

## Baza danych SQL Server w projekcie Visual Studio

Programista zamierzający korzystać w swoim projekcie z baz danych może wybrać jeden z trzech oferowanych przez Microsoft mechanizmów odwzorowania zawartości baz danych w klasach .NET lub jeden z wielu framework'ów niezależnych firm.

Omówimy techniki *odwzorowania obiektowo-relacyjnego* (ORM), które zostały przygotowane przez Microsoft i dostępne są w platformie .NET bez konieczności dodatkowej instalacji.

Czym są w ogóle technologie ORM? Mówiąc najprościej, jest to mechanizm, który umożliwia aplikacji dostęp do danych z tabel i widoków bazy danych bez konieczności nawiązywania niskopoziomowego „kontaktu” z bazą danych i wysyłania do niej zapytań SQL. ORM udostępnia dane w kolekcjach platformy .NET, z których mogą być nie tylko odczytywane, ale również w nich modyfikowane. Jest to więc pomost między światem aplikacji a światem bazy danych, warstwa abstrakcji zwalniająca programistę z obowiązku znajomości szczegółów dotyczących konkretnego typu używanej przez niego bazy danych.

Pierwszym rozwiązaniem jest technika ADO.NET oparta na kontrolce DataSet, która pozwala na tworzenie aplikacji bazodanowych przy minimalnej liczbie samodzielnie napisanych linii kodu. Pozwala na łączenie z bazami danych SQL Server, Microsoft Access poprzez mechanizm OLE DB, z obiektowymi bazami danych XML, a także bazami dostępnymi poprzez pomosty ODBC.

Kontrolka DataSet jest w zasadzie bezpośrednią reprezentacją tabel bazy danych, a dodatkowo jest zarządcą poleceń SQL wysyłanych do bazy danych. Zajmuje się także buforowaniem odbieranych z niej danych.

Drugim rozwiązaniem jest LINQ to SQL i jego klasa DataContext. Niestety jest ograniczone wyłącznie do baz danych SQL Server.

Entity Framework (EF) jest trzecim, obecnie najbardziej promowanym mechanizmem ORM dostępnym w platformie .NET. Od wersji 6.0 dołączanej do Visual Studio 2013 jego rozwój został „uwolniony” i zajmuje się nim teraz grupa programistów *open source*.

W przypadku wszystkich trzech technologii po utworzeniu obiektowej reprezentacji zawartości bazy danych możliwe jest utworzenie tzw. źródeł danych (podokno *Data Sources* w Visual Studio).

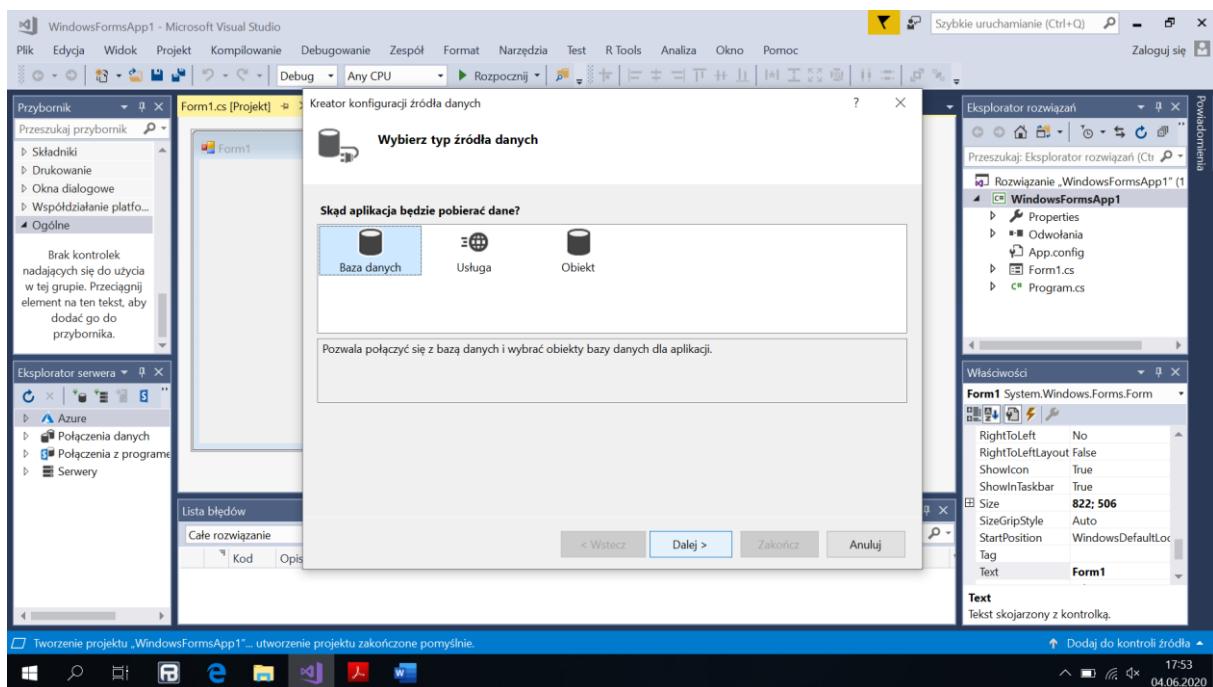
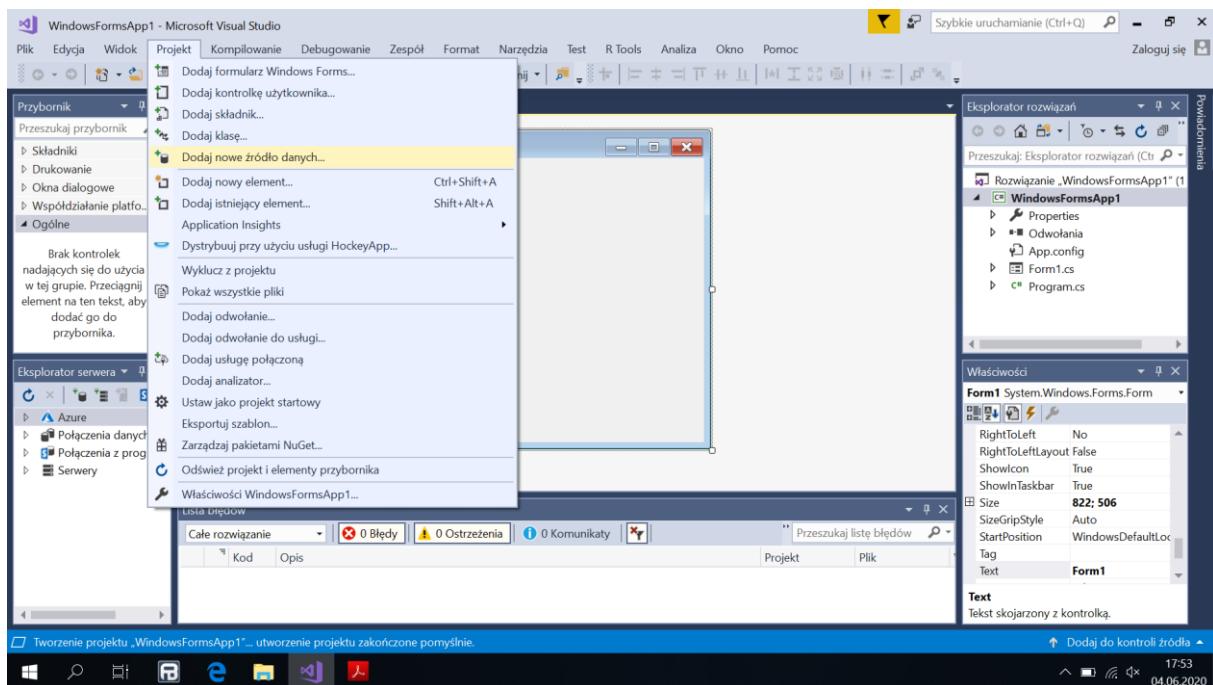
## Projekt aplikacji z bazą danych

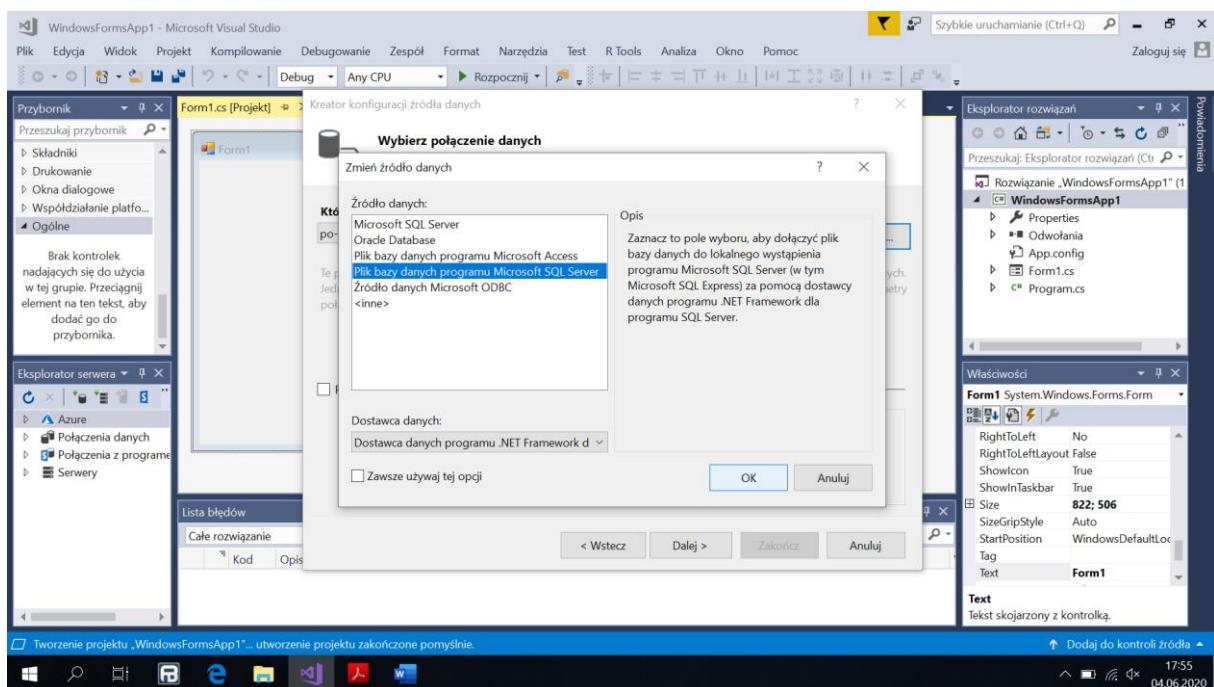
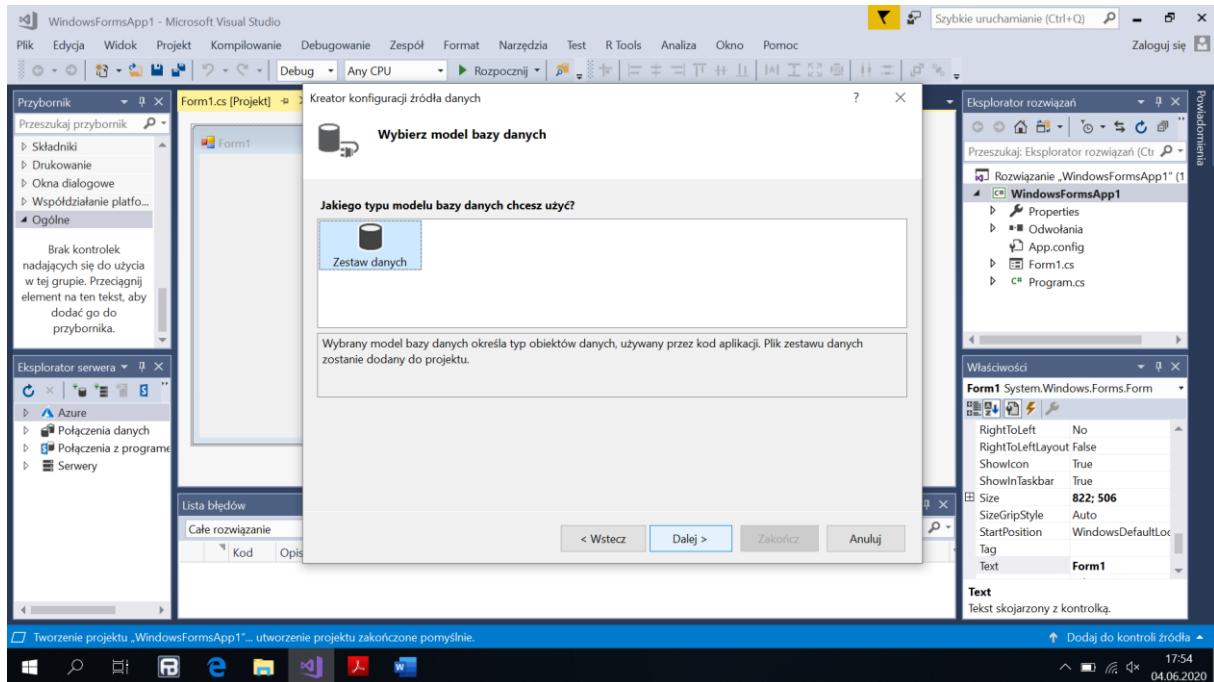
Z poziomu Visual Studio można utworzyć i dodać do projektu aplikacji instancję bazy danych SQL Server. SQL Server jest wyróżniony. Projekty .NET mogą wprawdzie korzystać z wielu rodzajów baz danych, ale tylko w przypadku SQL Servera Visual Studio pozwala na tworzenie bazy danych od zera bez konieczności używania dodatkowych narzędzi.

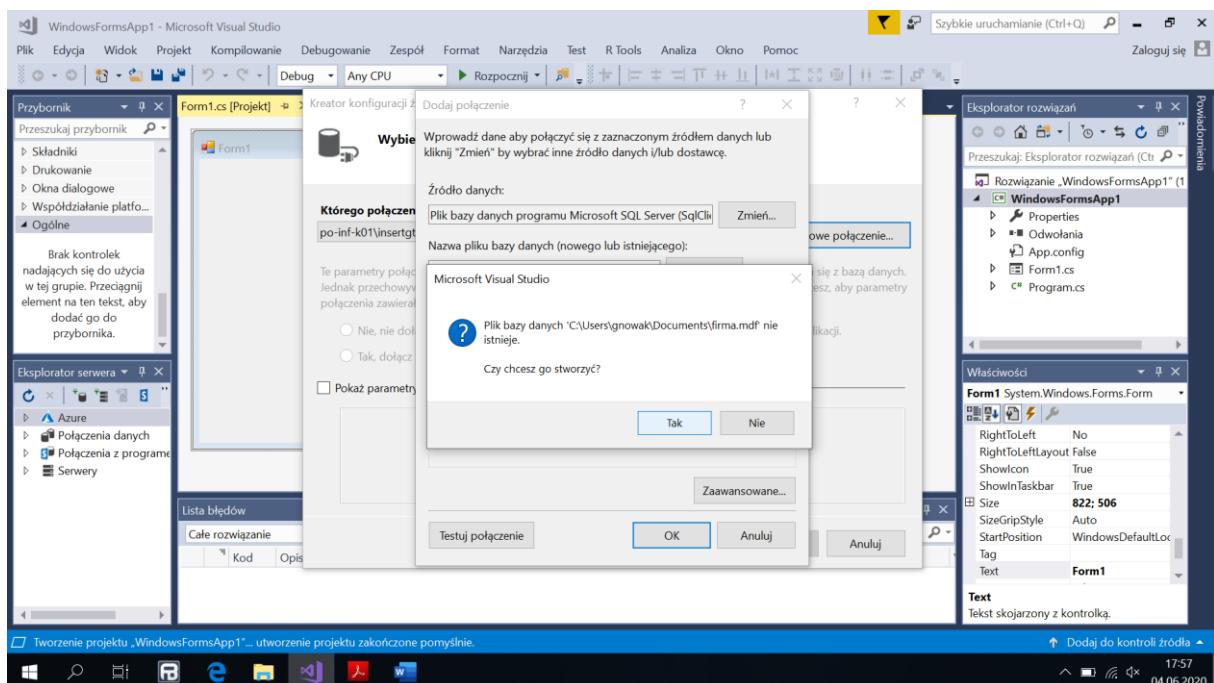
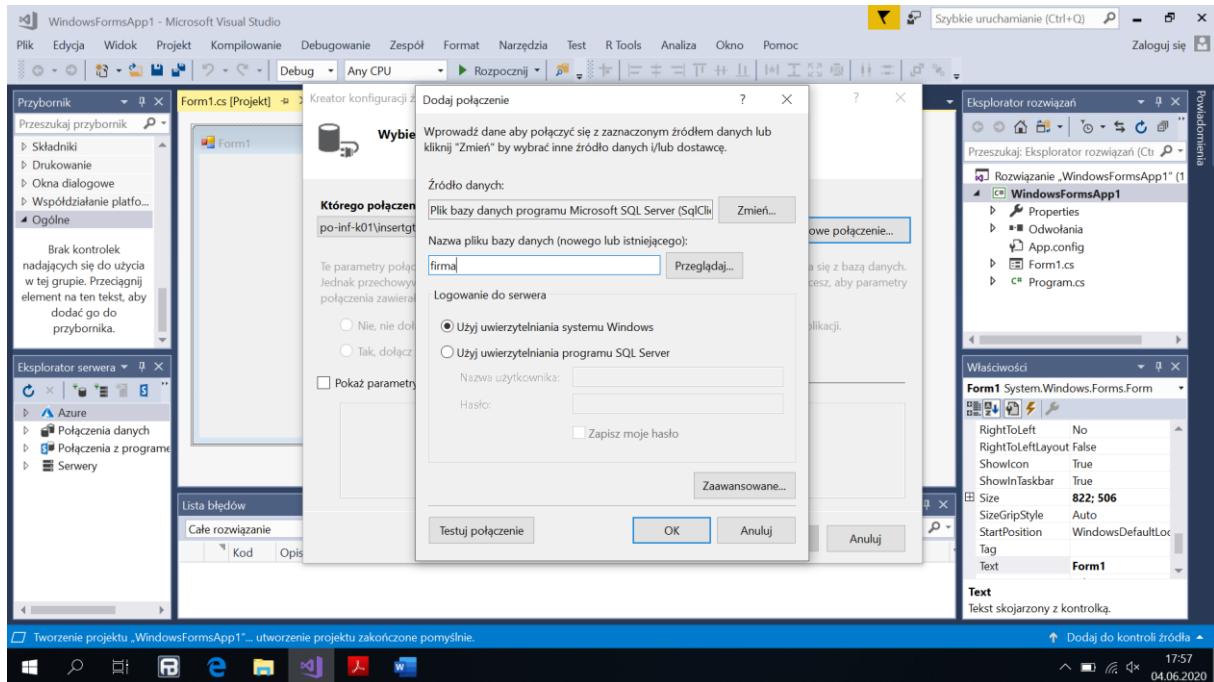
## Dodawanie bazy danych do projektu aplikacji

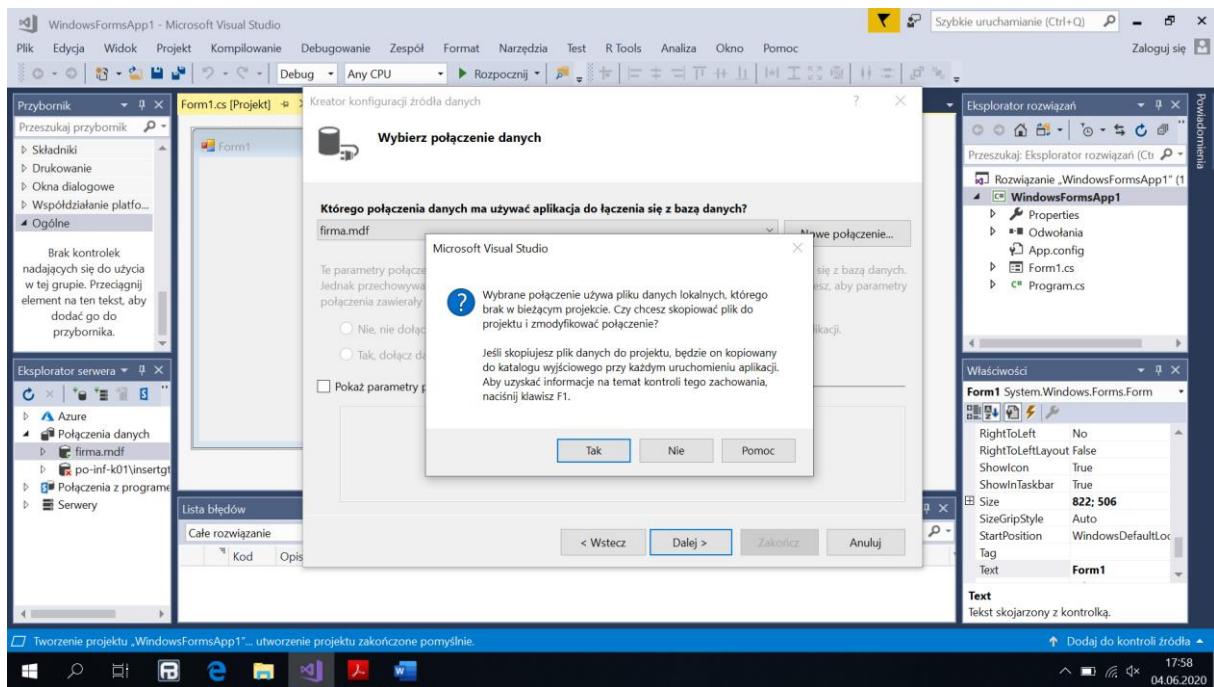
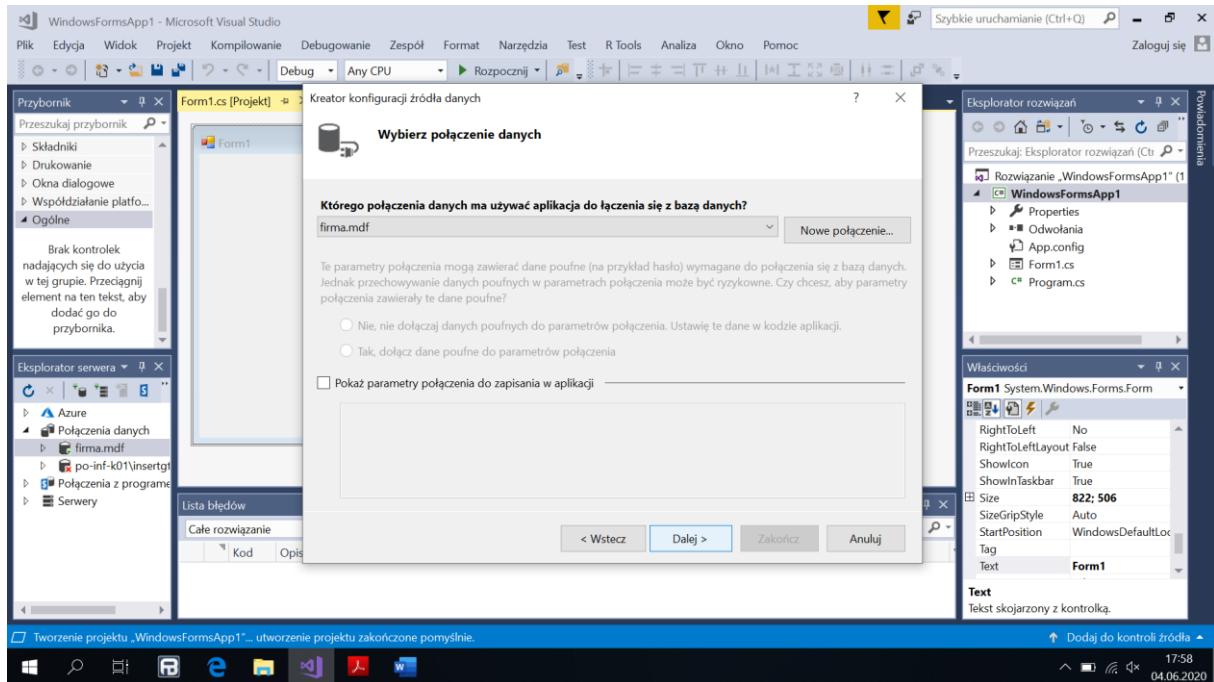
Utwórzmy nowy projekt aplikacji Windows Forms o nazwie *Firma*.

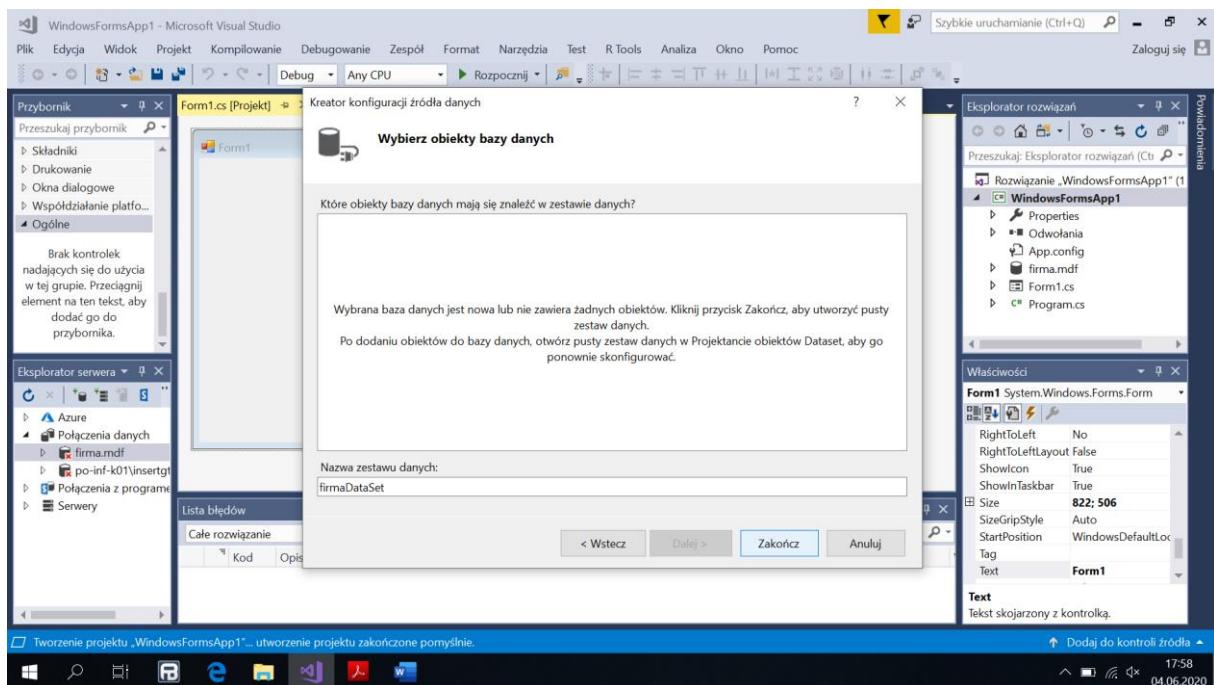
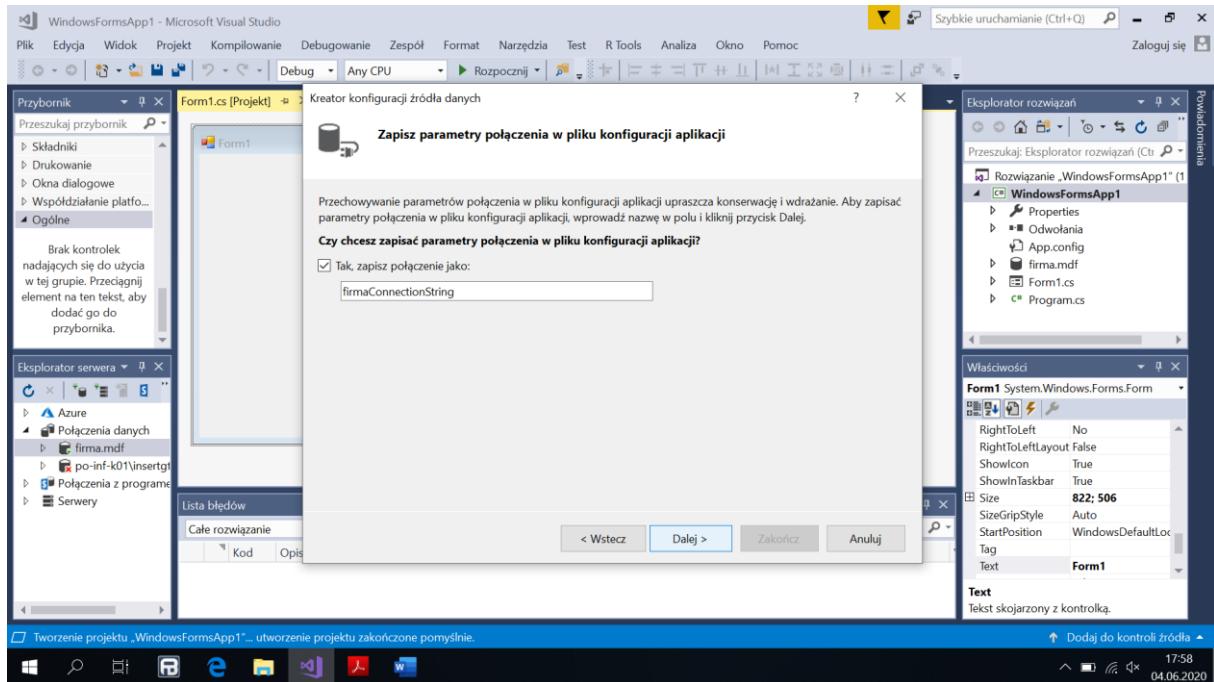
Dodajmy do projektu bazę danych SQL Server. W tym celu z menu Projekt wybieramy pozycję *Dodaj nowe źródło danych* i postępujmy jak pokazano poniżej:







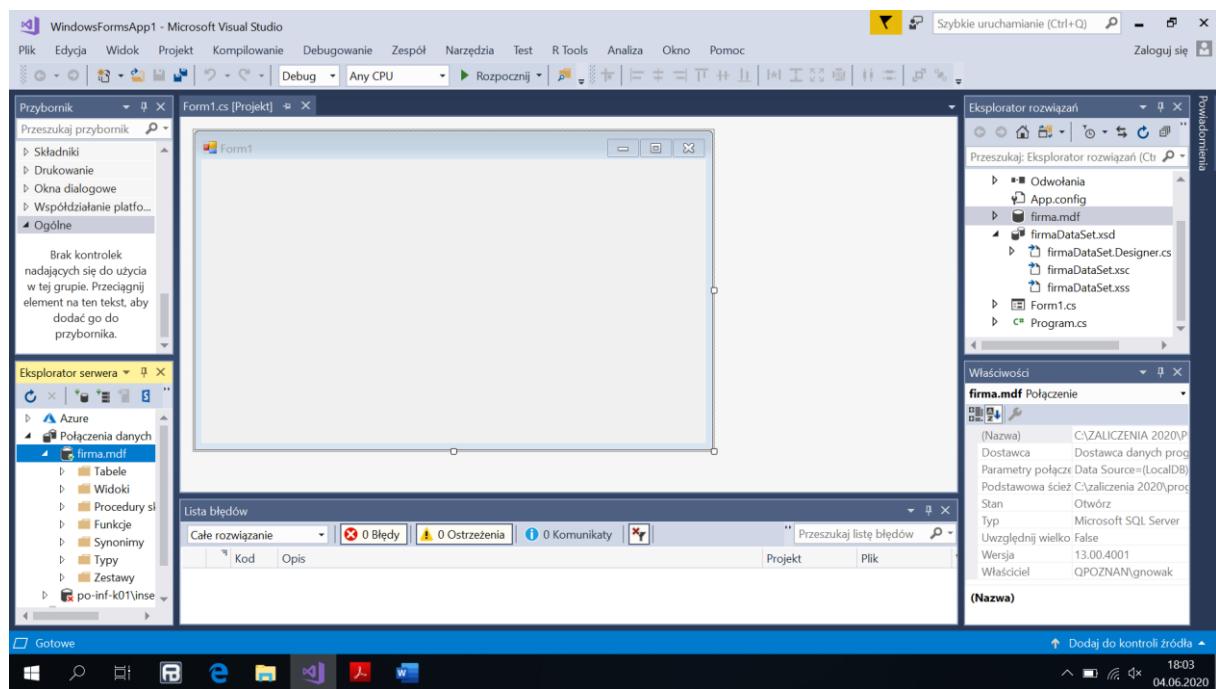




Do plików wymienionych w *Eksplorator rozwiązań* dodana została pozycja *Firma.mdf* wraz z towarzyszącym jej plikiem *Firma\_log.ldf*.

### Łańcuch połączenia (ang. connection string)

Kliknijmy dwukrotnie pozycję *Firma.mdf* w Eksploratorze rozwiązań (*Solution Explorer*). W miejscu, gdzie zwykle jest podokno Przybornik (*Toolbox*), pojawi się Eksplorator serwera (*Server Explorer*) - narzędzie, które umożliwia także przeglądanie i edycję baz danych SQL Server. Pierwsze otwarcie bazy danych w tym podoknie wiąże się z uruchomieniem serwera bazy danych i może potrwać dłuższą chwilę, czasem dłuższą niż przewidziany na to czas. Zatem jeżeli się nie powiedzie, warto spróbować jeszcze raz. Po rozwinięciu gałęzi *Firma.mdf* zobaczymy podgałęzie odpowiadające tabelom, widokom, procedurom składowanym itd. Wszystkie są na razie puste.



Jeżeli w Eksploratorze rozwiązań zaznaczmy *Firma.mdf*, w podoknie *Właściwości* pojawiają się szczegóły, dotyczące m.in. połączenia z tą bazą danych.

Własność Parametry połączenia (*Connection String*), określa szczegóły połączenia z bazą danych. Ma on następującą postać:

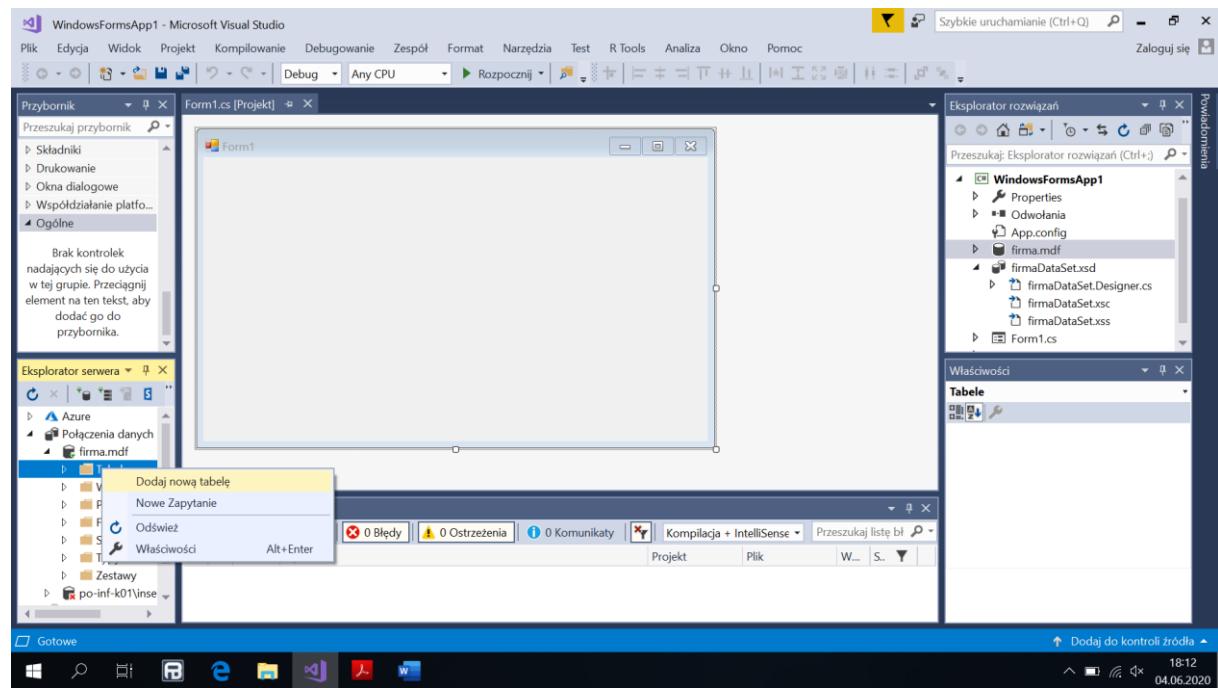
```
Data Source=(LocalDB)\MSSQLLocalDB;  
AttachDbFilename="C:\baza\WindowsFormsApp1\WindowsFormsApp1\firma.mdf";  
Integrated Security=True; Connect Timeout=30
```

Jak widać, łańcuch ten zawiera bezwzględną ścieżkę do pliku firma.mdf. Poza tym w podoknie Właściwości możemy sprawdzić informacje o użytym sterowniku (Dostawca danych programu .NET Framework dla programu SQL Server) i aktualnym stanie połączenia.

### Dodawanie tabeli do bazy danych

Aby do bazy danych dodać tabelę:

W Eksploratorze serwera zaznaczmy pozycję Tabele (Tables) i z jej menu kontekstowego wybierzmy *Dodaj nową tabelę*. W głównej części okna pojawi się edytor pozwalający na zdefiniowanie pól nowej tabeli.



Zdefiniujmy tabelę, ustalając nazwy pól i ich typy zgodnie ze wzorem przedstawionym na rysunku

The screenshot shows the SSMS interface with the 'dbo.FakturySprzedazy [Design]' tab selected. On the left, the Object Explorer displays the database structure under 'firma.mdf'. The 'Tabele' node is expanded, showing 'FakturySprzedazy' and other tables like 'Pracownicy', 'Widoki', etc. The 'Script File' dropdown at the top right is set to 'dbo.Table.sql'. The table design grid has columns for Name, Data Type, Allow Nulls, and Default. The columns defined are:

Name	Data Type	Allow Nulls	Default
Id	int	<input type="checkbox"/>	
Numer	varchar(16)	<input type="checkbox"/>	
Netto	float	<input type="checkbox"/>	
VAT	float	<input checked="" type="checkbox"/>	
Data	date	<input type="checkbox"/>	
Zaplacono	float	<input checked="" type="checkbox"/>	
KontrahentId	int	<input type="checkbox"/>	
PracownikId	int	<input type="checkbox"/>	

Below the design grid is the T-SQL code for creating the table:

```
1 CREATE TABLE [dbo].[FakturySprzedazy]
2 (
3     [Id] INT NOT NULL PRIMARY KEY,
4     [Numer] VARCHAR(16) NOT NULL,
5     [Netto] FLOAT NOT NULL,
6     [VAT] FLOAT NULL,
7     [Data] DATE NOT NULL,
8     [Zaplacono] FLOAT NULL,
9     [KontrahentId] INT NOT NULL,
10    [PracownikId] INT NOT NULL
11 )
```

Jedno z pól powinno być kluczem głównym. Domyślnie jest nim automatycznie dodane do tabeli pole *Id* (ikona z kluczem przy tym polu).

Przy polach *Zaplacono* oraz *Vat* pozostawiamy zaznaczoną opcję *Allow Nulls*. Wszystkie pozostałe będą musiały posiadać wartości.

Pod edytorem tabeli widoczny jest kod T-SQL, który powstaje w konsekwencji definiowania kolejnych kolumn.

Zmieńmy nazwę tabeli w kodzie T-SQL z *Table* na *fakturySprzedazy*

Aby utworzyć tabelę, kliknijmy przycisk *Update* nad edytorem tabeli.

Potwierdzmy, klikając *Update Database*.

W podoknie *Eksplorator serwera* w gałęzi *Tabele* zobaczymy tabelę *FakturySprzedazy*.

Postępując podobnie dodajmy do bazy danych firma.mdf jeszcze dwie tabele: Pracownicy i Kontrahenci

The screenshot shows the SQL Server Management Studio interface with four windows:

- dbo.Pracownicy [Data]**: Shows the table structure with columns: Id (int), Imię (varchar(32)), Nazwisko (varchar(32)), Email (varchar(32)), and Telefon (varchar(16)).
- dbo.Kontrahenci [Data]**: Shows the table structure with columns: Id (int), Nazwa (varchar(64)), Nip (varchar(16)), Ulica (varchar(32)), and Miasto (varchar(32)).
- dbo.Faktury-Sprzedaży [Data]**: A blank table structure window.
- dbo.Kontrahenci [Design]**: A table design window for the Kontrahenci table.

Below these windows, there are two code editors:

- dbo.Table\_1.sql**: T-SQL script for creating the Pracownicy table.
- dbo.Table\_2.sql\***: T-SQL script for creating the Kontrahenci table.

```

CREATE TABLE [dbo].[Pracownicy]
(
    [Id] INT NOT NULL PRIMARY KEY,
    [Imię] VARCHAR(32) NOT NULL,
    [Nazwisko] VARCHAR(32) NOT NULL,
    [Email] VARCHAR(32) NOT NULL,
    [Telefon] VARCHAR(16) NOT NULL
)

CREATE TABLE [dbo].[Kontrahenci]
(
    [Id] INT NOT NULL PRIMARY KEY,
    [Nazwa] VARCHAR(64) NOT NULL,
    [Nip] VARCHAR(16) NOT NULL,
    [Ulica] VARCHAR(32) NULL,
    [Miasto] VARCHAR(32) NOT NULL
)

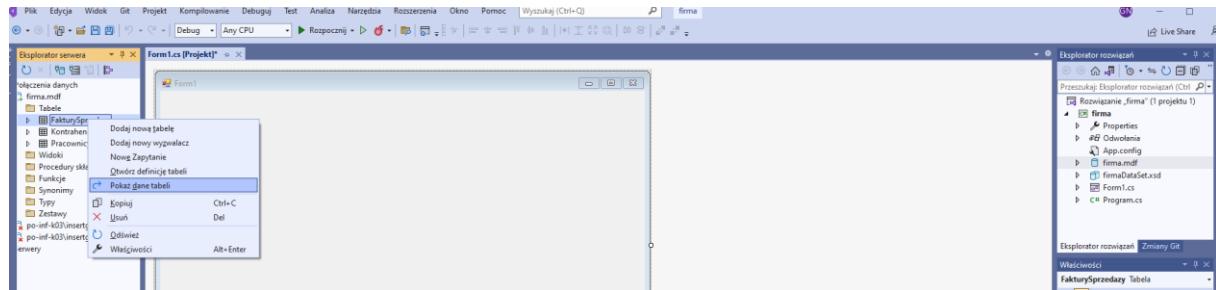
```

## Edycja danych w tabeli

Następnym krokiem może być wypełnienie tabeli przykładowymi danymi:

W Eksploratorze serwera rozwińmy gałąź *Tabele* i zaznaczmy pozycję *fakturySprzedazy*.

Z jej menu kontekstowego wybierzmy polecenie *Pokaż dane tabeli*.



Zobaczmy siatkę pozwalającą na edycję danych. Umieścmy w tabeli kilka przykładowych rekordów

	Id	Numer	Netto	VAT	Data	Zaplacono	KontrahentId	PracownikId
	1	V/01/23	1000	230	10.01.2023	1000	1	1
	2	V/02/23	500	115	12.01.2023	2000	1	2
	3	V/03/23	2000	0	12.01.2023	NULL	2	1
	4	V/04/23	10000	NULL	15.01.2023	NULL	2	2
	5	V/05/23	800	208	16.01.2023	1008	3	1
	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Także pozostałe dwie tabele wypełnijmy przykładowymi danymi

	Id	Nazwa	Nip	Ulica	Miasto
	1	Koala	777-222-33-45	Krótką 23	Poznań
	2	Albeco	555-888-12-45	Długa 12	Poznań
	NULL	NULL	NULL	NULL	NULL

	Id	Imię	Nazwisko	Email	Telefon
	1	Jan	Kowalski	jan.kowalski@...	600500800
	2	Piotr	Nowak	piotr.nowa@w...	600400800
	NULL	NULL	NULL	NULL	NULL

## Procedura składowana — pobieranie danych

Dodatkowo umieścmy w bazie danych w tabeli fakturySprzedazy procedury składowane. Pierwsza będzie zapytaniem służącym do pobierania danych, a druga posłuży do ich modyfikacji.

Zacznijmy od zapytania SQL pobierającego listę faktur niezapłaconych zapisanych w tabeli fakturySprzedazy. W tym celu w Eksplorator serwera, przejdźmy do pozycji *Procedury składowania* i prawym przyciskiem myszy rozwińmy jej menu kontekstowe. Wybierzmy z niego polecenie *Dodaj nową procedurę składową*.



Pojawi się edytor, w którym możemy wpisać kod SQL.

Wpiszmy proste zapytanie

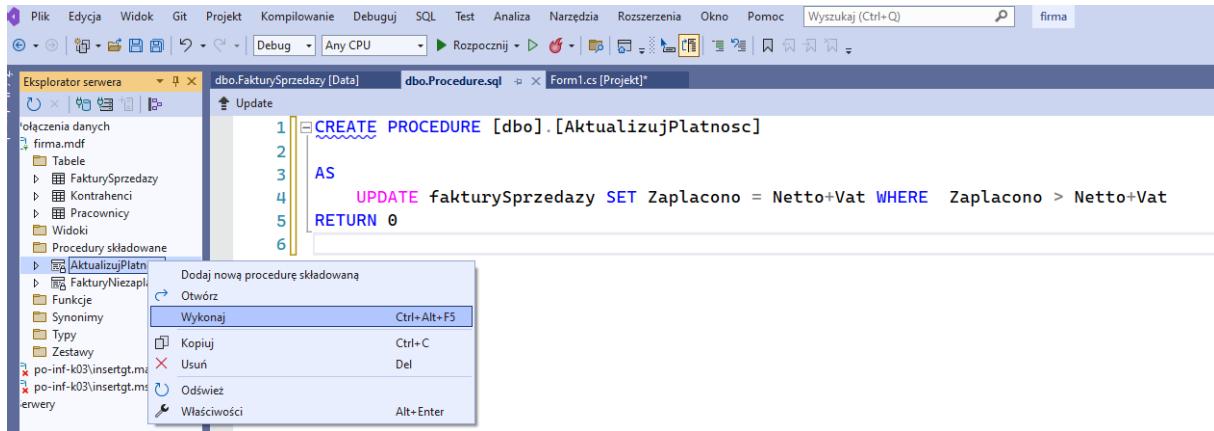
```
CREATE PROCEDURE [dbo].[FakturyNiezaplacone]
/*
    @param1 int = 0,
    @param2 int
*/
AS
    SELECT * FROM fakturySprzedazy WHERE netto+vat>zaplacono
RETURN 0
```

Zmieńmy też nazwę procedury na [dbo].[FakturyNiezaplacone].

Usuńmy też argumenty procedury. Zapiszmy zmiany w bazie danych, klikając przycisk *Update*. Procedurę tę można uruchomić z poziomu Eksploratora serwera. Wystarczy z jej menu kontekstowego wybrać polecenie *Wykonaj*. Pojawi się skrypt SQL (na jego pasku narzędzi jest ikona pozwalająca na ponowne uruchomienie), a pod nim dane będące wynikiem zapytania. Przygotujmy także procedurę składowaną aktualizującą pole Zaplacono (jeżeli kwota zapłacona za fakturę jest większa niż kwota faktury, to pole Zaplacono = Netto + Vat)

```
CREATE PROCEDURE [dbo].[AktualizujPlatnosc]
AS
    UPDATE fakturySprzedazy SET Zaplacono = Netto+Vat WHERE Zaplacono > Netto+Vat
RETURN 0
```

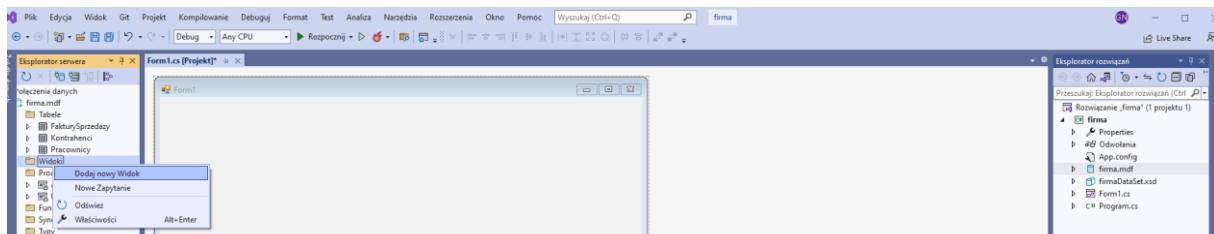
Sprawdźmy jej działanie polecienniem Wykonaj z menu kontekstowego. Zmiany zostaną wprowadzone do bazy danych.



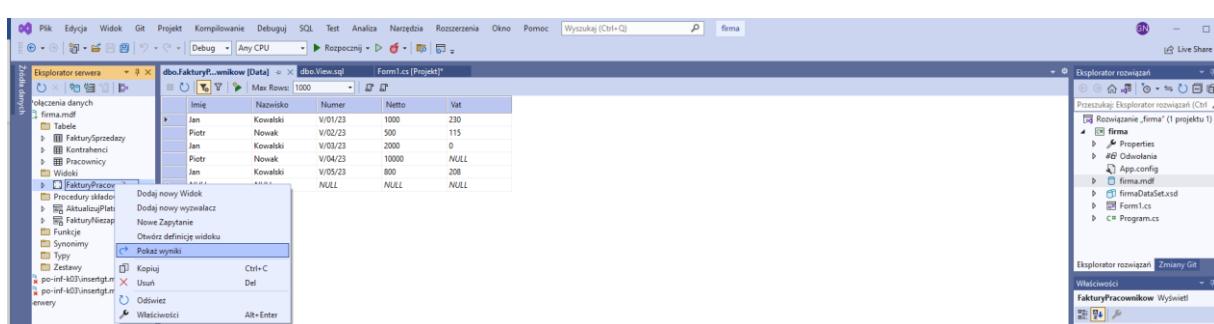
## Widok

Zamiast procedury składowanej będącej zapytaniem SQL do pobierania danych lepiej użyć widoku. Z punktu widzenia programisty widok zachowuje się tak jak tabela, ale tak naprawdę dostępne w nim dane są generowane dynamicznie. Można w ten sposób ograniczyć dostępność danych lub przygotować pseudotabelę łączącą dane z kilku tabel.

Utwórzmy widok udostępniający listę faktur łącznie z osobami, które je wystawiły.



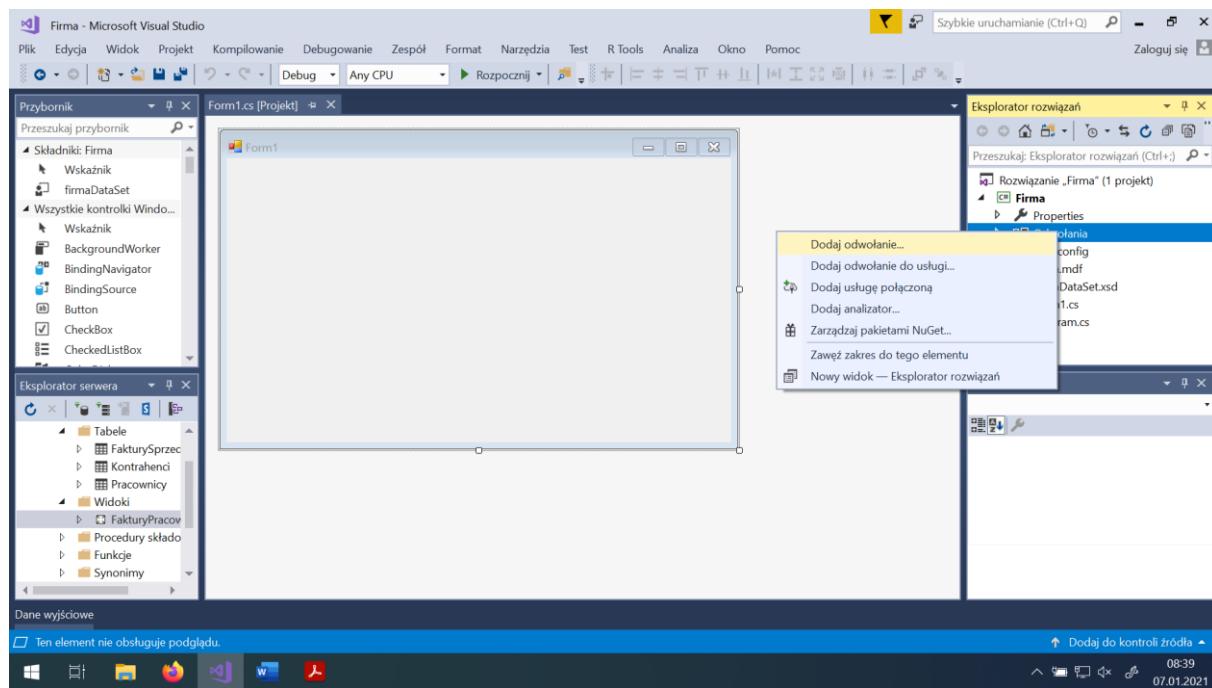
Sprawdźmy działanie widoku

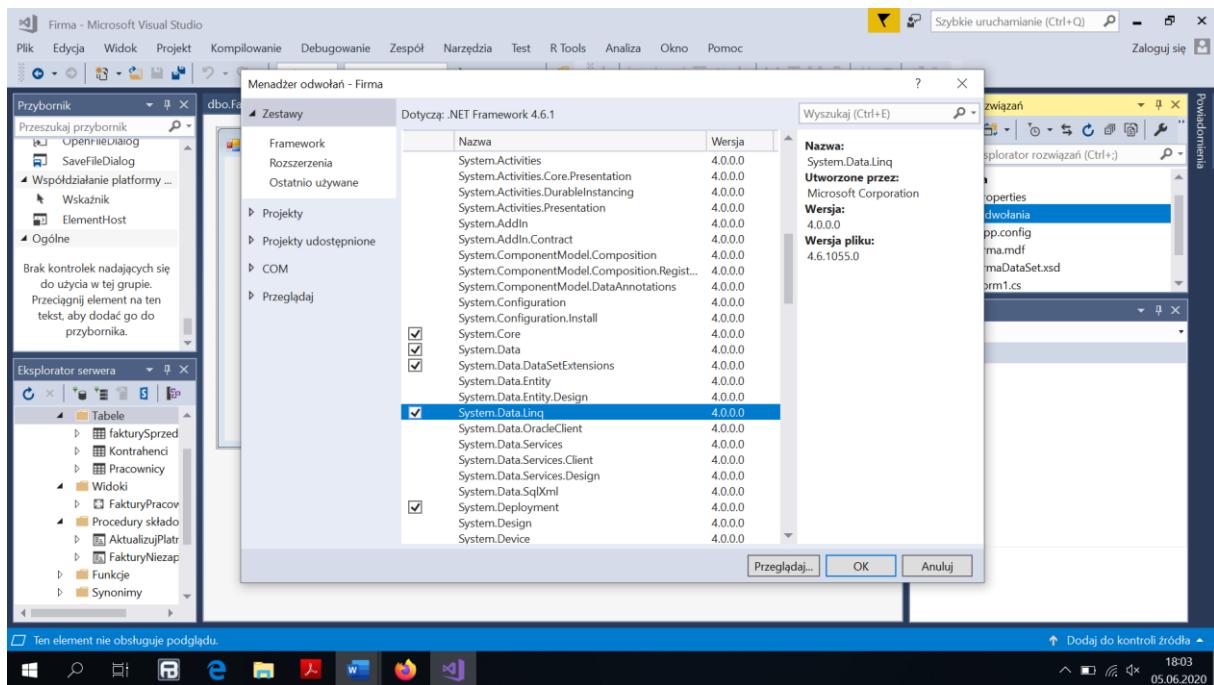


## Klasa encji

*Klasa encji* to zwykła klasa C#, w której pola klasy powiązane są za pomocą atrybutów z polami tabeli (kolumnami). Mamy więc do czynienia z modelowaniem zawartości relacyjnej bazy danych w typowych klasach języka, w którym przygotowujemy program. Dzięki tej relacji programista może w pewnym stopniu utożsamiać tę klasę z reprezentowaną przez nią tabelą. To pozwala również przygotowywać zapytania LINQ z wykorzystaniem nazw pól tabeli. Ich przetłumaczenie na zapytania SQL jest wtedy szczególnie proste, a jednocześnie w pełni zachowana zostaje kontrola typów. Domyślne wiązanie realizowane jest na podstawie nazw obu pól. Możliwa jest jednak zmiana tego domyślnego sposobu wiązania oraz zmiana wartości poszczególnych pól.

Atrybuty, których użyjemy do „oznakowania” klasy encji, należą do przestrzeni nazw System.Data.Linq.Mapping i umieszczone są w osobnej bibliotece platformy .NET o nazwie System.Data.Linq.dll. W przypadku „ręcznego” przygotowywania klasy encji, bibliotekę tę należy samodzielnie dodać do projektu





Utwórzmy nową klasę `FakturaSprzedazy`. Po dodaniu referencji należy zadeklarować użycie jej przestrzeni nazw (w `FakturaSprzedazy.cs`) instrukcją

```
using System.Data.Linq.Mapping;
```

Przed całą klasą encji powinien znaleźć się atrybut `Table`, w którym wskazujemy nazwę tabeli z bazy danych. Natomiast przed polami klasy odpowiadającymi kolumnom z tabeli należy umieścić atrybut `Column`, w którym możemy wskazać m.in. nazwę kolumny w tabeli (parametr `Name`), zaznaczyć, że kolumna ta jest kluczem głównym (`IsPrimaryKey`) lub że może przyjmować puste wartości (`CanBeNull`). Argumentu `Name` używam tylko przy polu `Id`. Nie jest on konieczny, jeżeli nazwy kolumn są takie same jak nazwy pól w klasie encji.

```
using System.Data.Linq.Mapping;

namespace firma
{
    [Table(Name = "FakturySprzedazy")]
    internal class FakturaSprzedazy
    {
        [Column(Name = "Id", IsPrimaryKey = true, CanBeNull = false)] public int Id;
        [Column(CanBeNull = false)] public string Numer;
        [Column(CanBeNull = false)] public double Netto;
        [Column(CanBeNull = true)] public double Vat;
        [Column(CanBeNull = true)] public double Zaplacono;
        [Column(CanBeNull = false)] public DateTime Data;
        [Column(CanBeNull = false)] public int KontrahentId;
        [Column(CanBeNull = false)] public int PracownikId;
    }
}
```

Definiując klasę encji, należy zadeklarować odpowiadające im pola klasy w taki sposób, aby dopuszczały przypisanie wartości null. W przypadku łańcuchów nie ma problemu. Typ String jest typem referencyjnym i zawsze można mu przypisać wartość null. Inaczej wygląda to np. w przypadku np. typu int, który jest typem wartościowym. Należy wówczas w deklaracji pola wykorzystać typ parametryczny Nullable<int>, który równoważnie może być zapisany jako int?

Wiązanie klasy z tabelą bazy danych przeprowadzane zostaje na podstawie atrybutów dołączanych do definicji klasy. W terminologii Microsoft nazywane jest to *mapowaniem opartym na atrybutach*. Możliwe jest również podejście alternatywne, w którym mapowanie odbywa się na podstawie struktury zapisanej w pliku XML. Takie podejście nosi nazwę mapowania zewnętrznego.

W analogiczny sposób utwórzmy klasy Kontrahent i Pracownik i powiążmy je z tabelami Kontrahenci i Pracownicy

```
using System.Data.Linq.Mapping;

namespace Firma
{
    [Table(Name = "Pracownicy")]
    class Pracownik
    {
        [Column(Name = "Id", IsPrimaryKey = true, CanBeNull = false)] public int Id;
        [Column(CanBeNull = false)] public string Imie;
        [Column(CanBeNull = false)] public string Nazwisko;
        [Column(CanBeNull = false)] public string Email;
        [Column(CanBeNull = false)] public string Telefon;
    }
}

using System.Data.Linq.Mapping;

namespace Firma
{
    [Table(Name = "Kontrahenci")]
    class Kontrahent
    {
        [Column(Name = "Id", IsPrimaryKey = true, CanBeNull = false)] public int Id;
        [Column(CanBeNull = false)] public string Nazwa;
        [Column(CanBeNull = true)] public string Nip;
        [Column(CanBeNull = false)] public string Ulica;
        [Column(CanBeNull = false)] public string Miasto;
    }
}
```

## Pobieranie danych

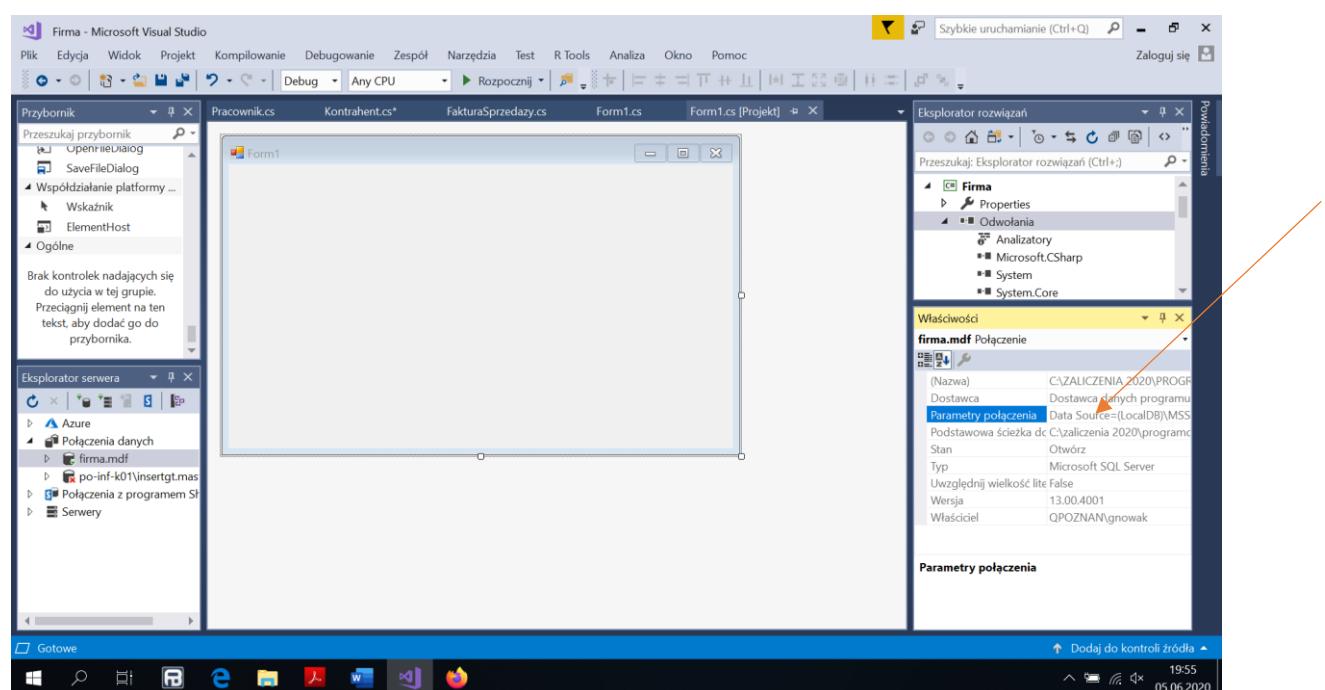
Do projektowania klasy encji wykorzystywany jest edytor O/R Designer.

Zrobimy to jednak samodzielnie (bez edytora).

Zdefiniujmy pola przechowujące referencje do instancji klasy `DataContext` i jej tabeli `Pracownicy`:

```
Data Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename="C:\zaliczenia  
2022\programowanie obiektowe\baza5\firma\firma.mdf";Integrated  
Security=True;Connect Timeout=30  
  
const string connectionString = @"Data Source=(LocalDB)\ MSSQLLocalDB;  
AttachDbFilename=C:\zaliczenia 2022\programowanie  
obiektowe\baza5\firma\firma.mdf;Integrated Security=True";  
static DataContext bazaDanychFirma = new DataContext(connectionString);  
static Table<Pracownik> listaPracownikow = bazaDanychFirma.GetTable<Pracownik>();
```

Wartość pola `connectionString` możemy odczytać z:



Tworzymy obiekt `DataContext` i pobieramy z niego referencję do tabeli, a ściślej jej reprezentacji w LINQ to SQL, czyli klasy `Table` parametryzowanej klasą encji `Pracownik`.

Klasa `DataContext` znajduje się w przestrzeni nazw `System.Data.Linq`, należy więc uwzględnić ją w grupie dyrektywy `using`.

```
using System.Data.Linq;
```

Konstruktor klasy `DataContext` wymaga podania łańcucha połączenia konfiguruującego połączenie z bazą danych.

Następnie tworzymy sam obiekt `DataContext` reprezentujący bazę danych oraz tworzymy pole o nazwie `listaPracownikow` reprezentujące tabelę i umożliwiające dostęp do danych pobranych z tabeli `Pracownicy`. Pole to stanowi kluczową informację. Będzie mogło być używane jako źródło danych w zapytaniach LINQ. Nazwa pobieranej tabeli wskazana została w atrybucie `Table`, którym poprzedziliśmy klasę encji.

Cały wpisany fragment kodu:

```
using System.Data.Linq;  
  
namespace Firma  
{  
    public partial class Form1 : Form  
    {  
        const string connectionString = @"Data Source=(LocalDB)\  
MSSQLLocalDB;AttachDbFilename= C:\zaliczenia 2020\programowanie  
obiektowe\baza4\Firma\Firma\firma.mdf;Integrated Security=True";  
        static DataContext bazaDanychFirma = new DataContext(connectionString);  
        static Table<Pracownik> listaPracownikow =  
bazaDanychFirma.GetTable<Pracownik>();  
  
        public Form1()  
        {  
            InitializeComponent();  
        }  
    }  
}
```

Obiekt typu `Table<Pracownik>` może być źródłem w zapytaniach LINQ. Dzięki temu możemy swobodnie pobrać interesujące nas dane z tabeli.

Umieścmy na formie button i utworzymy metodę zdarzeniową przycisku (kliknąć na nim dwa razy)

```
string s = "Lista pracowników: " + "\n";  
var lp = from Pracownik in listaPracownikow  
        orderby Pracownik.Nazwisko  
        select Pracownik;  
foreach (Pracownik pracownik in lp)  
    s += pracownik.Imie + " " + pracownik.Nazwisko + " " + "\n";  
MessageBox.Show(s);
```

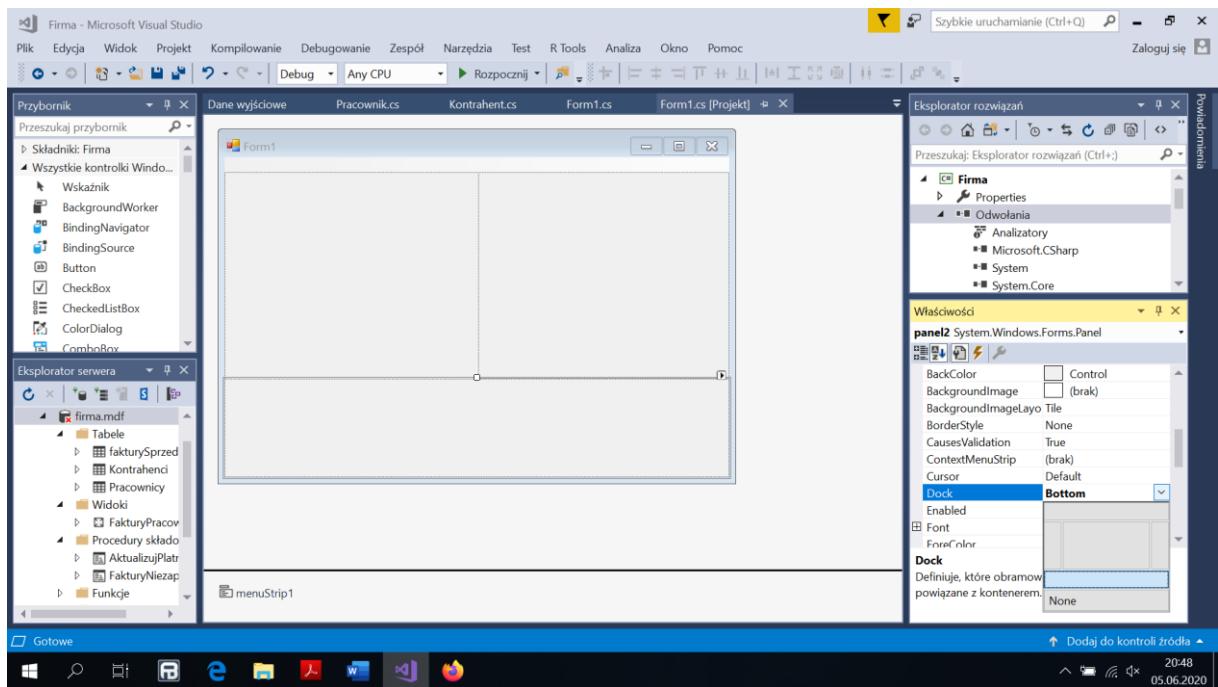
### Zapytanie LINQ

```
var lp = from Pracownik in listaPracownikow orderby Pracownik.Nazwisko  
        select Pracownik;
```

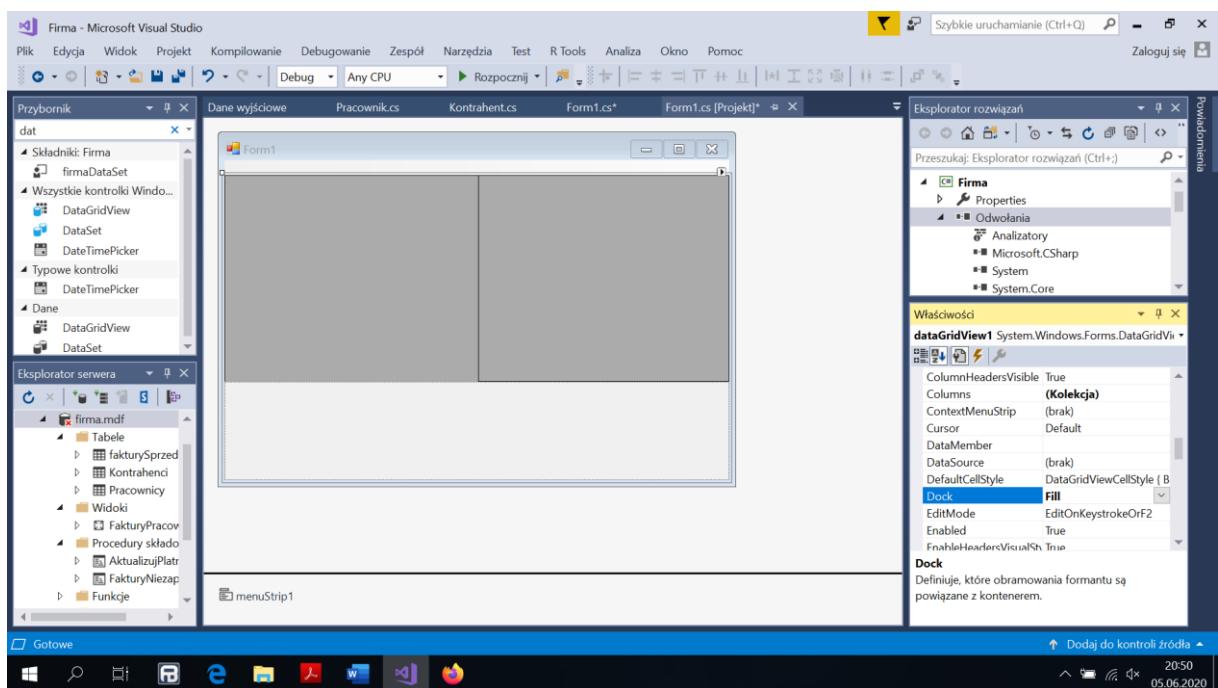
pobiera z tabeli listę osób i sortuje ich nazwiska alfabetycznie.

Oczywiście pobrane dane można prezentować nie tylko w okienkach komunikatu. Wygodniej użyć do tego kontrolki. Na przykład na siatce, czyli kontrolce `DataGridView`

Zaprojektujmy jednak najpierw formę. Umieścmy trzy panele na formie. Własność Dock Paneli ustawmy kolejno na: Bottom, Right, Fill



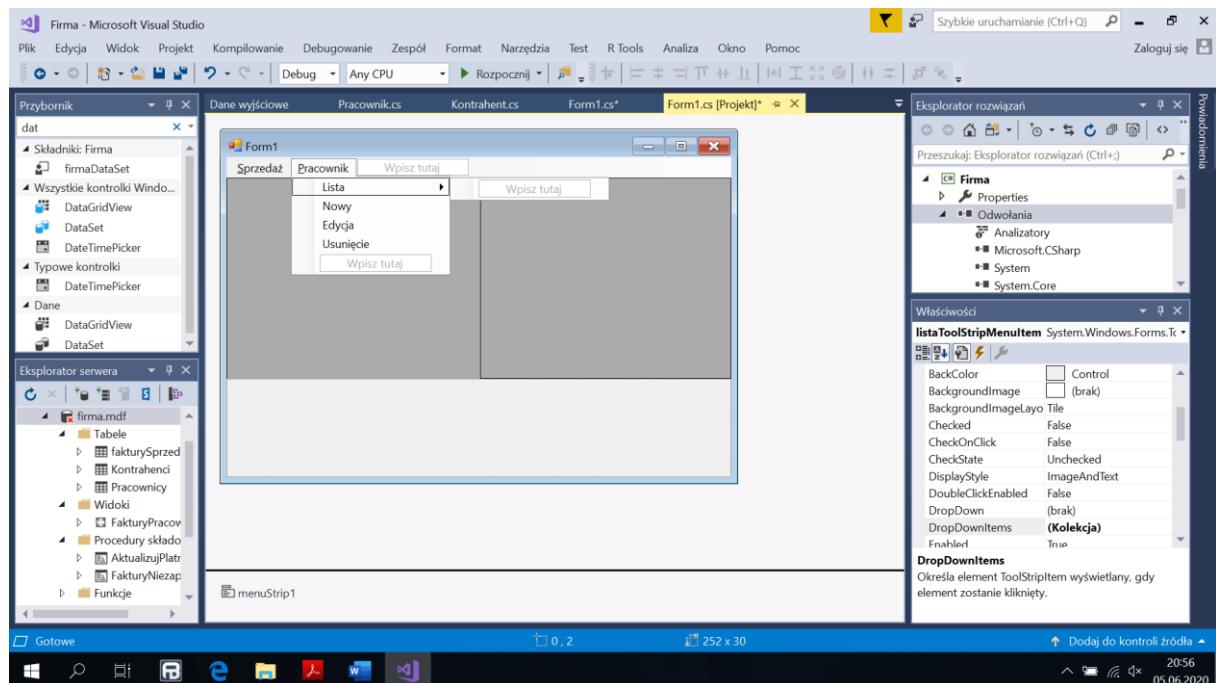
Umieścmy na panelu 1 i 2 siatkę, czyli kontrolkę DataGridView  
Ustawmy własność Dock obu DataGridView na Fill.



W widoku projektowania, zaznacz komponent MenuStrip i umieść go na formie. Komponent znajdzie się na dodatkowym pasku pod podglądem formy. Ten komponent ma także reprezentację na podglądzie formy - jest to zarazem edytor menu.

Wprowadźmy nazwę pierwszego podmenu: &Sprzedaż. Po rozpoczęciu pisania natychmiast pojawią się dodatkowe miejsca pozwalające na utworzenie kolejnego podmenu oraz pierwszej pozycji w podmenu *Sprzedaż*.

Następnie dodaj podmenu &*Pracownik* z pozycjami *Lista*, *Nowy*, *Edycja*, *Usunięcie*.



Umieszczony w nazwie pozycji menu znak & wskazuje na aktywny klawisz (oznaczona w ten sposób litera będzie podkreślona po naciśnięciu lewego klawisza *Alt* i uaktywnieniu menu).

Klawisze aktywne pozwalają na nawigację w menu bez użycia myszy - wystarczy nacisnąć klawisz odpowiadający podkreślonej literze, żeby uruchomić związaną z nią metodę zdarzeniową.

Prezentacja danych w siatce DataGridView

W metodzie zdarzeniowej Lista umieścmy kod:

```
// pobieranie kolekcji
var lp = from Pracownik in listaPracownikow
         orderby Pracownik.Nazwisko
         select new
        {
            Pracownik.Id,
            Pracownik.Imie,
            Pracownik.Nazwisko,
            Pracownik.Telefon,
            Pracownik.Email
        };
dataGridView1.DataSource = lp;
```

W zapytaniu pobieramy rekordy odpowiadające osobom. Wynik zapytania, czyli kolekcję, wskazujemy następnie jako źródło danych kontrolki.

Wymusiliśmy skopiowanie danych, dzięki użyciu typu anonimowego w zwracanej kolekcji. Bez tego nie uzyskalibyśmy pożądanego efektu.

## Aktualizacja danych w bazie

Pobieranie danych, wizytówka LINQ, to zwykle pierwsza czynność wykonywana przez aplikacje. Jednak technologia LINQ to SQL nie poprzestaje tylko na tym. Zmiany wprowadzone w pobranej zapytaniem LINQ kolekcji można w łatwy sposób przesłać z powrotem do bazy danych. Służy do tego metoda SubmitChanges obiektu DataContext. Tym samym wszelkie modyfikacje danych stają się bardzo naturalne: nie ma konieczności przygotowywania poleceń SQL odpowiedzialnych za aktualizację tabel w bazie danych - wszystkie operacje wykonujemy na obiektach C#. Zachowana jest w ten sposób spójność programu, co pozwala na kontrolę typów i pełną weryfikację kodu już w trakcie komplikacji.

## Dodawanie i usuwanie rekordów

Co jeszcze możemy zmienić w tabeli? Ważna jest możliwość dodawania nowych i usuwania istniejących rekordów. Również to zadanie jest dzięki LINQ to SQL bardzo proste. W tym przypadku zmiany muszą być jednak wprowadzane wprost w obiekcie reprezentującym tabelę - w instancji klasy DataContext, a nie w kolekcji pobranej z niego zapytaniem LINQ.

Wyznaczamy wartość pola Id dla nowego rekordu, dodając jednośc do największej wartości tego pola odczytanej z tabeli - korzystamy przy tym z metody rozszerzającej Max. Następnie tworzymy obiekt typu Pracownik i poleciem InsertOnSubmit dodajemy go do tabeli. Rzeczywista zmiana nastąpi w momencie najbliższego wywołania metody SubmitChanges obiektu DataContext.

```
// dodawanie osoby do tabeli
int noweId = listaPracownikow.Max(Pracownik => Pracownik.Id)+1;
Pracownik nowyPracownik = new Pracownik
{
    Id = noweId,
    Imie = "Anna",
    Nazwisko = "Kaczmarek",
    Email = "A.Kaczmarek@wp.pl",
    Telefon = "100100100",
};
listaPracownikow.InsertOnSubmit(nowyPracownik);
bazaDanychFirma.SubmitChanges(); // zapisywanie zmian
listaToolStripMenuItem.Click(this, null); // odświeżenie siatki
```

Równie łatwo usunąć z tabeli rekord lub całą grupę rekordów. Należy tylko zdobyć referencję do odpowiadającego im obiektu (względnie grupy obiektów). Dla pojedynczego rekordu należy użyć metody DeleteOnSubmit, dla kolekcji — DeleteAllOnSubmit. W obu przypadkach rekordy zostaną oznaczone jako przeznaczone do usunięcia i rzeczywiście usunięte z bazy danych przy najbliższym wywołaniu metody SubmitChanges.

```

// odczytujemy wiersz, w którym jest pozycja do usunięcia
int wiersz = Convert.ToInt32(dataGridView1.CurrentRow.Index);
// odczytujemy id osoby do usunięcia
int idPracownik =
Convert.ToInt32(dataGridView1.Rows[wiersz].Cells[0].Value.ToString());
// wybieranie elementów do usunięcia i ich oznaczanie
IEnumerable<Pracownik> doSkasowania = from pracownik in listaPracownikow
                                         where pracownik.Id == idPracownik
                                         select pracownik;
listaPracownikow.DeleteAllOnSubmit(doSkasowania);
// zapisywanie zmian
bazaDanychFirma.SubmitChanges();
// wyświetlanie tabeli
listaToolStripMenuItem_Click(this, null);

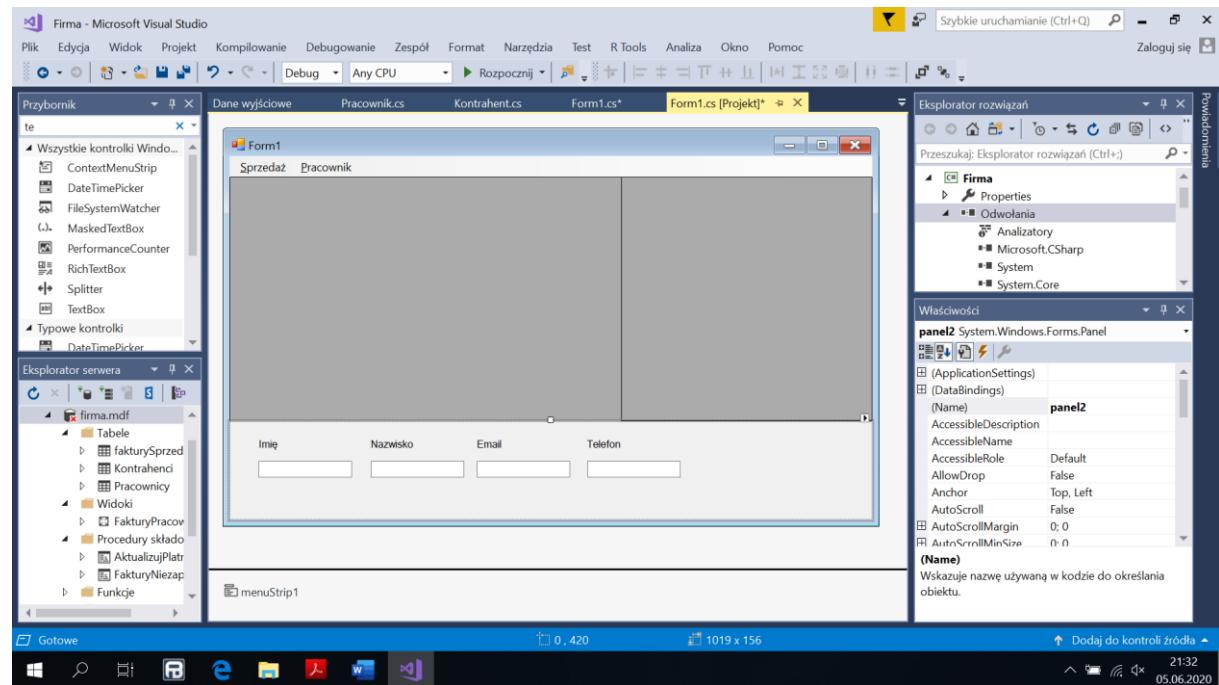
```

Zmodyfikujmy metodę dodawania rekordów

Dodajmy 4 Labelle i 4 TextBox (jak na rysunku).

Dla Labeli zmieńmy pole Text odpowiednio na: Imię, Nazwisko, Email, Telefon.

Można także zmienić pole Name dla TextBox odpowiednio na: textBoxImie, textBoxNazwisko, textBoxEmail, textBoxTelefon.



W metodzie zdarzeniowej Click DataGridView1

```
int wiersz = Convert.ToInt32(dataGridView1.CurrentRow.Index);
string imie = dataGridView1.Rows[wiersz].Cells[1].Value.ToString();
string nazwisko = dataGridView1.Rows[wiersz].Cells[2].Value.ToString();
string email = dataGridView1.Rows[wiersz].Cells[3].Value.ToString();
string telefon = dataGridView1.Rows[wiersz].Cells[1].Value.ToString();

textBoxImie.Text = imie;
textBoxNazwisko.Text = nazwisko;
textBoxEmail.Text = email;
textBoxTelefon.Text = telefon;
```

Najpierw odczytujemy ze StringGrida odpowiednie pola, a następnie umieszczamy je w TextBox.

Zmodyfikujmy zdarzenie dodawania pracownika (dane nowego pracownika odczytujemy z TextBox)

```
// dodawanie osoby do tabeli
int noweId = listaPracownikow.Max(Pracownik => Pracownik.Id)+1;
Pracownik nowyPracownik = new Pracownik
{
    Id = noweId,
    Imie = textBoxImie.Text,
    Nazwisko = textBoxNazwisko.Text,
    Email = textBoxEmail.Text,
    Telefon = textBoxTelefon.Text,
};
listaPracownikow.InsertOnSubmit(nowyPracownik);
// zapisywanie zmian
bazaDanychFirma.SubmitChanges(); // w bazie dodawany jest nowy rekord
listaToolStripMenuItem.Click(this, null); // odświeżenie siatki
```

## Modyfikacje istniejących rekordów

Aby zapisać nowe wartości do pliku bazy danych, wystarczy tylko wywołać metodę SubmitChanges na rzecz obiektu bazaDanychFirma.

```
int wiersz = Convert.ToInt32(dataGridView1.CurrentRow.Index);
int idPracownik =
Convert.ToInt32(dataGridView1.Rows[wiersz].Cells[0].Value.ToString());
foreach (Pracownik pracownik in listaPracownikow)
{
    if (pracownik.Id == idPracownik)
    {
        pracownik.Imie = textBoxImie.Text;
        pracownik.Nazwisko = textBoxNazwisko.Text;
        pracownik.Email = textBoxEmail.Text;
        pracownik.Telefon = textBoxTelefon.Text;
    }
}
// zapisywanie zmian
bazaDanychFirma.SubmitChanges();
listaToolStripMenuItem.Click(this, null);
```

Kolekcja listaPracownikow przechowuje tylko referencje do obiektów tabeli, a nie kopie tych obiektów. Tak więc wszystkie zmiany wprowadzane do kolekcji automatycznie wprowadzane są w buforze obiektu bazaDanychFirma, przechowującym dane pobrane z bazy danych. Obiekt ten potrafi zapisać je z powrotem do tabeli bazy SQL Server. Warto przy tym pamiętać, że wszystkie zmiany wprowadzane w kolekcji są przechowywane w niej aż do momentu wywołania metody SubmitChanges. Możemy więc dowolnie (wielokrotnie) modyfikować kolekcję bez obawy, że nadużywamy zasobów bazy danych i komputera, obniżając tym samym wydajność programu.

Zmodyfikujmy jeszcze zdarzenie `dataGridView1_Click` tak, żeby w drugim DataGridView wyświetlały się faktury wystawione przez tę osobę.

Najpierw dodajmy pola przechowujące referencje do instancji klasy `DataContext` i jej tabeli `FakturaSprzedazy`:

```
static Table<FakturySprzedazy> listaFaktursprzedazy =
    bazaDanychFirma.GetTable<FakturySprzedazy>();
```

Następnie zmodyfikujmy zdarzenie Click DataGridView1

```
private void dataGridView1_Click(object sender, EventArgs e)
{
    int wiersz = Convert.ToInt32(dataGridView1.CurrentRow.Index);
    string imie = dataGridView1.Rows[wiersz].Cells[1].Value.ToString();
    string nazwisko = dataGridView1.Rows[wiersz].Cells[2].Value.ToString();
    string email = dataGridView1.Rows[wiersz].Cells[3].Value.ToString();
    string telefon = dataGridView1.Rows[wiersz].Cells[1].Value.ToString();
    int idPracownik =
Convert.ToInt32(dataGridView1.Rows[wiersz].Cells[0].Value.ToString());
    textBoxImie.Text = imie;
    textBoxNazwisko.Text = nazwisko;
    textBoxEmail.Text = email;
    textBoxTelefon.Text = telefon;

    var ls = from FakturySprzedazy in listaFaktursprzedazy
        where (FakturySprzedazy.PracownikId == idPracownik)
        select new
        {
            FakturySprzedazy.Id,
            FakturySprzedazy.Numer,
            FakturySprzedazy.Netto,
            FakturySprzedazy.Vat,
            FakturySprzedazy.Data,
            FakturySprzedazy.Zaplacono
        };
    dataGridView2.DataSource = ls;
}
```

Zdarzenie to możemy przypisać także do innych metod zdarzeniowych DataGridView1 (np. do KeyUp – zwolnienia klawisza)

```
dataGridView1_Click(this, null);
```

- 1. Uzupełnić projekt o możliwość wystawiania faktur i dopisywania dostawcy.**
- 2. Uzupełnić projekt o informację na DataGridView o fakturach nieuregulowanych oraz o fakturach danego dostawcy.**