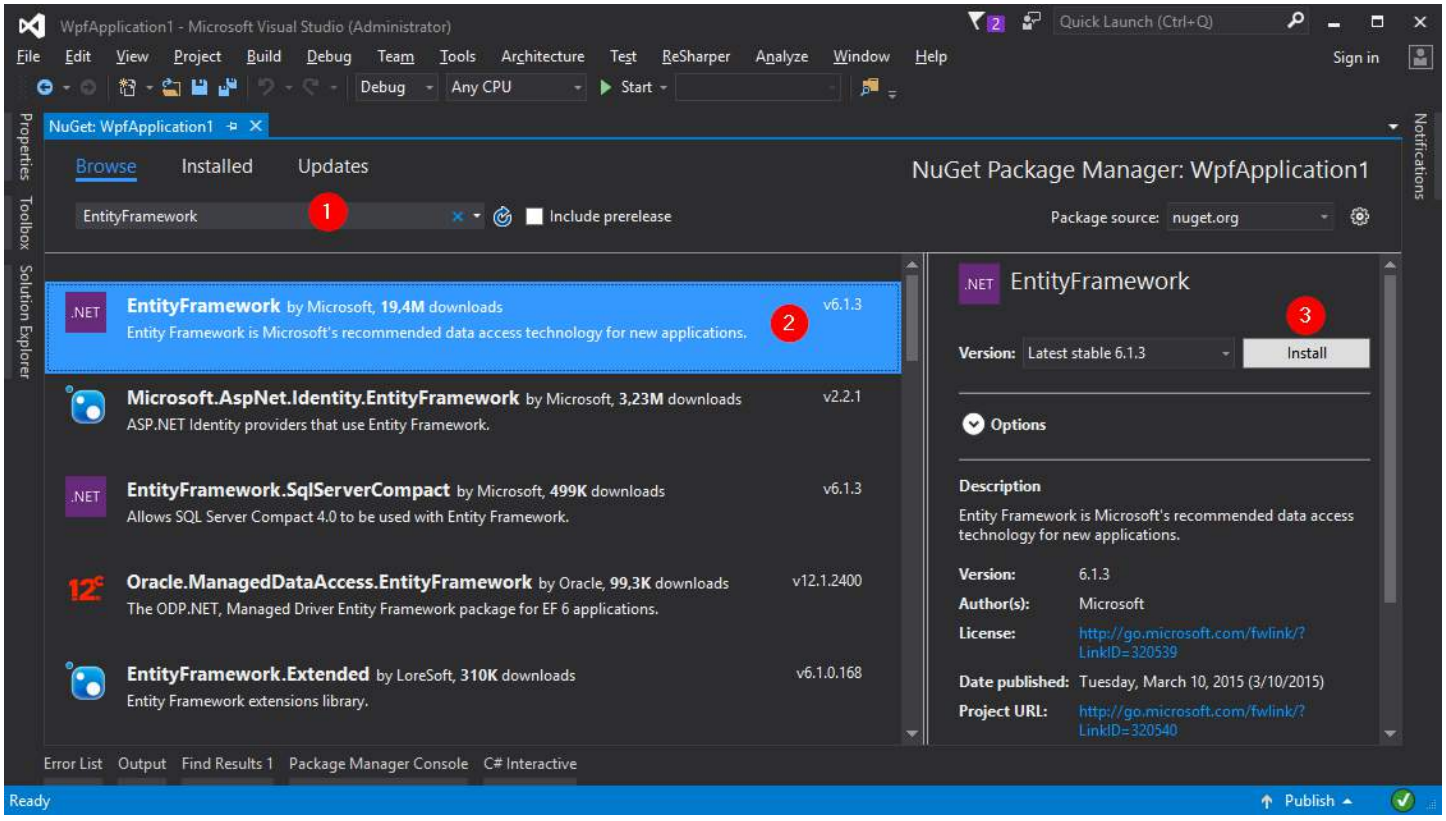


In the window that opens type `EntityFramework` in the search box in the top right.

(c) ketabton.com: The Digital Library



Or if you are using Visual Studio 2015 you'll see something like this:

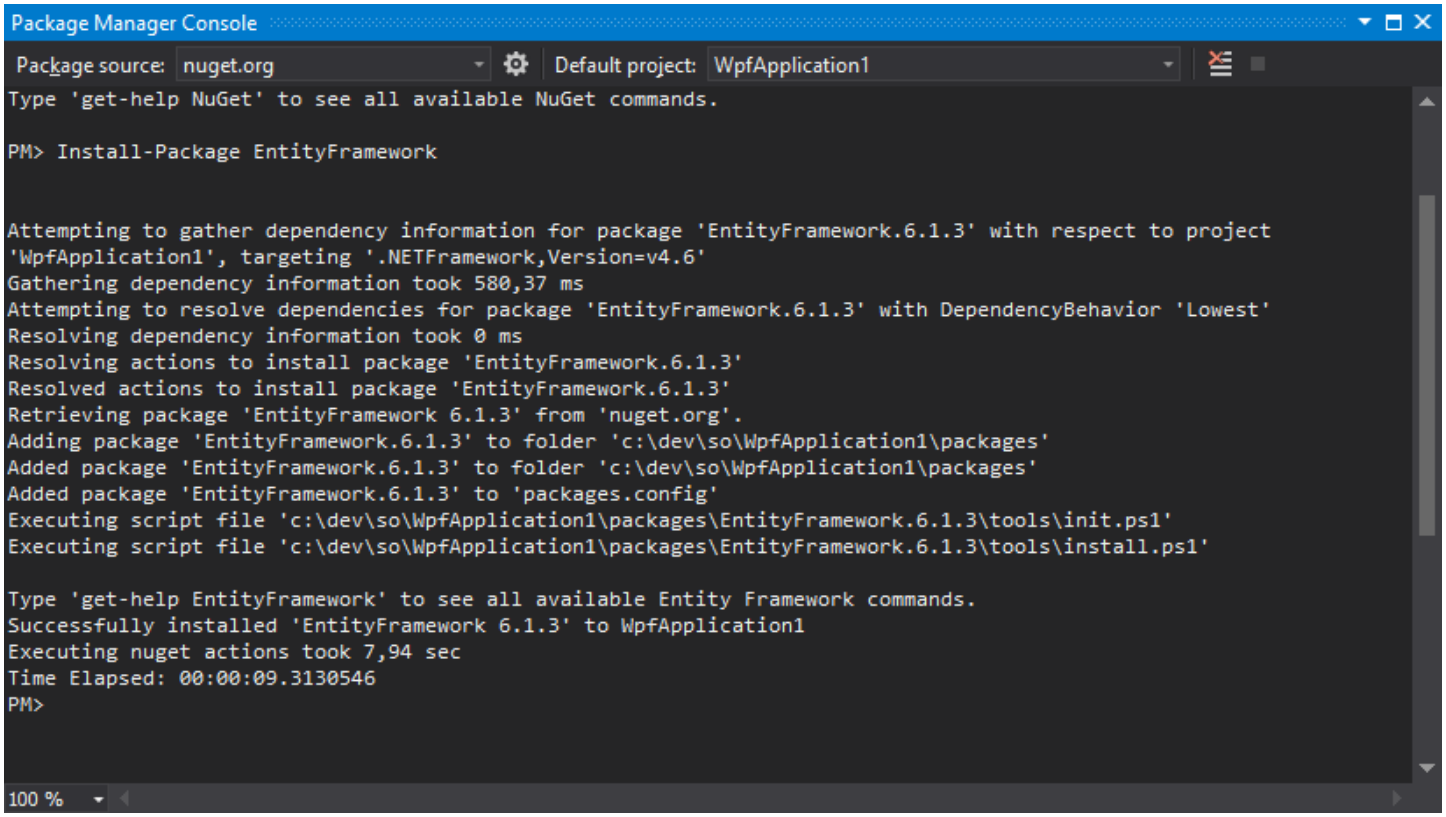


Then click Install.

We can also install entity framework using the package manager console. To do you have first to open it using the *Tools menu -> NuGet Package Manager -> Package Manager Console* then enter this:

(c) ketabton.com: The Digital Library

```
Install-Package EntityFramework
```



This will install Entity Framework and automatically add a reference to the assembly in your project.

Section 1.2: Using Entity Framework from C# (Code First)

Code first allows you to create your entities (classes) without using a GUI designer or a .edmx file. It is named *Code first*, because you can create your models *first* and *Entity framework* will create database according to mappings for you automatically. Or you can also use this approach with existing database, which is called *code first with existing database* For example, if you want a table to hold a list of planets:

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

Now create your context which is the bridge between your entity classes and the database. Give it one or more `DbSet<>` properties:

```
using System.Data.Entity;

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

We can use this by doing the following:

```
using(var context = new PlanetContext())
{
    var jupiter = new Planet
    {
```

(c) ketabton.com: The Digital Library

```
        Name = "Jupiter",
        AverageDistanceFromSun = 778.5
    };

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

In this example we create a new `Planet` with the `Name` property with the value of `"Jupiter"` and the `AverageDistanceFromSun` property with the value of `778.5`

We can then add this `Planet` to the context by using the `DbSet`'s `Add()` method and commit our changes to the database by using the `SaveChanges()` method.

Or we can retrieve rows from the database:

```
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.");
}
```

Section 1.3: What is Entity Framework?

Writing and managing ADO.Net code for data access is a tedious and monotonous job. Microsoft has provided an **O/RM framework called "Entity Framework"** to automate database related activities for your application.

Entity framework is an Object/Relational Mapping (O/RM) framework. It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing & storing the data in the database.

What is O/RM?

ORM is a tool for storing data from domain objects to the relational database like MS SQL Server, in an automated way, without much programming. O/RM includes three main parts:

1. Domain class objects
2. Relational database objects
3. Mapping information on how domain objects map to relational database objects(**e.x** tables, views & stored procedures)

ORM allows us to keep our database design separate from our domain class design. This makes the application maintainable and extendable. It also automates standard CRUD operation (Create, Read, Update & Delete) so that the developer doesn't need to write it manually.

(c) ketabton.com: The Digital Library

Chapter 2: Code First Conventions

Section 2.1: Removing Conventions

You can remove any of the conventions defined in the System.Data.Entity.ModelConfiguration.Conventions namespace, by overriding `OnModelCreating` method.

The following example removes PluralizingTableNameConvention.

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

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

By default EF will create DB table with entity class name suffixed by 's'. In this example, Code First is configured to ignore PluralizingTableName convention so, instead of `dbo.Products` table `dbo.Product` table will be created.

Section 2.2: Primary Key Convention

By default a property is a primary key if a property on a class is named “ID” (not case sensitive), or the class name followed by "ID". If the type of the primary key property is numeric or GUID it will be configured as an identity column. Simple Example:

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

Section 2.3: Type Discovery

By default Code First includes in model

1. Types defined as a DbSet property in context class.
2. Reference types included in entity types even if they are defined in different assembly.
3. Derived classes even if only the base class is defined as DbSet property

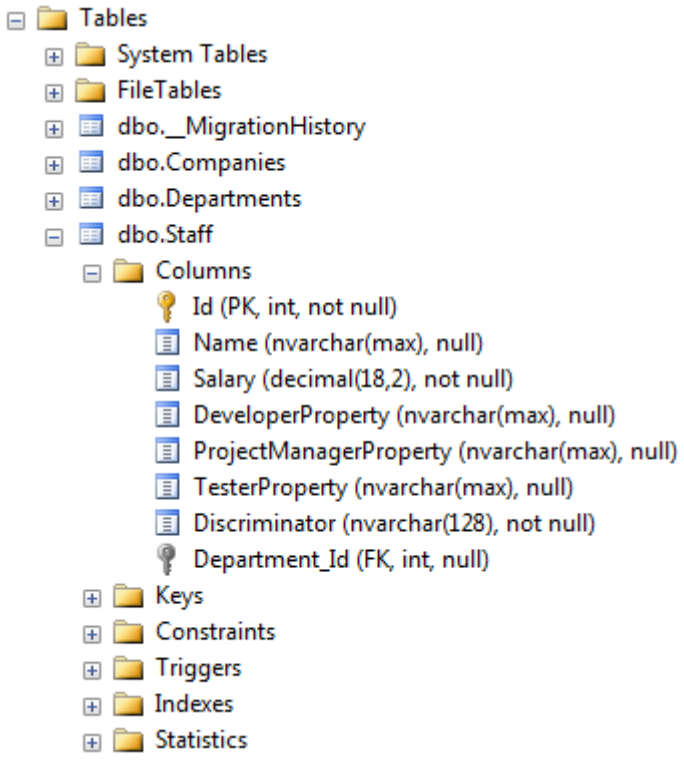
Here is an example, that we are only adding `Company` as `DbSet<Company>` in our context class:

```
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; }
```

(c) ketabton.com: The Digital Library

```
    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; }
}
```

We can see that all the classes are included in model

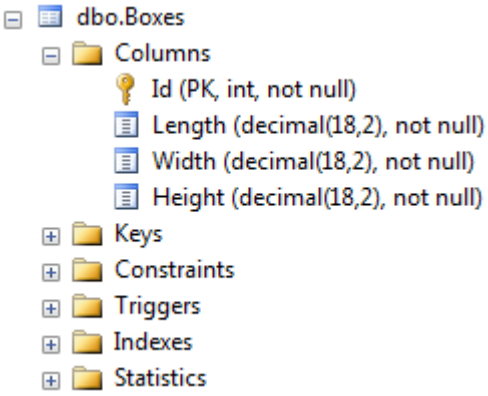


Section 2.4: DecimalPropertyConvention

By default Entity Framework maps decimal properties to decimal(18,2) columns in database tables.

(c) ketabton.com: The Digital Library

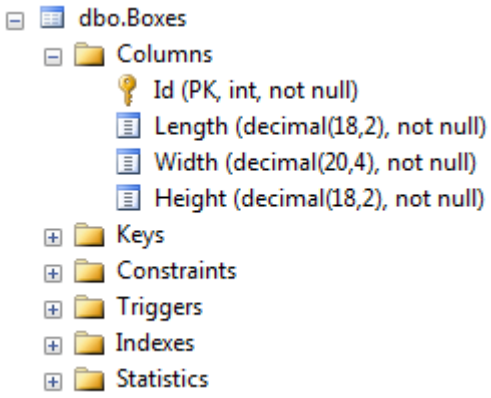
```
public class Box
{
    public int Id { set; get; }
    public decimal Length { set; get; }
    public decimal Width { set; get; }
    public decimal Height { set; get; }
}
```



We can change the precision of decimal properties:

1.Use Fluent API:

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

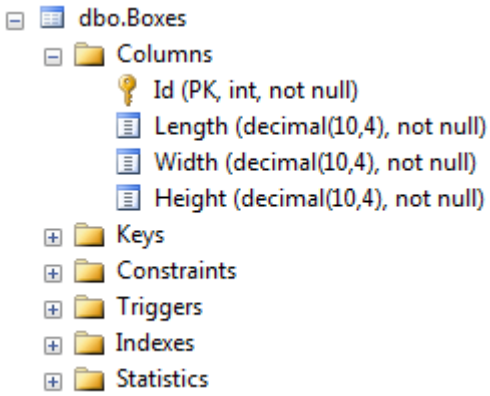


Only "Width" Property is mapped to decimal(20, 4).

2.Replace the convention:

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

(c) ketabton.com: The Digital Library



Every decimal property is mapped to decimal(10,4) columns.

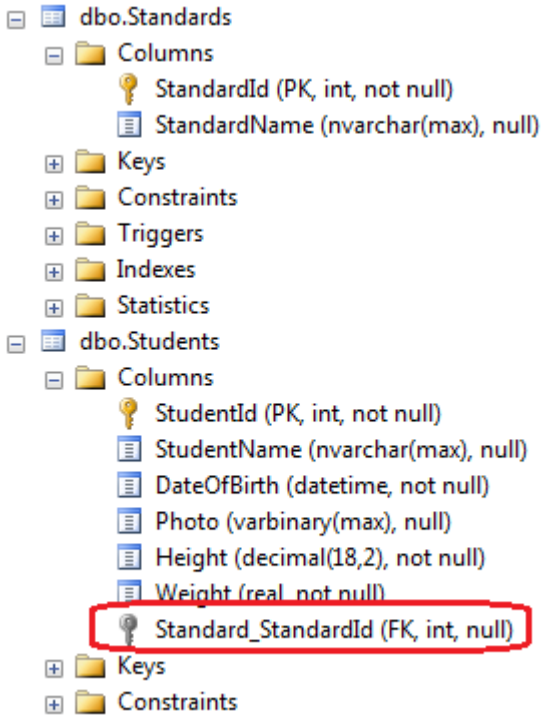
Section 2.5: Relationship Convention

Code First infer the relationship between the two entities using navigation property. This navigation property can be a simple reference type or collection type. For example, we defined Standard navigation property in Student class and ICollection navigation property in Standard class. So, Code First automatically created one-to-many relationship between Standards and Students DB table by inserting Standard_StandardId foreign key column in the Students table.

```
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; }

}
```

The above entities created the following relationship using Standard_StandardId foreign key.

(c) ketabton.com: The Digital Library



Section 2.6: Foreign Key Convention

If class A is in relationship with class B and class B has property with the same name and type as the primary key of A, then EF automatically assumes that property is foreign key.

```
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; }
}
```

In this case DepartmentId is foreign key without explicit specification.

(c) ketabton.com: The Digital Library

Chapter 3: Code First DataAnnotations

Section 3.1: [Column] attribute

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

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

Tells Entity Framework to use a specific column name instead using the name of the property. You can also specify the database data type and the order of the column in table:

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

Section 3.2: [DatabaseGenerated] attribute

Specifies how the database generates values for the property. There are three possible values:

1. `None` specifies that the values are not generated by the database.
2. `Identity` specifies that the column is an identity column, which is typically used for integer primary keys.
3. `Computed` specifies that the database generates the value for the column.

If the value is anything other than `None`, Entity Framework will not commit changes made to the property back to the database.

By default (based on the `StoreGeneratedIdentityKeyConvention`) an integer key property will be treated as an identity column. To override this convention and force it to be treated as a non-identity column you can use the `DatabaseGenerated` attribute with a value of `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
}
```

The following SQL creates a table with a computed column:

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

(c) ketabton.com: The Digital Library

```
GO
```

To create an entity for representing the records in the above table, you would need to use the `DatabaseGenerated` attribute with a value of `Computed`.

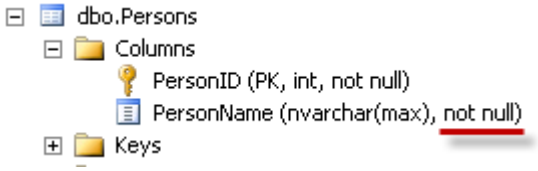
```
[Table("Person")]
public class Person
{
    [Key, StringLength(100)]
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public int Age { get; set; }
}
```

Section 3.3: [Required] attribute

When applied to a property of a domain class, the database will create a NOT NULL column.

```
using System.ComponentModel.DataAnnotations;

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

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

The resulting column with the NOT NULL constraint:



Note: It can also be used with asp.net-mvc as a validation attribute.

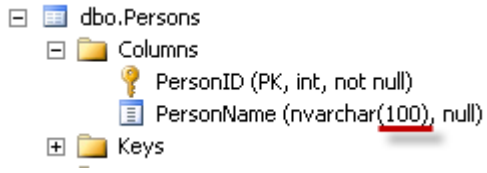
Section 3.4: [MaxLength] and [MinLength] attributes

[MaxLength(int)] attribute can be applied to a string or array type property of a domain class. Entity Framework will set the size of a column to the specified value.

```
using System.ComponentModel.DataAnnotations;

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

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

The resulting column with the specified column length:

(c) ketabton.com: The Digital Library



[MinLength(int)] attribute is a validation attribute, it does not affect the database structure. If we try to insert/update a Person with PersonName with length less than 3 characters, this commit will fail. We'll get a `DbUpdateConcurrencyException` that we'll need to handle.

```
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'."
     }
}
```

Both **[MaxLength]** and **[MinLength]** attributes can also be used with asp.net-mvc as a validation attribute.

Section 3.5: [InverseProperty(string)] attribute

```
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 can be used to identify *two way* relationships when **multiple** *two way* relationships exist between two entities.

It tells Entity Framework which navigation properties it should match with properties on the other side.

Entity Framework doesn't know which navigation property map with which properties on the other side when multiple bidirectional relationships exist between two entities.

It needs the name of the corresponding navigation property in the related class as its parameter.

(c) ketabton.com: The Digital Library

This can also be used for entities that have a relationship to other entities of the same type, forming a recursive relationship.

```
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; }
}
```

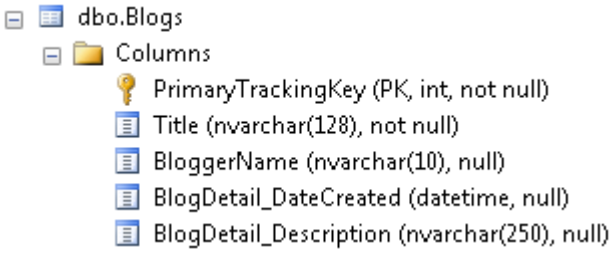
Note also the use of the `ForeignKey` attribute to specify the column that is used for the foreign key on the table. In the first example, the two properties on the `Employee` class could have had the `ForeignKey` attribute applied to define the column names.

Section 3.6: [ComplexType] attribute

```
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

(c) ketabton.com: The Digital Library

complete entity.

Section 3.7: [ForeignKey(string)] attribute

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).

Section 3.8: [Range(min,max)] attribute

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 });
```

(c) ketabton.com: The Digital Library

```
    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:

Section 3.9: [NotMapped] attribute

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.

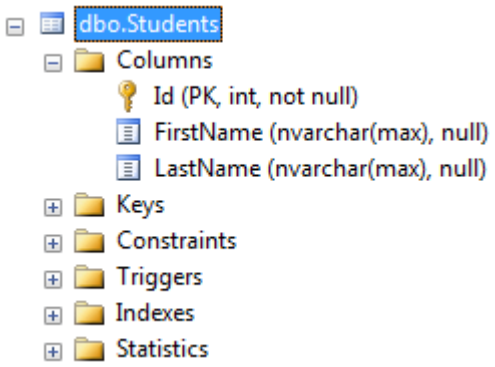
```
using System.ComponentModel.DataAnnotations.Schema;

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`

(c) ketabton.com: The Digital Library

```
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



Section 3.10: [Table] attribute

```
[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")]
```

Section 3.11: [Index] attribute

```
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.

(c) ketabton.com: The Digital Library

```
[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.

Section 3.12: [Key] attribute

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
```

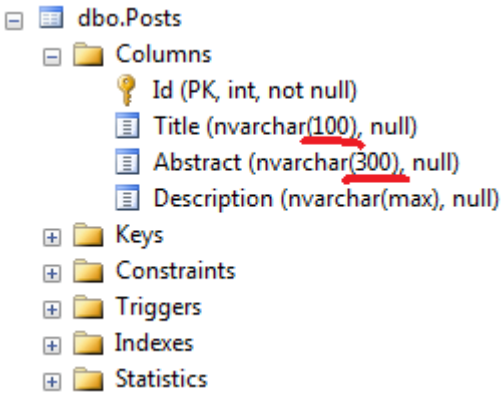
(c) ketabton.com: The Digital Library

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

Section 3.13: [StringLength(int)] attribute

```
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.



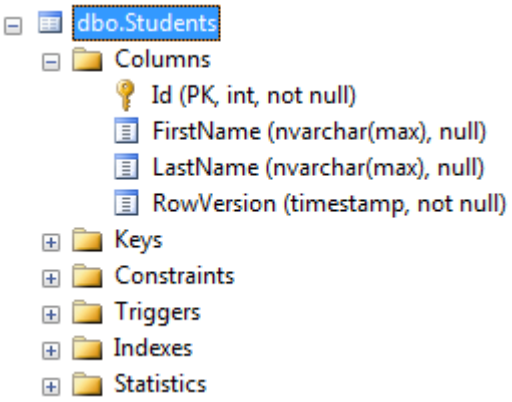
Note: It can also be used with asp.net-mvc as a validation attribute.

Section 3.14: [Timestamp] attribute

[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; }
}
```

(c) ketabton.com: The Digital Library



Section 3.15: [ConcurrencyCheck] Attribute

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, Code-First will include AuthorName column in update command (where clause) to check for optimistic concurrency.

(c) ketabton.com: The Digital Library

Chapter 4: Entity Framework Code First

Section 4.1: Connect to an existing database

To achieve the simplest task in Entity Framework - to connect to an existing database `ExampleDatabase` on your local instance of MSSQL you have to implement two classes only.

First is the entity class, that will be mapped to our database table `dbo.People`.

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

The class will use Entity Framework's conventions and map to table `dbo.People` which is expected to have primary key `PersonId` and a varchar(max) property `FirstName`.

Second is the context class which derives from `System.Data.Entity.DbContext` and which will manage the entity objects during runtime, pupulate them from database, handle concurrency and save them back to the database.

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

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

Please mind, that in the constructor of our context we need to set database initializer to null - we don't want Entity Framework to create the database, we just want to access it.

Now you are able manipulate data from that table, e.g. change the `FirstName` of first person in the database from a console application like this:

```
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();
            }
        }
    }
}
```

In the code above we created instance of Context with an argument "DbConnectionString". This has to be specified in our `app.config` file like this:

```
<connectionStrings>
```

(c) ketabton.com: The Digital Library

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

(c) ketabton.com: The Digital Library

Chapter 5: Entity framework Code First Migrations

Section 5.1: Enable Migrations

To enable Code First Migrations in entity framework, use the command

```
Enable-Migrations
```

on the *Package Manager Console*.

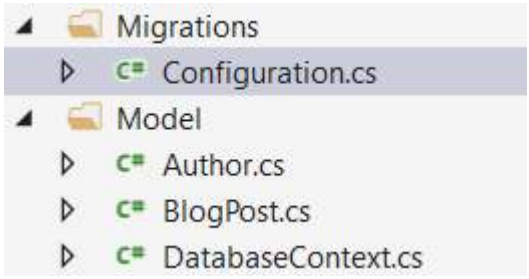
You need to have a valid `DbContext` implementation containing your database objects managed by EF. In this example the database context will contain to objects `BlogPost` and `Author`:

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

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

After executing the command, the following output should appear:

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

In addition, a new folder `Migrations` should appear with a single file `Configuration.cs` inside:



The next step would be to create your first database migration script which will create the initial database (see next example).

Section 5.2: Add your first migration

After you've enabled migrations (please refer to this example) you are now able to create your first migration containing an initial creation of all database tables, indexes and connections.

A migration can be created by using the command

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

This command will create a new class containing two methods `Up` and `Down` that are used to apply and remove the migration.

Now apply the command based on the example above to create a migration called *Initial*:

(c) ketabton.com: The Digital Library

```
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.
```

A new file *timestamp*_Initial.cs is created (only the important stuff is shown here):

```
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");
}
```

As you can see, in method `Up()` two tables `Authors` and `BlogPosts` are created and the fields are created accordingly. In addition, the relation between the two tables is created by adding the field `Author_AuthorId`. On the other side the method `Down()` tries to reverse the migration activities.

If you feel confident with your migration, you can apply the migration to the database by using the command:

```
Update-Database
```

All pending migrations (in this case the *Initial*-migration) are applied to the database and afterwards the seed method is applied (the appropriate example)

```
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.
```

(c) ketabton.com: The Digital Library

```
Running Seed method.
```

You can see the results of the activities in the SQL explorer:

For the commands `Add-Migration` and `Update-Database` several options are available which can be used to tweak the activities. To see all options, please use

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

and

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

Section 5.3: Doing "Update-Database" within your code

Applications running in non-development environments often require database updates. After using the `Add-Migration` command to create your database patches there's the need to run the updates on other environments, and then the test environment as well.

Challenges commonly faced are:

- no Visual Studio installed on production environments, and
- no connections allowed to connection/customer environments in real life.

A workaround is the following code sequence which checks for updates to be performed, and executes them in order. Please ensure proper transactions & exception handling to ensure no data gets lost in case of errors.

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

where `MyDbConfiguration` is your migration setup somewhere in your sources:

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

Section 5.4: Seeding Data during migrations

After enabling and creating migrations there might be a need to initially fill or migrate data in your database. There are several possibilities but for simple migrations you can use the method 'Seed()' in the file Configuration created after calling `enable-migrations`.

(c) ketabton.com: The Digital Library

The `Seed()` function retrieves a database context as it's only parameter and you are able to perform EF operations inside this function:

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

You can perform all types of activities inside `Seed()`. In case of any failure the complete transaction (even the applied patches) are being rolled back.

An example function that creates data only if a table is empty might look like this:

```
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();
    }
}
```

A nice feature provided by the EF developers is the extension method `AddOrUpdate()`. This method allows to update data based on the primary key or to insert data if it does not exist already (the example is taken from the generated source code of 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" }
    );
}
```

> Please be aware that `Seed()` is called after the **last** patch has been applied. If you need to migration or seed data during patches, other approaches need to be used.

Section 5.5: Initial Entity Framework Code First Migration Step by Step

1. Create a console application.
2. Install EntityFramework nuget package by running `Install-Package EntityFramework` in "Package Manager Console"
3. Add your connection string in app.config file , It's important to include `providerName="System.Data.SqlClient"` in your connection.
4. Create a public class as you wish , some thing like "`Blog`"
5. Create Your ContextClass which inherit from DbContext , some thing like "`BlogContext`"
6. Define a property in your context of DbSet type , some thing like this :

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

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

(c) ketabton.com: The Digital Library

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

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

7. It's important to pass the connection name in constructor (here Your_Connection_Name)
8. In Package Manager Console run `Enable-Migration` command , This will create a migration folder in your project
9. Run `Add-Migration Your_Arbitrary_Migraiton_Name` command , this will create a migration class in migrations folder with two method Up() and Down()
10. Run `Update-Database` command in order to create a database with a blog table

Section 5.6: Using Sql() during migrations

For example: You are going to migrate an existing column from non-required to required. In this case you might need to fill some default values in your migration for rows where the altered fields are actually `NULL`. In case the default value is simple (e.g. "0") you might use a `default` or `defaultSql` property in your column definition. In case it's not so easy, you may use the `Sql()` function in `Up()` or `Down()` member functions of your migrations.

Here's an example. Assuming a class *Author* which contains an email-address as part of the data set. Now we decide to have the email-address as a required field. To migrate existing columns the business has the *smart* idea of creating dummy email-addresses like `fullname@example.com`, where full name is the authors full name without spaces. Adding the `[Required]` attribute to the field `Email` would create the following migration:

```
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));
    }
}
```

This would fail in case some NULL fields are inside the database:

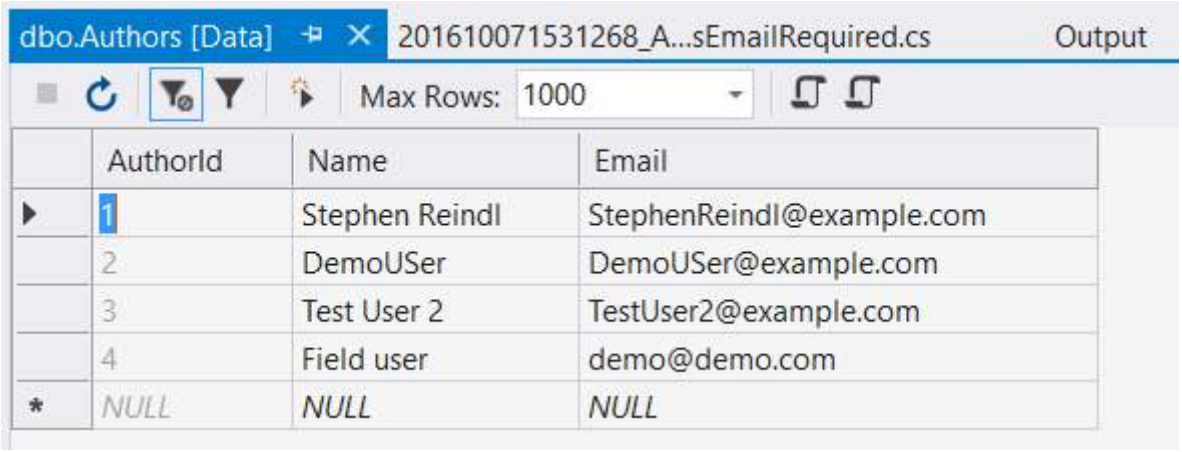
> Cannot insert the value NULL into column 'Email', table 'App.Model.DatabaseContext.dbo.Authors'; column does not allow nulls. UPDATE fails.

Adding the following like **before** the `AlterColumn` command will help:

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

The `update-database` call succeeds and the table looks like this (example data shown):

(c) ketabton.com: The Digital Library



Other Usage

You may use the `Sql()` function for all types of DML and DDL actibities in your database. It is executed as part of the migration transaction; If the SQL fails, the complete migration fails and a rollback is done.

(c) ketabton.com: The Digital Library

Chapter 6: Inheritance with EntityFramework (Code First)

Section 6.1: Table per hierarchy

This approach will generate one table on the database to represent all the inheritance structure.

Example:

```
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; }
```

The table generated will be:

Table: People Fields: Id Name BirthDate Discrimitator AdmissionDate JobDescription LastPurchaseDate TotalVisits

Where 'Discriminator' will hold the name of the subclass on the inheritance and 'AdmissionDate', 'JobDescription', 'LastPurchaseDate', 'TotalVisits' are nullable.

Advantages

- Better performance since no joins are required although for to many columns the database might require many paging operations.
- Simple to use and create
- Easy to add more subclasses and fields

Disadvantages

- Violates the 3rd Normal Form Wikipedia: Third normal form
- Creates lots of nullable fields

Section 6.2: Table per type

This approach will generate (n+1) tables on the database to represent all the inheritance structure where n is the

(c) ketabton.com: The Digital Library

number of subclasses.

How to:

```
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; }
```

The table generated will be:

Table: People Fields: Id Name BirthDate

Table: Employees Fields: PersonId AdmissionDate JobDescription

Table: Customers: Fields: PersonId LastPurchaseDate TotalVisits

Where 'PersonId' on all tables will be a primary key and a constraint to People.Id

Advantages

- Normalized tables
- Easy to add columns and subclasses
- No nullable columns

Disadvantages

- Join is required to retrieve the data
- Subclass inference is more expensive

(c) ketabton.com: The Digital Library

Chapter 7: Code First - Fluent API

Section 7.1: Mapping models

EntityFramewok Fluent API is a powerful and elegant way of mapping your **code-first** domain models to underlying database. This also can be used with *code-first with existing database*. You have two options when using *Fluent API*: you can directly map your models on *OnModelCreating* method or you can create mapper classes which inherits from *EntityTypeConfiguration* and then add that models to *modelBuilder* on *OnModelCreating* method. *Second* option is which I prefer and am going to show example of it.

Step one: Create model.

```
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}";
        }
    }
}
```

Step two: Create mapper class

```
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")
```

(c) ketabton.com: The Digital Library

```
            .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");
    }
}
```

Let us explain mappings:

- **HasKey** - defines the primary key. *Composite primary keys* can also be used. For example: *this.HasKey(m => new { m.DepartmentId, m.PositionId })*.
- **Property** - lets us to configure model properties.
- **HasColumnType** - specify database level column type. Please note that, it can be different for different databases like Oracle and MS SQL.
- **HasDatabaseGeneratedOption** - specifies if property is calculated at database level. Numeric PKs are *DatabaseGeneratedOption.Identity* by default, you should specify *DatabaseGeneratedOption.None* if you do not want them to be so.
- **HasMaxLength** - limits the length of string.
- **IsRequired** - marks the property as requiered.
- **HasPrecision** - lets us to specify precision for decimals.
- **Ignore** - Ignores property completely and does not map it to database. We ignored FullName, because we do not want this column at our table.
- **ToTable** - specify table name and schema name (optional) for model.
- **HasColumnName** - relate property with column name. This is not needed when property names and column names are identical.

Step three: Add mapping class to configurations.

We need to tell EntityFramework to use our mapper class. To do so, we have to add it to *modelBuilder.Configurations* on *OnModelCreating* method:

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

And that is it. We are all set to go.

Section 7.2: Composite Primary Key

By using the .HasKey() method, a set of properties can be explicitly configured as the composite primary key of the entity.

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

(c) ketabton.com: The Digital Library

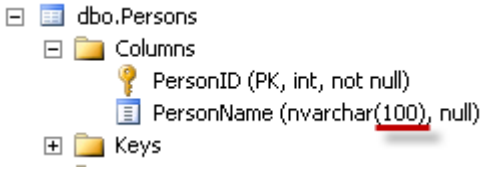
```
// ..

public class PersonContext : DbContext
{
    // ..

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

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

Section 7.3: Maximum Length

By using the .HasMaxLength() method, the maximum character count can be configured for a property.

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

public class PersonContext : DbContext
{
    // ..

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

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

The resulting column with the specified column length:



Section 7.4: Primary Key

By using the .HasKey() method, a property can be explicitly configured as primary key of the entity.

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

public class PersonContext : DbContext
{
    // ..

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

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

(c) ketabton.com: The Digital Library

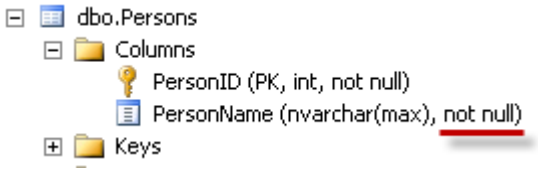
```
}
```

Section 7.5: Required properties (NOT NULL)

By using the .IsRequired() method, properties can be specified as mandatory, which means that the column will have a NOT NULL constraint.

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

public class PersonContext : DbContext
{
    // ..

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

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

The resulting column with the NOT NULL constraint:



Section 7.6: Explict Foreign Key naming

When a navigation property exist on a model, Entity Framework will automatically create a Foreign Key column. If a specific Foreign Key name is desired but is not contained as a property in the model, it can be set explicitly using the Fluent API. By utilizing the `Map` method while establishing the Foreign Key relationship, any unique name can be used for Foreign Keys.

```
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"));
    }
```

(c) ketabton.com: The Digital Library

```
}
```

After specifying the relationship, the `Map` method allows the Foreign Key name to be explicitly set by executing `MapKey`. In this example, what would have resulted in a column name of Employer_Id is now EmployerId.

(c) ketabton.com: The Digital Library

Chapter 8: Mapping relationship with Entity Framework Code First: One-to-one and variations

This topic discusses how to map one-to-one type relationships using Entity Framework.

Section 8.1: Mapping one-to-zero or one

So let's say again that you have the following model:

```
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; }
}
```

And now you want to set it up so that you can express the following specification: one person can have one or zero car, and every car belongs to one person exactly (relationships are bidirectional, so if CarA belongs to PersonA, then PersonA 'owns' CarA).

So let's modify the model a bit: add the navigation properties and the foreign key properties:

```
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; }
}
```

And the configuration:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
  public CarEntityTypeConfiguration()
  {
```

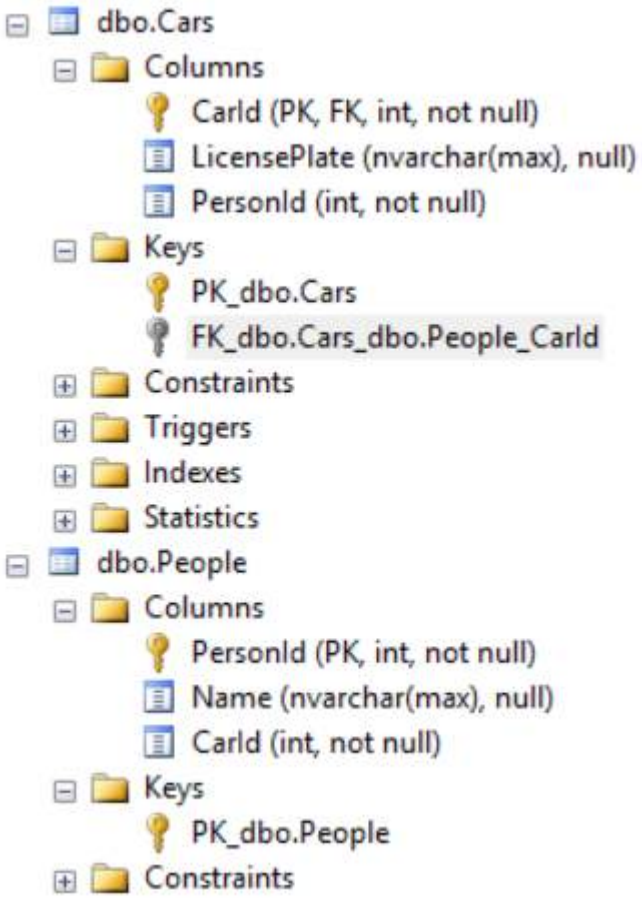
(c) ketabton.com: The Digital Library

```
        this.HasRequired(c => c.Person).WithOptional(p => p.Car);
    }
}
```

By this time this should be self-explanatory. The car has a required person (*HasRequired()*), with the person having an optional car (*WithOptional()*). Again, it doesn't matter which side you configure this relationship from, just be careful when you use the right combination of Has/With and Required/Optional. From the `Person` side, it would look like this:

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

Now let's check out the db schema:



Look closely: you can see that there is no FK in `People` to refer to `Car`. Also, the FK in `Car` is not the `PersonId`, but the `CarId`. Here's the actual script for the FK:

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

So this means that the `CarId` and `PersonId` foregn key properties we have in the model are basically ignored. They are in the database, but they are not foreign keys, as it might be expected. That's because one-to-one mappings does not support adding the FK into your EF model. And that's because one-to-one mappings are quite problematic in a relational database.

(c) ketabton.com: The Digital Library

The idea is that every person can have exactly one car, and that car can only belong to that person. Or there might be person records, which do not have cars associated with them.

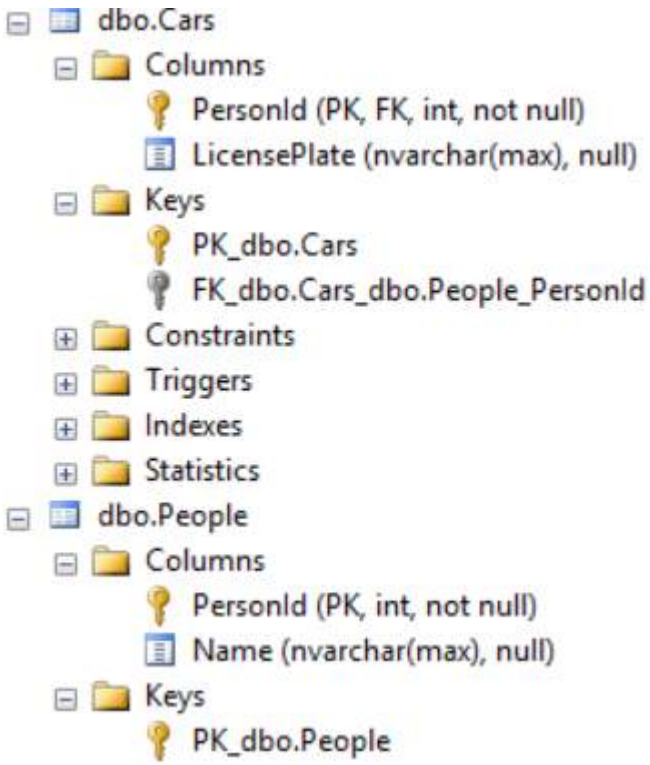
So how could this be represented with foreign keys? Obviously, there could be a `PersonId` in `Car`, and a `CarId` in `People`. To enforce that every person can have only one car, `PersonId` would have to be unique in `Car`. But if `PersonId` is unique in `People`, then how can you add two or more records where `PersonId` is `NULL`(more than one car that don't have owners)? Answer: you can't (well actually, you can create a filtered unique index in SQL Server 2008 and newer, but let's forget about this technicality for a moment; not to mention other RDBMS). Not to mention the case where you specify both ends of the relationship...

The only real way to enforce this rule if the `People` and the `Car` tables have the 'same' primary key (same values in the connected records). And to do this, `CarId` in `Car` must be both a PK and an FK to the PK of People. And this makes the whole schema a mess. When I use this I rather name the PK/FK in `Car` `PersonId`, and configure it accordingly:

```
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);
  }
}
```

Not ideal, but maybe a bit better. Still, you have to be alert when using this solution, because it goes against the usual naming conventions, which might lead you astray. Here's the schema generated from this model:

(c) ketabton.com: The Digital Library



So this relationship is not enforced by the database schema, but by Entity Framework itself. That's why you have to be very careful when you use this, not to let anybody temper directly with the database.

Section 8.2: Mapping one-to-one

Mapping one-to-one (when both sides are required) is also a tricky thing.

Let's imagine how this could be represented with foreign keys. Again, a `CarId` in `People` that refers to `CarId` in `Car`, and a `PersonId` in Car that refers to the `PersonId` in `People`.

Now what happens if you want to insert a car record? In order for this to succeed, there must be a `PersonId` specified in this car record, because it is required. And for this `PersonId` to be valid, the corresponding record in `People` must exist. OK, so let's go ahead and insert the person record. But for this to succeed, a valid `CarId` must be in the person record — but that car is not inserted yet! It cannot be, because we have to insert the referred person record first. But we cannot insert the referred person record, because it refers back to the car record, so that must be inserted first (foreign key-ception :)).

So this cannot be represented the 'logical' way either. Again, you have to drop one of the foreign keys. Which one you drop is up to you. The side that is left with a foreign key is called the 'dependent', the side that is left without a foreign key is called the 'principal'. And again, to ensure the uniqueness in the dependent, the PK has to be the FK, so adding an FK column and importing that to your model is not supported.

So here's the configuration:

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

By now you really should have gotten the logic of it :) Just remember that you can choose the other side as well, just

(c) ketabton.com: The Digital Library

be careful to use the Dependent/Principal versions of WithRequired (and you still have to configure the PK in Car).

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

If you check the DB schema, you'll find that it's exactly the same as it was in the case of the one-to-one or zero solution. That's because again, this is not enforced by the schema, but by EF itself. So again, be careful :)

Section 8.3: Mapping one or zero-to-one or zero

And to finish off, let's briefly look at the case when both sides are optional.

By now you should be really bored with these examples :), so I'm not going into the details and play with the idea of having two FK-s and the potential problems and warn you about the dangers of not enforcing these rules in the schema but in just EF itself.

Here's the config you need to apply:

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

Again, you can configure from the other side as well, just be careful to use the right methods :)

(c) ketabton.com: The Digital Library

Chapter 9: Mapping relationship with Entity Framework Code First: One-to-many and Many-to-many

The topic discusses how you can map one-to-many and many-to-many relationships using Entity Framework Code First.

Section 9.1: Mapping one-to-many

So let's say you have two different entities, something like this:

```
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; }
}
```

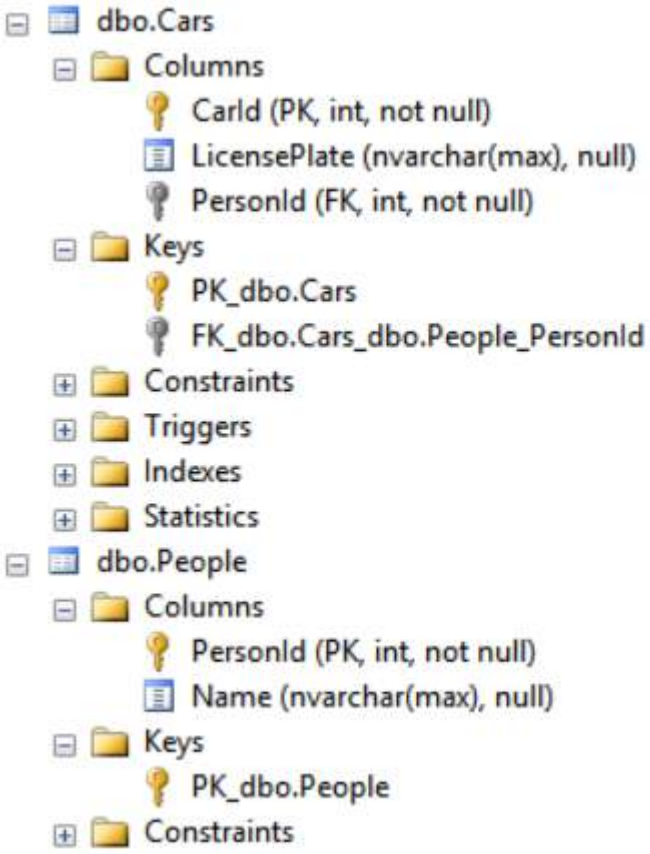
And you want to setup a one-to-many relationship between them, that is, one person can have zero, one or more cars, and one car belongs to one person exactly. Every relationship is bidirectional, so if a person has a car, the car belongs to that person.

To do this just modify your model classes:

```
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; }
}
```

And that's it :) You already have your relationship set up. In the database, this is represented with foreign keys, of course.

(c) ketabton.com: The Digital Library



Section 9.2: Mapping one-to-many: against the convention

In the last example, you can see that EF figures out which column is the foreign key and where should it point to. How? By using conventions. Having a property of type `Person` that is named `Person` with a `PersonId` property leads EF to conclude that `PersonId` is a foreign key, and it points to the primary key of the table represented by the type `Person`.

But what if you were to change *PersonId* to *OwnerId* and *Person* to *Owner* in the **Car** type?

public class Car { public int CarId { get; set; } public string LicensePlate { get; set; } public int OwnerId { get; set; } public virtual Person Owner { get; set; } }

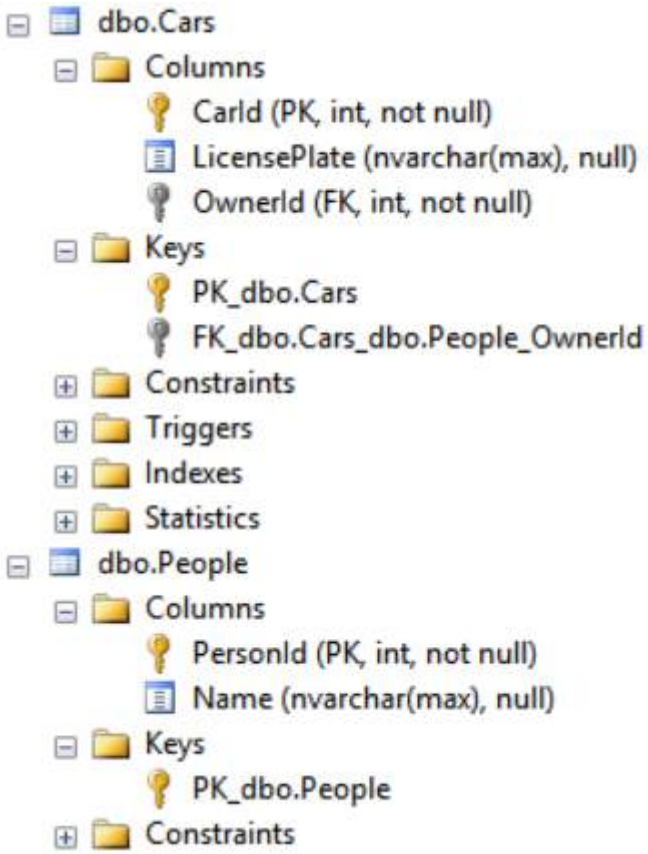
Well, unfortunately in this case, the conventions are not enough to produce the correct DB schema:

No worries; you can help EF with some hints about your relationships and keys in the model. Simply configure your `Car` type to use the `OwnerId` property as the FK. Create an entity type configuration and apply it in your `OnModelCreating()`:

(c) ketabton.com: The Digital Library

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

This basically says that `Car` has a required property, `Owner` (*HasRequired()*) and in the type of `Owner`, the `Cars` property is used to refer back to the car entities (*WithMany()*). And finally the property representing the foreign key is specified (*HasForeignKey()*). This gives us the schema we want:



You could configure the relationship from the `Person` side as well:

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

The idea is the same, just the sides are different (note how you can read the whole thing: 'this person has many cars, each car with a required owner'). Doesn't matter if you configure the relationship from the `Person` side or the `Car` side. You can even include both, but in this case be careful to specify the same relationship on both sides!

Section 9.3: Mapping zero or one-to-many

In the previous examples a car cannot exist without a person. What if you wanted the person to be optional from the car side? Well, it's kind of easy, knowing how to do one-to-many. Just change the `PersonId` in `Car` to be nullable:

```
public class Car
{
```

(c) ketabton.com: The Digital Library

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

And then use the *HasOptional()* (or *WithOptional()*, depending from which side you do the configuration):

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

Section 9.4: Many-to-many

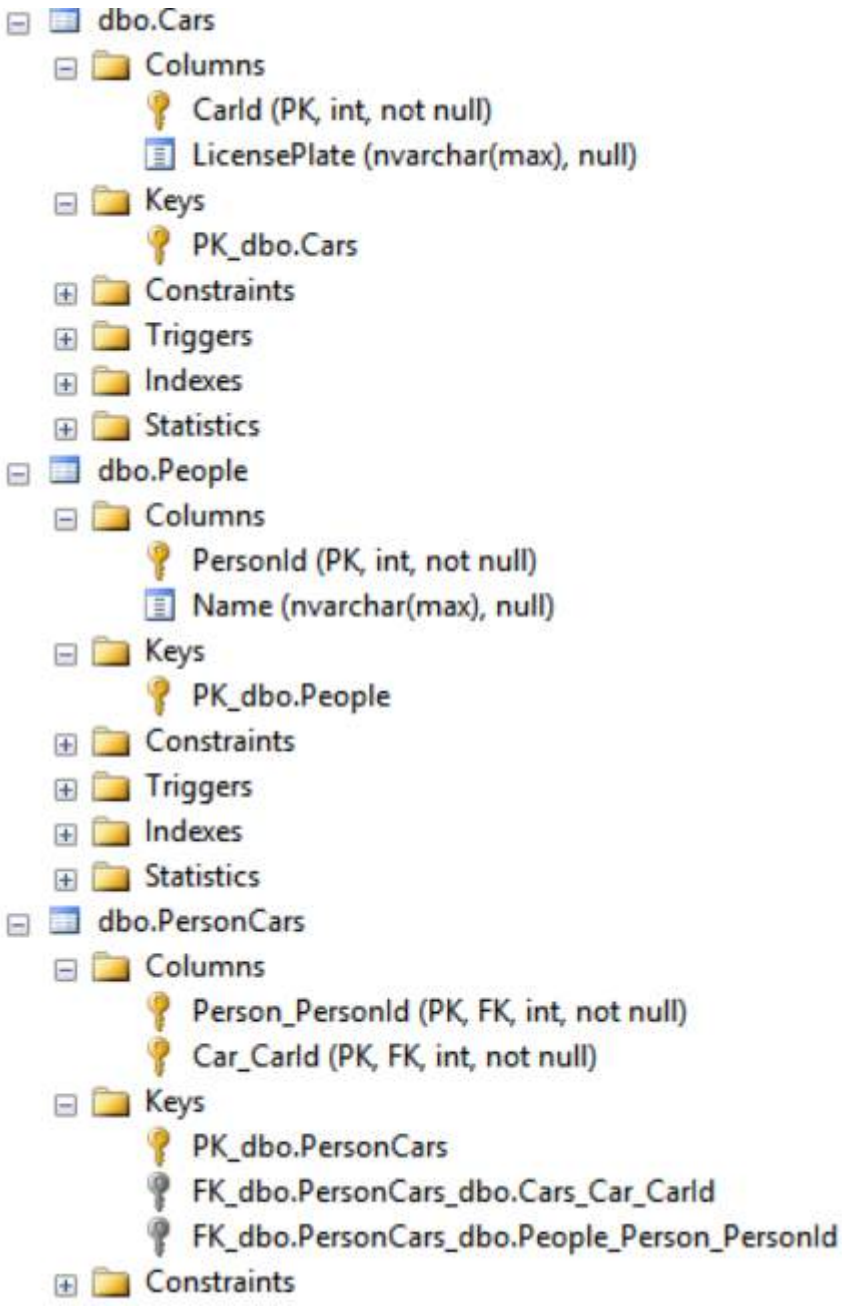
Let's move on to the other scenario, where every person can have multiple cars and every car can have multiple owners (but again, the relationship is bidirectional). This is a many-to-many relationship. The easiest way is to let EF do it's magic using conventions.

Just change the model like this:

```
 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; }
}
```

And the schema:

(c) ketabton.com: The Digital Library



Almost perfect. As you can see, EF recognized the need for a join table, where you can keep track of person-car pairings.

Section 9.5: Many-to-many: customizing the join table

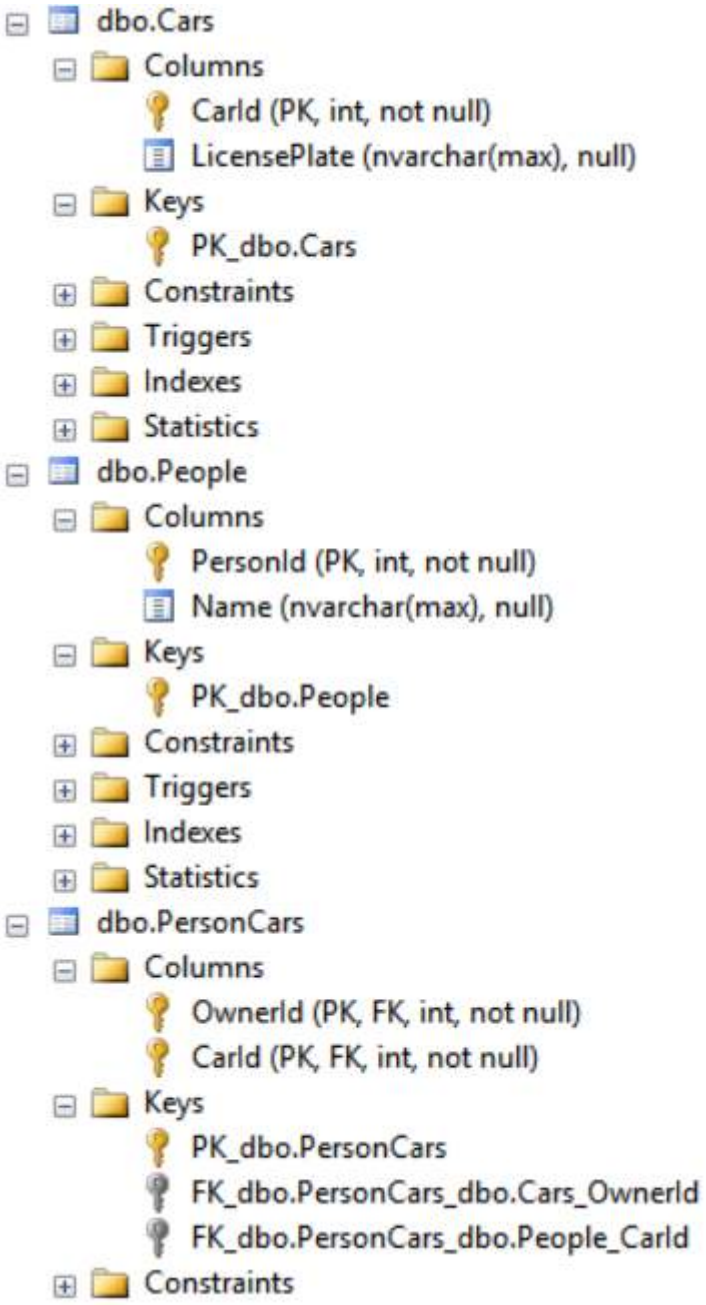
You might want to rename the fields in the join table to be a little more friendly. You can do this by using the usual configuration methods (again, it doesn't matter which side you do the configuration from):

```
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");
              }
        );
  }
```

(c) ketabton.com: The Digital Library

```
}
```

Quite easy to read even: this car has many owners (*HasMany()*), with each owner having many cars (*WithMany()*). Map this so that you map the left key to OwnerId (*MapLeftKey()*), the right key to CarId (*MapRightKey()*) and the whole thing to the table PersonCars (*ToTable()*). And this gives you exactly that schema:



Section 9.6: Many-to-many: custom join entity

I have to admit, I'm not really a fan of letting EF infer the join table wihtout a join entity. You cannot track extra information to a person-car association (let's say the date from which it is valid), because you can't modify the table.

Also, the `CarId` in the join table is part of the primary key, so if the family buys a new car, you have to first delete the old associations and add new ones. EF hides this from you, but this means that you have to do these two operations instead of a simple update (not to mention that frequent inserts/deletes might lead to index fragmentation — good thing there is an easy fix for that).

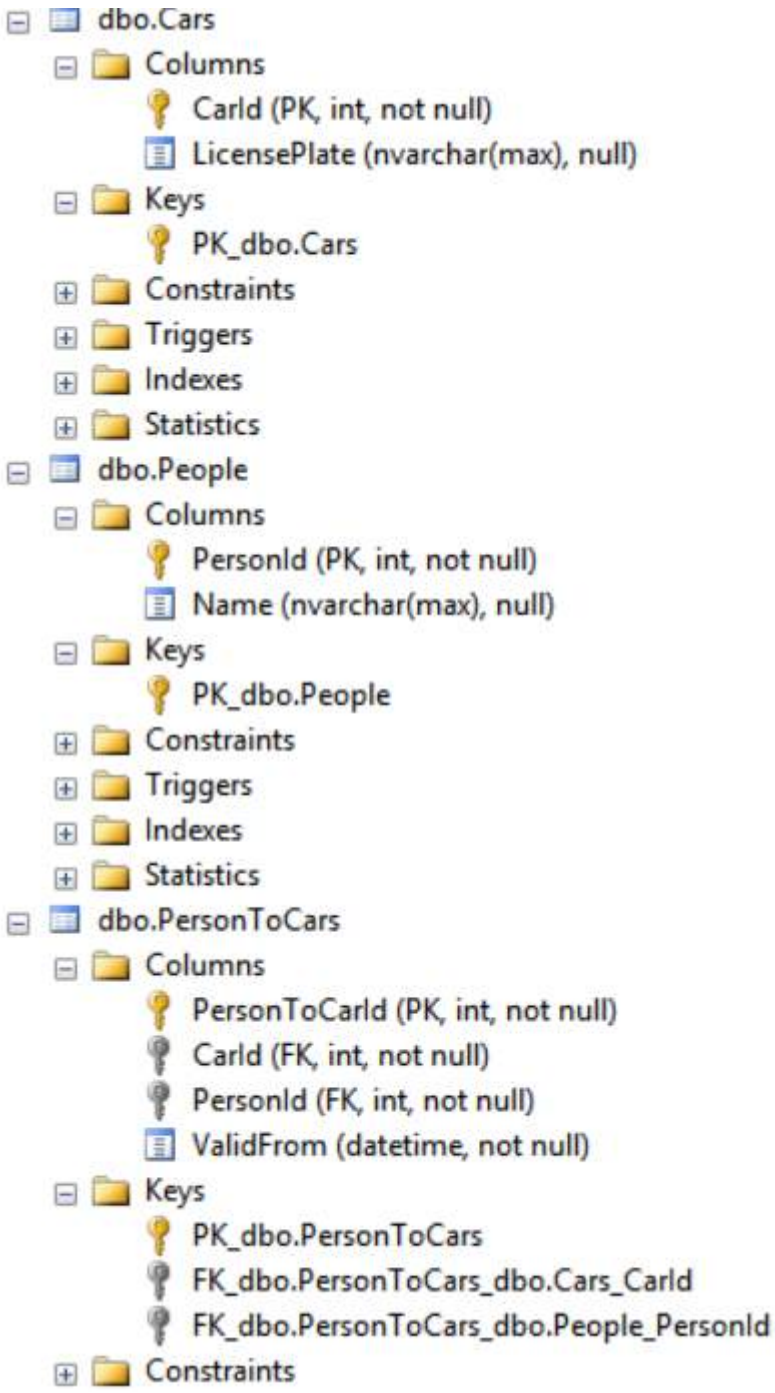
In this case what you can do is create a join entity that has a reference to both one specific car and one specific person. Basically you look at your many-to-many association as a combinations of two one-to-many associations:

(c) ketabton.com: The Digital Library

```
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; }
}
```

This gives me much more control and it's a lot more flexible. I can now add custom data to the association and every association has its own primary key, so I can update the car or the owner reference in them.

(c) ketabton.com: The Digital Library



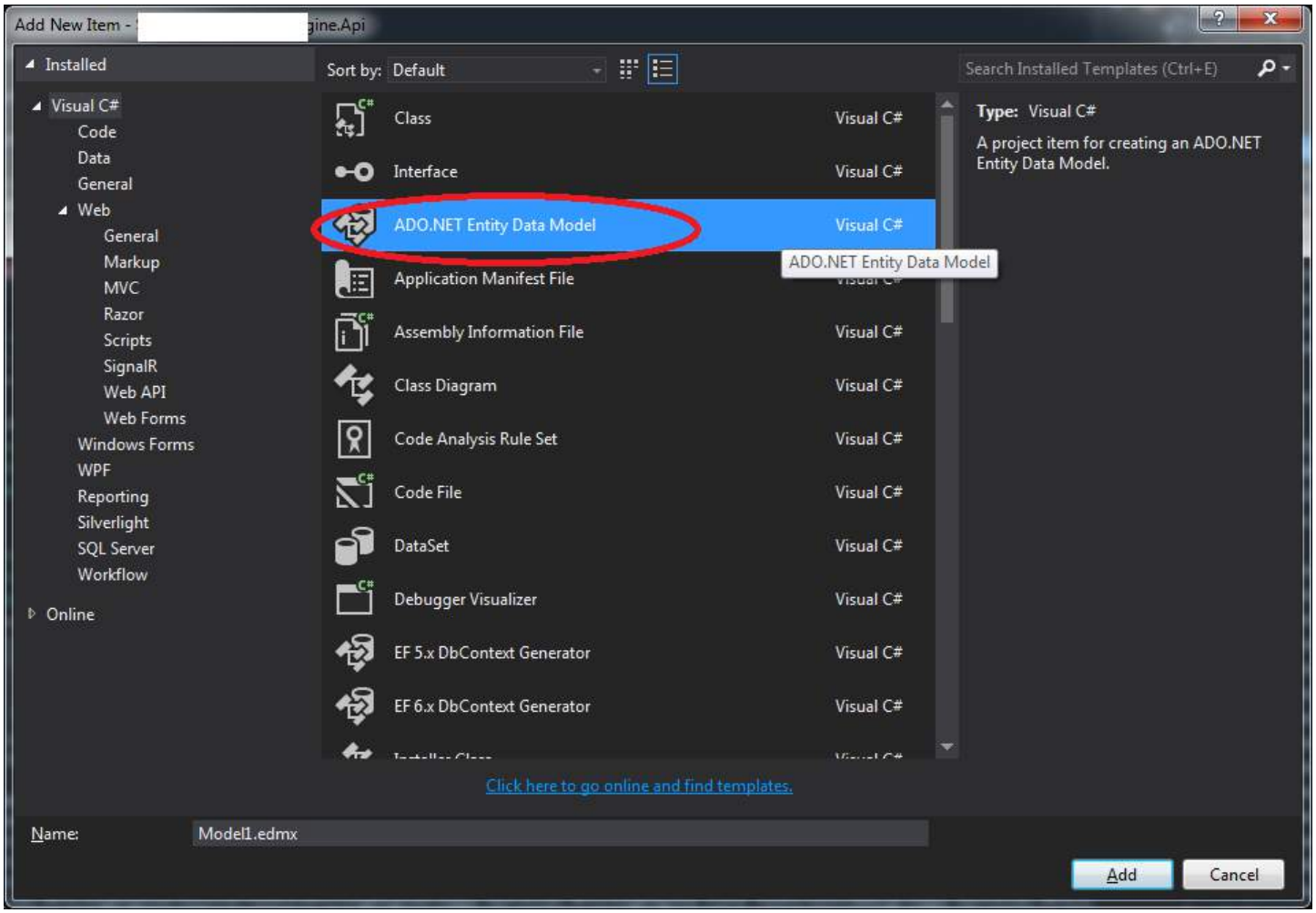
Note that this really is just a combination of two one-to-many relationships, so you can use all the configuration options discussed in the previous examples.

(c) ketabton.com: The Digital Library

Chapter 10: Database first model generation

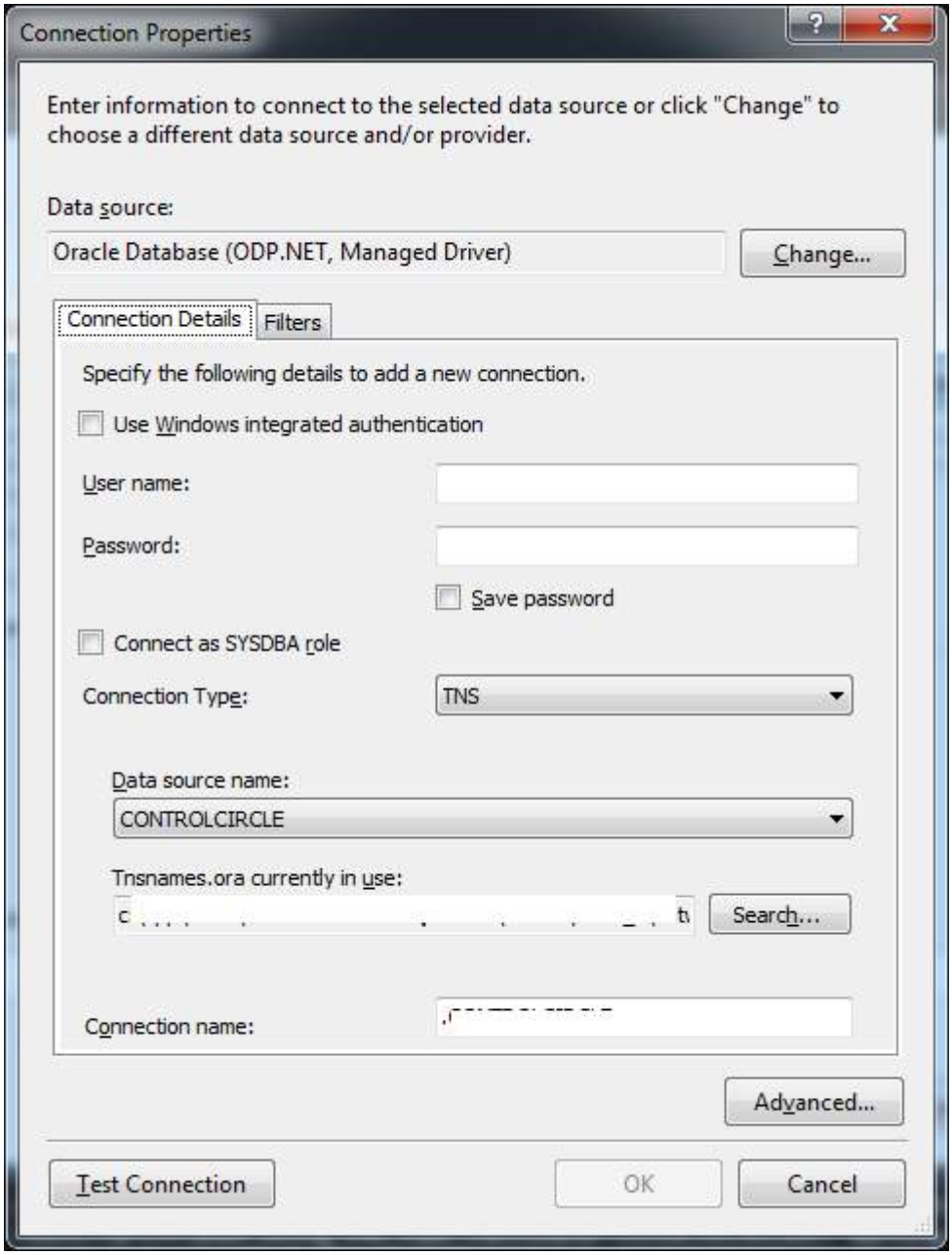
Section 10.1: Generating model from database

In `Visual Studio` go to your `Solution Explorer` then click on `Project` you will be adding model `Right mouse`. Choose `ADO.NET Entity Data Model`



Then choose `Generate from database` and click `Next` in next window click `New Connection...` and point to the database you want to generate model from (Could be `MSSQL`, `MySQL` or `Oracle`)

(c) ketabton.com: The Digital Library



After you done this click `Test Connection` to see if you have configured connection properly (do not go any further if it fails here).

Click `Next` then choose options that you want (like style for generating entity names or to add foreign keys).

Click `Next` again, at this point you should have model generated from database.

Section 10.2: Adding data annotations to the generated model

In T4 code-generation strategy used by Entity Framework 5 and higher, data annotation attributes are not included by default. To include data annotations on top of certain property every model regeneration, open template file included with EDMX (with `.tt` extension) then add a `using` statement under `UsingDirectives` method like below:

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

(c) ketabton.com: The Digital Library

```
using System.ComponentModel.DataAnnotations;  // --> add this line
```

As an example, suppose the template should include `KeyAttribute` which indicates a primary key property. To insert `KeyAttribute` automatically while regenerating model, find part of code containing `codeStringGenerator.Property` as below:

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

Then, insert an if-condition to check key property as this:

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

By applying changes above, all generated model classes will have `KeyAttribute` on their primary key property after updating model from database.

Before

```
using System;

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

After

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

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

(c) ketabton.com: The Digital Library

Chapter 11: Complex Types

Section 11.1: Code First Complex Types

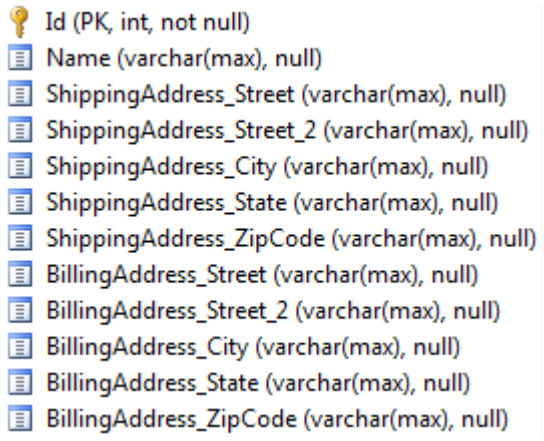
A complex type allows you to map selected fields of a database table into a single type that is a child of the main type.

```
[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; }
}
```

This complex type can then be used in multiple entity types. It can even be used more than once in the same entity type.

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

This entity type would then be stored in a table in the database that would look something like this.



Of course, in this case, a 1:n association (Customer-Address) would be the preferred model, but the example shows how complex types can be used.

(c) ketabton.com: The Digital Library

Chapter 12: Database Initialisers

Section 12.1: CreateDatabaseIfNotExists

Implementation of `IDatabaseInitializer` that is used in EntityFramework by default. As the name implies, it creates the database if none exists. However when you change the model, it throws an exception.

Usage:

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

Section 12.2: DropCreateDatabaseIfModelChanges

This implementation of `IDatabaseInitializer` drops and recreates the database if the model changes automatically.

Usage:

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

Section 12.3: DropCreateDatabaseAlways

This implementation of `IDatabaseInitializer` drops and recreates the database every time your context is used in applications app domain. Beware of the data loss due to the fact, that the database is recreated.

Usage:

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

Section 12.4: Custom database initializer

You can create your own implementation of `IDatabaseInitializer`.

Example implementation of an initializer, that will migrate the database to 0 and then migrate all the way to the newest migration (usefull e.g. when running integration tests). In order to do that you would need a `DbMigrationsConfiguration` type too.

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

(c) ketabton.com: The Digital Library

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

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

Section 12.5: MigrateDatabaseToLatestVersion

An implementation of `IDatabaseInitializer` that will use Code First Migrations to update the database to the latest version. To use this initializer you have to use `DbMigrationsConfiguration` type too.

Usage:

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

(c) ketabton.com: The Digital Library

Chapter 13: Tracking vs. No-Tracking

Section 13.1: No-tracking queries

- No tracking queries are useful when the results are used in a `read-only` scenario
- They are `quicker to execute` because there is no need to setup change tracking information

Example :

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

With EF Core 1.0 you are also able to change the default tracking behavior at the `context instance` level.

Example :

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

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

Section 13.2: Tracking queries

- By default, queries that return entity types are **tracking**
- This means you can make changes to those entity instances and have those changes persisted by `SaveChanges()`

Example :

- The change to the `book` rating will be detected and persisted to the database during `SaveChanges()`.

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

Section 13.3: Tracking and projections

- Even if the result type of the query isn't an entity type, if the result `contains entity` types they will still be `tracked by default`

Example :

- In the following query, which returns an `anonymous type`, the instances of `Book` in the result set `will be tracked`

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

(c) ketabton.com: The Digital Library

```
}
```

- If the result set `does not` contain any `entity` types, then `no tracking` is performed

Example :

- In the following query, which returns an `anonymous type` with some of the values from the entity (but `no instances` of the actual `entity` type), there is **no tracking** performed.

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

(c) ketabton.com: The Digital Library

Chapter 14: Transactions

Section 14.1: Database.BeginTransaction()

Multiple operations can be executed against a single transaction so that changes can be rolled back if any of the operations fail.

```
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
        }
    }
}
```

Note that it may be a developers' convention to call `transaction.Rollback()` explicitly, because it makes the code more self-explanatory. Also, there *may* be (less well-known) query providers for Entity Framework out there that don't implement `Dipsose` correctly, which would also require an explicit `transaction.Rollback()` call.

(c) ketabton.com: The Digital Library

Chapter 15: Managing entity state

Section 15.1: Setting state Added of a single entity

`EntityState.Added` can be set in two fully equivalent ways:

1. By setting the state of its entry in the context:

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

2. By adding it to a `DbSet` of the context:

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

When calling `SaveChanges`, the entity will be inserted into the database. When it's got an identity column (an auto-set, auto-incrementing primary key), then after `SaveChanges`, the primary key property of the entity will contain the newly generated value, *even when this property already had a value*.

Section 15.2: Setting state Added of an object graph

Setting the state of an *object graph* (a collection of related entities) to `Added` is different than setting a single entity as `Added` (see this example).

In the example, we store planets and their moons:

Class model

```
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; }
}
```

Context

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

We use an instance of this context to add planets and their moons:

(c) ketabton.com: The Digital Library

Example

```
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);
```

Output:

```
Added
Added
```

What we see here is that adding a `Planet` also sets the state of a moon to `Added`.

When setting an entity's state to `Added`, all entities in its navigation properties (properties that "navigate" to other entities, like `Planet.Moons`) are also marked as `Added`, *unless they already are attached to the context*.

(c) ketabton.com: The Digital Library

Chapter 16: Loading related entities

Section 16.1: Eager loading

Eager loading lets you load all your needed entities at once. If you prefer to get all your entities to work on in one database call, then *Eager loading* is the way to go. It also lets you load multiple levels.

You have *two options* to load related entities, you can choose either *strongly typed* or *string* overloads of the *Include* method.

Strongly typed.

```
// 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();
```

This method is available since Entity Framework 4.1. Make sure you have the reference `using System.Data.Entity;` set.

String overload.

```
// 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();
```

Section 16.2: Explicit loading

After turning *Lazy loading* off you can lazily load entities by explicitly calling *Load* method for entries. *Reference* is used to load single navigation properties, whereas *Collection* is used to get collections.

```
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();
```

As it is on *Eager loading* you can use overloads of above methods to load entiteis by their names:

(c) ketabton.com: The Digital Library

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

Filter related entities.

Using *Query* method we can filter loaded related entities:

```
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();
```

Section 16.3: Lazy loading

Lazy loading is enabled by default. Lazy loading is achieved by creating derived proxy classes and overriding virtual navigation proeprties. Lazy loading occurs when property is accessed for the first time.

```
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.
}
```

To turn Lazy loading off for specific navigation properties just remove virtual keyword from property declaration:

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

If you want to completely turn off Lazy loading, then you have to change Configuration, for example, at *Context constructor*:

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

Note: Please remember to *turn off* Lazy loading if your are using serialization. Because serializers access every property you are going to load all of them from database. Additionally, you can run into loop between navigation properties.

Section 16.4: Projection Queries

If one needs related data in a denormalized type, or e.g. only a subset of columns one can use projection queries. If there is no reason for using an extra type, there is the possibility to join the values into an anonymous type.

(c) ketabton.com: The Digital Library

```
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
                       };
```

Or with query-syntax:

```
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);
```

(c) ketabton.com: The Digital Library

Chapter 17: Model Restraints

Section 17.1: One-to-many relationships

UserType belongs to many Users <-> Users have one UserType

One way navigation property with required

```
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);
```

One way navigation property with optional (foreign key must be `Nullable` type)

```
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);
```

Two way navigation property with (required/optional change the foreign key property as needed)

```
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;}
}
```

Required

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

Optional

(c) ketabton.com: The Digital Library

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

(c) ketabton.com: The Digital Library

Chapter 18: Entity Framework with PostgreSQL

Section 18.1: Pre-Steps needed in order to use Entity Framework 6.1.3 with PostgresSql using Npgsqlddexprovider

1)Took backup of Machine.config from locations C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config and C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config

2)Copy them to different location and edit them as

> a)locate and add under **<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)if already exist above entry, check verison and update it.

3. Replace original files with changed ones.
4. run Developer Command Prompt for VS2013 as Administrator.
5. if Npgsql already installed use command " gacutil -u Npgsql " to uninstall then install new version of Npgsql 2.5.0 by command " gacutil -i [path of dll] "
6. Do above for Mono.Security 4.0.0.0
7. Download NpgsqlDdexProvider-2.2.0-VS2013.zip and run NpgsqlDdexProvider.vsix from it(Do close all instances of visual studio)
8. Found EFTools6.1.3-beta1ForVS2013.msi and run it.
9. After crating new project, Install version of EntityFramework(6.1.3), NpgSql(2.5.0) and NpgSql.EntityFramework(2.5.0) from Manage Nuget Packages.10)Its Done go ahead...Add new Entity Data Model in your MVc project

(c) ketabton.com: The Digital Library

Chapter 19: Entity Framework with SQLite

SQLite is a self-contained, serverless, transactional SQL database. It can be used within a .NET application by utilizing both a freely available .NET SQLite library and Entity Framework SQLite provider. This topic will go into setup and usage of the Entity Framework SQLite provider.

Section 19.1: Setting up a project to use Entity Framework with an SQLite provider

The Entity Framework library comes only with an SQL Server provider. To use SQLite will require additional dependencies and configuration. All required dependencies are available on NuGet.

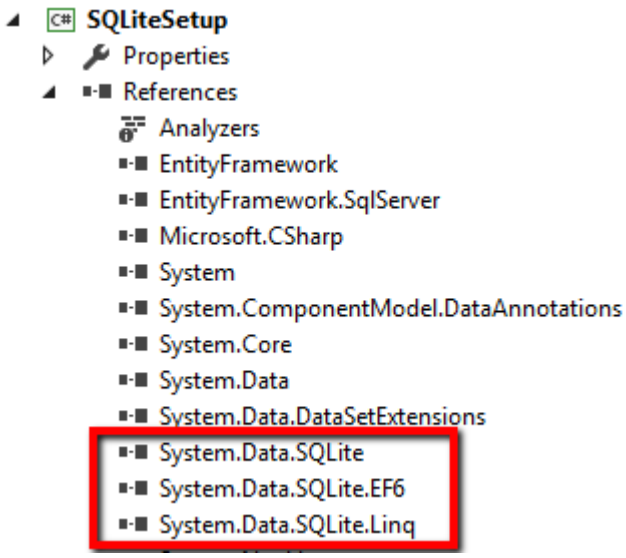
Install SQLite Managed Libraries

All of the mananged depedencies can be installed using the NuGet Package Manager Console. Run the command `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 project
Attempting to resolve dependencies for package 'System.Data.SQLite.1.0.104' with DependencyBehavior 'Lowest'
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\Projects
Added package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Projects\
Added package 'EntityFramework.6.0.0' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Entit
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Entit

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 201
Added package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015
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 2015
Added package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\
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\Syste
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 201
Added package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015
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\Pro
Added package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Proj
Added package 'System.Data.SQLite.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite 1.0.104' to SQLiteSetup
```

As shown above, when installing `System.Data.SQLite`, all related managed libraries are installed with it. This includes `System.Data.SQLite.EF6`, the EF provider for SQLite. The project also now references the assemblies required to use the SQLite provider.

(c) ketabton.com: The Digital Library



Including Unmanaged Library

The SQLite managed libraries are dependent on an unmanaged assembly named `SQLite.Interop.dll`. It is included with the package assemblies downloaded with the SQLite package, and they are automatically copied into your build directory when you build the project. However, because it's unmanaged, it will not be included in your reference list. But make note, this assembly most be distributed with the application for the SQLite assemblies to work.

Note: This assembly is bit-dependent, meaning you will need to include a specific assembly for each bitness you plan to support (x86/x64).

Editing the project's App.config

The `app.config` file will require some modifications before SQLite can be used as an Entity Framework provider.

Required Fixes

When installing the package, the `app.config` file is automatically updated to include the necessary entries for SQLite and SQLite EF. Unfortunately these entries contain some errors. They need to be modified before it will work correctly.

First, locate the `DbProviderFactories`element in the config file. It is within the `system.data` element and will contain the following

```
<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>
```

This can be simplified to contain a single entry

```
<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>
```

(c) ketabton.com: The Digital Library

With this, we have specified the EF6 SQLite providers should use the SQLite factory.

Add SQLite connection string

Connection strings can be added to the configuration file within the root element. Add a connection string for accessing an SQLite database.

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

The important thing to note here is the `provider`. It has been set to `System.Data.SQLite.EF6`. This tells EF that when we use this connection string, we want to use SQLite. The `data source` specified is just an example and will be dependent on the location and name of your SQLite database.

Your first SQLite DbContext

With all the installation and configuration complete, you can now start using a `DbContext` that will work on your SQLite database.

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

By specifying `name=TestContext`, I have indicating that the TestContext connection string located in the `app.config`file should be used to create the context. That connection string was configured to use SQLite, so this context will use an SQLite database.

(c) ketabton.com: The Digital Library

Chapter 20: .t4 templates in entity framework

Section 20.1: Dynamically adding Interfaces to model

When working with existing model that is quite big and is being regenerated quite often in cases where abstraction needed it might be costly to manually go around redecorating model with interfaces. In such cases one might want to add some dynamic behavior to model generation.

Following example will show how automatically add interfaces on classes that have specific column names:

In your model go to `.tt` file modify the `EntityClassOpening` method in following way, this will add `IPolicyNumber` interface on entities that have `POLICY_NO` column, and `IUniqueId` on `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);
}
```

This is one specific case but it shows a power of being able to modify `.tt` templates.

Section 20.2: Adding XML Documentation to Entity Classes

On every generated model classes there are no documentation comments added by default. If you want to use XML documentation comments for every generated entity classes, find this part inside *[modelname]*.tt (*modelname* is current EDMX file name):

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

You can add the XML documentation comments before `UsingDirectives` line as shown in example below:

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

(c) ketabton.com: The Digital Library

```
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
```

The generated documentation comment should be includes entity name as given below.

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

(c) ketabton.com: The Digital Library

Chapter 21: Advanced mapping scenarios: entity splitting, table splitting

How to configure your EF model to support entity splitting or table splitting.

Section 21.1: Entity splitting

So let's say you have an entity class like this:

```
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; }
}
```

And then let's say that you want to map this Person entity into two tables — one with the PersonId and the Name, and another one with the address details. Of course you would need the PersonId here as well in order to identify which person the address belongs to. So basically what you want is to split the entity into two (or even more) parts. Hence the name, entity splitting. You can do this by mapping each of the properties to a different table:

```
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");
      });
  }
}
```

This will create two tables: People and PersonDetails. Person has two fields, PersonId and Name, PersonDetails has four columns, PersonId, AddressLine, City and ZipCode. In People, PersonId is the primary key. In PersonDetails the primary key is also PersonId, but it is also a foreign key referencing PersonId in the Person table.

If you query the People DbSet, EF will do a join on the PersonIds to get the data from both tables to populate the entities.

You can also change the name of the columns:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
  modelBuilder.Entity<Person>().Map(m =>
  {
    m.Properties(t => new { t.PersonId });
    m.Property(t => t.Name).HasColumnName("PersonName");
```

(c) ketabton.com: The Digital Library

```
    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");
  });
}
```

This will create the same table structure, but in the People table there will be a PersonName column instead of the Name column, and in the PersonDetails table there will be a ProprietorId instead of the PersonId column.

Section 21.2: Table splitting

And now let's say you want to do the opposite of entity splitting: instead of mapping one entity into two tables, you would like to map one table into two entities. This is called table splitting. Let's say you have one table with five columns: PersonId, Name, AddressLine, City, ZipCode, where PersonId is the primary key. And then you would like to create an EF model like this:

```
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; }
}
```

One thing jumps right out: there is no AddressId in Address. That's because the two entities are mapped to the same table, so they must have the same primary key as well. If you do table splitting, this is something you just have to deal with. So besides table splitting, you also have to configure the Address entity and specify the primary key. And here's how:

```
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"));
  }
}
```

(c) ketabton.com: The Digital Library

Chapter 22: Best Practices For Entity Framework (Simple & Professional)

This article is to introduce a simple and professional practice to use Entity Framework.

Simple: because it only needs one class (with one interface)

Professional: because it applies SOLID architecture principles

I don't wish to talk more.... let's enjoy it!

Section 22.1: 1- Entity Framework @ Data layer (Basics)

In this article we will use a simple database called “Company” with two tables:

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

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

1-1 Generate Entity Framework code

In this layer we generate the Entity Framework code (in project library) (see this article in how can you do that) then you will have the following classes

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

1-2 Create basic Interface

We will create one interface for our basics functions

```
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
```

(c) ketabton.com: The Digital Library

```
}
```

1-3 Implementing basic Interface

```
/// <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);
    }
```

(c) ketabton.com: The Digital Library

```
    /// <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>
```

(c) ketabton.com: The Digital Library

```
    /// 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

}
```

Section 22.2: 2- Entity Framework @ Business layer

In this layer we will write the application business.

It is recommended for each presentation screen, you create the business interface and implementation class that contain all required functions for the screen.

Below we will write the business for product screen as example

```
/// <summary>
```

(c) ketabton.com: The Digital Library

```
/// 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)
    {
```

(c) ketabton.com: The Digital Library

```
        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>
```

(c) ketabton.com: The Digital Library

```
    /// 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
}
```

Section 22.3: 3- Using Business layer @ Presentation layer (MVC)

In this example we will use the Business layer in Presentation layer. And we will use MVC as example of Presentation layer (but you can use any other Presentation layer).

We need first to register the IoC (we will use Unity, but you can use any IoC), then write our Presentation layer

3-1 Register Unity types within MVC

(c) ketabton.com: The Digital Library

3-1-1 Add “Unity bootstrapper for ASP.NET MVC” NuGet backage

3-1-2 Add UnityWebActivator.Start(); in Global.asax.cs file (Application_Start() function)

3-1-3 Modify UnityConfig.RegisterTypes function as following

```
    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 Using Business layer @ Presentation layer (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
```

(c) ketabton.com: The Digital Library

```
            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
}
```

Section 22.4: 4- Entity Framework @ Unit Test Layer

In Unit Test layer we usually test the Business Layer functionalities. And in order to do this, we will remove the Data Layer (Entity Framework) dependencies.

And the question now is: How can I remove the Entity Framework dependencies in order to unit test the Business Layer functions?

And the answer is simple: we will a fake implementation for IDbRepository Interface then we can do our unit test

4-1 Implementing basic Interface (fake implementation)

```
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
```

(c) ketabton.com: The Digital Library

```
    #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)
    {
```

(c) ketabton.com: The Digital Library

```
        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 Run your unit testing

```
[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();
```

(c) ketabton.com: The Digital Library

```
        _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());
    }
}
```

(c) ketabton.com: The Digital Library

Chapter 23: Optimization Techniques in EF

Section 23.1: Using AsNoTracking

Bad Example:

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

return location;
```

Since the above code is simply returning an entity without modifying or adding it, we can avoid tracking cost.

Good Example:

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

return location;
```

When we use function `AsNoTracking()` we are explicitly telling Entity Framework that the entities are not tracked by the context. This can be especially useful when retrieving large amounts of data from your data store. If you want to make changes to un-tracked entities however, you must remember to attach them before calling `SaveChanges`.

Section 23.2: Execute queries in the database when possible, not in memory

Suppose we want to count how many counties are there in Texas:

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

The query is correct, but inefficient. `States.Single(...)` loads a state from the database. Next, `Counties` loads all 254 counties with all of their fields in a second query. `.Count()` is then performed *in memory* on the loaded `Counties` collection.
We've loaded a lot of data we don't need, and we can do better:

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

Here we only do one query, which in SQL translates to a count and a join. We return only the count from the database - we've saved returning rows, fields, and creation of objects.

It is easy to see where the query is made by looking at the collection type: `IQueryable<T>` vs. `IEnumerable<T>`.

Section 23.3: Loading Only Required Data

One problem often seen in code is loading all the data. This will greatly increase the load on the server.

Let's say I have a model called "location" that has 10 fields in it, but not all the fields are required at the same time. Let's say I only want the 'LocationName' parameter of that model.

Bad Example

(c) ketabton.com: The Digital Library

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

return location.Name;
```

Good Example

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

return location;
```

The code in the "good example" will only fetch 'LocationName' and nothing else.

Note that since no entity is materialized in this example, `AsNoTracking()` isn't necessary. There's nothing to be tracked anyway.

Fetching more fields with Anonymous Types

```
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;
```

Same as the example before, only the fields 'LocationName' and 'LocationArea' will be retrieved from the database, the Anonymous Type can hold as many values you want.

Section 23.4: Execute multiple queries async and in parallel

When using async queries, you can execute multiple queries at the same time, but not on the same context. If the execution time of one query is 10s, the time for the bad example will be 20s, while the time for the good example will be 10s.

Bad Example

```
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);
}
```

Good Example

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

(c) ketabton.com: The Digital Library

```
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;
```

Section 23.5: Working with stub entities

Say we have `Products` and `Categorys` in a many-to-many relationship:

```
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; }
}
```

If we want to add a `Category` to a `Product`, we have to load the product and add the category to its `Categories`, for example:

Bad Example:

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

(where `db` is a `DbContext` subclass).

This creates one record in the junction table between `Product` and `Category`. However, this table only contains two `Id` values. It's a waste of resources to load two full entities in order to create one tiny record.

A more efficient way is to use *stub entities*, i.e. entity objects, created in memory, containing only the bare minimum of data, usually only an `Id` value. This is what it looks like:

Good example:

```
// Create two stub entities
```

(c) ketabton.com: The Digital Library

```
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();
```

The end result is the same, but it avoids two roundtrips to the database.

Prevent duplicates

It you want to check if the association already exists, a cheap query suffices. For example:

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

Again, this won't load full entities into memory. It effectively queries the junction table and only returns a boolean.

Section 23.6: Disable change tracking and proxy generation

If you just want to get data, but not modify anything, you can turn off change tracking and proxy creation. This will improve your performance and also prevent lazy loading.

Bad Example:

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

Good Example:

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

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

It is particularly common to turn these off from within the constructor of your context, especially if you wish these to be set across your solution:

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

    //snip
}
```

(c) ketabton.com: The Digital Library

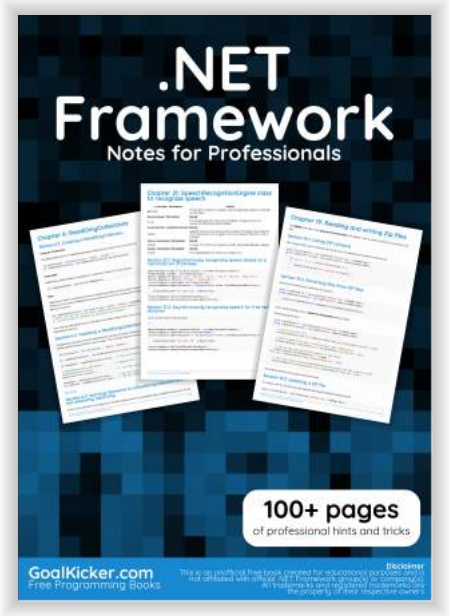
Credits

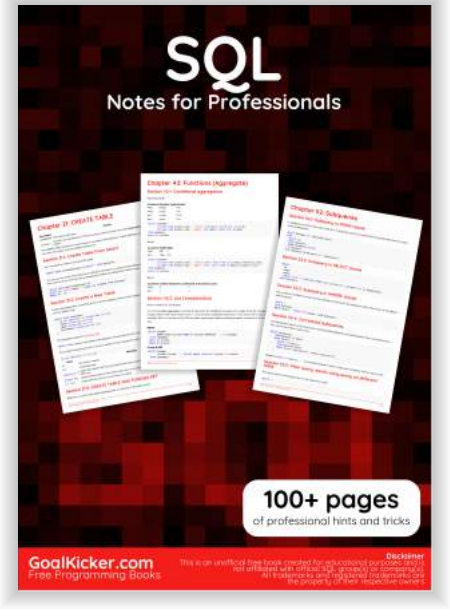
Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

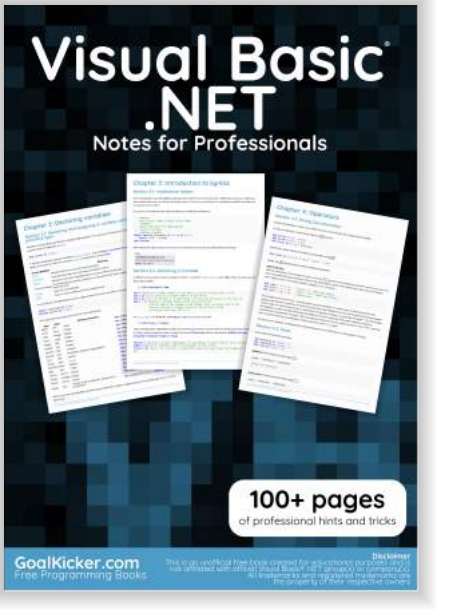
Adil Mammadov	Chapters 1, 7 and 16
Akos Nagy	Chapters 8, 9 and 21
Anshul Nigam	Chapter 23
CptRobby	Chapters 3, 11 and 14
Daniel Lemke	Chapters 3 and 7
DavidG	Chapters 1, 3 and 23
Diego	Chapter 3
Eldho	Chapter 1
Florian Haider	Chapter 16
Gert Arnold	Chapters 3, 11, 14, 15 and 23
Jacob Linney	Chapter 1
Jason Tyler	Chapters 7 and 19
Joshit	Chapter 16
Jozef Lačný	Chapters 3, 4 and 12
Kobi	Chapter 23
lucavgobbi	Chapters 6 and 23
MacakM	Chapter 2
Mark Shevchenko	Chapter 3
Matas Vaitkevicius	Chapters 1, 3, 10 and 20
Mina Matta	Chapter 22
Mostafa	Chapter 5
Nasreddine	Chapter 1
Parth Patel	Chapters 1, 2 and 3
Piotrek	Chapter 3
Sampath	Chapter 13
skj123	Chapter 18
SOfanatic	Chapter 17
Stephen Reindl	Chapter 5
Tetsuya Yamamoto	Chapters 10 and 20
tmg	Chapters 1, 2, 3 and 13
Tushar patel	Chapter 3
wertzui	Chapter 23

(c) ketabton.com: The Digital Library

You may also like







Get more e-books from www.ketabton.com
Ketabton.com: The Digital Library