

# Jakub Barber

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
    public class Product
    {
        public int ProductID { get; set; }
        public string ProductName { get; set; }
        public int UnitsOnStock { get; set; }
    }
}
```

Następnie stworzyłem nową klasę, pobrałem wymagany pakiet i dodałem odpowiednie usingi

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
    public class ProductContext:DbContext
    {
        public DbSet<Product> Products { get; set; }
    }
}
```

W celu przeprowadzenia migracji doinstalowałem odpowiednie moduły i wykonałem pierwszą migrację (zmodyfikowana klasa contextowa niżej ):

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
```

```

public class ProductContext:DbContext
{
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        base.OnConfiguring(optionsBuilder);
        optionsBuilder.UseSqlite("DataSource=ProductsDatabase");
    }
}

```

### Wywołanie migracji:

```

PS C:\Users\jakub\Desktop\studia sem 4\bazki\ENTITY_test\laby\JakubBarberEFProducts> dotnet ef migrations add InitProductDatabase
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
PS C:\Users\jakub\Desktop\studia sem 4\bazki\ENTITY_test\laby\JakubBarberEFProducts>

```

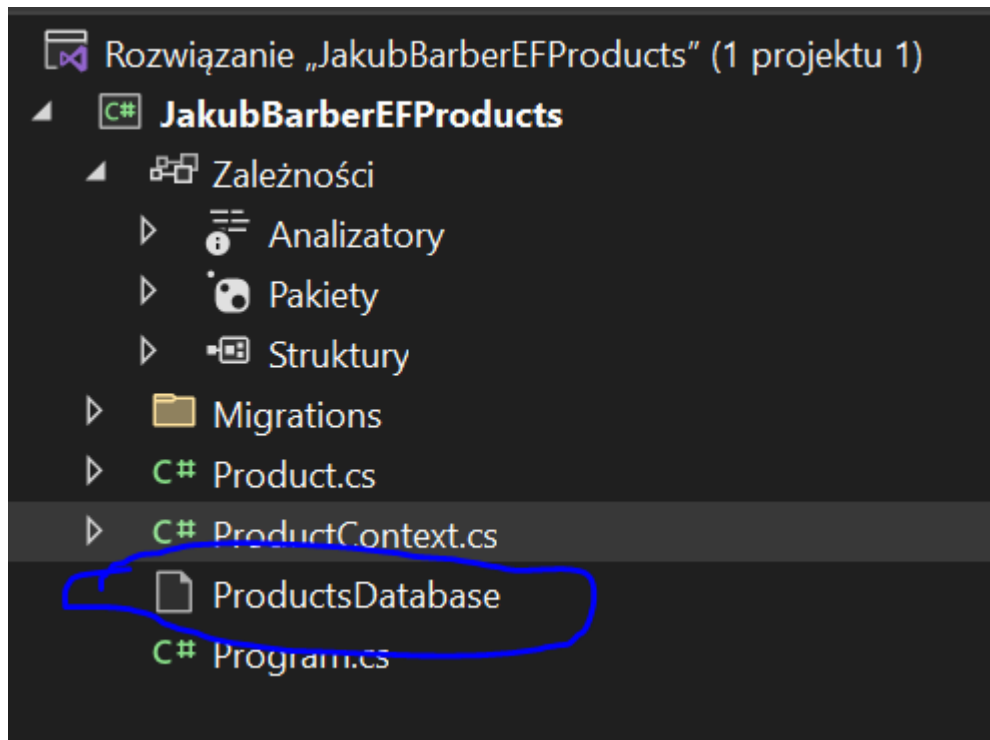
### Wywołanie update database:

```

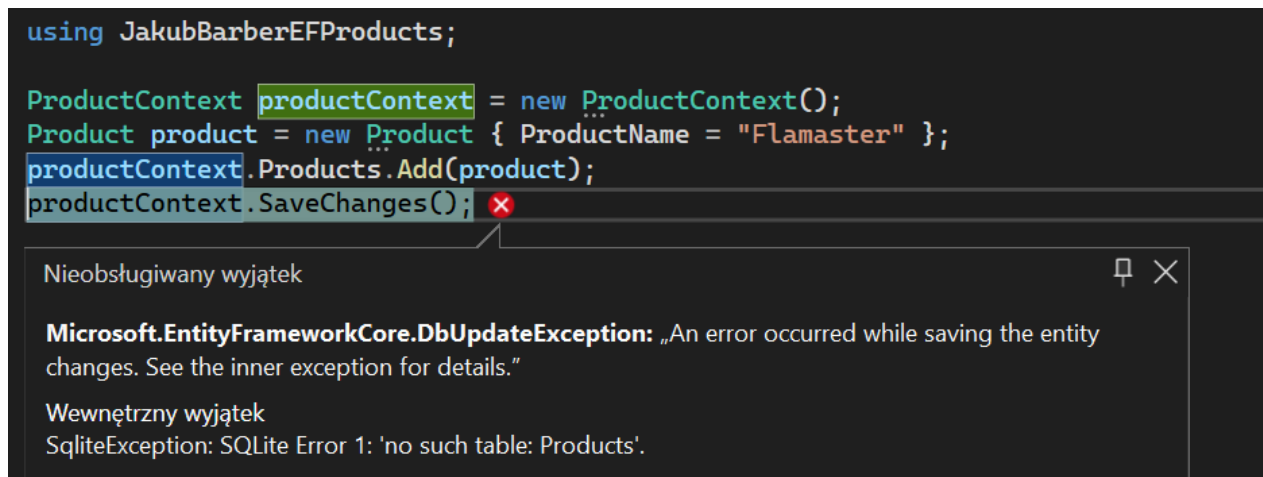
PS C:\Users\jakub\Desktop\studia sem 4\bazki\ENTITY_test\laby\JakubBarberEFProducts> dotnet ef database update
Build started...
Build succeeded.
Applying migration '20230329112338_InitProductDatabase'.
Done.
PS C:\Users\jakub\Desktop\studia sem 4\bazki\ENTITY_test\laby\JakubBarberEFProducts>

```

### Efekt oczekiwany:



Po uruchomieniu zmodyfikowanego kodu w pliku Program.cs jak przewidywaliśmy, otrzymujemy wyjątek:



Faktycznie w bazie sqlite mamy dane struktury

```
Enter ".help" for usage hints.
sqlite> .tables
Products                __EFMigrationsHistory
sqlite>
```

Po dodaniu kodu, wynik faktycznie działa, co ciekawe nie musiałem dodawać System.Linq

```
// See https://aka.ms/new-console-template for more information
//Console.WriteLine("Hello, World!");
```

```
using JakubBarberEFProducts;
//using System.Linq
```

```
ProductContext productContext = new ProductContext();
Product product = new Product { ProductName = "Flamaster" };
productContext.Products.Add(product);
productContext.SaveChanges();
```

```
var query = from prod in productContext.Products
            select prod.ProductName;
```

```
foreach (var pName in query)
{
    Console.WriteLine(pName);
}
```

Zmieniony kod w pliku Program.cs

```
// See https://aka.ms/new-console-template for more information
//Console.WriteLine("Hello, World!");
```

```
using JakubBarberEFProducts;
//using System.Linq
```

```
Console.WriteLine("Podaj nazwę produktu: ");
string Prodname = Console.ReadLine();
Console.WriteLine("Ponizej lista produktow w naszej bazie : ");
```

```
ProductContext productContext = new ProductContext();
Product product = new Product { ProductName = Prodname };
productContext.Products.Add(product);
productContext.SaveChanges();
```

```
var query = from prod in productContext.Products
            select prod.ProductName;
```

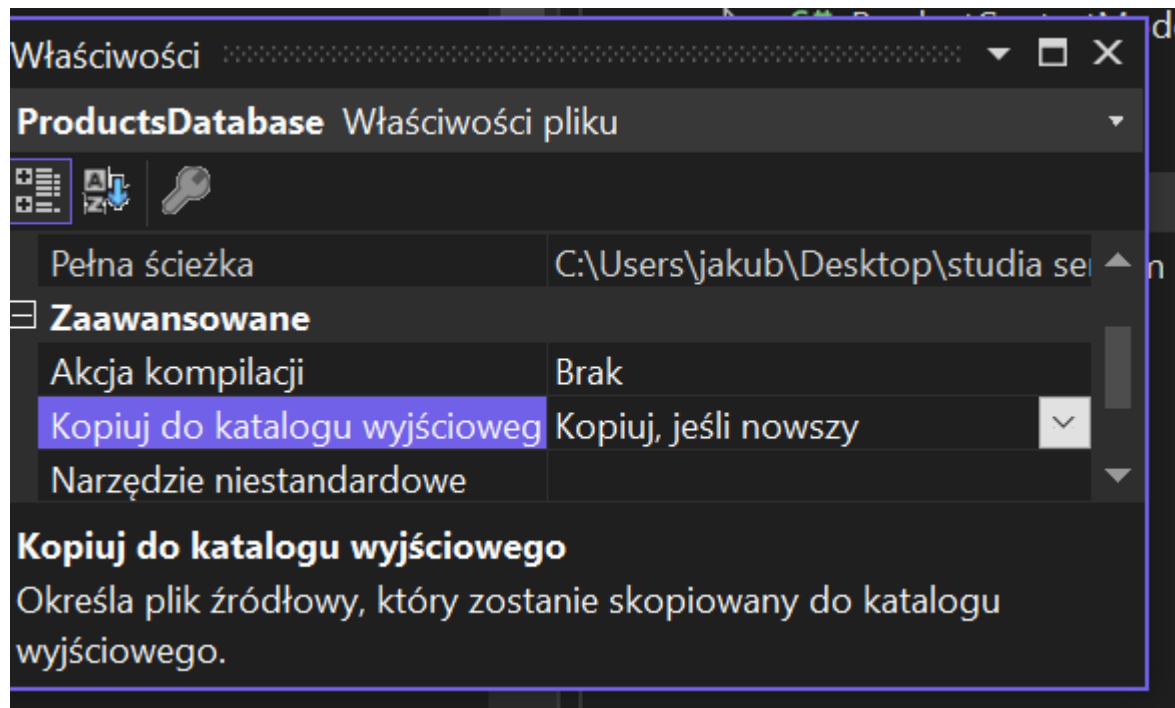
```
foreach (var pName in query)
{
    Console.WriteLine(pName);
}
```

Faktycznie dodało ( odpaliłem 2 razy )

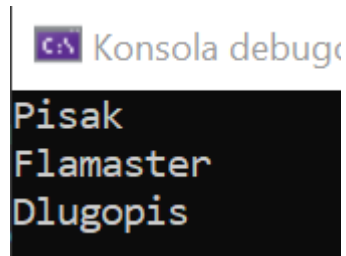
```
Flamaster
Flamaster

C:\Users\jakub\Desktop\studia sem 4\bazki\ENTITY_test\laby\JakubBarberEFProducts\bin\Debug\net6.0\JakubBarberEFProducts.exe (proces 16408) zakończono z kodem 0.
Aby automatycznie zamknąć konsolę po zatrzymaniu debugowania, włącz opcję Narzędzia -> Opcje -> Debugowanie -> Automatycznie zamknij konsolę po zatrzymaniu debugowania.
Naciśnij dowolny klawisz, aby zamknąć to okno...
```

Zgodnie z zaleceniami zmieniam we właściwościach pliku bazodanowego opcje nadpisywania:



Wynik faktycznie działa:



```
C:\> Konsola debug  
Pisak  
Flamaster  
Długopis
```

# Koniec pracy z przewodnikiem:

# Praca indywidulana:

## ZADANIE I Relacja MANY to ONE:

W zadaniu, w celu realizacji relacji jednokierunkowej n:1, najpierw utworzyłem nową klasę Supplier:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace JakubBarberEFProducts  
{  
    public class Supplier  
    {  
        public int SupplierID { get; set; }  
        public String CompanyName { get; set; }  
        public String Street { get; set; }  
        public String City { get; set; }  
    }  
}
```

Następnie zmodyfikowałem klasę Product ( dodałem pole będące referencją na odpowiedniego dostawce i na jego podstawie został utworzony klucz obcy wskazujący na SupplierID)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
    public class Product
    {
        public int ProductID { get; set; }
        public string ProductName { get; set; }
        public int UnitsOnStock { get; set; }

        public Supplier Supplier { get; set; }
    }
}

```

Następnie w klasie ProductContext dodałem kolekcję Suppliers:

```

using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
    public class ProductContext:DbContext
    {
        public DbSet<Product> Products { get; set; }
        public DbSet<Supplier> Suppliers { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            base.OnConfiguring(optionsBuilder);
            optionsBuilder.UseSqlite("DataSource=ProductsDatabase");
        }
    }
}

```

Po wykonaniu migracji i zupdateowaniu bazy danych odpowiednimi poleceniami, schemat bazy danych prezentuje się następująco:

```

sqlite> .schema Products
CREATE TABLE IF NOT EXISTS "Products" (
    "ProductID" INTEGER NOT NULL CONSTRAINT "PK_Products" PRIMARY KEY AUTOINCREMENT,
    "ProductName" TEXT NOT NULL,
    "SupplierID" INTEGER NOT NULL,
    "UnitsOnStock" INTEGER NOT NULL,
    CONSTRAINT "FK_Products_Suppliers_SupplierID" FOREIGN KEY ("SupplierID")
REFERENCES "Suppliers" ("SupplierID") ON DELETE CASCADE
);

```

```
CREATE INDEX "IX_Products_SupplierID" ON "Products" ("SupplierID");
sqlite> .schema Suppliers
CREATE TABLE IF NOT EXISTS "Suppliers" (
    "SupplierID" INTEGER NOT NULL CONSTRAINT "PK_Suppliers" PRIMARY KEY
    AUTOINCREMENT,
    "CompanyName" TEXT NOT NULL,
    "Street" TEXT NOT NULL,
    "City" TEXT NOT NULL
);
sqlite> ;
```

Widzimy, że do tabeli Products został dodany klucz obcy wskazujący na rekordy w tabeli Suppliers po SupplierID.

W celu przetestowania działania metod wykonuje kod z programu Main, gdzie tworzę nowy produkt tusz i nowego suppliera „Papierniczy PL” oraz zapisuję referencję na nowego dostawcę w nowym Produkcie.

```
using JakubBarberEFProducts;

ProductContext productContext = new ProductContext();
Product product = new Product { ProductName="tusz"};
Supplier supplier = new Supplier
{
    CompanyName = "Papierniczy PL",
    Street = "Wadowicka 123",
    City = "Krakow"
};
product.Supplier = supplier;
productContext.Suppliers.Add(supplier);
productContext.Products.Add(product);
productContext.SaveChanges();


var queryProd = from prod in productContext.Products
                select prod.ProductName;

var querySupp = from comp in productContext.Suppliers
                select comp.CompanyName;

Console.WriteLine("Ponizej lista produktow w naszej bazie : ");
foreach (var pName in queryProd)
{
    Console.WriteLine(pName);
}

Console.WriteLine("Ponizej lista dostawcow w naszej bazie : ");
foreach (var cName in querySupp)
{
    Console.WriteLine(cName);
}
```



 Konsola debugowania programu Microsoft Visual Studio

```
Ponizej lista produktow w naszej bazie :  
tusz  
Ponizej lista dostawcow w naszej bazie :  
Papierniczy PL
```

Wynik działania programu jest zgodny z oczekiwaniami, do bazy został dodany nowy produkt, a jego dostawca został ustawiony jako nowo dodany do bazy dostawca. Dla pewności wykonamy parę zapytań w terminalowym sqlite3:

```
sqlite> SELECT * FROM Products;  
8|tusz|3|0  
sqlite> SELECT * FROM Suppliers;  
3|Papierniczy PL|Wadowicka 123|Krakow  
sqlite> SELECT ProductName, CompanyName FROM Products INNER JOIN Suppliers on  
Products.SupplierID=Suppliers.SupplierID;  
tusz|Papierniczy PL  
sqlite>
```

Widzimy, że wyniki są zgodne z oczekiwaniami i wszystko działa jak należy.

## ZADANIE II

W zadaniu, w celu realizacji relacji jednokierunkowej 1:n, najpierw zmodyfikowałem klasę Products usuwając referencję na Supplier, a następnie dodałem w klasie Supplier kolekcję przechowującą referencje na obiekty typu Product.

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace JakubBarberEFProducts  
{  
    public class Supplier  
    {  
        public int SupplierID { get; set; }  
        public String CompanyName { get; set; }  
        public String Street { get; set; }  
        public String City { get; set; }  
    }  
}
```

```

        public ICollection<Product>? Products { get; set; }
    }
}

```

Po wykonaniu migracji i zupdateowaniu bazy danych sprawdzamy strukturę naszej bazy danych przy pomocy terminalowego sqlite3.

```

sqlite> .schema Suppliers
CREATE TABLE IF NOT EXISTS "Suppliers" (
    "SupplierID" INTEGER NOT NULL CONSTRAINT "PK_Suppliers" PRIMARY KEY
    AUTOINCREMENT,
    "CompanyName" TEXT NOT NULL,
    "Street" TEXT NOT NULL,
    "City" TEXT NOT NULL
);
sqlite> .schema Products
CREATE TABLE IF NOT EXISTS "Products" (
    "ProductID" INTEGER NOT NULL CONSTRAINT "PK_Products" PRIMARY KEY AUTOINCREMENT,
    "ProductName" TEXT NOT NULL,
    "SupplierID" INTEGER NOT NULL,
    "UnitsOnStock" INTEGER NOT NULL,
    CONSTRAINT "FK_Products_Suppliers_SupplierID" FOREIGN KEY ("SupplierID")
    REFERENCES "Suppliers" ("SupplierID") ON DELETE CASCADE
);
CREATE INDEX "IX_Products_SupplierID" ON "Products" ("SupplierID");
sqlite>

```

Widzimy, że wygenerowany schemat jest dokładnie taki sam jak we wcześniejszym przykładzie.

Sprawdzamy działanie naszego nowego schematu wykonując poniższy kod w programie Main:

```

using JakubBarberEFProducts;
ProductContext productContext = new ProductContext();
Product product1 = new Product { ProductName="kotlet" };
Product product2 = new Product { ProductName="spaghetti" };
Product product3 = new Product { ProductName="ziemniaki" };
Product product4 = new Product { ProductName="ketchup" };
Supplier supplier = new Supplier
{
    CompanyName = "Papierniczy PL",
    Street = "Wadowicka 123",
    City = "Krakow"
};
supplier.Products = new HashSet<Product>
{
    product1,
    product2,
    product3,

```

```

        product4
    };
    productContext.Suppliers.Add(supplier);
    productContext.Products.Add(product1);
    productContext.Products.Add(product2);
    productContext.Products.Add(product3);
    productContext.Products.Add(product4);
    productContext.SaveChanges();

    var queryProd = from prod in productContext.Products
                    select prod.ProductName;

    var querySupp = from comp in productContext.Suppliers
                    select comp.CompanyName;

    Console.WriteLine("Ponizej lista produktow w naszej bazie : ");
    foreach (var pName in queryProd)
    {
        Console.WriteLine(pName);
    }

    Console.WriteLine("Ponizej lista dostawcow w naszej bazie : ");
    foreach (var cName in querySupp)
    {
        Console.WriteLine(cName);
    }

```

Wynik działania jest zgodny z oczekiwaniami

Teraz sprawdzając przy użyciu terminalowego sqlite3 mamy:

```

sqlite> SELECT * FROM Products
...> ;
13|kotlet|5|0
14|spaghetti|5|0
15|ziemniaki|5|0
16|ketchup|5|0
sqlite> SELECT * FROM Suppliers;
5|Spozywczy PL|Jedzenie 123|Torun
sqlite> SELECT CompanyName, ProductName FROM Products INNER JOIN Suppliers ON
Suppliers.SupplierID = Products.SupplierID;
Spozywczy PL|kotlet
Spozywczy PL|spaghetti
Spozywczy PL|ziemniaki
Spozywczy PL|ketchup
sqlite>

```

Wszystko działa zgodnie z oczekiwaniami, przechodzimy do 3 wariantu relacji „ONE to MANY”

## ZADANIE III

W zadaniu, w celu realizacji relacji dwukierunkowej 1:n-n:1, zmodyfikowałem klasę Supplier:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
    public class Supplier
    {
        public int SupplierID { get; set; }
        public String CompanyName { get; set; }
        public String Street { get; set; }
        public String City { get; set; }

        public ICollection<Product> Products { get; set; }
    }
}
```

Następnie zmodyfikowałem klasę Product ( dodałem pole będące referencją na odpowiedniego dostawce i na jego podstawie został utworzony klucz obcy wskazujący na SupplierID)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
    public class Product
    {
        public int ProductID { get; set; }
        public string ProductName { get; set; }
        public int UnitsOnStock { get; set; }

        public Supplier Supplier { get; set; }
    }
}
```

Po wykonaniu migracji i zupdateowaniu bazy danych odpowiednimi poleceniami, schemat bazy danych prezentuje się następująco:

```
sqlite> .tables
Products          Suppliers          __EFMigrationsHistory
```

```

sqlite> .schema Products
CREATE TABLE IF NOT EXISTS "Products" (
    "ProductID" INTEGER NOT NULL CONSTRAINT "PK_Products" PRIMARY KEY AUTOINCREMENT,
    "ProductName" TEXT NOT NULL,
    "SupplierID" INTEGER NOT NULL,
    "UnitsOnStock" INTEGER NOT NULL,
    CONSTRAINT "FK_Products_Suppliers_SupplierID" FOREIGN KEY ("SupplierID")
REFERENCES "Suppliers" ("SupplierID") ON DELETE CASCADE
);
CREATE INDEX "IX_Products_SupplierID" ON "Products" ("SupplierID");
sqlite> .schema Suppliers
CREATE TABLE IF NOT EXISTS "Suppliers" (
    "SupplierID" INTEGER NOT NULL CONSTRAINT "PK_Suppliers" PRIMARY KEY
AUTOINCREMENT,
    "CompanyName" TEXT NOT NULL,
    "Street" TEXT NOT NULL,
    "City" TEXT NOT NULL
);
sqlite>

```

Widzimy, że do tabeli Products został dodany klucz obcy wskazujący na rekordy w tabeli Suppliers po SupplierID.

W celu przetestowanie działania wykonałem poniższy kod, dodając jako nowe produkty 6 nazw producentów piw i Supliera o nazwie „Browar PL”:

```

// See https://aka.ms/new-console-template for more information
//Console.WriteLine("Hello, World!");

```

```

using JakubBarberEFProducts;
//using System.Linq

```

```

/*
Console.WriteLine("Podaj nazwę produktu: ");
string Prodname = Console.ReadLine();
*/

```

```

ProductContext productContext = new ProductContext();

```

```

Product product1 = new Product { ProductName = "Tyskie" };
Product product2 = new Product { ProductName = "Zywiec" };
Product product3 = new Product { ProductName = "Harnas" };
Product product4 = new Product { ProductName = "Zatecky" };
Product product5 = new Product { ProductName = "Heineken" };
Product product6 = new Product { ProductName = "Lech" };
Supplier supplier = new Supplier
{
    CompanyName = "Browar PL",
    Street = "Krakowska 123",
    City = "Warszawa"
}

```

```

};
product1.Supplier = supplier;
product2.Supplier = supplier;
product3.Supplier = supplier;
product4.Supplier = supplier;
product5.Supplier = supplier;
product6.Supplier = supplier;

productContext.Suppliers.Add(supplier);
productContext.Products.Add(product1);
productContext.Products.Add(product2);
productContext.Products.Add(product3);
productContext.Products.Add(product4);
productContext.Products.Add(product5);
productContext.Products.Add(product6);

//Console.WriteLine(supplier);
productContext.SaveChanges();

var queryProd = from prod in productContext.Products
                select prod.ProductName;

var querySupp = from comp in productContext.Suppliers
                select comp.CompanyName;

Console.WriteLine("Ponizej lista produktow w naszej bazie : ");
foreach (var pName in queryProd)
{
    Console.WriteLine(pName);
}

Console.WriteLine("Ponizej lista dostawcow w naszej bazie : ");
foreach (var cName in querySupp)
{
    Console.WriteLine(cName);
}

```

W wyniku wykonania powyższego programu, zgodnie z przewidywaniami dodałem do bazy danych podane produkty i suppliera oraz połączyłem dodawane produkty z nowym supplierem.

## Konsola debugowania programu Microsoft Visual Studio

Ponizej lista produktow w naszej bazie :

Tyskie

Zywiec

Harnas

Zatecky

Heineken

Lech

Ponizej lista dostawcow w naszej bazie :

Browar PL

Dla pewności sprawdzamy w konsoli podpinając się przy pomocy sqlite3 do naszej bazy w podkatalogu/bin/Debug/net6.0

```
Products          Suppliers          __EFMigrationsHistory
sqlite> SELECT * FROM Suppliers
...> ;
5|Spozywczy PL|Jedzenie 123|Torun
sqlite> SELECT * FROM Products
...> ;
1|Tyskie|1|0
2|Zywiec|1|0
3|Harnas|1|0
4|Zatecky|1|0
5|Heineken|1|0
6|Lech|1|0
sqlite> SELECT * FROM Suppliers;
1|Browar PL|Krakowska 123|Warszawa
sqlite>
```

## Wnioski:

We wszystkich przypadkach wygenerowany w schemacie bazy danych kod DDL był taki sam, mamy zatem 3 sposoby na zdefiniowanie relacji 1 do wielu. Z tego co poszukałem na intercenie najczęstszym podejściem jest podejście 3 ( z kolekcją w jednej klasie oraz referencją w drugiej). W przypadku zadania drugiego musiałem zmienić deklaracje kolekcji z nullish operatorem, ponieważ bez niego migracje nie

chciały się wykonać. Po wykonaniu każdej z migracji, rekordy obecne w bazie danych zostały wyczyszczone.

## ZADANIE IV

W celu zamodelowania relacji wiele do wiele, utworzyłem nową klasę Invoice z kolekcją obiektów typu Product oraz w klasie Product dodałem kolekcję obiektów typu Invoice.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
    public class Invoice
    {
        public int InvoiceID { get; set; }
        public int Quantity { get; set; }

        public ICollection<Product> Products { get; set; }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
    public class Product
    {
        public int ProductID { get; set; }
        public string ProductName { get; set; }
        public int UnitsOnStock { get; set; }

        public Supplier Supplier { get; set; }
        public ICollection<Invoice> Invoices { get; set; }
    }
}
```



Po wykonaniu migracji i updatcie bazy danych otrzymujemy następujący, tym razem inny schemat bazy danych.

```
sqlite> .schema Products
CREATE TABLE IF NOT EXISTS "Products" (
  "ProductID" INTEGER NOT NULL CONSTRAINT "PK_Products" PRIMARY KEY AUTOINCREMENT,
  "ProductName" TEXT NOT NULL,
  "SupplierID" INTEGER NOT NULL,
  "UnitsOnStock" INTEGER NOT NULL,
  CONSTRAINT "FK_Products_Suppliers_SupplierID" FOREIGN KEY ("SupplierID")
REFERENCES "Suppliers" ("SupplierID") ON DELETE CASCADE
);
CREATE INDEX "IX_Products_SupplierID" ON "Products" ("SupplierID");
sqlite> .schema Invoices
Invoices          InvoicesInvoiceID
sqlite> .schema Invoices
CREATE TABLE IF NOT EXISTS "Invoices" (
  "InvoiceID" INTEGER NOT NULL CONSTRAINT "PK_Invoices" PRIMARY KEY AUTOINCREMENT,
  "Quantity" INTEGER NOT NULL
);
sqlite> .schema Invoices
Invoices          InvoicesInvoiceID
sqlite> .schema InvoicesInvoiceID
sqlite> .schema Suppliers
CREATE TABLE IF NOT EXISTS "Suppliers" (
  "SupplierID" INTEGER NOT NULL CONSTRAINT "PK_Suppliers" PRIMARY KEY
AUTOINCREMENT,
  "CompanyName" TEXT NOT NULL,
  "Street" TEXT NOT NULL,
  "City" TEXT NOT NULL
);
sqlite> .tables
InvoiceProduct    Products          __EFMigrationsHistory
Invoices          Suppliers
sqlite> .schema InvoiceProduct
CREATE TABLE IF NOT EXISTS "InvoiceProduct" (
  "InvoicesInvoiceID" INTEGER NOT NULL,
  "ProductsProductID" INTEGER NOT NULL,
  CONSTRAINT "PK_InvoiceProduct" PRIMARY KEY ("InvoicesInvoiceID",
"ProductsProductID"),
  CONSTRAINT "FK_InvoiceProduct_Invoices_InvoicesInvoiceID" FOREIGN KEY
("InvoicesInvoiceID") REFERENCES "Invoices" ("InvoiceID") ON DELETE CASCADE,
  CONSTRAINT "FK_InvoiceProduct_Products_ProductsProductID" FOREIGN KEY
("ProductsProductID") REFERENCES "Products" ("ProductID") ON DELETE CASCADE
);
CREATE INDEX "IX_InvoiceProduct_ProductsProductID" ON "InvoiceProduct"
```

```
("ProductsProductID");  
sqlite>
```

Widzimy, że została utworzona nowa tabela InvoiceProduct, która realizuje nam relację wiele do wiele.

W celu przetestowania rozwiązania, w programie Main stworzę 3 produkty i 3 faktury, każdy produkt będzie powiązany z jednym Supplierem.

```
using JakubBarberEFProducts;  
ProductContext productContext = new ProductContext();  
  
Supplier supplier = new Supplier  
{  
    CompanyName = "Browar PL",  
    Street = "Krakowska 123",  
    City = "Warszawa"  
};  
  
Product product1 = new Product { ProductName = "Tyskie" };  
Product product2 = new Product { ProductName = "Zywiec" };  
Product product3 = new Product { ProductName = "Harnas" };  
  
product1.Supplier = supplier;  
product2.Supplier = supplier;  
product3.Supplier = supplier;  
  
supplier.Products.Add(product1);  
supplier.Products.Add(product2);  
supplier.Products.Add(product3);  
  
Invoice invoice1 = new Invoice { Quantity=10 };  
Invoice invoice2 = new Invoice { Quantity=7 };  
Invoice invoice3 = new Invoice { Quantity=4 };  
invoice1.Products.Add(product1);  
product1.Invoices.Add(invoice1);  
invoice1.Products.Add(product2);  
product2.Invoices.Add(invoice1);  
invoice1.Products.Add(product3);  
product3.Invoices.Add(invoice1);  
invoice2.Products.Add(product1);  
product1.Invoices.Add(invoice2);  
invoice2.Products.Add(product3);  
product3.Invoices.Add(invoice2);  
invoice3.Products.Add(product3);  
product3.Invoices.Add(invoice3);  
productContext.Suppliers.Add(supplier);  
productContext.Products.Add(product1);  
productContext.Products.Add(product2);  
productContext.Products.Add(product3);  
productContext.Invoices.Add(invoice2);  
productContext.Invoices.Add(invoice1);  
productContext.Invoices.Add(invoice3);
```

```

productContext.SaveChanges();

var queryProd = from prod in productContext.Products
                select prod.ProductName;

var queryInv = from inv in productContext.Invoices
                select inv.InvoiceID;

var querySupp = from comp in productContext.Suppliers
                select comp.CompanyName;


Console.WriteLine("Ponizej lista produktow w naszej bazie : ");
foreach (var pName in queryProd)
{
    Console.WriteLine(pName);
}

Console.WriteLine("Ponizej lista dostawcow w naszej bazie : ");
foreach (var cName in querySupp)
{
    Console.WriteLine(cName);
}

Console.WriteLine("Ponizej lista faktur w naszej bazie : ");
foreach (var Inum in queryInv)
{
    Console.WriteLine(Inum);
}

```

Wynik działania dla wywołania programu zgodny z oczekiwaniami:

 Konsola debugowania programu Microsoft Visual Studio

```

Ponizej lista produktow w naszej bazie :
Tyskie
Zywiec
Harnas
Ponizej lista dostawcow w naszej bazie :
Browar PL
Ponizej lista faktur w naszej bazie :
1
2
3

```

```

sqlite> SELECT * FROM Products;
1|Tyskie|1|0
2|Zywiec|1|0
3|Harnas|1|0
sqlite> SELECT * FROM Invoices;
1|10
2|7
3|4
sqlite> SELECT * FROM Suppliers;
1|Browar PL|Krakowska 123|Warszawa
sqlite> SELECT * FROM InvoiceProduct
...> ;
1|1
1|2
1|3
2|1
2|3
3|3

```

W ramach testu sprawdzamy w terminalowym sqlite3 wyniki dla zapytań o faktury do których należą produkty o danym ID

```

<INNER JOIN InvoiceProduct ON InvoiceProduct.InvoicesInvoiceID = Invoices.InvoiceID
WHERE ProductsProductID = 1;
1|10
2|7
<INNER JOIN InvoiceProduct ON InvoiceProduct.InvoicesInvoiceID = Invoices.InvoiceID
WHERE ProductsProductID = 2;
1|10
<INNER JOIN InvoiceProduct ON InvoiceProduct.InvoicesInvoiceID = Invoices.InvoiceID
WHERE ProductsProductID = 3;
1|10
2|7
3|4
<INNER JOIN InvoiceProduct ON InvoiceProduct.InvoicesInvoiceID = Invoices.InvoiceID
WHERE ProductsProductID = 4;
4|10
<INNER JOIN InvoiceProduct ON InvoiceProduct.InvoicesInvoiceID = Invoices.InvoiceID
WHERE ProductsProductID = 5;
4|10
sqlite>

```

# ZADANIE V

Zaczynamy od zmodyfikowania pliku Supplier.cs ( dodajemy dziedziczenie i odpowiednio dostosowujemy pola do zadania):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
    public class Company
    {
        public int CompanyID { get; set; }
        public String? CompanyName { get; set; }
        public String? Street { get; set; }
        public String? City { get; set; }

        public String? zipCode { get; set; }
    }

    public class Supplier : Company
    {
        public Supplier() {
            this.Products = new HashSet<Product>();
        }
        public long BankAccountNumber { get; set; }

        public ICollection<Product> Products { get; set; }
    }

    public class Customer : Company
    {
        public double Discount { get; set; }
    }
}
```

Po wykonaniu migracji i zupdateowaniu bazy danych widzimy pierwszą różnicę. Została utworzona nowa tabela Companies, ale może zaniepokoić nas brak tabel Suppliers czy Customers. Po analizie kodu DDL tabeli Companies, widzimy, że jest ona pewnym kontenerem na rekordy będące reprezentantami wszystkich możliwych obiektów, będących instancjami różnych klas z hierarchii dziedziczenia. Nowo utworzona kolumna Discriminator będzie przechowywać nazwę klasy, której dany rekord jest instancją.

```
sqlite> .tables
Companies          Invoices          __EFMigrationsHistory
InvoiceProduct     Products
sqlite> SELECT * FROM Companies;
sqlite> SELECT * FROM Products;
sqlite> .schema Companies
CREATE TABLE IF NOT EXISTS "Companies" (
  "CompanyID" INTEGER NOT NULL CONSTRAINT "PK_Companies" PRIMARY KEY AUTOINCREMENT,
  "BankAccountNumber" INTEGER NULL,
  "City" TEXT NULL,
  "CompanyName" TEXT NULL,
  "Discount" REAL NULL,
  "Discriminator" TEXT NOT NULL,
  "Street" TEXT NULL,
  "zipCode" TEXT NULL
);
sqlite>
```

Sprawdzamy, czy kod w klasie Program.cs dalej się wykonuje:

```

Konsola debugowania programu Microsoft Visual Studio

Ponizej lista produktow w naszej bazie :
buty sportowe
narty
Ponizej lista dostawcow w naszej bazie :
Sportowy PL
Ponizej lista faktur w naszej bazie :
1
```

Widzimy, że faktycznie wyniki są takie same jak w poprzednim zadaniu, zmiana sposobu reprezentacji danych nie wpłynęła na funkcjonalność naszego modelu.

Następnie dodałem po 3 Suppliersów i Customersów do naszej bazy i wyświetliłem wyniki wyszukując na trzy różne sposoby : po Customers, Suppliers oraz Companies. Wykonywany kod oraz wyniki zamieszczam poniżej.

```
ProductContext productContext = new ProductContext();
Supplier supplier1 = new Supplier
{
    CompanyName = "Butik PL",
    Street = "Diamentowa 123",
    City = "Chocholow"
};
Supplier supplier2 = new Supplier
{
    CompanyName = "Remont PL",
    Street = "Betonowa 123",
    City = "Katowice"
};
Supplier supplier3 = new Supplier
{
    CompanyName = "Klub PL",
    Street = "Melanzowa 123",
    City = "Sopot"
};

Customer customer1 = new Customer
{
    CompanyName = "Customer PL",
    Street = "Klientowa 123",
    City = "Sopot"
};

Customer customer2 = new Customer
{
    CompanyName = "Customer PL",
    Street = "Klientowa 123",
    City = "Sopot",
    Discount = 0.2
};

Customer customer3 = new Customer
{
    CompanyName = "Customer PL",
    Street = "Klientowa 123",
    City = "Sopot",
    Discount = 0.3
};

productContext.Suppliers.Add(supplier1);
productContext.Suppliers.Add(supplier2);
productContext.Suppliers.Add(supplier3);
```

```
productContext.Customers.Add(customer1);
productContext.Customers.Add(customer2);
productContext.Customers.Add(customer3);

productContext.SaveChanges();

var queryProd = from prod in productContext.Products
                select prod.ProductName;

var queryInv = from inv in productContext.Invoices
                select inv.InvoiceID;

var querySupp = from comp in productContext.Suppliers
                 select comp.CompanyName;

var queryCustomers = from cust in productContext.Customers
                     select cust.CompanyName;

var queryCompanies = from comp in productContext.Companies
                      select comp.CompanyName;

Console.WriteLine("Ponizej lista dostawcow w naszej bazie : ");
foreach (var pName in querySupp)
{
    Console.WriteLine(pName);
}

Console.WriteLine("Ponizej lista Klientow w naszej bazie : ");
foreach (var cName in queryCustomers)
{
    Console.WriteLine(cName);
}

Console.WriteLine("Ponizej lista firm w naszej bazie : ");
foreach (var Inum in queryCompanies)
{
    Console.WriteLine(Inum);
}
```





Konsola debugowania programu Microsoft Visual Studio

Ponizej lista dostawcow w naszej bazie :

Sportowy PL

Butik PL

Remont PL

Klub PL

Ponizej lista Klientow w naszej bazie :

Customer PL

Customer PL

Customer PL

Ponizej lista firm w naszej bazie :

Sportowy PL

Customer PL

Customer PL

Customer PL

Butik PL

Remont PL

Klub PL

Wyniki są zgodne z oczekiwaniami. Dla formalności sprawdzimy jeszcze aktualny stan bazy danych przy pomocy terminalowego sqlite3.

```
sqlite> SELECT * FROM Companies;
1|0|Poznan|Sportowy PL||Supplier|Daliowa 123|
sqlite> SELECT * FROM COMPANIES;
1|0|Poznan|Sportowy PL||Supplier|Daliowa 123|
2||Sopot|Customer PL|0.0|Customer|Klientowa 123|
3||Sopot|Customer PL|0.2|Customer|Klientowa 123|
4||Sopot|Customer PL|0.3|Customer|Klientowa 123|
5|0|Chocholow|Butik PL||Supplier|Diamentowa 123|
6|0|Katowice|Remont PL||Supplier|Betonowa 123|
7|0|Sopot|Klub PL||Supplier|Melanzowa 123|
sqlite>
```

Widzimy, że wszystko działa jak należy.

# Wnioski:

W przypadku podejścia Table-Per-Hierarchy, powiązanie w hierarchii dziedziczenia DBsety są mapowane na jedną tabelę w bazie danych, defaultowo nazwaną jak kolekcja klasy najwyżej w hierachii dziedziczenia. Tabela ta zawiera dodatkową kolumnę Discriminator, której wartość tekstowa określa z instancją której klasy powiązany jest dany obiekt. Każdy rekord zawiera wszystkie tabele będące sumą ilościową cech wszystkich klas z drzewa dziedziczenia. W wyniku tego zabiegu, jeżeli klasy dziedziczące posiadają relatywnie dużą ilość własnych pól, dochodzimy do sytuacji, w której w naszej tabeli znajduje się potencjalnie bardzo duża ilość redundantnych danych ( wartości null). Rozwiązaniem tego problemu będzie kolejna strategia podejścia do hierarchii dziedziczenia tzw. Table-Per-Type (TPT).

## ZADANIE VI

W zadaniu 6 rozpocząłem od zmodyfikowania klasy ProductContext.cs poprzez nadpisanie metody OnModelCreating():

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace JakubBarberEFProducts
{
    public class ProductContext:DbContext
    {
        public DbSet<Product> Products { get; set; }
        public DbSet<Company> Companies { get; set; }
        public DbSet<Customer> Customers { get; set; }
        public DbSet<Supplier> Suppliers { get; set; }
        public DbSet<Invoice> Invoices { get; set; }
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Company>().UseTptMappingStrategy();
            modelBuilder.Entity<Customer>().UseTptMappingStrategy();
            modelBuilder.Entity<Supplier>().UseTptMappingStrategy();
        }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
```

```

    {
        base.OnConfiguring(optionsBuilder);
        optionsBuilder.UseSqlite("DataSource=ProductsDatabase");
    }
}

```

Po wykonaniu migracji i zupdateowaniu bazy danych schemat bazy wygląda teraz następująco:

(pokazuje tylko 3 tabele, które się zmieniły i które są elementem zainteresowania)

```


sqlite> .tables
Companies          Invoices          __EFMigrationsHistory
Customers          Products
InvoiceProduct     Suppliers
sqlite> .schema Companies;
sqlite> .schema Companies
CREATE TABLE IF NOT EXISTS "Companies" (
    "CompanyID" INTEGER NOT NULL CONSTRAINT "PK_Companies" PRIMARY KEY AUTOINCREMENT,
    "City" TEXT NULL,
    "CompanyName" TEXT NULL,
    "Street" TEXT NULL,
    "zipCode" TEXT NULL
);
sqlite> .schema Suppliers
CREATE TABLE IF NOT EXISTS "Suppliers" (
    "CompanyID" INTEGER NOT NULL CONSTRAINT "PK_Suppliers" PRIMARY KEY AUTOINCREMENT,
    "BankAccountNumber" INTEGER NOT NULL,
    CONSTRAINT "FK_Suppliers_Companies_CompanyID" FOREIGN KEY ("CompanyID")
REFERENCES "Companies" ("CompanyID") ON DELETE CASCADE
);
sqlite> .schema Customers
CREATE TABLE IF NOT EXISTS "Customers" (
    "CompanyID" INTEGER NOT NULL CONSTRAINT "PK_Customers" PRIMARY KEY AUTOINCREMENT,
    "Discount" REAL NOT NULL,
    CONSTRAINT "FK_Customers_Companies_CompanyID" FOREIGN KEY ("CompanyID")
REFERENCES "Companies" ("CompanyID") ON DELETE CASCADE
);

```

Widzimy, że w przeciwieństwie do strategii TPH, nie mamy w tym przypadku utworzonej jednej wspólnej tabeli, dla wszystkich rekordów. W tym przypadku rozwiązanie polega na przechowywaniu wszystkich danych specyficznych dla deklaracji konkretnej klasy w tabeli powiązanej z tą właśnie klasą. Widzimy, że dla

tabeli reprezentujących klasy dziedziczące z Company, mamy referencję na rekord w tabeli company z każdej z tych tabel w postaci klucza obcego. Tabele te zawierają kolumny specyficzne dla konkretnej implementacji superklasy : Discount dla Customers, czy bankAccountNumber w przypadku Suppliers.

Testujemy czy kod z poprzedniego zadania również wykona się bez przeszkód w tym przypadku:

 Konsola debugowania programu Microsoft Visual Studio

```
Ponizej lista dostawcow w naszej bazie :
```

```
Butik PL
```

```
Remont PL
```

```
Klub PL
```

```
Ponizej lista Klientow w naszej bazie :
```

```
Customer PL
```

```
Customer PL
```

```
Customer PL
```

```
Ponizej lista firm w naszej bazie :
```

```
Customer PL
```

```
Customer PL
```

```
Customer PL
```

```
Butik PL
```

```
Remont PL
```

```
Klub PL
```

Widzimy, że program działa, mimo że parę razy trzeba było zastosować tę samą migrację, żeby baza załapała zmianę .

Sprawdźmy dla formalności przy pomocy terminalowego sqlite3:

```

sqlite> .tables
Companies          Invoices          __EFMigrationsHistory
Customers          Products
InvoiceProduct     Suppliers
sqlite> SELECT * FROM Companies;
1|Sopot|Customer PL|Klientowa 123|
2|Sopot|Customer PL|Klientowa 123|
3|Sopot|Customer PL|Klientowa 123|
4|Chocholow|Butik PL|Diamentowa 123|
5|Katowice|Remont PL|Betonowa 123|
6|Sopot|Klub PL|Melanzowa 123|
sqlite> SELECT * FROM Companies INNER JOIN Suppliers ON Suppliers.CompanyID =
Companies.CompanyID
...> ;
4|Chocholow|Butik PL|Diamentowa 123||4|0
5|Katowice|Remont PL|Betonowa 123||5|0
6|Sopot|Klub PL|Melanzowa 123||6|0
sqlite> SELECT * FROM Companies INNER JOIN Customers ON Customers.CompanyID =
Companies.CompanyID
...> ;
1|Sopot|Customer PL|Klientowa 123||1|0.0
2|Sopot|Customer PL|Klientowa 123||2|0.2
3|Sopot|Customer PL|Klientowa 123||3|0.3
sqlite>

```

Wszystko działa bez przeszkód, a wyniki są zgodne z naszymi oczekiwaniami.

## Wnioski

Tak jak podkreśliłem wcześniej w przypadku strategii TPH powielamy niepotrzebne i redundantne dane. Z drugiej strony, bardziej skomplikowane zapytania w strategii TPP mogą być mniej efektywne i optymalne w związku z potencjalnie dużą ilością joinów między tabelami. Poza opisanymi dwoma strategiami istnieje jeszcze strategia TPC, w której dla każdej klasy tworzona jest odrębna tabela zawierająca wszystkie wartości pól dla danej instancji. W pewnym sensie odpowiada to ewentualnym problemom z wydajnością w przypadku TPP, oraz jest dużo bardziej przejrzyste niż podejście TPH. Schematy TCP są zdenormalizowane. Widoczną różnicą jest również fakt, że TPC nie tworzy osobnych kolumn dla klas abstrakcyjnych, gdzie w TPP mamy taką sytuację.