Entity Framework - sprawozdanie

Jakub Białecki, Przemysław Popowski, Jakub Worek

1. Dostawca i Produkty
   Kod:
   Klasa Product

```
namespace lab{

  internal class Product{
    public int ProductID { get; set; }
    public string ProductName { get; set; }
    public int UnitsOnStock { get; set; }
    public Supplier? Supplier { get; set; } = null;
    public override string ToString(){
      return $"{ProductName} ({UnitsOnStock} szt.)";
    }
  }
}
```

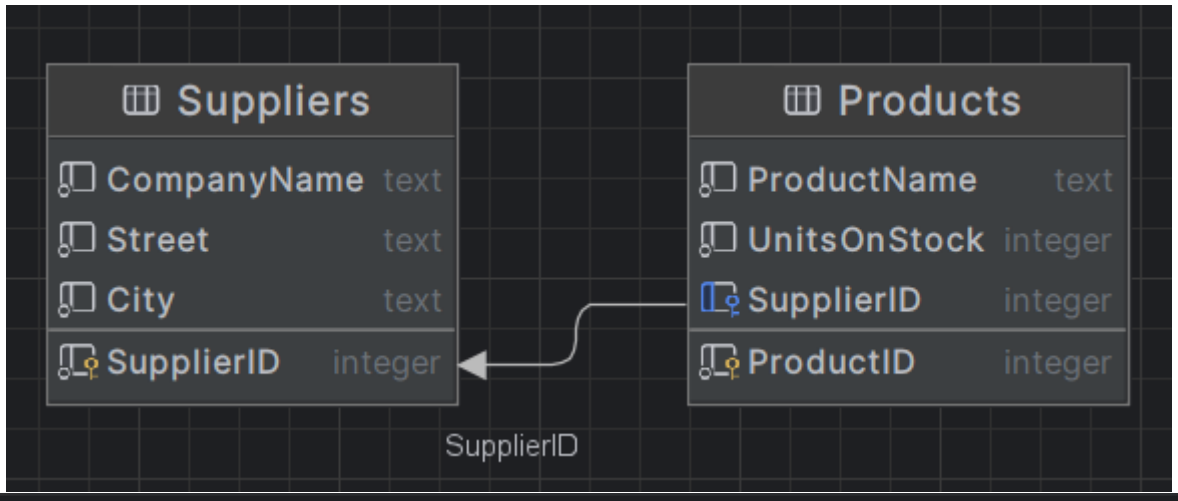
Klasa Supplier

```
namespace lab{
  internal class Supplier{
    public int SupplierID { get; set; }
    public string CompanyName { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public override string ToString(){
      return CompanyName;
    }
  }
}
```

Przykład działania i schemat bazy danych:

```
Podaj nazwę produktu
>>> Kredki
Podaj liczbę dostępnych sztuk produktu
>>> 10
Tworzę nowy produkt...
Stworzono produkt: Kredki (10 szt.)
Dodać nowego dostawcę? (tak/nie)
tak

Podaj nazwę dostawcy
>>> KredkowyRaj
Podaj miasto
>>> Krakow
Podaj ulicę
>>> Polna
Tworzę nowego dostawcę...
Stworzono dostawcę: KredkowyRaj
Dodaję dostawcę do produktu...
Zapisuję dane do bazy...
```



Products [MyProductDatabase]

3 rows | Tx: Auto | DDL | CSV

WHERE ORDER BY

	ProductID	ProductName	UnitsOnStock	SupplierID
1	1	Kredki	10	1
2	2	Flamastry	32	1
3	3	Rower	4	2

Suppliers [MyProductDatabase]

2 rows | Tx: Auto | DDL | CSV

WHERE ORDER BY

	SupplierID	CompanyName	Street	City
1	1	KredkowyRaj	Polna	Krakow
2	2	RowerowyRaj	Czarnowiejska	Krakow

2. Odwrócenie relacji dostawca-produkt

   Kod:

   Klasa Product

```
namespace lab{
  internal class Product{
    public int ProductID { get; set; }
    public string ProductName { get; set; }
    public int UnitsOnStock { get; set; }
    public override string ToString(){
      return $"{ProductName} ({UnitsOnStock} szt.)";
    }
  }
}
```

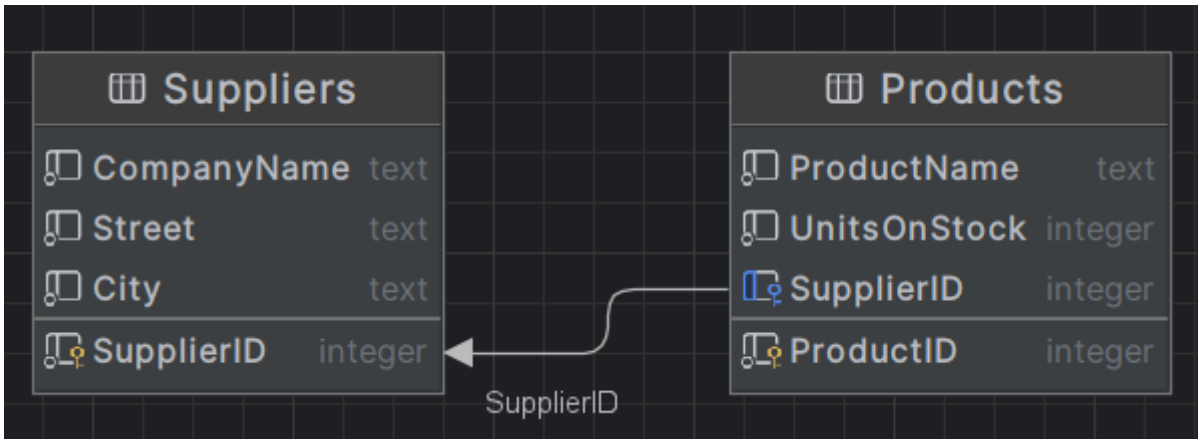
   Klasa Supplier

```
namespace lab{
  internal 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; } = new List<Product>();
    public override string ToString(){
      return CompanyName;
    }
  }
}
```

Przykład działania i schemat bazy danych:

```
Podaj nazwę produktu
>>> Kredki
Podaj liczbę dostępnych sztuk produktu
>>> 15
Tworzę nowy produkt...
Stworzono produkt: Kredki (15 szt.)
Dodać nowego dostawcę? (tak/nie)
tak

Podaj nazwę dostawcy
>>> KredkiCom
Podaj miasto
>>> Krakow
Podaj ulicę
>>> Kawiory
Tworzę nowego dostawcę...
Stworzono dostawcę: KredkiCom
Dodaję produkt to dostawcy...
Zapisuję dane do bazy...
```



Możemy zauważyć, że pomimo zapisania relacji w EF w odwrotny sposób, w bazie danych relacja wciąż wygląda tak samo. EF „pod spodem” dokonuje optymalizacji dzięki czemu nie musimy trzymać w tabeli Supplier powielonych danych dostawców różniących się kluczem obcym wskazującym na produkt z tabeli Products. W takiej sytuacji jeden dostawca znajdowałby się wielokrotnie w tabeli mając przypisane inne id.

3. Relacja dwustronna

   Kod:

   Klasa Product

```
namespace lab{
  internal class Product{
    public int ProductID { get; set; }
    public string ProductName { get; set; }
    public int UnitsOnStock { get; set; }
    public Supplier? Supplier { get; set; } = null;
    public override string ToString(){
      return $"{ProductName} ({UnitsOnStock} szt.)";
    }
  }
}
```

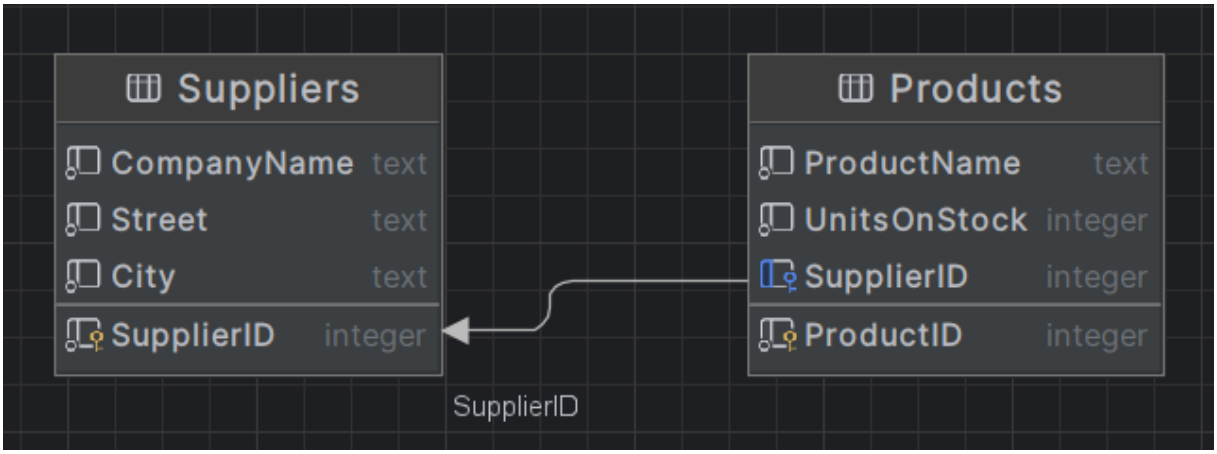
   Klasa Supplier

```
namespace lab{
  internal 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; } = new List<Product>();
    public override string ToString(){
      return CompanyName;
    }
  }
}
```

Przykład działania i schemat bazy danych:

```
Podaj nazwę produktu
>>> Rower
Podaj liczbę dostępnych sztuk produktu
>>> 2
Tworzę nowy produkt...
Stworzono produkt: Rower (2 szt.)
Dodać nowego dostawcę? (tak/nie)
tak

Podaj nazwę dostawcy
>>> RowerowySklep
Podaj miasto
>>> Krakow
Podaj ulicę
>>> Kawiory
Tworzę nowego dostawcę...
Stworzono dostawcę: RowerowySklep
Dodaję produkt to dostawcy...
Zapisuję dane do bazy...
```



Ponownie obserwujemy taki sam schemat bazy danych. Możemy dojść do wniosku, że EF pozwala na stworzenie relacji dwukierunkowej po to aby móc łatwiej manipulować obiektami, a mimo to „pod spodem” relacje są przekształcane.

4. Relacja wiele-do-wielu
   Kod:
   Klasa Product

```
internal class Product{
  public int ProductID { get; set; }
  public string ProductName { get; set; }
  public int UnitsInStock { get; set; }
  public virtual ICollection<InvoiceItem> InvoiceItems { get; set; }
  public override string ToString(){
    return $"{ProductName} ({UnitsInStock} szt.)";
  }
}
```

   Klasa Invoice

```
internal class Invoice{
  [Key]
  public int InvoiceNumber { get; set; }
  public virtual ICollection<InvoiceItem> InvoiceItems { get; set; }
  public override string ToString(){
    StringBuilder sn = new($"Invoice {InvoiceNumber}:");
    foreach (InvoiceItem item in InvoiceItems){
      sn.Append($"\t- {item}");
    }
    return sn.ToString();
  }
}
```

   Klasa InvoiceItem – pomocnicza, reprezentuje pozycje faktury

```
class InvoiceItem{
  [Key, Column(Order = 0)]
  public int InvoiceNumber { get; set; }
  [Key, Column(Order = 1)]
  public int ProductID { get; set; }
  public virtual Invoice Invoice { get; set; }
  public virtual Product Product { get; set; }
  public int Quantity { get; set; }
  public override string ToString()
  {
    return $"{Product} ({Quantity} szt.)";
  }
}
```

Przykład działania i schemat bazy danych:

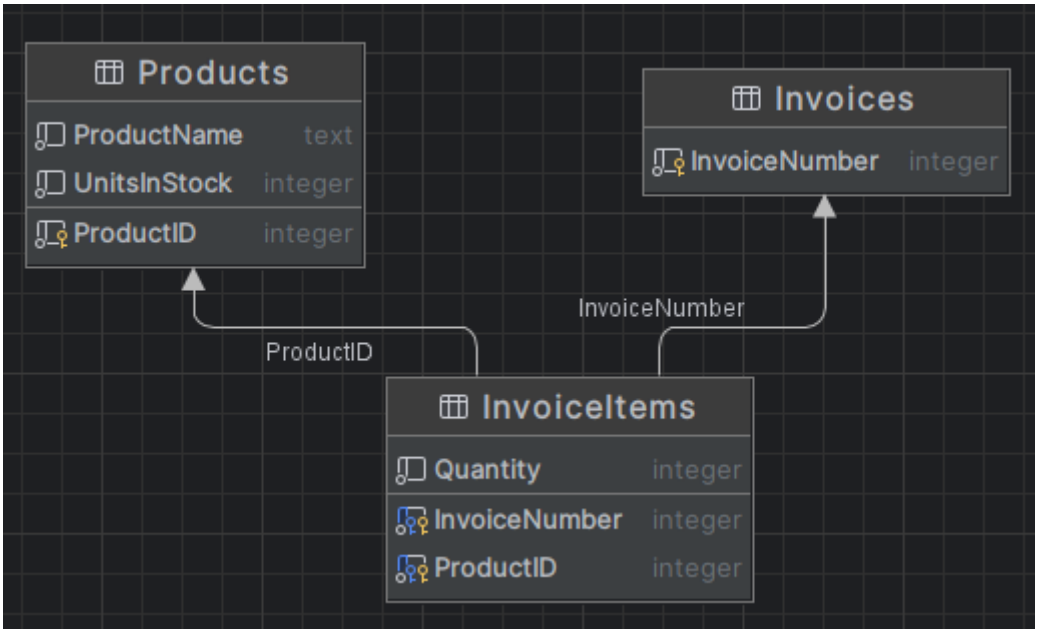
```
Napisz, co chcesz zrobić. Lista dostępnych komend:
        - add,
        - remove,
        - sell,
        - all,
        - available,
        - exit,
>>> sell
Z poniższej listy wybierz produkty, które mają zostać dodane do faktury
- wprowadź id produktu, a po spacji liczbę sprzedawanych sztuk
- aby zakończyć wybieranie produktów, naciśniej Enter

[1] flamaster (dostępne: 7)
[2] kredka (dostępne: 6)

>>> 2 1
Dodano 1 szt. kredka do faktury. Razem na fakturze jest 1 szt.

>>> 1 2
Dodano 2 szt. flamaster do faktury. Razem na fakturze jest 2 szt.
```

```
Zakończono wybieranie produktów
Aktualizuję liczbę dostępnych produktów...
Pozostało 5 szt. kredka
Pozostało 5 szt. flamaster
```



5. Dziedziczenie Table-Per-Hierarchy

Kod:

Klasa Company

```
namespace lab{
  internal abstract class Company{
    public int CompanyID { get; set; }
    public string CompanyName { get; set; } = String.Empty;
    public string Street { get; set; } = String.Empty;
    public string City { get; set; } = String.Empty;
    public string ZipCode { get; set; } = String.Empty;
    public override string ToString(){
      return $"[{CompanyID}] {CompanyName}";
    }
  }
}
```

Enum CompanyType

```
namespace lab{
  internal static class CompanyType{
    public const string CUSTOMER = "customer";
    public const string SUPPLIER = "supplier";
    public static List<string> ALL_TYPES = new(){
      CUSTOMER,
      SUPPLIER
    };
  }
}
```

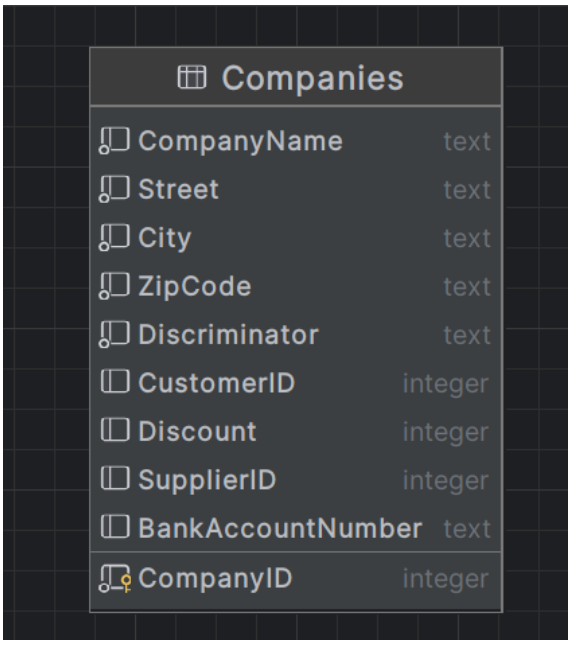
Klasa Customer

```
namespace lab{
  internal class Customer : Company{
    public int CustomerID { get; set; }
    public int Discount { get; set; }
    public override string ToString(){
      return $"{base.ToString()} (klient)";
    }
  }
}
```

Klasa Supplier

```
namespace lab{
  internal class Supplier : Company{
    public int SupplierID { get; set; }
    public string BankAccountNumber { get; set; } = String.Empty;
    public override string ToString(){
      return $"{base.ToString()} (dostawca)";
    }
  }
}
```

Schemat bazy danych:

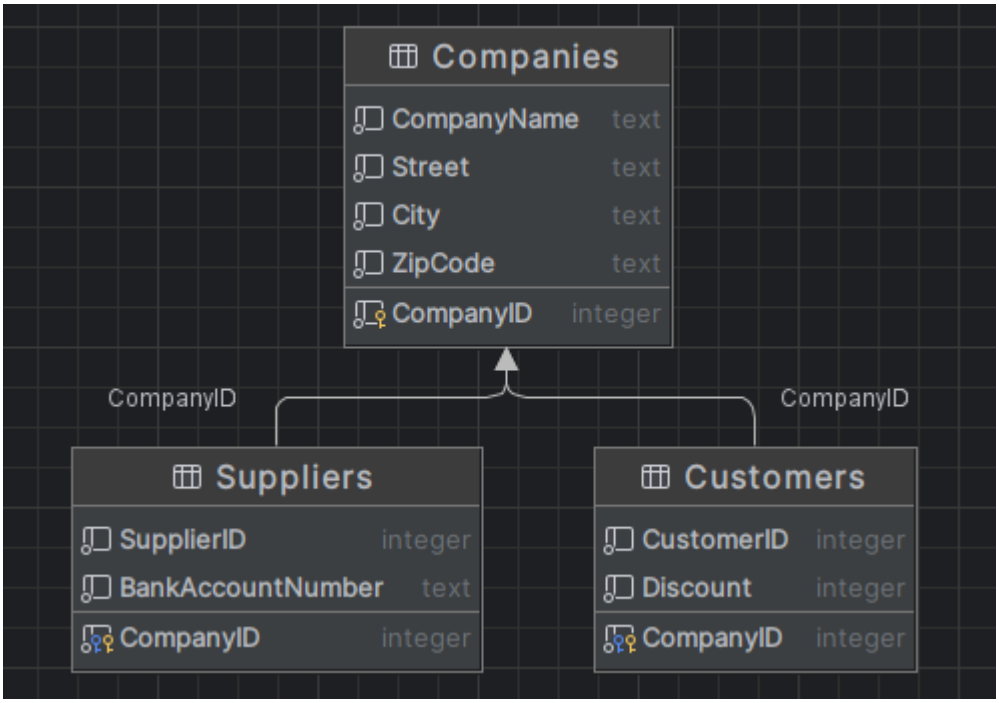


6. Dziedziczenie Table-Per-Type

Zmiany w kodzie są minimalne. Polegają na dodaniu adnotacji nad deklaracjami klas Customer i Supplier:

```
// w klasie Customer
[Table("Customers")]
// w klasie Supplier
[Table("Suppliers")]
```

Otrzymujemy wówczas taki schemat bazy danych:



7. Porównanie obu sposobów dziedziczenia

Table-Per-Hierarchy:

- Tworzona jest jedna tabela, która zawiera wspólne dla klas dziedziczących dane oraz dane, charakteryzujące każdą z klas dziedziczących z osobna,
- W przypadku, gdy klasa dziedzicząca posiada atrybut, którego nie ma w klasie, z której dziedziczy, dodawana jest osobna kolumna, w której dla pozostałych klas wpisane są wartości null, a dla tej klasy, odpowiednie wartości tego parametru
- Takie podejście do modelowania pozwala na zmniejszenie liczby wykonywanych operacji join na tabelach, w porównaniu do modelowania z wykorzystaniem Table-Per-Type (gdzie tworzone są osobne tabele dla każdego z typów).
- W przypadku wielu klas dziedziczących z tej samej klasy, jedna tabela nie jest dobrym rozwiązaniem, ponieważ będzie zawierała bardzo dużo wartości null (marnowanie miejsca),
- Grupowanie danych, w przypadku wielu klas dziedziczących, zmniejsza przejrzystość schematu bazy danych.

Table-Per-Type:

- Tworzone jest kilka tabel (osobne tabele dla każdej z klas, zarówno tej, z której dziedziczą klasy, jak i klas dziedziczących),
- Tabele klas dziedziczących są łączone z tabelą klasy, z której dziedziczą, przy pomocy relacji 1 do 1.
- Takie podejście nie wymaga trzymania pustych wartości w tabelach (null), dzięki czemu zapisywane są tylko wartości, stanowiące dane,
- W przypadku wielu klas dziedziczących z jednej klasy, takie podejście pozwala na zwiększenie czytelności schematu bazy danych.
- Konieczne jest wykonywanie wielu operacji join (łączenie tabel klas dziedziczących z tabelą klasy nadrzędnej – tej, z której klasy dziedziczą).