Mariusz Wieczorek

[Email address]

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document.   
When you’re ready to add your content, just click here and start typing.]

Entity Framework core

Course of Entity Framework

# **Czym jest Entity Framework ?**

**ORM** (Object Relational Mapping) to sposób odwzorowania, mapowania architektury obiektowej  
na relacyjną bazę danych.

**Entity Framework** jest maperem ORM dla aplikacji .NET wspieranym przez Microsoft.

EF umożliwia programistom pracę z danymi przy użyciu obiektów klas domenowych bez potrzeby skupiania się na tabelach i kolumnach bazy danych, w których te dane są przechowywane.  
Dzięki EF deweloperzy mogą pracować na wyższym poziomie abstrakcji, mogą tworzyć aplikacje zorientowane na dane z mniejszą ilością kodu w porównaniu z tradycyjnymi aplikacjami.

Dla programistów c# oznacza to, że operujemy na zwykłych obiektach a „pod spodem” wywoływane są zapytania SQL do bazy danych.

Diagram

Description automatically generated

Dzięki EF

możemy tworzyć zapytania na bazie danych za pomocą obiektowego paradygmatu programowania.  
oszczędzamy na kodzie – mniej się napiszemy niż w przypadku zapytania SQL  
kod zostanie zweryfikowany w momencie pisania, nie popełnimy literówki   
Framework zapewni wsparcie transakcji, migracji, puli połączeń.

# Jak działa Entity Framework API?

Entity Framework API (EF6 i EF Core) obejmuje możliwość:

* mapowania klas domenowych do schematu bazy danych,
* tłumaczenia i wykonywania zapytań LINQ do SQL,
* śledzenia zmian wprowadzonych w encjach
* zapisywania zmian w bazie danych.

Diagram

Description automatically generated with medium confidence

**Entity Data Model**

Pierwszym zadaniem EF API jest zbudowanie modelu EDM.  
Model EDM to reprezentacja w pamięci całych metadanych: modelu koncepcyjnego, modelu przechowywania i mapowania między nimi.

Diagram

Description automatically generated

**Model koncepcyjny**: EF buduje model koncepcyjny na podstawie:

* klas domenowych
* klasy kontekstu DbContext()
* domyślnych konwencji stosowanych w klasach domenowych
* konfiguracji.

**Model przechowywania**: EF buduje model magazynu dla bazowego schematu bazy danych.  
W podejściu code-first będzie to wywnioskowane z modelu koncepcyjnego.   
W podejściu database-first będzie to wywnioskowane z docelowej bazy danych.

Możemy tworzyć tabele za pomocą różnych konwencji  
Każdą konwencję możemy nadpisać za pomocą odpowiedniej konfiguracji  
Konfigurację możemy tworzyć za pomocą:  
**data annotation** a także za pomocą  
**fluent API**.

Jeżeli chcemy wykonać zapytanie na bazie danych, to możemy posłużyć się składnią LINQ.  
EF wykona odpowiednie zapytanie i zmapuje wynik na obiekty modelu.  
Wspiera wszystkie relacje   
1:1   
1:wielu  
wiele:wielu  
  
umożliwia zadawanie sparametryzowanych zapytań  
śledzi zmianę na obiektach w pamięci  
umożliwia tworzenie indeksów, transakcji, migracji  
pracę z widokami, procedurami czy funkcjami

# **Entity Framework Core versus Entity Framework**

|  |
| --- |
| **Entity Framework i Entity Framework Core to dwie oddzielne biblioteki.** |

EF Core został wprowadzony w roku 2016.   
Początkowo miał być kolejną po szóstej wersją EF, miał dostać numer 7, ale został napisany od zera.  
Jest to lekki, opensource’owy, wspierający wiele platform framework,  
Jest dużo szybszy od Entity Framework.  
Jest częścią .NET Core.  
Jest dostępny także za pomocą osobnego pakietu NuGet, co wprowadza trochę zamieszania.  
EF Core jest to ciągle rozwijany posiada już większość funkcjonalności Entity Framework,  
oraz wiele nowych nie dostępnych w starym frameworku.

Wspiera takie bazy jak: SQL Lite, SQL Server, PostgreSQL, MySQL, Oracle, Firebird.

Najwyższa wersja **Entity Framework** to wersja **6** została ona wprowadzona **2013** roku,   
od tego czasu nie jest rozwijana, wprowadzane są tylko drobne poprawki.

## **Wspierane rodzaje aplikacji**

Jest przede wszystkim stworzona do aplikacji na framework’u .NET Core  
Może być również używana w aplikacjach desktopowych, na platformie .Net 4.7.2 i wyższymi.

A screenshot of a computer

Description automatically generated with low confidence

|  |
| --- |
| **Entity Framework Core** jest obecnie zalecane do stosowania w każdym typie aplikacji, oprócz aplikacji webowych także w desktopowych czy mobilnych. |

|  |
| --- |
| Aplikacje napisane na framework’u **.NET Core 5** nie wspierają już **Entity Framework 6**. |

## Wspierane podejścia do tworzenia bazy

Ma dwa podejścia do tworzenia bazy danych:

* **Code First** – tworzymy bazę od zera za pomocą klas w c#

Najpierw stworzymy kod w c#, następnie na podstawie tego kodu wygenerujemy bazę danych wraz z odpowiednimi tabelami i kolumnami. Jest to podejście bardzo intuicyjne zwłaszcza dla programistów obiektowych. Klasy domenowe zamieniają się na tabele a właściwości tych klas na odpowiednie pola w tych tabelach.

* **Database First** – zaczynamy pracować na już istniejącej bazie danych

Diagram

Description automatically generated

## Inne frameworki

Czysty ADO.NET, Dapper, NHibernate.

Entity Framework przebył długą drogę i kiedyś nie był aż tak dobry, łatwy w użyciu i wydajny jak teraz.  
Sporo programistów nie ma wystarczające wiedzy na temat Entity Framework Core i tworzą mało wydajne zapytania.

# Przykładowa solucja Blog

Cała solucja będzie podzielona na trzy projekty.

## Pierwszy projekt: Blog.Domain

Będzie zawierał modele domenowe.  
Klasy, które później będą miały odwzorowane tabele w bazie danych.  
Będzie to rdzeń aplikacji, który nie będzie miał żadnych powiązań z EF Core.

## Drugi projekt: Blog.DataLayer

To biblioteki, mapowanie i cała konfiguracja dla EF Core

## Trzeci projekt: Blog.UI

To interfejs użytkownika np. konsolowa

## Powiązania między projektami

Graphical user interface, text, application

Description automatically generated

Blog.Domain nie ma żadnych odwołań do innych projektów.  
Blog.DataLayer ma odwołanie do projektu Blog.Domain.  
Blog.UI ma odwołanie do projektu Blog.DataLayer.

## Instalacja wymaganych pakietów NuGet

Możemy to zrobić wpisując odpowiednie komendy w **Package Manage Console**   
lub w oknie zarządzania NuGet’ami.

### Blog.DataLayer – 3 pakiety



To pakiet podstawowy



Pakiet dedykowany dla systemu bazodanowego, inny dla MS SQL inny dla bazy PostgreSQL



Pakiet dzięki któremu komendy wpisywane w Package Manage Console będą prawidłowo interpretowane

### Blog.UI – 1 pakiet

Pakiet dzięki któremu komendy wpisywane w Package Manage Console będą prawidłowo interpretowane. Dzięki niemu będziemy mogli pobrać ustawienia konfiguracyjne z pliku appsettings.json

## Model Domenowy i Relacje w projekcie Blog

|  |
| --- |
| Model domenowy jest to model koncepcyjny przedstawiający podmioty wchodzące w dziedzinę biznesową naszej aplikacji i relacje pomiędzy nimi. |

### Encje w projekcie Blog

**Post** – artykuły na blogu  
**Category** – kategorie wypisów  
**Tag** – tagi przypisane do wpisów  
**User** – informacje o użytkownikach  
**ContactInfo** – z informacjami kontaktowymi użytkowników

### Relacje pomiędzy encjami w projekcie Blog

Pomiędzy encjami **Post** i **Category** będzie relacja jeden do wielu :  
 każdy artykuł może mieć jedną kategorię  
 każda kategoria może zawierać wiele artykułów

Pomiędzy encjami **Post** i **Tag** będzie relacja wiele do wielu :  
 każdy artykuł może mieć wiele tagów  
 każdy tag może być przypisany do wielu artykułów.

Pomiędzy encjami **Post** i **User** będzie relacja jeden do wielu  
 każdy User może mieć wiele artykułów  
 każdy artykuł może mieć jednego Usera który utworzył artykuł

Pomiędzy encjami **User** i **ContactInfo** będzie relacja jeden do jeden  
 każdy adres należy tylko do jednego użytkownika  
 każdy użytkownik może mieć tylko jeden adres

# Konwencje w **Entity Framework Core**

Konwencje są **domyślnymi regułami**, które używa framework do budowania modelu.  
EF Core API tworzy bazę danych bazując na modelu i klasie context bez żadnych dodatkowych konfiguracji.

## Schemat

Domyślnie EF utworzy bazę w schemacie dbo

## Typy kolumn

Zależą od tego jak dostawca zmapował typy c# na typy w konkretnym typie bazy.  
Np. dla MSSQL

|  |  |
| --- | --- |
| **C# Data Type** | **Mapping to SQL Server Data Type** |
| int | int |
| string | nvarchar(Max) |
| decimal | decimal(18,2) |

## Klucz główny

Jak dodamy właściwość o nazwie Id lub <nazwa encji>Id (wielkość znaków nie ma znaczenia) to zgodnie z konwencją w tabeli zostanie utworzony klucz główny jako pole auto inkrementowane.  
Dla przykładu zostanie utworzona kolumna będąca kluczem głównym w tabeli Posts, jeżeli klasa Post będzie zawierał właściwość Id lub ID lub iD lub PostId lub POSTID lub pOstId itp.

## Index

EF utworzy klastrowy indeks dla kolumny będącej kluczem głównym i nieklastrowy indeks dla kolumny będącej kluczem obcym.

## Klucz obcy

EF utworzy kolumnę będącą kluczem obcym dla każdej właściwości nawigacyjnej w encji

## Kolumny Nullable

EF utworzy kolumny dopuszczające null dla wszystkich typów referencyjnych oraz dla typów prymitywnych oznaczonych w klasie c# jako nulowalne np. Nullable<int>, int? Decimal? .  
Pola typu string są domyślnie nulowalne – ale można to zmienić ustawiając te właściwości jako wymagane.

## Kolumny NotNull

EF utworzy kolumny jako not null dla wszystkich właściwości będących kluczem głównym i wszystkich prymitywnych typów typu int, decimal, DataTime

## Relacja jeden do jeden

W przykładowym projekcie mamy tylko jedną relację tego typu pomiędzy encjami User i ContactInfo.

Jeden kontakt może należeć do jednego użytkownika.   
Jeden użytkownik może mieć tylko jeden kontakt.

Zgodnie z konwencją w klasie ContactInfo tworzymy właściwość UserId lub UserUserId**,**będzie ona traktowane jako klucz obcy do tabeli **Users.**W relacji jeden-do-jeden musimy zdefiniować, która klasa jest zależna.  
Klasa ContactInfo jest klasą zależną od klasy User, dlatego w tej klasie wstawiamy zarówno właściwość nawigacyjną typu User oraz klucz obcy UserId.

|  |
| --- |
| public class ContactInfo  {  public int Id { get; set; }  public string Email { get; set; }  // klucz obcy  public int **UserId** { get; set; }  // właściwość nawigacyjna  public **User** **User** { get; set; }  } |

W klasie User wstawiamy tylko właściwość nawigacyjną ContactInfo.

|  |
| --- |
| public class User  {  public int Id { get; set; }  public string Login { get; set; }  public string Password { get; set; }  // właściwość nawigacyjna  public **ContactInfo** **ContactInfo** { get; set; }  } |

## Relacja jeden-do-wielu

W projekcie mamy relację jeden-do-wielu pomiędzy encjami Post i Category.  
Jeden artykuł może mieć jedną kategorię,  
Jedna kategoria może zawierać wiele artykułów.

Relację taką tworzymy wstawiając po obu stronach właściwości nawigacyjne.

W klasie Post musimy zdefiniować właściwość nawigacyjną Category  
oraz dodatkowo klucz obcy CategoryId.

|  |
| --- |
| public class Post  {  public int Id { get; set; }  // klucz obcy  public int CategorId { get; set; }  // właściwość nawigacyjna  public Category Category { get; set; }  } |

W encji Category wstawiamy kolekcję nawigacyjną.   
Używamy do tego najprostszej kolekcji np. ICollection.  
Inicjujemy kolekcję już w tym miejscu, aby później nie było problemu z null’ami.

|  |
| --- |
| public class Category  {  public int Id { get; set; }  public string Name { get; set; }  public string Url { get; set; }  public string Description { get; set; }  // właściwości nawigacyjne  public ICollection<Post> Posts { get; set; } = new HashSet<Post>();  } |

W przykładowym projekcie mamy relację jeden-do-wielu pomiędzy encjami User i Post.  
Artykuł ma jednego użytkownika,   
jeden użytkownik ma wiele artykułów.

|  |
| --- |
| public class User  {  public int Id { get; set; }  // właściwość nawigacyjna  public ICollection<**Post**> **Posts** { get; set; } = new HashSet<Post>();  } |

|  |
| --- |
| public class Post  {  // klucz obcy do Kategorii  public int CategorId { get; set; }  // właściwość nawigacyjna  public Category Category { get; set; }  // klucz obcy do użytkownika  public int UserId { get; set; }  // właściwość nawigacyjna  public **User** **User** { get; set; }  } |

## Relacja wiele do wielu

W przykładowym projekcie mamy jedną relację typu wiele do wielu pomiędzy encjami Post i Tag.  
Każdy artykuł może mieć wiele tagów,  
Każdy tag może zostać przypisany do wielu artykułów.

W EF Core począwszy od wersji 5.0 istniej konwencja do tego typu relacji.   
Wystarczy po każdej ze stron dodać jedną kolekcję i EF Core wie, że jest to relacja wiele do wielu.  
W tagach mamy kolekcję artykułów a w artykułach kolekcję tagów.

W poprzednich wersjach utworzenie takiej relacji wymagało utworzenia dodatkowej encji,  
która przechowywała dwie relacje jeden do wielu.  
Czyli np. encja o nazwie PostTag i zawierała ona klucz obcy do encji Tag i do encji Post

Po obu stronach wstawiamy kolekcje nawigacyjne.

|  |
| --- |
| public class Post  {  public int Id { get; set; }  // właściwość nawigacyjna  public ICollection<**Tag**> **Tags** { get; set; } = new HashSet<Tag>();  } |

|  |
| --- |
| public class Tag  {  public int Id { get; set; }  public string Name { get; set; }  public string Url { get; set; }  // właściwość nawigacyjna  public ICollection<**Post**> **Posts** { get; set; } = new HashSet<Post>();  } |

# **Klasa DbContext**

DbContext jest integralną częścią Entity Framework.

Na początek musimy dodać klasę dziedziczącą po DbContext.  
Nazwa tej klasy jest dowolna może to być np. ApplicationDbContext.

|  |
| --- |
| public class ApplicationDbContext : DbContext  {  } |

Klasa będzie w warstwie dostępu do danych, ponieważ jest ona ściśle powiązana z EF Core.  
Klasa ta reprezentuje sesję z bazą danych, na instancji tej klasy możemy wykonywać zapytania  
i różne operacje na bazie danych.  
Klasę tą musimy odpowiednio skonfigurować. **DbContext** reprezentuje wzorzec UnitOfWork  
który wewnątrz zawiera klasy repozytoryjne.

DbContext w EF Core pozwala nam wykonywać następujące zadania:

* zarządzać połączeniami z bazą danych
* konfigurować modele i relacje
* tworzyć zapytania do baz danych
* śledzić zmiany w bazie
* zarządzać transakcjami
* tworzyć komendy, zapisywać je w bazie danych

## Właściwość DbSet

Za pomocą właściwości **DbSet** wskazujemy, które klasy w projekcie c# mają zostać zmapowane  
na tabele w bazie danych.  
Właściwości typu DbSet są to kolekcje na których będziemy pracować. Mamy różne metody dodawania, wyszukiwania, usuwanie czy aktualizowania danych, ale co ważne tutaj nie mamy możliwości trwałego zapisu tych zmian do bazy.

Dopiero sam DbContext może dokonać trwałego zapisu danych to jest właśnie implementacja  
wzorca UnitOfWork.

|  |
| --- |
| public class ApplicationDbContext : DbContext  {  public DbSet<**Category**> **Categories** { get; set; }  public DbSet<**ContactInfo**> **ContactInfo** { get; set; }  public DbSet<**Post**> **Posts** { get; set; }  public DbSet<**Tag**> **Tags** { get; set; }  public DbSet<**User**> **Users** { get; set; }  } |

## ConnectionString

Connection string odpowiedni do danej bazy możemy pobrać ze strony :  
**www.conectionstrings.com**

Dla MS SQL Server taki łańcuch wygląda jak poniżej

"Server = 127.0.0.1; Database = MwBlog; Uid = user1; Pwd = alamakota;"

Kropka na początku oznacza localwięc zapis.\SQLEXPRESS   
oznacza to samo co (local)\SQLEXPRESSi to samo co127.0.0.1\SQLEXPRESS

Aby móc wpisać znak ukośnika \ wstawiamy przed łańcuchem znak @ lub w miejsce ukośnika podwójny ukośnik \\

@"Server =.\SQLEXPRESS; Database = MwBlog; Uid = user1; Pwd = alamakota;"

"Server = .\\SQLEXPRESS; Database = MwBlog; Uid = user1; Pwd = alamakota;"

Możemy go ustawić na kilka sposobów

### W klasie DbContext

|  |
| --- |
| public class ApplicationDbContext : DbContext  {  public DbSet<Category> Categories { get; set; }  public DbSet<ContactInfo> ContactInfo { get; set; }  public DbSet<Post> Posts { get; set; }  public DbSet<Tag> Tags { get; set; }  public DbSet<User> Users { get; set; }  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  optionsBuilder.UseSqlServer(  @"Server = 127.0.0.1; Database = MwMaintenance; Uid = user1; Pwd = alamakota;"  );  }  } |

Wadą tego rozwiązania jest to, że przy zmianie tego łańcucha musimy ponownie kompilować aplikację.

### W wydzielonym pliku konfiguracyjnym

Tworzymy plik appsettings.json

Graphical user interface, application

Description automatically generated

We właściwościach tego pliku ustawiamy cechę

Table

Description automatically generated

|  |
| --- |
| {  "ConnectionString": "Server=192.168.1.186;Database=MwBlog;Uid=user1;Pwd=alamakota;"  } |

Musimy teraz odczytać ten plik konfiguracyjny w naszej klasie DbContext.  
Aby to zrobić musimy najpierw zainstalować dwa pakiety:





Następnie modyfikujemy klasę DbContext

|  |
| --- |
| public class ApplicationDbContext : DbContext  {  public DbSet<Category> Categories { get; set; }  public DbSet<ContactInfo> ContactInfo { get; set; }  public DbSet<Post> Posts { get; set; }  public DbSet<Tag> Tags { get; set; }  public DbSet<User> Users { get; set; }  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  var builder = new ConfigurationBuilder()  .AddJsonFile("appsettings.json", true, true);  var config = builder.Build();  optionsBuilder.UseSqlServer(config["ConnectionString"]);  }  } |

### Plik **appsettings.json** w ASP.NET Core

|  |
| --- |
| {  "ConnectionStrings": {  "DefaultConnection": "Server=192.168.1.186;Database=MwBlog;Uid=user1;Pwd=alamakota;",  "LocalConnection": "Server=127.0.0.1;Database=MwBlog;Uid=user1;Pwd=alamakota;"  },  "Logging": {  "LogLevel": {  "Default": "Information",  "Microsoft": "Warning",  "Microsoft.Hosting.Lifetime": "Information"  }  },  "EmailConfiguration1": {  "SmtpServer": "smtp.gmail.com",  "Port": 587,  "EnableSsl": true,  "SenderName": "Mariusz Wieczorek",  "SenderEmail": "mariusz.wieczorek.testy@gmail.com",  "SenderEmailPassword": "rmhfvaurzyxnuztn"  },  "AllowedHosts": "\*"  } |

# **Seed Data**

Jeżeli chcemy dodać jakieś domyślne dane do aplikacji to możemy skorzystać z mechanizmu  
seed data.  
Aby to zrobić najpierw w klasie DbContext przesłaniamy metodę OnModelCreating.

Możemy to zrobić bezpośrednio w klasie DbContext tak jak poniżej.

|  |
| --- |
| protected override void OnModelCreating(ModelBuilder modelBuilder)  {  modelBuilder.Entity<Category>().HasData(  new Category  {  Id = 1,  Name = "general",  Description = "all general posts"  },  new Category  {  Id = 2,  Name = "other",  Description = "all other posts"  });  } |

Aby nie zaśmiecać klasy DbContext dodajemy metodę rozszerzającą.  
Tworzymy katalog Extensions a w nim klasę ModelBuilderExtensions.

|  |
| --- |
| public static class ModelBuilderExtensions  {  public static void SeedCategories(this ModelBuilder modelBuilder)  {  modelBuilder.Entity<Category>().HasData(  new Category  {  Id = 1,  Name = "general",  Description = "all general posts"  },  new Category  {  Id = 2,  Name = "other",  Description = "all other posts"  });  }  } |

a w DbContext wywołujemy tą metodę.

|  |
| --- |
| protected override void OnModelCreating(ModelBuilder modelBuilder)  {  modelBuilder.SeedCategories();  } |

# **Komendy Package Managment Console (PMC)**

Wydawać komendy możemy za pomocą PMC lub CLI.  
Są to dwa odmienne sposoby. Gdy używamy Visual Studio zaleca się stosować PMC.

Gdy w zainstalowanym Visual Studio nie mamy jeszcze przypiętego okna  
z Package Managment Console, uaktywniamy je jak poniżej.

Graphical user interface, application, table

Description automatically generated

Aby zobaczyć pełną listę komend wpisujemy komendę:

|  |
| --- |
| PM> get-help entityframework |

|  |
| --- |
| **PM> get-help entityframework**  \_/\\_\_  ---==/ \\  \_\_\_ \_\_\_ |. \|\  | \_\_|| \_\_| | ) \\\  | \_| | \_| \\_/ | //|\\  |\_\_\_||\_| / \\\/\\  TOPIC  about\_EntityFrameworkCore  SHORT DESCRIPTION  Provides information about the Entity Framework Core Package Manager Console Tools.  LONG DESCRIPTION  This topic describes the Entity Framework Core Package Manager Console Tools. See https://docs.efproject.net for  information on Entity Framework Core.  The following Entity Framework Core commands are available.  Cmdlet Description  -------------------------- ---------------------------------------------------  Add-Migration Adds a new migration.  Drop-Database Drops the database.  Get-DbContext Lists and gets information about available DbContext types.  Get-Migration Lists available migrations.  Remove-Migration Removes the last migration.  Scaffold-DbContext Scaffolds a DbContext and entity types for a database.  Script-DbContext Generates a SQL script from the DbContext. Bypasses any migrations.  Script-Migration Generates a SQL script from migrations.  Update-Database Updates the database to a specified migration.  SEE ALSO  Add-Migration  Drop-Database  Get-DbContext  Get-Migration  Remove-Migration  Scaffold-DbContext  Script-DbContext  Script-Migration  Update-Database |

# **Tworzenie Bazy Danych**

Bazę tworzymy za pomocą komendy:

|  |
| --- |
| PM> update-database |

Musimy jednak najpierw zwrócić uwagę na dwa szczegóły.  
**Na górze musimy ustawić projekt w którym jest klasa DbContext**

Graphical user interface, application, Word, website

Description automatically generated

Po drugi musimy ustawić projekt Blog.UI jako startowy, ponieważ z jego pliku konfiguracyjnego appsettings.json zostanie odczytany connectionstring.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Właśnie dlatego do tego projektu instalowaliśmy pakiet Microsoft.EntityFrameworkCore.Design

|  |
| --- |
| PM> update-database  Build started...  Build succeeded.  No migrations were applied. The database is already up to date.  Done.  PM> |

W Management Studio widzimy, że powstała baza danych z tabelą zawierającą informacje o migracjach. Nie powstała jeszcze żadna baza powiązana z naszym blogiem.

Graphical user interface, text, application

Description automatically generated

Aby powstały tabele musimy najpierw dodać migracje.

Zadanie

Zdefiniuj model domenowy, tworzymy kilka encji takich jak :

Invoice, InvoicePossition, Product, Attribute, Customer, Address

Relacje:  
Każda faktura może mieć wiele pozycji, a pozycja może być tylko na jednej fakturze  
Każda faktura może mieć jednego klienta, ale klient może być na wielu fakturach  
Każdy klient ma jeden adres a każdy adres ma tylko jednego klienta  
Każda pozycja ma jeden produkt, ale każdy produkt może wystąpić na wielu pozycjach faktury  
Każdy produkt może mieć wiele atrybutów, a każdy atrybut może mieć wiele wyrobów

Utwórz po kilka właściwości dla każdej z encji, aby były różnych typów string, bool, int DataTime, Decimal, Enum

# Migracje

Dzięki migracjom mamy wersjonowanie bazy danych podobnie jak dzięki Git mamy wersjonowanie kodu.

Graphical user interface, application

Description automatically generated

Używamy komend wywołanych w Package Managment Console lub w CLI.

Lista najważniejszych komend

|  |  |  |
| --- | --- | --- |
| **PMC Command** | **dotnet CLI command** | **Usage** |
| add-migration <migration name> | Add <migration name> | Creates a migration by adding a migration snapshot. |
| Remove-migration | Remove | Removes the last migration snapshot. |
| Update-database | Update | Updates the database schema based on the last migration snapshot. |
| Script-migration | Script | Generates a SQL script using all the migration snapshots. |

## Pierwsza Migracja





Podobnie jak w przypadku komendy **update-database**, migracje muszą być wywoływane na projekcie w którym mamy zdefiniowaną klasę DbContext czyli w naszym projekcie jest to **Blog.DataLayer**

**Na górze w okienku Package Managment Console musimy ustawić odpowiedni projekt.**

Obraz zawierający tekst

Opis wygenerowany automatycznie

Aby dodać migrację wywołujemy komendę **add-migration** oraz po niej nazwę migracji.  
Ważne aby wpisana nazwa była jak najbardziej opisowa i mówiła za co dana migracja odpowiada.

|  |
| --- |
| PM> **add-migration** InitMigration  Build started...  Build succeeded.  To undo this action, use Remove-Migration. |

Po tej komendzie została utworzona migracja, ale baza danych nie została jeszcze zmieniona.

Obraz zawierający tekst

Opis wygenerowany automatycznie

Do projektu został dodany nowy katalog **Migrations**, który zawiera wszystkie informacje o migracjach. Jest tu plik snapshot – czyli migawka aktualnego modelu, zawiera metadane o migracjach, pomaga frameworkowi w ustaleniu co zmieniło się od ostatniej aktualizacji.  
Oprócz snapshot’a jest plik z datą i nazwą migracji zawierający kod w c# dokonujący zmian.  
Jeżeli chcesz zachować poprawne wersjonowanie, to tych plików nie można usuwać.  
Podczas generowania nowej migracji następuje porównanie z poprzednią wersją.

W wygenerowanym pliku są dwie metody:  
**Up** – kod aktualizujące bazę  
**Down** – kod wycofujący zmiany

Możemy te metody modyfikować, uwzględniać swoje własne skrypty.

## Aktualizowanie Bazy Danych

Aby wprowadzić zmiany wydajemy komendę update-database   
Komenda ta zastosuje wszystkie niezaaplikowane migracje na schemacie bazy danych.

Background pattern

Description automatically generated

|  |
| --- |
| PM> **update-database**  Build started...  Build succeeded.  Applying migration '20210813122237\_InitMigration'.  Done. |

W tabeli \_\_EFMigrationsHistory mamy historię zaaplikowanych migracji.

|  |
| --- |
| SELECT TOP (1000) [MigrationId]  ,[ProductVersion]  FROM [MwBlog].[dbo].[\_\_EFMigrationsHistory] |

Obraz zawierający tekst

Opis wygenerowany automatycznie

Zgodnie z naszym modelem domenowym zostały utworzone tabele, wszystkie właściwości zostały zmapowane na kolumny w tych tabelach.

**Categories**

Obraz zawierający tekst

Opis wygenerowany automatycznie

Mamy również dodane wiersze, które zdefiniowaliśmy w seeddata

Obraz zawierający stół

Opis wygenerowany automatycznie

Obraz zawierający tekst

Opis wygenerowany automatycznie

**ContactInfo**

Obraz zawierający tekst

Opis wygenerowany automatycznie

Id jest kluczem głównym, UserId jest kluczem obcym

**Posts**



Mamy klucz główny, dwa klucze obce, kolumnę Type, która w projekcie jest enum’em a tutaj typu int.  
Kolumna PostedOn jest not null, a kolumna Modified umożliwia stosowanie nulli.

## Nowa Migracja

Aby zmienić cokolwiek w bazie, będziemy musieli najpierw dodać nową migrację, a później zaktualizować bazę danych.

Dodajemy do klasy domenowej Post nową właściwość **ShortDescription**

Obraz zawierający tekst

Opis wygenerowany automatycznie

|  |
| --- |
| PM> **add-migration** **AddShortDesvriptionInPost**  Build started...  Build succeeded.  To undo this action, use Remove-Migration. |

Zostaje dodana nowa migracja, powstaje nowy plik w katalogu Migrations

Obraz zawierający tekst

Opis wygenerowany automatycznie

|  |
| --- |
| public partial class AddShortDesvriptionInPost : Migration  {  protected override void Up(MigrationBuilder migrationBuilder)  {  migrationBuilder.AddColumn<string>(  name: "ShortDescription",  table: "Posts",  type: "nvarchar(max)",  nullable: true);  }  protected override void Down(MigrationBuilder migrationBuilder)  {  migrationBuilder.DropColumn(  name: "ShortDescription",  table: "Posts");  }  } |

Również doszły zmiany w snapshot’cie, który zawiera aktualny obraz naszej bazy, zawiera on już informacje o nowym polu.

|  |
| --- |
| b.Property<string>("Description")  .HasColumnType("nvarchar(max)");  b.Property<DateTime?>("Modified")  .HasColumnType("datetime2");  b.Property<DateTime>("PostedOn")  .HasColumnType("datetime2");  b.Property<bool>("Published")  .HasColumnType("bit");  b.Property<string>("ShortDescription")  .HasColumnType("nvarchar(max)");  b.Property<string>("Title")  .HasColumnType("nvarchar(max)"); |

Możemy zaktualizować bazę,  
**update-database -v  
update-database -verbose**

opcja -v lub -verbose powoduje wyświelenie komend , które są wywołane na bazie danych.

|  |
| --- |
| PM> **update-database -v**  Using project 'Blog.DataLayer'.  Using startup project 'Blog.UI'.  Build started...  Build succeeded.  C:\Program Files\dotnet\dotnet.exe exec --depsfile D:\vs\EFCore\Blog\Blog.UI\bin\Debug\net5.0\Blog.UI.deps.json --additionalprobingpath C:\Users\mariuszw\.nuget\packages --additionalprobingpath "C:\Program Files (x86)\Microsoft Visual Studio\Shared\NuGetPackages" --additionalprobingpath "C:\Program Files (x86)\Microsoft\Xamarin\NuGet" --runtimeconfig D:\vs\EFCore\Blog\Blog.UI\bin\Debug\net5.0\Blog.UI.runtimeconfig.json C:\Users\mariuszw\.nuget\packages\microsoft.entityframeworkcore.tools\5.0.9\tools\netcoreapp2.0\any\ef.dll database update --verbose --no-color --prefix-output --assembly D:\vs\EFCore\Blog\Blog.UI\bin\Debug\net5.0\Blog.DataLayer.dll --startup-assembly D:\vs\EFCore\Blog\Blog.UI\bin\Debug\net5.0\Blog.UI.dll --project-dir D:\vs\EFCore\Blog\Blog.Datalayer\ --language C# --working-dir D:\vs\EFCore\Blog --root-namespace Blog.DataLayer  Using assembly 'Blog.DataLayer'.  Using startup assembly 'Blog.UI'.  Using application base 'D:\vs\EFCore\Blog\Blog.UI\bin\Debug\net5.0'.  Using working directory 'D:\vs\EFCore\Blog\Blog.UI'.  Using root namespace 'Blog.DataLayer'.  Using project directory 'D:\vs\EFCore\Blog\Blog.Datalayer\'.  Remaining arguments: .  Finding DbContext classes...  Finding IDesignTimeDbContextFactory implementations...  Finding application service provider in assembly 'Blog.UI'...  Finding Microsoft.Extensions.Hosting service provider...  No static method 'CreateHostBuilder(string[])' was found on class 'Program'.  No application service provider was found.  Finding DbContext classes in the project...  Found DbContext 'ApplicationDbContext'.  Using context 'ApplicationDbContext'.  Finding design-time services for provider 'Microsoft.EntityFrameworkCore.SqlServer'...  Using design-time services from provider 'Microsoft.EntityFrameworkCore.SqlServer'.  Finding design-time services referenced by assembly 'Blog.UI'...  Finding design-time services referenced by assembly 'Blog.DataLayer'...  No referenced design-time services were found.  Finding IDesignTimeServices implementations in assembly 'Blog.UI'...  No design-time services were found.  Migrating using database 'MwBlog' on server '127.0.0.1'.  Opening connection to database 'MwBlog' on server '127.0.0.1'.  Opened connection to database 'MwBlog' on server '127.0.0.1'.  Creating DbCommand for 'ExecuteNonQuery'.  Created DbCommand for 'ExecuteNonQuery' (2ms).  Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT 1  Executed DbCommand (11ms) [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT 1  Closing connection to database 'MwBlog' on server '127.0.0.1'.  Closed connection to database 'MwBlog' on server '127.0.0.1'.  Creating DbCommand for 'ExecuteScalar'.  Created DbCommand for 'ExecuteScalar' (0ms).  Opening connection to database 'MwBlog' on server '127.0.0.1'.  Opened connection to database 'MwBlog' on server '127.0.0.1'.  Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT OBJECT\_ID(N'[\_\_EFMigrationsHistory]');  Executed DbCommand (8ms) [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT OBJECT\_ID(N'[\_\_EFMigrationsHistory]');  Closing connection to database 'MwBlog' on server '127.0.0.1'.  Closed connection to database 'MwBlog' on server '127.0.0.1'.  Opening connection to database 'MwBlog' on server '127.0.0.1'.  Opened connection to database 'MwBlog' on server '127.0.0.1'.  Creating DbCommand for 'ExecuteNonQuery'.  Created DbCommand for 'ExecuteNonQuery' (0ms).  Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT 1  Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT 1  Closing connection to database 'MwBlog' on server '127.0.0.1'.  Closed connection to database 'MwBlog' on server '127.0.0.1'.  Creating DbCommand for 'ExecuteScalar'.  Created DbCommand for 'ExecuteScalar' (0ms).  Opening connection to database 'MwBlog' on server '127.0.0.1'.  Opened connection to database 'MwBlog' on server '127.0.0.1'.  Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT OBJECT\_ID(N'[\_\_EFMigrationsHistory]');  Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT OBJECT\_ID(N'[\_\_EFMigrationsHistory]');  Closing connection to database 'MwBlog' on server '127.0.0.1'.  Closed connection to database 'MwBlog' on server '127.0.0.1'.  Creating DbCommand for 'ExecuteReader'.  Created DbCommand for 'ExecuteReader' (0ms).  Opening connection to database 'MwBlog' on server '127.0.0.1'.  Opened connection to database 'MwBlog' on server '127.0.0.1'.  Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT [MigrationId], [ProductVersion]  FROM [\_\_EFMigrationsHistory]  ORDER BY [MigrationId];  Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']  SELECT [MigrationId], [ProductVersion]  FROM [\_\_EFMigrationsHistory]  ORDER BY [MigrationId];  A data reader was disposed.  Closing connection to database 'MwBlog' on server '127.0.0.1'.  Closed connection to database 'MwBlog' on server '127.0.0.1'.  Applying migration '20210813151450\_AddShortDesvriptionInPost'.  Opening connection to database 'MwBlog' on server '127.0.0.1'.  Opened connection to database 'MwBlog' on server '127.0.0.1'.  Beginning transaction with isolation level 'Unspecified'.  Began transaction with isolation level 'ReadCommitted'.  Creating DbCommand for 'ExecuteNonQuery'.  Created DbCommand for 'ExecuteNonQuery' (0ms).  Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']  ALTER TABLE [Posts] ADD [ShortDescription] nvarchar(max) NULL;  Executed DbCommand (5ms) [Parameters=[], CommandType='Text', CommandTimeout='30']  ALTER TABLE [Posts] ADD [ShortDescription] nvarchar(max) NULL;  Creating DbCommand for 'ExecuteNonQuery'.  Created DbCommand for 'ExecuteNonQuery' (0ms).  Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']  INSERT INTO [\_\_EFMigrationsHistory] ([MigrationId], [ProductVersion])  VALUES (N'20210813151450\_AddShortDesvriptionInPost', N'5.0.9');  Executed DbCommand (0ms) [Parameters=[], CommandType='Text', CommandTimeout='30']  INSERT INTO [\_\_EFMigrationsHistory] ([MigrationId], [ProductVersion])  VALUES (N'20210813151450\_AddShortDesvriptionInPost', N'5.0.9');  Committing transaction.  Committed transaction.  Disposing transaction.  Closing connection to database 'MwBlog' on server '127.0.0.1'.  Closed connection to database 'MwBlog' on server '127.0.0.1'.  'ApplicationDbContext' disposed.  Done.  PM> |

## Powrót Do Wcześniejszych Migracji

Aby wrócić do dowolnej wcześniejszej migracji możemy wywołać komendę update-database z nazwą dowolnej wcześniejszej migracji (wystarczy sama nazwa, nie trzeba podawać daty).

Background pattern

Description automatically generated

|  |
| --- |
| PM> **update-database InitMigration**  Build started...  Build succeeded.  Reverting migration '20210813151450\_AddShortDesvriptionInPost'.  Done.  PM> |

została wykonana metoda Down() klasy AddShortDesvriptionInPost.

Możemy usunąć z projektu ostatnią migrację, która jeszcze nie została wykonana na bazie danych.

|  |
| --- |
| PM> **remove-migration**  Build started...  Build succeeded.  Removing migration '20210813151450\_AddShortDesvriptionInPost'.  Reverting the model snapshot.  Done.  PM> |

Migracja zniknęła z katalogu Migration oraz został zaktualizowany nasz snapshot.

Kolejne wywołanie tej komendy usunie z projektu następną z kolei niewykonaną migrację.   
Możemy po przez kolejne wywołania **remove-migration** usuwać kolejne migracje z projektu.  
Usunięcie ostatniej pozostałej migracji spowoduje usunięcie katalogu Migration z projektu.  
Pozostanie pusta baza baz tabel i pusta tabela z historią migracji.

Jeżeli chciałbym usunąć migrację, która została już wykonana na bazie musiał bym wywołać  
**remove-migration -force.** Można tę komendę również wywołać wielokrotnie w każdym kroku cofając się do poprzedniej migracji. Opcja ta automatycznie aktualizuje bazę danych.

Możemy jedną komendą wycofać z bazy wszystkie migracje

|  |
| --- |
| PM> **update-database -migration 0**  Build started...  Build succeeded.  Reverting migration '20210813122237\_InitMigration'.  Done.  PM> |

## Zmiana domyślnego miejsca katalogu z migracjami

Możemy utworzyć katalog z migracjami w innym niż domyślnym miejscu

|  |
| --- |
| PM> **add-migration InitMigration -outputdir "Persistence/Migrations"**  Build started...  Build succeeded.  To undo this action, use Remove-Migration.  PM> |

Obraz zawierający tekst

Opis wygenerowany automatycznie

## Zaawansowane Migracje

Bardziej zaawansowane scenariusze.  
Mamy już rekordy w tabeli Posts  
dodajemy pole longDescriptions  
i chcemy aby było ono wypełnione tymi samymi danymi co w Description

|  |
| --- |
| public partial class AddLongDescriptionInPost : Migration  {  protected override void Up(MigrationBuilder migrationBuilder)  {  migrationBuilder.AddColumn<string>(  name: "LongDescription",  table: "Posts",  type: "nvarchar(max)",  nullable: true);  migrationBuilder.Sql(@"UPDATE POSTS SET LongDescription = Description");  }  protected override void Down(MigrationBuilder migrationBuilder)  {  migrationBuilder.DropColumn(  name: "LongDescription",  table: "Posts");  }  }  } |

Usuwanie tabeli, można skopiować do tabeli historycznej.  
Dodanie nowej kolumny, która jest wymagana, tutaj możemy nadać kolumnie wartość domyślną lub napisać SQL’a, który zaktualizuje wartość.

## Niezsynchronizowana Baza Danych

Gdy dodamy do naszego modelu nową właściwość

|  |
| --- |
| public class Post  {  public int Id { get; set; }  public string Title { get; set; }  public string Description { get; set; }  public string ShortDescription { get; set; }  public string LongDescription { get; set; }  public string Url { get; set; }  public string ImageUrl { get; set; }  public bool Published { get; set; }  public DateTime PostedOn { get; set; }  public DateTime? Modified { get; set; } |

i nie zaktualizujemy bazy, to podczas zapytania do bazy zostanie rzucony wyjątek

|  |
| --- |
| static void Main(string[] args)  {  using (var context = new ApplicationDbContext() )  {  var post = context.Posts.FirstOrDefault();  Console.WriteLine(post.Description);  }  } |

Obraz zawierający tekst

Opis wygenerowany automatycznie

## Komenda **get-migration**

Komenda pozwala uzyskać informacje czy wszystkie migracje zostały zastosowane na bazie

|  |
| --- |
| PM> **get-migration**  Build started...  Build succeeded.  id name safeName applied  -- ---- -------- -------  20210813161644\_InitMigration InitMigration InitMigration True  20210813162415\_AddLongDescriptionInPost AddLongDescriptionInPost AddLongDescriptionInPost True  PM> |

Po dodaniu migracji

|  |
| --- |
| PM> **get-migration**  Build started...  Build succeeded.  id name safeName applied  -- ---- -------- -------  20210813161644\_InitMigration InitMigration InitMigration True  20210813162415\_AddLongDescriptionInPost AddLongDescriptionInPost AddLongDescriptionInPost True  20210813164952\_AddImageUrlInPost AddImageUrlInPost AddImageUrlInPost False  PM> |

Model domenowy I baza z którą się łączysz muszą być zsynchronizowane.

## Automatyczne Migracje

Nie są wspierane przez EF Core.  
Można je wywołać w sposób:

|  |
| --- |
| Using Microsoft.EntityFrameworkCore;  using (var context = new ApplicationDbContext() )  {  **context.Database.Migrate();**  } |

Jednak zalecane jest uruchamianie migracji ręcznie.

## Generowanie Skryptów

Gdy u klienta nie masz zainstalowanego Visual Studio z Packages Manager’em będziesz potrzebował mieć możliwość wygenerować skrypt, który uruchomisz w Ms SQL Management Studio.

Tworzymy migrację, a później zamiast update-database piszemy script-migration

|  |
| --- |
| PM> **script-migration**  Build started...  Build succeeded.  PM> |

Zostaje utworzony skrypt, który zawiera wszystkie migracje od początku

|  |
| --- |
| IF OBJECT\_ID(N'[\_\_EFMigrationsHistory]') IS NULL  BEGIN  CREATE TABLE [\_\_EFMigrationsHistory] (  [MigrationId] nvarchar(150) NOT NULL,  [ProductVersion] nvarchar(32) NOT NULL,  CONSTRAINT [PK\_\_\_EFMigrationsHistory] PRIMARY KEY ([MigrationId])  );  END;  GO  BEGIN TRANSACTION;  GO  CREATE TABLE [Categories] (  [Id] int NOT NULL IDENTITY,  [Name] nvarchar(max) NULL,  [Url] nvarchar(max) NULL,  [Description] nvarchar(max) NULL,  CONSTRAINT [PK\_Categories] PRIMARY KEY ([Id])  );  GO  CREATE TABLE [Tags] (  [Id] int NOT NULL IDENTITY,  [Name] nvarchar(max) NULL,  [Url] nvarchar(max) NULL,  CONSTRAINT [PK\_Tags] PRIMARY KEY ([Id])  );  GO  CREATE TABLE [Users] (  [Id] int NOT NULL IDENTITY,  [Login] nvarchar(max) NULL,  [Password] nvarchar(max) NULL,  CONSTRAINT [PK\_Users] PRIMARY KEY ([Id])  );  GO  CREATE TABLE [ContactInfo] (  [Id] int NOT NULL IDENTITY,  [Email] nvarchar(max) NULL,  [UserId] int NOT NULL,  CONSTRAINT [PK\_ContactInfo] PRIMARY KEY ([Id]),  CONSTRAINT [FK\_ContactInfo\_Users\_UserId] FOREIGN KEY ([UserId]) REFERENCES [Users] ([Id]) ON DELETE CASCADE  );  GO  CREATE TABLE [Posts] (  [Id] int NOT NULL IDENTITY,  [Title] nvarchar(max) NULL,  [Description] nvarchar(max) NULL,  [ShortDescription] nvarchar(max) NULL,  [Url] nvarchar(max) NULL,  [Published] bit NOT NULL,  [PostedOn] datetime2 NOT NULL,  [Modified] datetime2 NULL,  [Type] int NOT NULL,  [CategoryId] int NOT NULL,  [UserId] int NOT NULL,  CONSTRAINT [PK\_Posts] PRIMARY KEY ([Id]),  CONSTRAINT [FK\_Posts\_Categories\_CategoryId] FOREIGN KEY ([CategoryId]) REFERENCES [Categories] ([Id]) ON DELETE CASCADE,  CONSTRAINT [FK\_Posts\_Users\_UserId] FOREIGN KEY ([UserId]) REFERENCES [Users] ([Id]) ON DELETE CASCADE  );  GO  CREATE TABLE [PostTag] (  [PostsId] int NOT NULL,  [TagsId] int NOT NULL,  CONSTRAINT [PK\_PostTag] PRIMARY KEY ([PostsId], [TagsId]),  CONSTRAINT [FK\_PostTag\_Posts\_PostsId] FOREIGN KEY ([PostsId]) REFERENCES [Posts] ([Id]) ON DELETE CASCADE,  CONSTRAINT [FK\_PostTag\_Tags\_TagsId] FOREIGN KEY ([TagsId]) REFERENCES [Tags] ([Id]) ON DELETE CASCADE  );  GO  IF EXISTS (SELECT \* FROM [sys].[identity\_columns] WHERE [name] IN (N'Id', N'Description', N'Name', N'Url') AND [object\_id] = OBJECT\_ID(N'[Categories]'))  SET IDENTITY\_INSERT [Categories] ON;  INSERT INTO [Categories] ([Id], [Description], [Name], [Url])  VALUES (1, N'all general posts', N'general', NULL);  IF EXISTS (SELECT \* FROM [sys].[identity\_columns] WHERE [name] IN (N'Id', N'Description', N'Name', N'Url') AND [object\_id] = OBJECT\_ID(N'[Categories]'))  SET IDENTITY\_INSERT [Categories] OFF;  GO  IF EXISTS (SELECT \* FROM [sys].[identity\_columns] WHERE [name] IN (N'Id', N'Description', N'Name', N'Url') AND [object\_id] = OBJECT\_ID(N'[Categories]'))  SET IDENTITY\_INSERT [Categories] ON;  INSERT INTO [Categories] ([Id], [Description], [Name], [Url])  VALUES (2, N'all other posts', N'other', NULL);  IF EXISTS (SELECT \* FROM [sys].[identity\_columns] WHERE [name] IN (N'Id', N'Description', N'Name', N'Url') AND [object\_id] = OBJECT\_ID(N'[Categories]'))  SET IDENTITY\_INSERT [Categories] OFF;  GO  CREATE UNIQUE INDEX [IX\_ContactInfo\_UserId] ON [ContactInfo] ([UserId]);  GO  CREATE INDEX [IX\_Posts\_CategoryId] ON [Posts] ([CategoryId]);  GO  CREATE INDEX [IX\_Posts\_UserId] ON [Posts] ([UserId]);  GO  CREATE INDEX [IX\_PostTag\_TagsId] ON [PostTag] ([TagsId]);  GO  INSERT INTO [\_\_EFMigrationsHistory] ([MigrationId], [ProductVersion])  VALUES (N'20210813161644\_InitMigration', N'5.0.9');  GO  COMMIT;  GO  BEGIN TRANSACTION;  GO  ALTER TABLE [Posts] ADD [LongDescription] nvarchar(max) NULL;  GO  UPDATE POSTS SET LongDescription = Description  GO  INSERT INTO [\_\_EFMigrationsHistory] ([MigrationId], [ProductVersion])  VALUES (N'20210813162415\_AddLongDescriptionInPost', N'5.0.9');  GO  COMMIT;  GO  BEGIN TRANSACTION;  GO  ALTER TABLE [Posts] ADD [ImageUrl] nvarchar(max) NULL;  GO  INSERT INTO [\_\_EFMigrationsHistory] ([MigrationId], [ProductVersion])  VALUES (N'20210813164952\_AddImageUrlInPost', N'5.0.9');  GO  COMMIT;  GO  BEGIN TRANSACTION;  GO  DECLARE @var0 sysname;  SELECT @var0 = [d].[name]  FROM [sys].[default\_constraints] [d]  INNER JOIN [sys].[columns] [c] ON [d].[parent\_column\_id] = [c].[column\_id] AND [d].[parent\_object\_id] = [c].[object\_id]  WHERE ([d].[parent\_object\_id] = OBJECT\_ID(N'[Posts]') AND [c].[name] = N'LongDescription');  IF @var0 IS NOT NULL EXEC(N'ALTER TABLE [Posts] DROP CONSTRAINT [' + @var0 + '];');  ALTER TABLE [Posts] DROP COLUMN [LongDescription];  GO  INSERT INTO [\_\_EFMigrationsHistory] ([MigrationId], [ProductVersion])  VALUES (N'20210813184516\_RemoveLongDescriptionInPost', N'5.0.9');  GO  COMMIT;  GO |

Możemy też podać nazwę migracji np.

|  |
| --- |
| PM> **script-migration AddImageUrlInPost**  Build started...  Build succeeded.  PM> |

|  |
| --- |
| BEGIN TRANSACTION;  GO  DECLARE @var0 sysname;  SELECT @var0 = [d].[name]  FROM [sys].[default\_constraints] [d]  INNER JOIN [sys].[columns] [c] ON [d].[parent\_column\_id] = [c].[column\_id] AND [d].[parent\_object\_id] = [c].[object\_id]  WHERE ([d].[parent\_object\_id] = OBJECT\_ID(N'[Posts]') AND [c].[name] = N'LongDescription');  IF @var0 IS NOT NULL EXEC(N'ALTER TABLE [Posts] DROP CONSTRAINT [' + @var0 + '];');  ALTER TABLE [Posts] DROP COLUMN [LongDescription];  GO  INSERT INTO [\_\_EFMigrationsHistory] ([MigrationId], [ProductVersion])  VALUES (N'20210813184516\_RemoveLongDescriptionInPost', N'5.0.9');  GO  COMMIT;  GO |

Możemy też wygenerować skrypt między migracjami, jako parametry podajemy najpierw nazwę migracji początkowej, później nazwę migracji końcowej.

### Skrypt z podaniem nazwy pliku jaki ma powstać

Możemy podać nazwę skryptu jaki ma być utworzony np.

|  |
| --- |
| PM> **script-migration AddImageUrlInPost -output Blog.DataLayer/Migrations/Scripts/AddImageUrlInPost.sql**  Build started...  Build succeeded.  PM> |

### Skrypty idempotentne

Skrypty warunkowe, wprowadzą zmiany jeżeli ich jeszcze nie ma

|  |
| --- |
| PM> **script-migration -idempotent**  Build started...  Build succeeded.  PM> |

# **Mapowanie danych**

Zawsze w EF obowiązywała zasada **Convention over Configuration**.  
Jeżeli będziemy trzymać się konwencji, to nie będziemy potrzebować żadnych dodatkowych konfiguracji.

Niektóre konwencje:  
Id lub nazwa klasy + Id będzie traktowana jako klucz główny.  
Nazwa właściwości nawigacyjnej + Id to klucz obcy.

Dzięki odpowiedniemu mapowaniu możemy nadpisać te konwencje i oznaczyć inne właściwości jako klucz główny, czy klucz obcy. Oprócz tego możemy oznaczyć pola jako wymagane, szczegółowo określić typy naszych kolumn, relacje, walidacje, nazwy tabel i nazwy kolumn, mamy pełną kontrolę nad bazą danych, którą będziemy tworzyć.

Mamy dwa sposoby mapowania danych  
Data Annotation  
Fluent Api

## Data Annotations vs. Fluent API

W przykładzie dla porównania, ustawiamy pole Title w klasie Post jako wymagane, za pomocą  
Data Annoatation a później za pomocą Fluent Api.

|  |
| --- |
| using System.ComponentModel.DataAnnotations;  public class Post  {  [Required]  public string Title { get; set; } |

Wykonanie migracji wygeneruje kod jak poniżej.

|  |
| --- |
| public partial class SetTitleAsRequiredInPost : Migration  {  protected override void Up(MigrationBuilder migrationBuilder)  {  migrationBuilder.AlterColumn<string>(  name: "Title",  table: "Posts",  type: "nvarchar(max)",  nullable: false,  defaultValue: "",  oldClrType: typeof(string),  oldType: "nvarchar(max)",  oldNullable: true);  }  protected override void Down(MigrationBuilder migrationBuilder)  {  migrationBuilder.AlterColumn<string>(  name: "Title",  table: "Posts",  type: "nvarchar(max)",  nullable: true,  oldClrType: typeof(string),  oldType: "nvarchar(max)");  }  } |

Co po aktualizacji skutkuje odpowiednimi zmianami w bazie danych

Graphical user interface, text, application

Description automatically generated

Wycofujemy tą migrację

|  |
| --- |
| PM> remove-migration -force  Build started...  Build succeeded.  Reverting migration '20210816055511\_SetTitleAsRequiredInPost'.  Removing migration '20210816055511\_SetTitleAsRequiredInPost'.  Reverting the model snapshot.  Done.  PM> |

Na bazie danych mamy ponownie null

Text

Description automatically generated

Usuwamy annotacje [Required] z klasy domenowej.

Ustawiamy to samo za pomocą fluent API.  
Zmian dokonujemy w klasie DbContext w metodzie OnModelCreating.

|  |
| --- |
| protected override void OnModelCreating(ModelBuilder modelBuilder)  {  modelBuilder.SeedCategories();  **modelBuilder.Entity<Post>()**  **.Property(x => x.Title)**  **.IsRequired();**  } |

Po wykonaniu migracji i aktualizacji bazy uzyskujemy ten sam efekt.

## **Dlaczego Fluent API ?**

### Dobre wzorce, czysty kod

Model domenowy nie powinien być zależny od żadnych framework’ów, nie powinien zawierać żadnych dodatkowych danych konfiguracyjnych.  
Dlatego lepiej stosować Fluent API, ponieważ kod konfiguracyjny powiązany z EF jest w osobnym miejscu, umieszczanie go w klasach domenowych nie jest dobrą praktyką.

### Możliwości

Fluent Api ma większe możliwości, zrobimy w nim wszystko to co w Data Annotation natomiast odwrotnie w Data Annotation nie zrobimy wszystkiego tego, co jest możliwe w FluentAPI.

### Mieszanie kodu

W związku z tym jeżeli będziesz chciał użyć Data Annotation to czasami również będziesz musiał użyć Fluent API, co jest najgorszym połączeniem, wówczas część kodu będzie w modelu domenowym a część w zupełnie innym miejscu, co wprowadzi dużo bałaganu w kodzie i będzie on cięższy w utrzymaniu i rozwijaniu.

|  |
| --- |
| **Fluent API** wymaga napisania trochę większej ilości kodu, ale będzie on lepszej jakości |

## **Fluent Api: Najlepsze Praktyki**

Jeżeli będziemy stosować Fluent API, to ilość kodu w metodzie OnModelCreating() będzie duża i klasa będzie puchnąć i stanie się przez to nieczytelna.  
W związku z najlepszą praktyką jest utworzenie osobnego pliku dla każdej z encji.

W DataLayer tworzymy katalog **Configurations**. W nim będziemy tworzyć pliki konfiguracyjne dla każdej z encji trzymając się konwencji nazwa encji + „configuration”.

Graphical user interface, text, application, chat or text message

Description automatically generated

|  |
| --- |
| Taka klasa musi implementować interfejs **IEntityTypeConfiguration<T>** przekazując encję jako typ generyczny. |

|  |
| --- |
| class PostConfiguration : IEntityTypeConfiguration<Post>  {  public void Configure(EntityTypeBuilder<Post> builder)  {  throw new NotImplementedException();  }  } |

|  |
| --- |
| class PostConfiguration : IEntityTypeConfiguration<Post>  {  public void Configure(EntityTypeBuilder<Post> builder)  {  builder  .Property(x => x.Title)  .IsRequired();  }  } |

Aby to zadziałało musimy wskazać w ApplicationDbContext w Metodzie OnModelCreating().  
Można to zrobić na kilka sposobów.

|  |
| --- |
| protected override void OnModelCreating(ModelBuilder modelBuilder)  {  modelBuilder.SeedCategories();  modelBuilder.ApplyConfiguration(new PostConfiguration());    } |

Z biegiem czasu może się nazbierać trochę kodu

Możemy również dodać za pomocą **mechanizmu refleksji** wszystkie konfiguracje.  
Wtedy taka metoda przeskanuje cały projekt w poszukiwaniu klas implementujących interfejs  
IEntityTypeConfiguration.

|  |
| --- |
| **using System.Reflection;**  protected override void OnModelCreating(ModelBuilder modelBuilder)  {  modelBuilder.SeedCategories();  modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());    } |

## Fluent API: nazwy tabel, pól, wielkości pól, indeksy, unikalność

Domyślnie stringi są mapowane na nvchar, możemy to zmienić ustawiając IsUnicode na false

Ważniejsza jest konfiguracja niż konwencja.

|  |
| --- |
| class PostConfiguration : IEntityTypeConfiguration<Post>  {  public void Configure(EntityTypeBuilder<Post> builder)  {  builder.ToTable("Posts2");    builder.Property(x => x.Title)  .HasMaxLength(100)  .HasColumnName("Title2")  .IsRequired();  builder.Property(x => x.PostedOn)  .HasColumnType("datetime");  builder.Property(x => x.ShortDescription)  .HasMaxLength(50);  builder.Property(x => x.Description)  .HasMaxLength(200);  builder.Property(x => x.ImageUrl)  .IsUnicode(false)  .HasDefaultValue("/content/image.png");  builder.Property(x => x.Published)  .IsRequired(false);  }  } |

|  |
| --- |
| class CategoryConfiguration : IEntityTypeConfiguration<Category>  {  public void Configure(EntityTypeBuilder<Category> builder)  {  // ustawiamy do jakiej tabeli, nic się nie zmieni,  // bo podaliśmy nazwę zgodną z konwencją  builder.ToTable("Categories");  // ustawiamy klucz główny, również się nic nie zmieni  // bo wg konwencji byłoba to również kolumna Id  builder.HasKey(x => x.Id);  // Ustawiamy indeks i unikalność na polu Name  builder.HasIndex(x => x.Name)  .IsUnique();  builder.Property(x => x.Name)  .HasMaxLength(20);  builder.Property(x => x.Url)  .HasMaxLength(500);  builder.Property(x => x.Description)  .HasMaxLength(20);    }  } |

## Fluent API: konfigurowanie relacji

EF zawiera wystarczająco dużo konwencji, aby skonfigurować relacje bez dodatkowego kodu.  
Jednakże możesz użyć Fluent API w celu łatwiejszego utrzymania kodu.

## **Fluent API: konfiguracja relacji jeden-do-jeden**

Zgodnie z konwencją musimy wskazać, która encja jest główna a która zależna.  
W encji **User**, która jest główna wystarczy dodać tylko właściwość nawigacyjną.

|  |
| --- |
| public class User  {  public int Id { get; set; }  // właściwość nawigacyjna  public ContactInfo ContactInfo { get; set; }  } |

W encji **ContactInfo**, która jest zależna oprócz właściwości nawigacyjnej dodajemy też klucz obcy

|  |
| --- |
| public class ContactInfo  {  public int Id { get; set; }  // klucz obcy  public int UserId { get; set; }  // właściwość nawigacyjna  public User User { get; set; }  } |

Text

Description automatically generated

W bazie danych mamy kolumnę UserId, która jest kluczem obcym do użytkownika i ta kolumna jest not null, czyli nie jesteśmy w stanie najpierw utworzyć kontaktu, musimy najpierw utworzyć użytkownika.

**UserId** jest kluczem obcym, ponieważ nazwa jest zgodna z konwencją to nie musimy konfigurować relacji za pomocą Fluent API.  
Jeżeli właściwość będąca kluczem obcym miałaby inną nazwę musielibyśmy wskazać to za pomocą Fluent API.

Wpis dodajemy tylko w konfiguracji klasy nadrzędnej.

|  |
| --- |
| class UserConfiguration : IEntityTypeConfiguration<User>  {  public void Configure(EntityTypeBuilder<User> builder)  {    // właściwość nawigacyjną w klasie User  // właściwość nawigacyjną w klasie ContactInfo  // klucz obcy w ContactInfo  **builder**  **.HasOne(u => u.ContactInfo)**  **.WithOne(c => c.User)**  **.HasForeignKey<ContactInfo>(c => c.UserId);**  }  } |

Graphical user interface, text, application

Description automatically generated  
Powyższy wpis nic nie zmieni bo mamy nazwy pól nadal zgodne z konwencją.

## **Fluent API: konfigurowanie relacji jeden-do-wielu**

W naszym przykładzie mamy dwie takie relacje. Pierwsza pomiędzy encjami **Post -> Category**  
Każdy post ma jedną kategorię natomiast każda kategoria ma wiele postów.  
W Post mamy właściwość nawigacyjną Category i klucz obcy CategoryId.

|  |
| --- |
| public class Post  {  public int Id { get; set; }  // klucz obcy do Kategorii  public int **CategoryId** { get; set; }  // właściwość nawigacyjna  public Category **Category** { get; set; }  } |

W Category mamy kolekcję artykułów, jest ona od razu inicjowana aby zapobiec błędom związanym z nulami. Stosujemy najprostszą kolekcję.

|  |
| --- |
| Public class Category  {  public int Id { get; set; }  // właściwości nawigacyjne  public ICollection<Post> Posts { get; set; } = new HashSet<Post>();  } |

Klucz obcy ma nazwę zgodną z konwencją, dlatego ta właściwość została automatycznie zmapowana  
w bazie danych na klucz obcy

Text

Description automatically generated

A picture containing graphical user interface

Description automatically generated

Konfigurację możemy umieścić w PostConfiguration lub w CategoryConfiguration

**Konfiguracja w PostConfiguration**

|  |
| --- |
| class PostConfiguration : IEntityTypeConfiguration<Post>  {  public void Configure(EntityTypeBuilder<Post> builder)  {  // relacja z kategorią  **builder.HasOne<Category>(p => p.Category)**  **.WithMany(c => c.Posts)**  **.HasForeignKey(p => p.CategoryId)**  **.OnDelete(DeleteBehavior.Restrict);**  }  } |

Graphical user interface, text, application

Description automatically generated

**Konfiguracja w CategoryConfiguration**

|  |
| --- |
| class CategoryConfiguration : IEntityTypeConfiguration<Category>  {  public void Configure(EntityTypeBuilder<Category> builder)  {  **builder**  **.HasMany<Post>(c => c.Posts)**  **.WithOne(p => p.Category)**  **.HasForeignKey(p => p.CategoryId);**  }  } |

Text

Description automatically generated with medium confidence

Post->User – każdy Post ma jednego Usera, każdy user ma wiele postów  
W User mamy właściwość nawigacyjną User i klucz obcy UserId

|  |
| --- |
| public class Post  {  public int Id { get; set; }  // klucz obcy do użytkownika  public int UserId { get; set; }  // właściwość nawigacyjna  public User User { get; set; }  } |

Mamy kolekcję artykułów, które stworzył ten użytkownik.

|  |
| --- |
| public class User  {  public int Id { get; set; }  // właściwość nawigacyjna  public ICollection<Post> Posts { get; set; } = new HashSet<Post>();  } |

**Wiele relacji pomiędzy tymi samymi encjami.**

Jeżeli chciałbyś do artykułu dodać informację również o użytkowniku, który zatwierdził dany artykuł,  
to musiał byś stworzyć nową relację, czyli do encji Post dodać dwie nowe właściwości.

|  |
| --- |
| public class Post  {  public int Id { get; set; }  // klucz obcy do użytkownika  public int UserId { get; set; }  // właściwość nawigacyjna  public User User { get; set; }  // klucz obcy do użytkownika  **public int ApprovedByUserId { get; set; }**  // właściwość nawigacyjna  **public User ApprovedBy { get; set; }**  } |

A do encji User nową kolekcję PostApproved zawierającą zatwierdzone Posty.

|  |
| --- |
| public class User  {  public int Id { get; set; }  public ICollection<Post> Posts { get; set; } = new HashSet<Post>();  **public ICollection<Post> PostsApproved { get; set; } = new HashSet<Post>();**  } |

Teraz użytkownik ma oprócz kolekcji artykułów, które stworzył, kolekcję artykułów, które zatwierdził.

Taką konfigurację możemy ustawić w PostConfiguration lub w UserConfiguration.  
**Preferowana konfiguracja w encji, gdzie mamy klucz obcy.**

Relacja Post z Użytkownikiem, który utworzył post  
Konfiguracja w **PostConfiguration**

|  |
| --- |
| Class PostConfiguration : IEntityTypeConfiguration<Post>  {  public void Configure(EntityTypeBuilder<Post> builder)  {  **builder.HasOne(x => x.User)**  **.WithMany(x => x.Posts)**  **.HasForeignKey(x => x.UserId);**  }  } |

Graphical user interface, text, application

Description automatically generated

Relacja Post z Użytkownikiem, który utworzył post

Konfiguracja w **UserConfiguration**

|  |
| --- |
| class UserConfiguration : IEntityTypeConfiguration<User>  {  public void Configure(EntityTypeBuilder<User> builder)  {  **builder.HasMany(x => x.Posts)**  **.WithOne(x => x.User)**  **.HasForeignKey(x => x.UserId);**  }  } |

Graphical user interface, text, application

Description automatically generated

Relacja Post z Użytkownikiem, który zatwierdził post

Konfiguracja w **PostConfiguration**

|  |
| --- |
| class PostConfiguration : IEntityTypeConfiguration<Post>  {  public void Configure(EntityTypeBuilder<Post> builder)  {  // relacja z użytkownikiem, który zatwierdził  **builder.HasOne<User>(p => p.ApprovedBy)**  **.WithMany(u => u.PostsApproved)**  **.HasForeignKey(p => p.ApprovedByUserId)**  **.OnDelete(DeleteBehavior.Restrict);**  }  } |

Graphical user interface, text, application, email

Description automatically generated

Relacja Post z Użytkownikiem, który zatwierdził post

Konfiguracja w **UserConfiguration**

|  |
| --- |
| class UserConfiguration : IEntityTypeConfiguration<User>  {  public void Configure(EntityTypeBuilder<User> builder)  {  **builder.HasMany<Post>(u => u.PostsApproved)**  **.WithOne(p => p.ApprovedBy)**  **.HasForeignKey(p => p.ApprovedByUserId);**  }  } |

Timeline

Description automatically generated with medium confidence

Właściwość możemy oznaczyć jako nulowalną, dzięki czemu relacja te będzie opcjonalna

|  |
| --- |
| public class Post  {  public int Id { get; set; }  // klucz obcy do użytkownika  **public int? ApprovedByUserId { get; set; }**  // właściwość nawigacyjna  public User ApprovedBy { get; set; }  } |

Po migracji i update

Graphical user interface, text, application

Description automatically generated

## Fluent API: relacja Wiele Do Wielu

W przykładowym projekcie mamy jedną relację typu wiele do wielu pomiędzy encjami Post i Tag.  
Każdy artykuł może mieć wiele tagów,  
Każdy tag może zostać przypisany do wielu artykułów.

Zdefiniowaliśmy już to za pomocą konwencji.

|  |
| --- |
| public class Post  {  public int Id { get; set; }  // właściwość nawigacyjna  **public ICollection<Tag> Tags { get; set; } = new HashSet<Tag>();**  } |

|  |
| --- |
| public class Tag  {  public int Id { get; set; }  // właściwość nawigacyjna  **public ICollection<Post> Posts { get; set; } = new HashSet<Post>();**  } |

Do naszej bazy została dodana tabele PostTag, mimo, że takiej encji w naszym projekcie nie mamy.

A picture containing graphical user interface

Description automatically generated

Takie mapowanie możemy dodać również za pomocą Fluent API

### Standardowa tabela łącząca

Konfigurację dodajemy po stronie PostConfiguration lub TagConfiguration bez znaczenia gdzie.

|  |
| --- |
| // wiele do wielu  // HasMany - właściwość nawigacyjna  builder.HasMany(x => x.Tags)  .WithMany(x => x.Posts)  .UsingEntity(x => x.ToTable("PostsTagMaps")); |

Podajemy jako parametr nazwę tabeli.

Text

Description automatically generated

### Tabele łącząca z dodatkowym polem

Jeżeli potrzebujemy dodatkowego pola w tabeli łączącej to  
musimy dodać nową encję np. PostsTagsMaps   
i tam zdefiniować wszystkie właściwości i skonfigurować mapowanie.

Czyli tworzymy nową klasę

|  |
| --- |
| public class PostTag  {  public int PostId { get; set; }  public int TagId { get; set; }  public DateTime CreatedDate { get; set; }  } |

Tworzymy klasę konfiguracyjną

|  |
| --- |
| class PostTagConfiguration : IEntityTypeConfiguration<PostTag>  {  public void Configure(EntityTypeBuilder<PostTag> builder)  {  builder.ToTable("PostsTagsMaps");  }  } |

Tworzymy relację

|  |
| --- |
| class PostConfiguration : IEntityTypeConfiguration<Post>  {  public void Configure(EntityTypeBuilder<Post> builder)  {  // wiele do wielu  // HasMany - właściwość nawigacyjna  // nasz post ma wiele tagów  // a każdy tag ma wiele postów  // i używamy encji łączącej PostTag  builder.HasMany(x => x.Tags)  .WithMany(x => x.Posts)  .UsingEntity<PostTag>(  x => x.HasOne(x => x.Tag).WithMany().HasForeignKey(x => x.TagId),  x => x.HasOne(x => x.Post).WithMany().HasForeignKey(x => x.PostId)  )  .Property(x => x.CreatedDate)  .HasDefaultValueSql("getdate()");  }  } |

Te działania wystarczą, aby zostały założone relacje.

Aby móc tworzyć zapytania dodajemy też kolejną właściwość DataSet do klasy DbContext.  
Będzie to tylko pomocne po stronie c#, w bazie danych nic się nie zmieni.

|  |
| --- |
| public class ApplicationDbContext : DbContext  {  public DbSet<Category> Categories { get; set; }  public DbSet<ContactInfo> ContactInfo { get; set; }  public DbSet<Post> Posts { get; set; }  public DbSet<Tag> Tags { get; set; }  public DbSet<User> Users { get; set; }  **public DbSet<PostTag> PostTags { get; set; }**  } |

## Usuwanie Rekordów Powiązanych

Konfiguracja kaskadowego usuwania rekordów.  
Problem usuwanie rekordów podrzędnych, które są przyporządkowane do rekordów nadrzędnych, w przypadku usunięcia rekordu nadrzędnego.

Mamy do wyboru 4 zachowania:  
Casdade  
ClientSetNull  
SetNull  
Restrict

Będą się one inaczej zachowywać przy kluczach obcych oznaczonych jako wymagane i niewymagane nullable.  
Dla kluczy obcych oznaczonych jako wymagane, domyślnym ustawieniem jest Cascade tak się zachowa w przypadku usunięcia użytkownika, który utworzył post.  
W przypadku klucza obcego typu nullable domyślnym ustawieniem jest ustawienie w kluczu obcym wartości null - ClientSetNull. Aby to zadziałało musimy przy zapytaniu załadować informacje o artykule.

|  |
| --- |
| class PostConfiguration : IEntityTypeConfiguration<Post>  {  public void Configure(EntityTypeBuilder<Post> builder)  {  // HasOne przekazujemy właściwość nawigacyjną z Post  // WithMany przekazujemy kolekcję, czyli właściwość nawigacyjną z User: Posts  builder.HasOne(x => x.User)  .WithMany(x => x.Posts)  .HasForeignKey(x => x.UserId)  **.OnDelete(DeleteBehavior.Restrict);**  // relacja z użytkownikiem, który zatwierdził  builder.HasOne(x => x.ApprovedBy)  .WithMany(x => x.PostsApproved)  .HasForeignKey(x => x.ApprovedByUserId);  }  } |

W przypadku próby usunięcia użytkownika, który ma artykuły zostanie rzucony wyjątek.

|  |
| --- |
| class PostConfiguration : IEntityTypeConfiguration<Post>  {  public void Configure(EntityTypeBuilder<Post> builder)  {  // HasOne przekazujemy właściwość nawigacyjną z Post  // WithMany przekazujemy kolekcję, czyli właściwość nawigacyjną z User: Posts  builder.HasOne(x => x.User)  .WithMany(x => x.Posts)  .HasForeignKey(x => x.UserId)  **.OnDelete(DeleteBehavior.Cascade);**  // relacja z użytkownikiem, który zatwierdził  builder.HasOne(x => x.ApprovedBy)  .WithMany(x => x.PostsApproved)  .HasForeignKey(x => x.ApprovedByUserId);  }  } |

W przypadku usunięcia użytkownika, który ma artykuły zostaną usunięte wszystkie jego artykuły.

# Inne

## Zakładanie użytkownika w MS SQL Server

Zakładamy użytkownika w **Security->logins**

Ważne w zakładce **Roles** dodać uprawnienie **sysadmin**

# Language-Integrated Query (LINQ)

Część technologii .Net, która umożliwia zadawanie zapytań na obiektach.  
Dwie składnie:

**Query syntax**   
Bazuje na zapytaniach SQL

|  |
| --- |
| var numbers = new List<int>() { 1, 5, 7, 22, 0, 13, 17, 2, 4, 9, 33, 44, 49, 50, 55, 99, 314 };  **var evenNumbersQuerySyntax = from x in numbers**  **where x % 2 == 0**  **select x;** |

**Method Syntax** = Lambda Expresion  
Bazuje na metodach rozszerzających i wyrażeniach lambda.

|  |
| --- |
| var numbers = new List<int>() { 1, 5, 7, 22, 0, 13, 17, 2, 4, 9, 33, 44, 49, 50, 55, 99, 314 };  **var evenNumbersMethodSyntax1 = numbers.Where(x => x % 2 == 0);** |

Oczekujemy delegata, możemy więc przekazać go w postaci wyrażenia lambda lub funkcji.

|  |
| --- |
| **var evenNumbersMethodSyntax2 = numbers.Where(GetEventNumber);**  private static bool GetEventNumber(int x)  {  return x%2 == 0;  } |

Where jest metodą rozszerzającą interfejsu IEnumerable, która przyjmuje Funk’a i sama zwraca też interfejs IEnumerable

|  |
| --- |
| public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate); |

Metoda składni oferuje większe możliwości i jest częściej stosowana.

# DbContext

Wszystkie zapytania będziemy wykonywać na klasie dziedziczącej po DbContext.  
Domyślnie context zarządza połączeniami z bazą danych  
Każda instancja po wykonaniu zapytania powinna zostać usunięta.  
Musimy pamiętać więc o wywołaniu metody Dispose().

W c# najłatwiej w tym celu użyć instrukcji using().

|  |
| --- |
| using (var context = new ApplicationDbContext() )  {    Console.ReadLine();  } |

Mamy pewność ,że po wykonaniu instrukcji wszystkie niezarządzane zasoby zostaną zwolnione  
i zapobiegniemy w ten sposób wyciekom pamięci. Gdybyć w całej swojej aplikacji pracował na jednej instancji DbContext, czyli na Singletonie to każde kolejne zapytanie byłoby coraz wolniejsze.  
Zasób ten powinien mieć jak najkrótszy czas życia.

Najlepszym sposobem użycia jest używanie Dependency Injection i wstrzykiwanie contextu do nowego requesta w aplikacji webowej.  
Wtedy po wykonaniu requesta zwalniane są wszystkie zasoby DbContextu.  
Wówczas będziemy pracować na jednej instancji context’u w jednym żądaniu.

W aplikacjach desktopowych najlepszym rozwiązaniem jest stosowanie jednego contextu na jeden formularz.

DbContext ma dostęp do repozytoriów czyli kolekcji obiektów.  
Odwoujemy się do dbset’ów, które zdefiniowaliśmy w modelu domenowym.

Pierwsze zapytanie za pomocą dbContext będzie dluzej trwało, ponieważ wykonuje się jeszcze kilka dodatkowych zadań.

Context ma też medtodę Find(), której nie ma w Linq

|  |
| --- |
| var userWithId3 = context.Users.Find(3);  Console.WriteLine(userWithId3.Login); |

SQL Server Profiler

Table

Description automatically generated