

Ćwiczenie: baza danych w c# - Entity Framework Core

- Krzysztof Molenda
- 2021-01-14 (ver. 1)

Realizując ćwiczenie *krok-po-kroku* dowiesz i nauczysz się, jak:

- łączyć się z bazą danych w SQLite
- definiować model bazy w EF Core, mapować encje (klasy) na tabele w bazie
 - generować automatycznie model (*scaffolding*)
- budować proste zapytania do bazy w oparciu o zdefiniowany model
- modyfikować zawartość bazy
- logować informacje o wykonywanych operacjach na bazie danych

Stworzysz prostą aplikację konsolową wykonującą podstawowe operacje na bazie (CRUD).

Wymagania:

- zainstalowany .Net 5 (wykorzystujemy fragmentarycznie C#9)
- w ramach `dotnet` zainstalowane `dotnet-eF` - środowisko EF Core 5
- środowisko programowania VS Code z obsługą C#
- SQLite oraz SQLiteStudio
- baza danych `Northwind`

Wprowadzenie

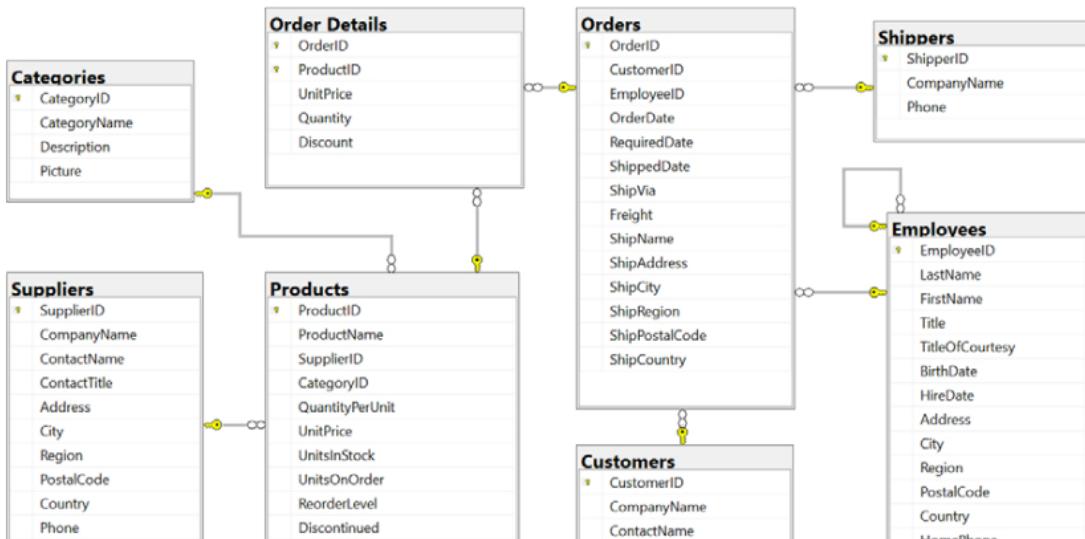
Entity Framework jest opensource-owym środowiskiem typu ORM (Object-Relational-Mapping) dla .Net. Początkowo był integralną częścią .Net Framework, obecnie stanowi niezależną część (od wersji 6).

EF Core to wersja EF niezależna od systemu operacyjnego (i w wielu aspektach różniąca się od *legacy EF*). EF Core 5 dostosowana jest do platformy .Net 5. Wspiera nie tylko relacyjne bazy danych, ale również bazy nierelacyjne i udostępniane w chmurze. EF Core nie współdziała z .Net Framework 4.8.

<https://docs.microsoft.com/en-us/ef/core/whatis-new/ef-core-5.0/whatsnew>

Northwind na SQLite

Ćwiczenie opiera się na szkoleniowej bazie danych `Northwind`. Poniżej przedstawiono jej diagram fizyczny (ERD):





Instalacja SQLite na komputerze z Windows

1. Strona producenta oprogramowania: <https://www.sqlite.org/download.html>

2. Pobierz ze strony *Precompiled Binaries for Windows*:

Precompiled Binaries for Windows

sqlite-dll-win32-x86-3340000.zip	32-bit DLL (x86) for SQLite version 3.34.0. (sha3: 725cc4107fd1e88fd0b05e150123050bc23cf37c29e31bf388ecae9f4849ca95) (491.81 KB)
sqlite-dll-win64-x64-3340000.zip	64-bit DLL (x64) for SQLite version 3.34.0. (sha3: 2ec319b69961af35fd86c0271e9c54d9b8115dfac40876a3bee4c5e51c05b8b4) (814.19 KB)
sqlite-tools-win32-x86-3340000.zip	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqlcipher.exe program, and the sqlite3_analyzer.exe program. (sha3: 75e13ae0cc38da453792aff2534fbdc8e1898939e9b5f4e540c1c12e15bb0112) (1.76 MB)

3. Wypakuj pobrany plik np. do `C:\SQLite\`

4. W zmiennych środowiskowych użytkownika dopisz do `PATH` ścieżkę do folderu z instalacją SQLite.

Tworzenie bazy danych Northwind

1. Skopiuj udostępniony skrypt SQL `Northwind.sql` do folderu projektu.

2. Utwórz bazę wydając polecenie `sqlite3 Northwind.db -init Northwind.sql`. Uzbrój się w cierpliwość 😊. Poczekaj na wyświetlenie poniższych informacji:

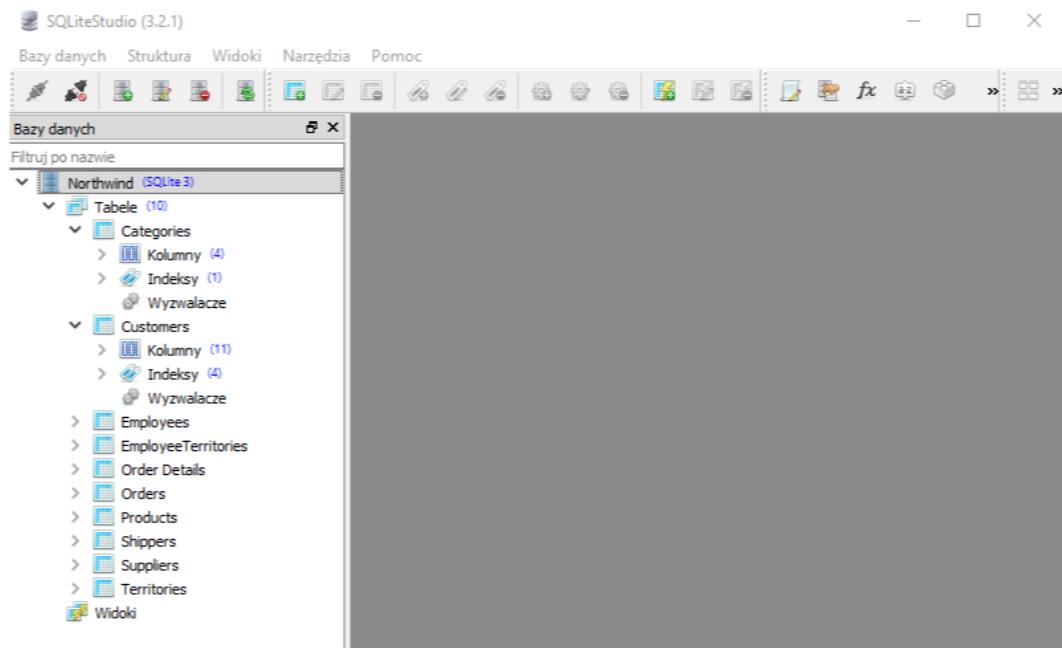
```
-- Loading resources from Northwind.sql
SQLite version 3.34.0 2020-12-01 16:14:00
Enter ".help" for usage hints.
sqlite>
```

Zakończysz działanie powłoki SQLite naciskając `Ctrl + C` w Windows lub `Ctrl + D` na macOS lub Linux.

3. Sprawdź, czy w folderze projektu znajduje się plik z bazą SQLite `Northwind.db`.

Zarządzanie bazą danych w SQLite

Do zarządzania bazami SQLite sugeruję używanie polskiego produktu, autorstwa Pawła Salawy: <http://sqlitestudio.pl/>.



Otwórz bazę, rzuć okiem na jej elementy (tabele, kolumny).

EF Core - instalacja narzędzi i środowiska

EF Core providers: <https://docs.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli>

1. W terminalu sprawdź, jakie narzędzia dotnet masz zainstalowane:

```
dotnet tool list --global
```

W moim przypadku:

Identyfikator pakietu	Wersja	Polecenia
dotnet-script	1.0.1	dotnet-script
dotnet-try	1.0.19553.4	dotnet-try
microsoft.dotnet-interactive	1.0.155302	dotnet-interactive

2. Zainstaluj `dotnet-ef` (<https://docs.microsoft.com/en-us/ef/core/cli/dotnet#installing-the-tools>) poleceniem:

```
dotnet tool install --global dotnet-ef --version 5.0.1
```

3. Jeśli miałeś już zainstalowaną wcześniejszą wersję `dotnet-ef` musisz ją usunąć (`dotnet tool uninstall --global dotnet-ef`) i ponownie przeprowadzić instalację.

EF Core - projekt aplikacji konsolowej

Określanie środowiska provider'a bazy

1. Utwórz projekt konsolowy: `dotnet new console`
2. W folderze projektu masz już plik z bazą SQLite: `Northwind.db`
3. Korygujesz plik konfiguracji projektu `.csproj` dopisując referencję do provider'a.

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>

<ItemGroup>
    <PackageReference
        Include="Microsoft.EntityFrameworkCore.Sqlite"
        Version="5.0.1" />
</ItemGroup>

</Project>
```

4. Należy teraz przywrócić zależności, budując projekt (pusty)

```
dotnet build
```

Do folderu `bin` pobrane zostaną wymagane paczki nu-get.

5. Czynności powyższe można również wykonać poleceniem:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

Więcej informacji o providerze SQLite: <https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Sqlite/>

Definiowanie modeli dla EF Core

EF core wykorzystuje pewne konwencje programowania:

- *Fluent API*,
- specjalne adnotacje (atrybuty) umożliwiając automatyczne generowanie kodu, odciążając programistę od pisania tzw. *boilerplate code*.

Entity class - pełni rolę opakowania obiektu dla tabeli bazy danych: opisuje strukturę tabeli i instancja tej klasy reprezentuje wiersz w tej tabeli.

Konwencje EF Core, nazewnictwo

<https://docs.microsoft.com/en-us/ef/core/modeling/>

Posłużmy się przykładem tabeli `Products` z bazy `Northwind`:

Nazwa tabeli: Products WITHOUT ROWID

	Nazwa	Typ danych	Klucz Główny	Klucz Obcy	Wartości unikalne	Warunek	Niepuste	Zestawienie	Domyślna wartość
1	ProductID	INTEGER	🔑						NULL
2	ProductName	nvarchar (40)				⚠️			NULL
3	SupplierID	"int"		🔗					NULL
4	CategoryID	"int"		🔗					NULL
5	QuantityPerUnit	nvarchar (20)							NULL
6	UnitPrice	"money"			📝				0
7	UnitsInStock	"smallint"			📝				0
8	UnitsOnOrder	"smallint"			📝				0
9	ReorderLevel	"smallint"			📝				0
10	Discontinued	"bit"				⚠️			0

Typ	Nazwa	Szczegóły
🔗 FOREIGN KEY	FK_Products_Categories	(CategoryID) REFERENCES Categories (CategoryID)
🔗 FOREIGN KEY	FK_Products_Suppliers	(SupplierID) REFERENCES Suppliers (SupplierID)
📝 CHECK	CK_Products_UnitPrice	(UnitPrice >= 0)
📝 CHECK	CK_ReorderLevel	(ReorderLevel >= 0)
📝 CHECK	CK_UnitsInStock	(UnitsInStock >= 0)
📝 CHECK	CK_UnitsOnOrder	(UnitsOnOrder >= 0)

- Nazwa tabeli → w klasie typu `DbContext` property typu `DBSet<T>` o takiej samej nazwie, np. u nas `Products`.
- Nazwy kolumn → nazwy properties w klasie (np. `ProductID`)
- typ `nvarchar` w SQLite → typ `string` w .Net
- typ `int` w SQLite → typ `int` w .Net
- klucz główny `ProductID` → property `ProductID` typu `guid` w .Net

Adnotacje (atrybuty) EF Core

Powyżej opisane zasady nazewnicze uzupełniane są dodatkowymi adnotacjami (aby umożliwić pełne mapowanie struktury tabel na obiekty).

Przykłady:

1. W bazie danych, w tabeli `Products`: `ProductName NVARCHAR(40) NOT NULL`.

W klasie `Product`:

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

2. W bazie danych, w tabeli `Products`: `UnitPrice MONEY CHECK (UnitPrice >= 0) DEFAULT (0)`

W klasie `Product`:

```
[Column(TypeName = "money")]
public decimal? UnitPrice { get; set; }
```

3. W bazie danych, w tabeli `Categories`: `Description NTEXT`

W klasie `Category`:

```
[Column(TypeName = "ntext")]
public string Description { get; set; }
```

EF Core Fluent API

Zamiast adnotacji, do konfiguracji mapowania można użyć *Fluent API*

Przykład:

Zamiast

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

można zapisać:

```
modelBuilder.Entity<Product>()
    .Property(product => product.ProductName)
    .IsRequired()
    .HasMaxLength(40);
```

Fluent API można wykorzystać do zainicjowania bazy. Jeśli chcemy być pewni, że baza ma co najmniej jeden wiersz w tabeli `Product`, wywołamy metodę `HasData()` jak w przykładzie:

```
modelBuilder.Entity<Product>()
    .HasData(new Product
    {
        ProductID = 1,
        ProductName = "Chai",
        UnitPrice = 8.99M
    });
});
```

<https://docs.microsoft.com/en-us/ef/core/modeling/data-seeding>

Budowanie modelu EF Core

Model opisywał będzie tylko fragment bazy `Northwind`: tabele `Products` oraz `Categories` (dla uproszczenia ćwiczenia).

1. Tworzymy 3 pliki w podfolderze `.\Model` projektu: `Northwind.cs`, `Category.cs` oraz `Product.cs` - **nazewnictwo w liczbie pojedynczej** (jak przy projektowaniu logicznym baz danych ER-D)

```
//plik: Northwind.cs
namespace NorthwindEFCore.Model
{
    public class Northwind
    {
    }
}
```

```
//plik: Category.cs
namespace NorthwindEFCore.Model
{
    public class Category
    {
    }
}
```

```
//plik: Product.cs
namespace NorthwindEFCore.Model
{
    public class Product
    {
    }
}
```

2. Skrypty DDL tych tabel wyglądają następująco. Ze skryptów tych odczytamy stosowne informacje potrzebne przy projektowaniu modelu bazy:

```
-- encja Category
CREATE TABLE "Categories" (
    "CategoryID" INTEGER PRIMARY KEY,
    "CategoryName" nvarchar (15) NOT NULL ,
    "Description" "ntext" NULL ,
    "Picture" "image" NULL
);

-- encja Product
CREATE TABLE "Products" (
    "ProductID" INTEGER PRIMARY KEY,
    "ProductName" nvarchar (40) NOT NULL ,
    "SupplierID" "int" NULL ,
    "CategoryID" "int" NULL ,
    "QuantityPerUnit" nvarchar (20) NULL ,
    "UnitPrice" float NULL ,
    "UnitsInStock" int NULL ,
    "UnitsOnOrder" int NULL ,
    "ReorderLevel" int NULL ,
    "Discontinued" bit NULL
);
```

```

    "UnitPrice" "money" NULL CONSTRAINT "DF_Products_UnitPrice" DEFAULT (0),
    "UnitsInStock" "smallint" NULL CONSTRAINT "DF_Products_UnitsInStock" DEFAULT (0),
    "UnitsOnOrder" "smallint" NULL CONSTRAINT "DF_Products_UnitsOnOrder" DEFAULT (0),
    "ReorderLevel" "smallint" NULL CONSTRAINT "DF_Products_ReorderLevel" DEFAULT (0),
    "Discontinued" "bit" NOT NULL CONSTRAINT "DF_Products_Discontinued" DEFAULT (0),
    CONSTRAINT "FK_Products_Categories" FOREIGN KEY ( "CategoryID" ) REFERENCES "Categories" ( "CategoryID" )
    CONSTRAINT "FK_Products_Suppliers" FOREIGN KEY ( "SupplierID" ) REFERENCES "Suppliers" ( "SupplierID" ),
    CONSTRAINT "CK_Products_UnitPrice" CHECK (UnitPrice >= 0),
    CONSTRAINT "CK_ReorderLevel" CHECK (ReorderLevel >= 0),
    CONSTRAINT "CK_UnitsInStock" CHECK (UnitsInStock >= 0),
    CONSTRAINT "CK_UnitsOnOrder" CHECK (UnitsOnOrder >= 0)
);

```

Między encjami `Category` i `Product` jest zależność jeden-do-wielu, zamodelowana ograniczeniem integralnościowym `FK_Products_Categories`.

3. Modyfikujemy klasę `Category`

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace NorthwindEFCore.Model
{
    public class Category
    {
        #region mapowanie kolumn
        public int CategoryID { get; set; }
        public string CategoryName { get; set; }

        [Column(TypeName = "ntext")]
        public string Description { get; set; }
        #endregion

        // navigation property for related rows
        public virtual ICollection<Product> Products { get; set; }

        // konstruktor
        public Category()
        {
            // to enable developers to add products to a Category we must
            // initialize the navigation property to an empty collection
            this.Products = new HashSet<Product>();
        }
    }
}

```

Nie mapujemy kolumny `Picture`, aby nie utrudniać projektu.

4. Modyfikujemy klasę `Product` - mapujemy tylko 6 wybranych kolumn: `ProductID`, `ProductName`, `UnitPrice`, `UnitsInStock`, `Discontinued` oraz `CategoryID`.

Kolumny nie zamapowane do `properties` nie będą mogły byćbrane pod uwagę używając instancji klasy. Jeśli użyjemy tej klasy do utworzenia obiektu, nowy wiersz w tabeli zawierać będzie `NULL` lub inną wartość domyślną dla niezamapowanych pól. W projekcie nie będziemy używać tych kolumn.

W klasie zmienimy nazwy mapowanych pól tabeli (adnotacja `[column]`).

```

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

namespace NorthwindEFCore.Model
{
    public class Product
    {
        #region mapowanie pól
        public int ProductID { get; set; }

        [Required]
        [StringLength(40)]
        public string ProductName { get; set; }

        [Column("UnitPrice", TypeName = "money")]
        public decimal? Cost { get; set; } // zmiana nazwy

        [Column("UnitsInStock")]
        public short? Stock { get; set; }

        public bool Discontinued { get; set; }

        #endregion
    }
}

```

```

    #endregion

    #region zależność od klucza obcego
    public int CategoryID { get; set; }
    public virtual Category Category { get; set; }
    #endregion

}
}

```

5. ⚠️ W klasie `Category` property `Products` oraz w klasie `Product` property `category` mają modyfikator `virtual`. Dzięki temu umożliwimy EF Core na dziedziczenie i przesłanianie tych *properties* umożliwiając wykorzystanie dodatkowych funkcjonalności, jak np. *lazy loading* (nie jest dostępne w EF Core 2.0 i wcześniejszych).
6. Klasa `Northwind` reprezentuje bazę danych. Aby użyć EF Core, musi dziedziczyć z `DbContext`. Klasa ta wie, w jaki sposób należy komunikować się z bazą danych i jak tworzyć dynamicznie zapytania SQL do manipulowania danymi.
 - W klasie dziedziczącej z `DbContext` musimy zdefiniować co najmniej jedno property typu `DbSet<T>` - reprezentuje ono tabelę. Aby poinformować EF Core, jakie kolumny ma każda tabela, właściwości `DbSet` używają typów generycznych do określania klasy, która reprezentuje wiersz w tabeli, z właściwościami reprezentującymi jego kolumny.
 - Klasa dziedzicząca z `DbContext` musi przesłonić również metodę `OnConfiguring`, która między innymi ustawia *database connection string*.
 - Opcjonalnie możemy przesłonić metodę `OnModelCreating`, aby mieć możliwość wykorzystania *Fluent API* do dekorowania klas encji atrybutami.

```

using Microsoft.EntityFrameworkCore;

namespace NorthwindEFCore.Model
{
    // klasa odpowiedzialna za zarządzanie połączeniem z bazą danych
    public class Northwind : DbContext
    {
        // properties mapujące na tabele w bazie danych
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }

        // konfiguracja connection string
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            string path = System.IO.Path.Combine(System.Environment.CurrentDirectory, "Northwind.db");
            optionsBuilder.UseSqlite($"Filename={path}");
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // przykład użycia Fluent API zamiast atrybutów (adnotacji)
            // do ograniczenia długości nazwy kategorii do 15 znaków
            modelBuilder.Entity<Category>()
                .Property(category => category.CategoryName)
                .IsRequired() // ponieważ NOT NULL
                .HasMaxLength(15);

            // dodane, aby skorygować działania na decima w SQLite
            modelBuilder.Entity<Product>()
                .Property(product => product.Cost)
                .HasConversion<double>();
        }
    }
}

```

W EF Core 3.0 i późniejszych, typ `decimal` nie jest wspierany do sortowania i innych operacji. Poprawiamy to, informując SQLite, że wartości dziesiętne można konwertować na wartości `double`. W rzeczywistości nie powoduje to żadnej konwersji w czasie wykonywania.

Scaffolding

Scaffolding (pol. rusztowanie)

metaprogramistyczna metoda budowania aplikacji wykorzystujących bazy danych; technika obsługiwana przez niektóre wzorce projektowe typu MVC, gdzie programista pisze specyfikację opisującą strukturę i zależności w bazie danych, a kompilator lub generator generuje kod umożliwiający tworzenie, czytanie,

aktualizowanie i usuwanie wpisów w bazie.

Technika scaffolding-u została spopularyzowana wraz z pojawieniem się Ruby on Rails.

[https://en.wikipedia.org/wiki/Scaffold_\(programming\)](https://en.wikipedia.org/wiki/Scaffold_(programming))

Scaffolding jest procesem używającym dedykowanego narzędzia do generowania klas reprezentujących model dla istniejącej bazy danych, wykorzystujący *inżynierię odwrotną (reverse engineering)*.

Dobre narzędzie *scaffolding-u* pozwala rozszerzyć automatycznie generowane klasy, a następnie zregenerować te klasy bez utraty klas rozszerzonych.

1. W VSCode do projektu dodajemy narzędzie do scaffolding-u (w terminalu, w folderze projektu):

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

2. Utworzymy model w podfolderze `ScaffoldModels` dla tabel: `Categories` ORAZ `Products`.

```
dotnet ef dbcontext scaffold "Filename=Northwind.db" Microsoft.EntityFrameworkCore.Sqlite --table Categories --table Products -
```

Wyjaśnienia:

- `dotnet ef` - narzędzie
- `dbcontext scaffold` - opis zadania
- `"Filename=Northwind.db"` - connection string
- `Microsoft.EntityFrameworkCore.Sqlite` - database provider
- `--table Categories --table Products` - które tabele
- `--output-dir ScaffoldModels` - do którego katalogu
- `--namespace NorthwindEFCore.ScaffoldModels` - w jakiej przestrzeni nazw
- `--data-annotations` - aby wykorzystać adnotacje (atrybuty) ale również FluentAPI
- `--context Northwind` - do określenia nazwy kontekstu

3. Wykonanie skryptu:

```
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string, you should move it out of source code. You can avoid sc
Skipping foreign key with identity '0' on table 'Products' since principal table 'Suppliers' was not found in the model. This u
```

W folderze `ScaffoldModels` utworzone zostały automatycznie wymagane klasy:

```
namespace NorthwindEFCore.ScaffoldModels
{
    [Index(nameof(CategoryName), Name = "CategoryName")]
    public partial class Category
    {
        public Category()
        {
            Products = new HashSet<Product>();
        }

        [Key]
        [Column("CategoryID")]
        public long CategoryId { get; set; }
        [Required]
        [Column(TypeName = "nvarchar (15)")]
        public string CategoryName { get; set; }
        [Column(TypeName = "ntext")]
        public string Description { get; set; }
        [Column(TypeName = "image")]
        public byte[] Picture { get; set; }

        [InverseProperty(nameof(Product.Category))]
        public virtual ICollection<Product> Products { get; set; }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;
```

```

#nullable disable

namespace NorthwindEFCore.ScaffoldModels
{
    [Index(nameof(CategoryId), Name = "CategoriesProducts")]
    [Index(nameof(CategoryId), Name = "CategoryID")]
    [Index(nameof(ProductName), Name = "ProductName")]
    [Index(nameof(SupplierId), Name = "SupplierID")]
    [Index(nameof(SupplierId), Name = "SuppliersProducts")]
    public partial class Product
    {
        [Key]
        [Column("ProductID")]
        public long ProductId { get; set; }
        [Required]
        [Column(TypeName = "nvarchar (40)")]
        public string ProductName { get; set; }
        [Column("SupplierID", TypeName = "int")]
        public long? SupplierId { get; set; }
        [Column("CategoryID", TypeName = "int")]
        public long? CategoryId { get; set; }
        [Column(TypeName = "nvarchar (20)")]
        public string QuantityPerUnit { get; set; }
        [Column(TypeName = "money")]
        public byte[] UnitPrice { get; set; }
        [Column(TypeName = "smallint")]
        public long? UnitsInStock { get; set; }
        [Column(TypeName = "smallint")]
        public long? UnitsOnOrder { get; set; }
        [Column(TypeName = "smallint")]
        public long? ReorderLevel { get; set; }
        [Required]
        [Column(TypeName = "bit")]
        public byte[] Discontinued { get; set; }

        [ForeignKey(nameof(CategoryId))]
        [InverseProperty("Products")]
        public virtual Category Category { get; set; }
    }
}

```

```

using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata;

#nullable disable

namespace NorthwindEFCore.ScaffoldModels
{
    public partial class Northwind : DbContext
    {
        public Northwind()
        {}

        public Northwind(DbContextOptions<Northwind> options)
            : base(options)
        {}

        public virtual DbSet<Category> Categories { get; set; }
        public virtual DbSet<Product> Products { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            if (!optionsBuilder.IsConfigured)
            {
#warning To protect potentially sensitive information in your connection string, you should move it out of s
                optionsBuilder.UseSqlite("Filename=Northwind.db");
            }
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Category>(entity =>
            {
                entity.Property(e => e.CategoryId).ValueGeneratedNever();
            });
        }
    }
}

```

```

        modelBuilder.Entity<Product>(entity =>
    {
        entity.Property(e => e.ProductId).ValueGeneratedNever();

        entity.Property(e => e.Discontinued).HasDefaultValueSql("0");

        entity.Property(e => e.ReorderLevel).HasDefaultValueSql("0");

        entity.Property(e => e.UnitPrice).HasDefaultValueSql("0");

        entity.Property(e => e.UnitsInStock).HasDefaultValueSql("0");

        entity.Property(e => e.UnitsOnOrder).HasDefaultValueSql("0");
    });

    OnModelCreatingPartial(modelBuilder);
}

partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
}
}

```

4. Porównajmy kod `NorthwindEFCore.ScaffoldModels.Category` z `NorthwindEFCore.Model.Category` i przyjrzyjmy się temu wygenerowanemu:

- Dyrektywa prekompilacji `#nullable disable` - EF Core aktualnie nie obsługuje *nullable reference types*, dlatego je wyłącza.
- Dekoracje atrybutami `[Index]` (od EF Core 5, wcześniej tylko *Fluent API*) wskazują *properties* które pasują do pól indeksowanych (kolumna `CategoryName` w bazie jest indeksowana). We wcześniejszych wersjach wyłączenie indeksowania tylko za pomocą *Fluent API*.
- Narzędzie `dotnet-ef` stara się samodzielnie określić liczbę pojedynczą/mnogą dla nazw (w oparciu o projekt `Humanizer`). Dlatego, dla tabeli o nazwie `Categories` wygenerowana została klasa `Category`.
- Klasa w pliku zapisana jest jako `partial`. Pozwala to programistie dodawać swobodnie dodatkowe składniki do tej klasy, ale w innym pliku (i innej części), nie naruszając wygenerowanego pliku i umożliwiając powtórne regenerowanie kodu.
- Property `[categoryId]` zostało udekorowane atrybutem `[Key]`, ponieważ kolumna `CategoryID` jest kluczem głównym w tabeli.
- Property `[Products]` jest oznaczone atrybutem `[InverseProperty]`, definiującym klucz obcy w zależności między property `Category` oraz klasą `Product`.

5. Wygenerowany kod w klasie `NorthwindEFCore.ScaffoldModels.Product` wygląda analogicznie do tego z `NorthwindEFCore.ScaffoldModels.Category`. Pojawiły się odwołania do tabeli `Suppliers` nie branej pod uwagę w tym projekcie.

6. Kod klasy `NorthwindEFCore.ScaffoldModels.Northwind`

- Są dwa konstruktory - bezargumentowy i z parametrem, umożliwiającym przekazanie informacji o sposobie połączenia z bazą (ten drugi w aplikacji ułatwi przekazanie *connection string* w czasie uruchomienia)
- W przesłoniętej metodzie `OnConfiguring`, jeśli w konstruktorze nie określono opcji połączenia, nastąpi przyłączenie do bazy `Northwind.db` zlokalizowanej w bieżącym katalogu aplikacji. W kodzie umieszczono dyrektywę prekompilacji z ostrzeżeniem, iż umieszczanie danych związanych z połączeniem do bazy bezpośrednio w kodzie nie jest dobrym pomysłem (np. hasło dostępu, możliwa dekomplikacja kodu i poznanie takiego hasła)
- W przesłoniętej metodzie `OnModelCreating`, za pomocą *Fluent API* skonfigurowano dwie klasy encji: `Category` oraz `Product`. Ostatnią linią kodu w tej metodzie jest wywołanie metody `OnModelCreatingPartial`, w której programista, w innym pliku, doda własną obsługę modelu bazy. Aby komplikacja przeszła bezbłędnie metoda ta zapowiedziana jest jako składnik klasy `Product`: `partial void OnModelCreatingPartial(ModelBuilder modelBuilder);`. Zapowiedź zrealizowano jako metodę częściową (`partial`). Ten mechanizm pozwala na bezpieczną regenerację kodu.

Więcej informacji na temat *scaffolding-u*: <https://docs.microsoft.com/en-us/ef/core/managing-schemas/scaffolding?tabs=dotnet-core-cli>

Zapytania do modelu

 W dalszej części wykorzystamy model opracowany ręcznie (a nie `ScaffoldModels`). Wykorzystamy *LINQ-to-SQL*.

Modyfikacja kodu klasy `Program`, prosta kwerenda

```
using static System.Console; // aby nie pisać Console.WriteLine() tylko WriteLine()
using NorthwindEFCore.Model; // model bazy
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace NorthwindEFCore
{
    class Program
    {
        static void Main(string[] args)
        {
            CategoriesAndProducts();
        }

        static void CategoriesAndProducts()
        {
            using var db = new Northwind(); //C#8

            WriteLine("Lista kategorii i liczba produktów:");
            // kwerenda pobiera wszystkie kategorie i powiązane z nimi produkty
            IQueryable<Category> categories = db.Categories
                .Include(c => c.Products);

            int i = 1;
            foreach (var c in categories)
            {
                WriteLine($"{i:D2}. {c.CategoryName}: {c.Products.Count}");
                i++;
            }

            //automatyczne zwolnienie zasobów dla zmiennej db
        }
    }
}
```

1. Importujemy przestrzenie nazw na początku pliku `Program.cs`.
2. Tworzymy metodę przeglądającą kategorie `CategoriesAndProducts`. Podłączamy się do bazy za pomocą obiektu `db` - instancji klasy `Northwind` dziedziczącej z `DbContext`. Klasa `DbContext` implementuje `IDisposable`. Używamy notacji C#8 dla `using`. W zapytaniu wykorzystujemy interfejs `IQueryable<T>` i metodę rozszerzającą `EntityFrameworkQueryables.Include< TEntity, TProperty >`.
3. Uruchamiamy program

```
Lista kategorii i liczba produktów w kategorii.
01. Beverages: 12
02. Condiments: 12
03. Confections: 13
04. Dairy Products: 10
05. Grains/Cereals: 7
06. Meat/Poultry: 6
07. Produce: 5
08. Seafood: 12
```

Filtrowanie i sortowanie

Odfiltrujemy te kategorie, których produkty są na magazynie w wymaganej liczbie sztuk.

```
static void CategoriesAndProductsWithMinimumInStock()
{
    using var db = new Northwind();

    WriteLine("Kategorie z minimalną liczbą produktów w magazynie.");
    Write("Podaj zakładaną minimalną liczbę sztuk: ");
    int stock = int.Parse(ReadLine());

    IQueryable<Category> categories = db.Categories
        .Include(c => c.Products)
        .Where(p => p.Stock >= stock)
        );

    foreach (Category c in categories)
    {
        WriteLine($"Kategoria: {c.CategoryName} - {c.Products.Count} produkt(y/ów) z minimum {stock} szt. w magazynie");
    }
}
```

```

        foreach (Product p in c.Products)
    {
        WriteLine($" - Produkt: {p.ProductName}: {p.Stock} szt.");
    }
}

```

Wynik uruchomienia metody dla liczby sztuk 50:

```

Kategorie z minimalną liczbą produktów w magazynie.
Podaj zakładaną minimalną liczbę sztuk: 50
Kategoria: Beverages - 5 produkt(y/ów) z minimum 50 szt. w magazynie.
- Produkt: Sasquatch Ale: 111 szt.
- Produkt: Chartreuse verte: 69 szt.
- Produkt: Laughing Lumberjack Lager: 52 szt.
- Produkt: Rhönbräu Klosterbier: 125 szt.
- Produkt: Lakaliköri: 57 szt.
Kategoria: Condiments - 4 produkt(y/ów) z minimum 50 szt. w magazynie.
- Produkt: Chef Anton's Cajun Seasoning: 53 szt.
- Produkt: Grandma's Boysenberry Spread: 120 szt.
- Produkt: Sirop d'érále: 113 szt.
- Produkt: Louisiana Fiery Hot Pepper Sauce: 76 szt.
Kategoria: Confections - 2 produkt(y/ów) z minimum 50 szt. w magazynie.
- Produkt: NuNuCafe Nuß-Nougat-Creme: 76 szt.
- Produkt: Valkoinen sukkila: 65 szt.
Kategoria: Dairy Products - 3 produkt(y/ów) z minimum 50 szt. w magazynie.
- Produkt: Queso Manchego La Pastor: 86 szt.
- Produkt: Geitost: 112 szt.
- Produkt: Raclette Courdavault: 79 szt.
Kategoria: Grains/Cereals - 2 produkt(y/ów) z minimum 50 szt. w magazynie.
- Produkt: Gustaf's Knäckebröd: 104 szt.
- Produkt: Tunnbröd: 61 szt.
Kategoria: Meat/Poultry - 1 produkt(y/ów) z minimum 50 szt. w magazynie.
- Produkt: Pâté chinois: 115 szt.
Kategoria: Produce - 0 produkt(y/ów) z minimum 50 szt. w magazynie.
Kategoria: Seafood - 6 produkt(y/ów) z minimum 50 szt. w magazynie.
- Produkt: Inlægd Sill: 112 szt.
- Produkt: Boston Crab Meat: 123 szt.
- Produkt: Jack's New England Clam Chowder: 85 szt.
- Produkt: Spegelsild: 95 szt.
- Produkt: Escargots de Bourgogne: 62 szt.
- Produkt: Röd Kaviar: 101 szt.

```

Więcej na temat *Filtered include* w EF Core 5: <https://docs.microsoft.com/en-us/ef/core/querying/related-data/eager#filtered-include>

Kolejne zapytanie filtruje produkty o cenie wyższej od granicznej, sortując wyniki.

```

static void ProductsWithHighestPriceOrdered()
{
    using var db = new Northwind();

    WriteLine("Produkty o cenie przekraczającej zadaną, w kolejności od najdroższego do najtańszego.");
    decimal price;
    do
    {
        Write("Podaj cenę minimalną produktu: ");
    } while (!decimal.TryParse(ReadLine(), out price));

    var products = db.Products
        .Where(product => product.Cost >= price)
        .OrderByDescending(product => product.Cost);

    var products1 =
        from product in db.Products
        where product.Cost >= price
        orderby product.Cost descending
        select new {product.ProductID, product.ProductName, product.Cost, product.Stock};

    foreach (var p in products)
    {
        WriteLine($"id: {p.ProductID}, Produkt: {p.ProductName}, cena: {p.Cost:$#,##0.00}, na stanie {p.Stock}");
    }
}

```

Wynik wykonania metody:

```
Produkty o cenie przekraczającej zadaną, w kolejności od najdroższego do najtańszego.  
Podaj cenę minimalną produktu: 50  
id: 38, Produkt: Côte de Blaye, cena: $263,50, na stanie 17 szt.  
id: 29, Produkt: Thüringer Rostbratwurst, cena: $123,79, na stanie 0 szt.  
id: 9, Produkt: Mishi Kobe Niku, cena: $97,00, na stanie 29 szt.  
id: 20, Produkt: Sir Rodney's Marmalade, cena: $81,00, na stanie 40 szt.  
id: 18, Produkt: Carnarvon Tigers, cena: $62,50, na stanie 42 szt.  
id: 59, Produkt: Raclette Courdavault, cena: $55,00, na stanie 79 szt.  
id: 51, Produkt: Manjimup Dried Apples, cena: $53,00, na stanie 20 szt.
```

W metodzie zawarto dwa równoważne warianty kwerendy: `products` i `products1`.

Eksport zapytania SQL

Zapytanie LINQ (*Linq-to-SQL*) operujące na modelu tłumaczone jest na zapytanie SQL dla wskazanego systemu zarządzania relacyjną bazą danych.

Postać zapytania możemy uzyskać, wywołując - jako ostatnią - metodę `ToQueryString()`.

```
static string CategoriesAndProductsWithMinimumInStockExportToQueryString(int stock)  
{  
    using var db = new Northwind();  
    return  
        db.Categories  
            .Include(c => c.Products.Where(p => p.Stock >= stock))  
            .ToQueryString();  
}  
  
// ... w Main()  
WriteLine( CategoriesAndProductsWithMinimumInStockExportToQueryString(stock: 50) );
```

Wynik wywołania:

```
.param set @_stock_0 50  
  
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description", "t"."ProductID", "t"."CategoryID", "t"."UnitPrice"  
FROM "Categories" AS "c"  
LEFT JOIN (  
    SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice", "p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"  
    FROM "Products" AS "p"  
    WHERE "p"."UnitsInStock" >= @_stock_0  
) AS "t" ON "c"."CategoryID" = "t"."CategoryID"  
ORDER BY "c"."CategoryID", "t"."ProductID"
```

Rejestrowanie (logowanie) współdziałania EF Core z bazą danych

W sytuacji, gdy potrzebujemy monitorowania komunikacji framework'a EF Core z bazą danych, wykorzystujemy wbudowane w C# mechanizmy logowania (dla .Net Core i ASP .Net Core Microsoft dostarcza własne mechanizmy: [Logging in .NET Core and ASP.NET Core](#)). Musimy:

- zarejestrować tzw. *logging provider'a* ([interfejs IServiceProvider](#)),
- zaimplementować własną klasę `Logger` bazując na [interfejsie ILogger](#).

1. Tworzymy w projekcie plik `EFConsoleLogger.cs` i umieszczamy w nim dwie definicje klas: `EFConsoleLoggerProvider`

```
: IServiceProvider ORAZ EFConsoleLogger : ILogger  
  
// file EFConsoleLogger.cs  
using System;  
using static System.Console;  
using Microsoft.Extensions.Logging;  
  
namespace NorthwindEFCore  
{  
  
    public class EFConsoleLoggerProvider : IServiceProvider  
    {  
  
    }  
}
```

```

public class EFConsoleLogger : ILogger
{
}

```

2. Implementujemy klasę `EFConsoleLoggerProvider`:

```

public class EFConsoleLoggerProvider : IServiceProvider
{
    public ILogger CreateLogger(string categoryName) => new EfConsoleLogger();

    // gdyby nasz Logger wykorzystywał zasoby niezarządzalne
    // konieczny byłby kod w Dispose
    public void Dispose() {}
}

```

Metoda `CreateLogger` zwraca instancję `EfConsoleLogger`. Interfejs `ILoggerProvider` dziedziczy z `IDisposable`, zatem konieczna jest również implementacja metody `Dispose()`. W naszym przypadku nie wykorzystujemy zasobów niezarządzalnych związanych z instancją `EFConsoleLogger`, zatem kod metody będzie pusty.

3. Implementujemy klasę `EFConsoleLogger`:

```

public class EFConsoleLogger : ILogger
{
    // gdyby nasz Logger wykorzystywał zasoby niezarządzalne
    // zwrócilibyśmy instancje klasy obsługującej te zasoby
    // nie jest to naszym przypadkiem, więc zwracamy null
    public IDisposable BeginScope<TState>(TState state) => null;

    // definiujemy, które poziomy logowania obsługujemy:
    // nie obsługujemy - None, Trace, Information
    // obsługujemy - Debug, Warning, Error, Critical
    public bool IsEnabled(LogLevel logLevel) =>
        (logLevel == LogLevel.Trace ||
         logLevel == LogLevel.Information ||
         logLevel == LogLevel.None
        ) ? false : true;

    // określamy sposób obsługi logów (wyrzucamy na konsolę)
    public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception exception, Func<TStat
    {
        ForegroundColor = ConsoleColor.DarkGray;
        Write( $"{DateTime.Now} -> Level: {logLevel}, Event Id: {eventId}");
        if( state is not null ) // `is not` dopiero w C#9
            Write($"", State: {state});
        if( exception is not null )
            Write($"", Exception: {exception.Message});
        WriteLine();
        ResetColor();
    }
}

```

- nie obsługujemy poziomów logowania: `None`, `Trace` oraz `Information`, obsługujemy pozostałe: `Debug`, `Warning`, `Error`, `Critical`,
- definiujemy metodę `Log` logującą wybrane zdarzenia na konsolę,
- komunikaty logowania wypisywane będą na konsolę kolorem szarym.

4. Modyfikujemy plik `Program.cs` dołączając monitorowanie obsługi bazy do aplikacji.

- Musimy uzupełnić importowane przestrzenie nazw:

```

using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

```

- Do metody `CategoriesAndProducts()`, **natychmiast** po deklaracji zmiennej `db` (tzn. `using var db = new Northwind();`), dopisujemy instrukcję: tworzącą obiekt logujący i konfigurującą go:

```

var efLogger = db.GetService<ILoggerFactory>();
efLogger.AddProvider(new EFConsoleLoggerProvider());

```

W zasadzie obiekt logujący nie będzie nam do niczego potrzebny - zostaje utworzony, skonfigurowany i zarejestrowany. Dwie instrukcje możemy zapisać jedną:

```
db.GetService<ILoggerFactory>().AddProvider(new EFConsoleLoggerProvider());
```

5. W `Main()` komentujemy wszystkie wywołania kwerend, za wyjątkiem `CategoriesAndProducts()` i uruchamiamy aplikację. Otrzymujemy przeplatane teksty - efekty działania aplikacji z komunikatami *loggera* (w kolorze szarym).

Na początku wydruku widzimy, w jaki sposób zapytanie przetłumaczone zostaje na odpowiedni kod SQL, następnie skompilowane i wysłane do wykonania przez serwer bazodanowy. Po zakończeniu wykonania i wypisaniu ostatniego wyniku następuje zamknięcie połączenia i zwolnienie zasobów.

6. Wypisywanych komunikatów jest zbyt wiele (z naszego punktu widzenia). Chcemy się ograniczyć wyłącznie do tych, w których zawarty jest kod SQL wykonywany przez serwer. Są to komunikaty o `eventId`

`Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting` (formalnie o takim polu `Name`). Zdarzenia są arbitralnie ponumerowane, w tym przypadku jest to zdarzenie `eventId.Id == 20100`.

Modyfikujemy kod metody `Log` w klasie `EFConsoleLogger`, odrzucając wszystkie zdarzenia za wyjątkiem powyższego:

```
public class EFConsoleLogger : ILogger
{
    ...
    public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception exception, Func<TState, string> formatter)
    {
        if( eventId.Id != 20100 ) return;
        ...
    }
}
```

Ponownie uruchamiamy aplikację.

7. Dodajemy logowanie zdarzeń do metody `ProductsWithHighestPriceOrdered()`, usuwamy komentarz dla tej metody w `Main` i ponownie uruchamiamy aplikację.

```
Lista kategorii i liczba produktów w kategorii.
13.01.2021 22:04:08 -> Level: Debug, Event Id: Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State: Executing
DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description", "p"."ProductID", "p"."CategoryID", "p"."UnitPrice", "p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Categories" AS "c"
LEFT JOIN "Products" AS "p" ON "c"."CategoryID" = "p"."CategoryID"
ORDER BY "c"."CategoryID", "p"."ProductID"
01. Beverages: 12
02. Condiments: 12
03. Confections: 13
04. Dairy Products: 10
05. Grains/Cereals: 7
06. Meat/Poultry: 6
07. Produce: 5
08. Seafood: 12
Produkty o cenie przekraczającej zadaną, w kolejności od najdroższego do najtańszego.
Podaj cenę minimalną produktu: 50
13.01.2021 22:04:13 -> Level: Debug, Event Id: Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State: Executing
DbCommand [Parameters=@_price_0=50, CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice", "p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE "p"."UnitPrice" >= @_price_0
ORDER BY "p"."UnitPrice" DESC
id: 38, Produkt: Côte de Blaye, cena: $263,50, na stanie 17 szt.
id: 29, Produkt: Thüringer Rostbratwurst, cena: $123,79, na stanie 0 szt.
id: 9, Produkt: Mishí Kobe Nikú, cena: $97,00, na stanie 29 szt.
id: 20, Produkt: Sir Rodney's Marmalade, cena: $81,00, na stanie 40 szt.
id: 18, Produkt: Carnarvon Tigers, cena: $62,50, na stanie 42 szt.
id: 59, Produkt: Raclette Courdavault, cena: $55,00, na stanie 79 szt.
id: 51, Produkt: Manjimup Dried Apples, cena: $53,00, na stanie 20 szt.
```

8. Kwerendy LINQ możemy tagować, np. dla potrzeb rejestrowania logów. Wprowadzone tagi umieszczane są w kodzie SQL jako komentarz. Jednocześnie wprowadzanie tagów bezpośrednio do kwerend LINQ zwiększa ich czytelność (<https://docs.microsoft.com/en-us/ef/core/querying/tags>).

Modyfikujemy kod metody `ProductsWithHighestPriceOrdered`, dopisując do zapytania LINQ sformułowanego w *Fluid API* jeszcze jedną instrukcję:

```
static void ProductsWithHighestPriceOrdered()
{
    ...
    var products = db.Products
        .TagWith("Produkty o cenie przekraczającej zadaną, w kolejności od najdroższego do najtańszego.")
        .Where(product => product.Cost >= price)
        .OrderByDescending(product => product.Cost);
    ...
}
```

W logach pojawi się wpis, jak poniżej:

```
-- Produkty o cenie przekraczającej zadaną, w kolejności od najdroższego do najtańszego.
```

```
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice", "p"."Discontinued", "p"."ProductName", "p"."Units
FROM "Products" AS "p"
WHERE "p"."UnitPrice" >= @_price_0
```

```
WHERE p . UnitPrice >= @_price_0
ORDER BY "p"."UnitPrice" DESC
id: 38, Produkt: Côte de Blaye, cena: $263,50, na stanie 17 szt.
```

9. EF Core wspiera instrukcje SQL wykorzystujące porównywanie do wzorca (użycie SQL-owego operatora `LIKE`). Napiszemy zapytanie do bazy, wyszukujące produkty, których nazwa pasować będzie do wprowadzonego wzorca:

```
static void ProductsWithName()
{
    using var db = new Northwind();
    db.GetService<ILoggerFactory>().AddProvider(new EFConsoleLoggerProvider());

    Write("Podaj nazwę produktu (lub jej część): ");
    string productName = ReadLine();
    IQueryables products =
        db.Products
            .TagWith("Produkty o zadanej nazwie (LIKE)")
            .Where( prod => EF.Functions.Like(prod.ProductName, $"'{productName}'"));

    foreach (var item in products)
    {
        WriteLine($"id: {item.ProductID}, Produkt: {item.ProductName}, cena: {item.Cost:$#,##0.00}, na stanie: {item.UnitsInStock}");
    }
}
```

```
Podaj nazwę produktu (lub jej część): Mix
13.01.2021 23:45:46 -> Level: Debug, Event Id: Microsoft.EntityFrameworkCore.Database.CommandExecuting, State: Executing
-- Produkty o zadanej nazwie (LIKE)

SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice", "p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE "p"."ProductName" LIKE @_Format_1
id: 5, Produkt: Chef Anton's Gumbo Mix, cena: $21,35, na stanie 0 szt., wycofany: True
id: 52, Produkt: Filo Mix, cena: $7,00, na stanie 38 szt., wycofany: False
```

W EF Core twórcy dostarczyli statyczną klasę `EF` zawierającą jedną właściwość `Functions` zwracającą obiekt `DbFunctions` oraz kilkadziesiąt przeciążonych wariantów metody `CompileQuery`. Obiekt `DbFunctions` zawiera mapowania funkcji bazodanowych EF na ich odpowiedniki SQL. Wywołanie `EF.Functions.Like` formalnie wywołuje tę z `DbFunctions`. Pierwszym parametrem jest `string` który ma zostać dopasowany do wzorca podanego jako drugi parametr (z użyciem właściwych *wildcards* dla danego systemu bazodanowego).

10. W poprzednich punktach pokazane zostały techniki filtrowania wyników kwerend (`Where` oraz z `LIKE`). Jeśli, projektując aplikację, oczekiwaliśmy globalnego filtrowania danych - np. w naszym przykładzie pomijania produktów wycofanych - należałoby to raczej zrobić na poziomie globalnym, już na poziomie modelu. Wystarczy zmodyfikować odpowiednio metodę `OnModelCreating` w klasie opisującej bazę danych (`context`) `Northwind`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // przykład użycia Fluent API zamiast atrybutów (adnotacji)
    // do ograniczenia długości nazwy kategorii do 15 znaków
    modelBuilder.Entity<Category>()
        .Property(category => category.CategoryName)
        .IsRequired() // ponieważ NOT NULL
        .HasMaxLength(15);

    // dodane, aby skorygować działania na decima w SQLite
    modelBuilder.Entity<Product>()
        .Property(product => product.Cost)
        .HasConversion<double>();

    // filtr globalny, biorący pod uwagę produkty, które nie są wycofane
    modelBuilder.Entity<Product>()
        .HasQueryFilter(prod => !prod.Discontinued);
}
```

Teraz żaden produkt, oznaczony w bazie jako `Discontinued==true` nie zostanie wzięty pod uwagę w aplikacji - wszystkie budowane zapytania odwołujące się do `Product` będą miały dopisany warunek

```
...
WHERE NOT (Product.Discontinued)
```

...

Strategie pobierania danych z bazy danych

EF Core przewiduje następujące strategie ładowania danych z bazy:

- zachłanne/gorliwe (*eager loading*) - zapytanie dotyczące jednej encji ładuje również powiązane encje jako swoją część, dzięki czemu nie musimy wykonywać oddzielnego zapytania dla powiązanych encji,
- leniwe (*lazy loading*) - leniwe ładowanie opóźnia ładowanie powiązanych encji, dopóki tego nie zażądasz. Jest przeciwieństwem gorliwego (zachłanego) ładowania,
- jawne (*explicit loading*) - jawnie określamy które encje, w jakiej kolejności i do jakie postaci powinny zostać załadowane z bazy danych.

Ładowanie zachłanne (*eager loading*)

Zachłanne ładowanie wymuszamy na poziomie kwerendy LINQ używając operatora/metody `Include()` (lub kontynuując `ThenInclude()`, jeśli łączymy więcej encji).

Przykład użycia pojawił się w metodzie `CategoriesAndProducts`:

```
static void CategoriesAndProducts()
{
    ...
    IQueryable<Category> categories = db.Categories
        .Include(c => c.Products);
    ...
}
```

Efektem komplikacji tego zapytania było wygenerowanie kodu SQL:

```
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description", "t"."ProductID", "t"."CategoryID", "t"."UnitPrice"
FROM "Categories" AS "c"
LEFT JOIN (
    SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice", "p"."Discontinued", "p"."ProductName", "p"."Units
    FROM "Products" AS "p"
    WHERE NOT ("p"."Discontinued")
) AS "t" ON "c"."CategoryID" = "t"."CategoryID"
ORDER BY "c"."CategoryID", "t"."ProductID"
```

Zapytanie zawiera podzapytanie (po klauzuli `JOIN` jeszcze jeden `SELECT`). Wszystkie dane jednym zapytaniem zostaną pobrane z bazy.

Jeśli usuneliśmy operator `Include` z zapytania:

```
IQueryable<Category> categories = db.Categories; // .Include(c => c.Products);
```

wypiszymy **tylko** listę kategorii, z zerową liczbą produktów:

```
Lista kategorii i liczba produktów w kategorii.
14.01.2021 00:27:33 -> Level: Debug, Event Id: Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State: Executing
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description"
FROM "Categories" AS "c"
01. Beverages: 0
02. Condiments: 0
03. Confections: 0
04. Dairy Products: 0
05. Grains/Cereals: 0
06. Meat/Poultry: 0
07. Produce: 0
08. Seafood: 0
```

Widzimy, w wygenerowanej instrukcji SQL, brak powiązania z tabelą `Products`.

Leniwe ładowanie (*lazy loading*)

Leniwe ładowanie w EF Core wymaga referencji do pakietu NuGet `Microsoft.EntityFrameworkCore.Proxies`, a następnie skonfigurowania proxy.

1. W pliku `Northwind.cs` dopisujemy na początku dyrektywę użycia przestrzeni nazw `using Microsoft.EntityFrameworkCore.Proxies;`. Pojawi się błąd braku takiej przestrzeni w EF Core.
2. Dopuszczamy w pliku `.csproj` konfiguracji projektu referencję do `Microsoft.EntityFrameworkCore.Proxies`:

```
<PackageReference  
    Include="Microsoft.EntityFrameworkCore.Proxies"  
    Version="5.0.1" />
```

Konieczne jest przeładowanie projektu (`dotnet restore`) oraz zbudowanie go od nowa (`dotnet build`) aby wymagana biblioteka została zaciągnięta (u mnie było konieczne zamknięcie VS Code i ponowne otworzenie).

3. Modyfikujemy metodę `OnConfiguring` w klasie `Northwind`, dopisując przed `.UseSqlite($"Filename={path}")` metodę `.UseLazyLoadingProxies()`:

```
optionsBuilder  
    .UseLazyLoadingProxies()  
    .UseSqlite($"Filename={path}");
```

Teraz za każdym razem, gdy pętla się wylicza i podejmowana jest próba odczytania właściwości `Products`, leniwe ładowanie proxy sprawdzi, czy dane są załadowane. Jeśli nie, załaduje je dla nas „leniwie”, wykonując instrukcję `SELECT`, aby załadować tylko ten zestaw produktów, który pasuje dla bieżącej kategorii, a następnie na wyjście zostanie zwrocona wyznaczona liczba.

4. Uruchamiamy aplikację, aby zobaczyć działanie:

```
Lista kategorii i liczba produktów w kategorii.  
14.01.2021 01:16:57 -> Level: Debug, Event Id: Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State: Executing  
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description"  
FROM "Categories" AS "c"  
14.01.2021 01:16:58 -> Level: Debug, Event Id: Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State: Executing  
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice", "p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"  
FROM "Products" AS "p"  
WHERE NOT ("p"."Discontinued") AND ("p"."CategoryID" = @_p_0)  
01. Beverages: 11  
  
...  
  
14.01.2021 01:16:59 -> Level: Debug, Event Id: Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting, State: Executing  
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice", "p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"  
FROM "Products" AS "p"  
WHERE NOT ("p"."Discontinued") AND ("p"."CategoryID" = @_p_0)  
08. Seafood: 12
```

Widzimy, że wykonywanych jest wiele zapytań (sparametryzowanych bieżącym produktem).

Jawne ładowanie (*explicit loading*)

Ładowanie jawne encji jest podobne do leniwego, z tym, że mamy kontrolę nad tym procesem - jak i dokąd dane zostaną załadowane.

Nawet jeśli ładowanie leniwe jest wyłączone, możemy je jawnie wywołać operatorem LINQ `Load()`.

Napiszemy nową metodę `CategoriesAndProductsEagerOrExplicitLoading` demonstrującą różne sposoby ładowania danych z bazy. W zależności od dokonanego wyboru (1 - zachłanne (eager), 2 - leniwe (lazy) lub jakikolwiek inny klawisz - wtedy żadne z podanych) uruchomiony zostanie odpowiedni fragment kodu. W przypadku *explicit loading* w pętli przeglądającej kolejno kategorie ładowana jest tabela z produktami (o ile nie została załadowana wcześniej). Tabela ta ładowana jest do pamięci w formie kolekcji (par: kategoria, pasujący produkt).

```
static void CategoriesAndProductsEagerOrExplicitLoading()  
{  
    using var db = new Northwind(); //C#8  
    db.GetService<ILoggerFactory>().AddProvider(new EFConsoleLoggerProvider());  
  
    WriteLine("Lista kategorii i liczba produktów w kategorii (explicit loading).");  
    // kwerenda pobiera wszystkie kategorie i powiązane z nimi produkty  
    IQueryables<Category> categories = db.Categories;  
    //.Include(c => c.Products);  
  
    db.ChangeTracker.LazyLoadingEnabled = false;  
    bool explicitLoading = false;  
  
    Write("Wybierz metodę ładowania: 1 - zachłanne (eager), 2 - leniwe (lazy): ");
```

```

var key = ReadKey().Key;
WriteLine();

switch(key)
{
    case System.ConsoleKey.D1 : //eager loading
        explicitLoading = false;
        categories = categories.Include(c => c.Products);
        WriteLine("* włączone ładowanie zachłanne");
        break;
    case System.ConsoleKey.D2 : //explicit loading
        explicitLoading = true;
        WriteLine("* włączone ładowanie jawnie");
        break;
    default:
        WriteLine("** nie wybrano ani zachłannego ani jawnego ładowania");
        break;
}

int i = 1;
foreach (var c in categories)
{
    if(explicitLoading)
    {
        var products = db.Entry(c).Collection(c1 => c1.Products);
        if (!products.IsLoaded) products.Load();
    }
    WriteLine($"{i:D2}. {c.CategoryName}: {c.Products.Count}");
    i++;
}

```

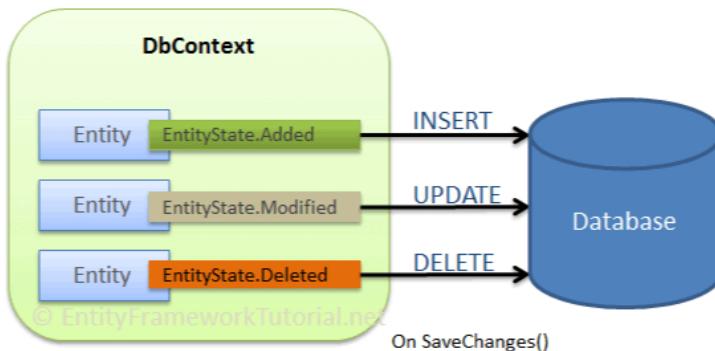
Modyfikowanie danych w bazie (CUD - create, update, delete)

Klasa `DbContext` (z której dziedziczy `Northwind`) obsługuje automatycznie śledzenie modyfikacji danych na poziomie aplikacji (*change tracking*). Aby przesłać zmiany w encjach do bazy, wystarczy wywołać metodę `SaveChanges()` - zwróci ona liczbę pomyślnie przyjętych zmian encji. Klasa `DbChangeTracker` przechowuje wszystkie informacje o śledzonych encjach.

W EF Core mamy dwa scenariusze trwałego zapisania danych w bazie:

- *connected* - ta sama instancja klasy dziedziczącej z `DbContext` jest wykorzystywana do pobierania i zapisywania danych w bazie. Kontekst kontroluje modyfikacje wszystkich encji. Operacje wykonywane są stosunkowo szybko. Baza cały czas pozostaje otwarta.
- *disconnected* - dwie różne instancje klasy dziedziczącej z `DbContext` są wykorzystywane do operowania na bazie danych - jedna do pobierania, druga do zapisywania. Ten scenariusz wykorzystuje mniej zasobów w porównaniu do poprzedniego, operacje wykonywane są wolniej. Baza pozostaje otwarta tylko na czas efektywnego wykonania działania. Model ten częściej wykorzystywany w aplikacjach webowych lub aplikacjach z odległą/zdalną bazą danych (por. [DbContext pooling](#)).

W ćwiczeniu realizujemy scenariusz pierwszy (*connected*). W tym scenariuszu `DbContext` śledzi wszelkie zmiany encji i automatycznie ustawia dla każdej z nich `EntityState` w zależności od wykonanej operacji.



Insert

W klasie `Program` utworzymy pomocniczą metodę `ListProducts` wypisującą wszystkie produkty w kolejności malejącej

ze względu na cene:

```
static void ListProducts()
{
    using var db = new Northwind();

    WriteLine("{0,-2} {1,-33} {2,8} {3,5} {4}",
        "ID", "Produkt", "Cena", "Szt.", "Wycofany");
    foreach (var p in db.Products.OrderByDescending(p1 => p1.Cost))
    {
        WriteLine("{0:00} {1,-33} {2,8:$#,##0.00} {3,5} {4}",
            p.ProductID, p.ProductName, p.Cost,
            p.Stock, p.Discontinued);
    }
}
```

Uruchomienie metody w `Main()`:

ID Produkt	Cena	Szt.	Wycofany
38 Côte de Blaye	\$263,50	17	False
20 Sir Rodney's Marmalade	\$81,00	40	False
18 Carnarvon Tigers	\$62,50	42	False
...			

W klasie `Program` utworzymy metodę `AddProduct` dodającą pojedynczy produkt do bazy (do tabeli `Products`):

```
static bool AddProduct(int categoryId, string productName, decimal? price)
{
    using var db = new Northwind();
    db.GetService<ILoggerFactory>().AddProvider(new EFConsoleLoggerProvider());

    var新产品 = new Product
    {
        CategoryID = categoryId,
        ProductName = productName,
        Cost = price
    };

    // oznacz produkt jako dodany w change tracking
    db.Products.Add(新产品);

    // zapisz śledzone zmiany w bazie
    int result = db.SaveChanges();
    return (result == 1);
}
```

Teraz, w metodzie `Main` możemy sprawdzić poprawność wykonania operacji dodania nowego produktu:

```
static void Main(string[] args)
{
    //CategoriesAndProducts();
    //CategoriesAndProductsWithMinimumInStock();
    //ProductsWithHighestPriceOrdered();
    //WriteLine(CategoriesAndProductsWithMinimumInStockExportToQueryString(stock: 50));
    //ProductsWithName();
    //CategoriesAndProductsEagerOrExplicitLoading();
    if( AddProduct(categoryID: 6, productName: "Tatar wołowy", price: 300M) )
        WriteLine("Produkt pomyslnie dopisany do bazy");
    ListProducts();
}
```

```
14.01.2021 01:26:56 -> Level: Debug, Event Id: Microsoft.EntityFrameworkCore.Database.CommandExecuting, State: Executing DbCommand
INSERT INTO "Products" ("CategoryID", "UnitPrice", "Discontinued", "ProductName", "UnitsInStock")
VALUES (@p0, @p1, @p2, @p3, @p4);
SELECT "ProductID"
FROM "Products"
WHERE changes() = 1 AND "rowid" = last_insert_rowid();
Produkt pomyslnie dopisany do bazy
ID Produkt              Cena Szt. Wycofany
78 Tatar wołowy          $300,00      False
38 Côte de Blaye         $263,50     17 False
20 Sir Rodney's Marmalade $81,00      40 False
```

Update

Chcemy zaktualizować istniejący wpis w bazie - np. zwiększyć o zadaną wartość liczbę sztuk na stanie. Dopusujemy metodę w klasie `Program`:

```
static bool IncreaseProductStock(int productId, short stockToIncrease)
{
    using var db = new Northwind();

    // pobieramy produkt o wskazanym id (będzie tylko jeden, bo ProductID jest kluczem)
    Product productToUpdate = db.Products
        .FirstOrDefault(p => p.ProductID == productId);
    if(productToUpdate is null) return false;

    if(productToUpdate.Stock is null)
        productToUpdate.Stock = 0;
    productToUpdate.Stock += stockToIncrease;

    // zapisz śledzone zmiany w bazie
    int result = db.SaveChanges();
    return (result == 1);
}
```

i wywołujemy w `Main`:

```
if (IncreaseProductStock(78, 5))
    Writeline("Aktualizacja stanu produktu przebiegła pomyślnie.");
ListProducts();
```

```
Aktualizacja stanu produktu przebiegła pomyślnie.
ID Produkt      Cena Szt. Wycofany
78 Tatar wołowy   $300,00    5 False
38 Côte de Blaye   $263,50   17 False
20 Sir Rodney's Marmalade   $81,00   40 False
...
```

W przypadku podania błędного `ProductId` aktualizacja nie wykona się (również nie zostanie zgłoszony wyjątek).

Delete

Załóżmy, że chcemy usunąć z bazy wszystkie produkty o nazwie rozpoczynającej się od określonego tekstu (np. "Tatar"). Do klasy `Program` dopisujemy stosowną metodę (tym razem zwracamy liczbę usuniętych wierszy):

```
static int DeleteProductsWithNameStartingWith(string name)
{
    using var db = new Northwind();

    Ienumerable<Product> productsToDelete = db.Products
        .TagWith("Wyliczenie produktów zaczynających się od name")
        .Where(p => p.ProductName.StartsWith(name));
    db.Products.RemoveRange(productsToDelete);

    int result = db.SaveChanges();
    return result;
}
```

Wywołujemy ją w `Main`:

```
int noOfDeletedRecords = DeleteProductsWithNameStartingWith("Tatar");
WriteLine($"{noOfDeletedRecords} produkt(y) usunięto.");
ListProducts();
```

```
1 produkt(y) usunięto.
ID Produkt      Cena Szt. Wycofany
38 Côte de Blaye   $263,50   17 False
20 Sir Rodney's Marmalade   $81,00   40 False
```

Ponowne uruchomienie programu zaraportuje

```
0 produkt(y) usunięto.
```

`productsToDelete` jest typu `IEnumerable<Product>` i w najgorszej sytuacji może być `empty` (nie `null`). Ponieważ metoda `RemoveRange` nie zgłasza wyjątków, również metoda `DeleteProductsWithNameStartingWith` ich nie zgłosi.

Transakcje

Wywołując `SaveChanges()` po stronie bazy uruchamiana jest domyślna transakcja (*implicit transaction*). Jeśli coś zawiedzie w momencie zapisu, zmiany automatycznie zostaną wycofane (*roll back*). Jeśli zmiany zostaną przyjęte, transakcja zostanie zatwierdzona (*commit*). Transakcje pozwalają zachować integralność bazy (np. blokują odczyty i zapisy w trakcie wykonywania `SaveChanges`).

W EF Core transakcje możemy kontrolować, wywołując je i zatwierdzając jawnie. Transakcje są obiektami klas implementujących `IDbContextTransaction`, [Są `IDisposable`](#).

Przykład:

Modyfikujemy operację zbiorowego usuwania produktów `DeleteProductsWithNameStartingWith` (może dugo trwać) otaczając ją transakcją. Po pierwsze, przyłączamy przestrzeń nazw `using Microsoft.EntityFrameworkCore.Storage;`. Następnie modyfikujemy kod:

```
static int TrasactDeleteProductsWithNameStartingWith(string name)
{
    using var db = new Northwind();

    ⑤     using( IDbContextTransaction t = db.Database.BeginTransaction() )
    {
        IEnumarable<Product> productsToDelete = db.Products
            .TagWith("Wyliczenie produktów zaczynających się od name")
            .Where(p => p.ProductName.StartsWith(name));
        db.Products.RemoveRange(productsToDelete);

        ⑬     int result = db.SaveChanges();
        t.Commit();
        return result;
    }
}
```

Referencje

- [Entity Framework Core](#) - oficjalne źródłowe materiały z docs.Microsoft
- [Logging in C# .NET Modern-day Practices: The Complete Guide](#)
- [Entity Framework Tutorial](#) - zwarty, z wieloma przykładami tutorial do EF 6 oraz EF Core. Uzupełnieniem tutoriala jest prosty projekt <https://github.com/entityframeworktutorial/EF6-DBFirst-Demo>.

Zadania

Zadanie 1.

Napisz metodę `ListCategories()` wypisującą wszystkie kategorie w formacie: najpierw `Id` kategorii, potem nazwa kategorii, sortując według nazwy alfabetycznie

Zadanie 2.

Spróbuj dopisać - w ramach jednej transakcji - nową kategorię oraz dodać do niej dwa nowe produkty. Dane wymyśl.

Zadanie 3.

Wzorując się na opisie, dodaj do modelu (ręcznie) encję `Supplier` mapującą tabelę `Suppliers`. Utwórz kwerendę wypisującą produkty oraz ich dostawców.

