Microsoft

Stopařův průvodce po ADO.NET a LINQ aneb Foxařův průvodce po práci s daty v .NET

Milan Kosina

Obsah

Úvod

Cílem této brožurky je poskytnout pomoc programátorům ve Visual FoxPro, kteří hledí s nedůvěrou na ostatní programovací nástroje a jsou přesvědčeni, že žádný jiný nástroj jim nemůže poskytnout takový komfort při práci s daty jako „stará dobrá Foxka“ ☺. Budu se snažit Vám (programátorům ve VFP) pomoci tím, že budu porovnávat způsob práce ve Visual FoxPro a v .NET. Budu používat jazyk C#, ale pokud byste chtěli používat VB.NET, nezoufejte – práce ve VB.NET je stejná jako v C#, jen syntaxe se trochu liší.

Z výše uvedeného je zřejmé, že pro čtení dalších kapitol je nezbytná znalost práce s Visual FoxPro. **Není** ale potřeba znát C#, neboť příklady budou psány tak, aby jim bylo možné rozumět i bez znalosti tohoto jazyka.

Co je nutné nainstalovat

Mým cílem bylo, abyste si mohli všechny příklady sami vyzkoušet – a to s minimem vynaložené námahy. Pro naše povídání proto bude třeba nainstalovat pouze následující programy (vše se dá pořídit zdarma):

- Data budou uložena na SQL Serveru, takže je nezbytné mít libovolnou verzi SQL Serveru - buď Express verzi (která je zdarma) nebo plnou verzi. Samozřejmě je možné použít i MSDE. Vzhledem k tomu, že většina z nás už nějakou (třeba Express) verzi na nějakém CD určitě má, URL neuvádím.
- Potřebujeme mít stejnou databázi pro naše příklady. Zkontrolujte proto, zda vaše instalace SQL Serveru obsahuje databázi NorthWind. Pokud ji neobsahuje, stáhněte si ji z http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en (1,5 MB) a nainstalujte ji podle pokynů v dokumentaci.
- Pro poslední kapitolku („Vytvoření malé aplikace“) budeme potřebovat Visual Studio 2008. Pro Vaše první pokusy bude určitě stačit Express verze, která je zdarma (http://www.microsoft.com/express/vcsharp/). Pro první dvě kapitoly („ADO.NET“ a „LINQ“) je třeba mít nainstalován minimálně .NET Framework 3.5 SP1, který lze stáhnout z http://download.microsoft.com/download/2/0/e/20e90413-712f-438c-988e-fdaa79a8ac3d/dotnetfx35.exe. (**Pozor!** Je to 230 MB.) Musí se jednat o verzi 3.5 - bude nás totiž zajímat LINQ.
- Vzhledem k tomu, že mým cílem je usnadnit vám pokusy s .NET, budeme v prvních dvou kapitolách pracovat v programu LINQPad, který je zdarma a můžete si ho stáhnout z http://www.linqpad.net/. Více si o tomto skvělém programu povíme v samostatné kapitole.

O čem se budeme bavit

Společně se podíváme na dva způsoby práce s daty: ADO.NET a LINQ.

ADO.NET (Active Data Objects for .NET): Umožňuje objektovou práci s relačními daty na nejnižší úrovni. *Překlad pro foxaře:* Odpovídá zhruba SQL-Pass-Through a vzdáleným pohledům. Rozdíl je v tom, že výsledkem není kursor, ale objekt vytvořený z nějaké třídy (podstatně bohatší než tabulka).

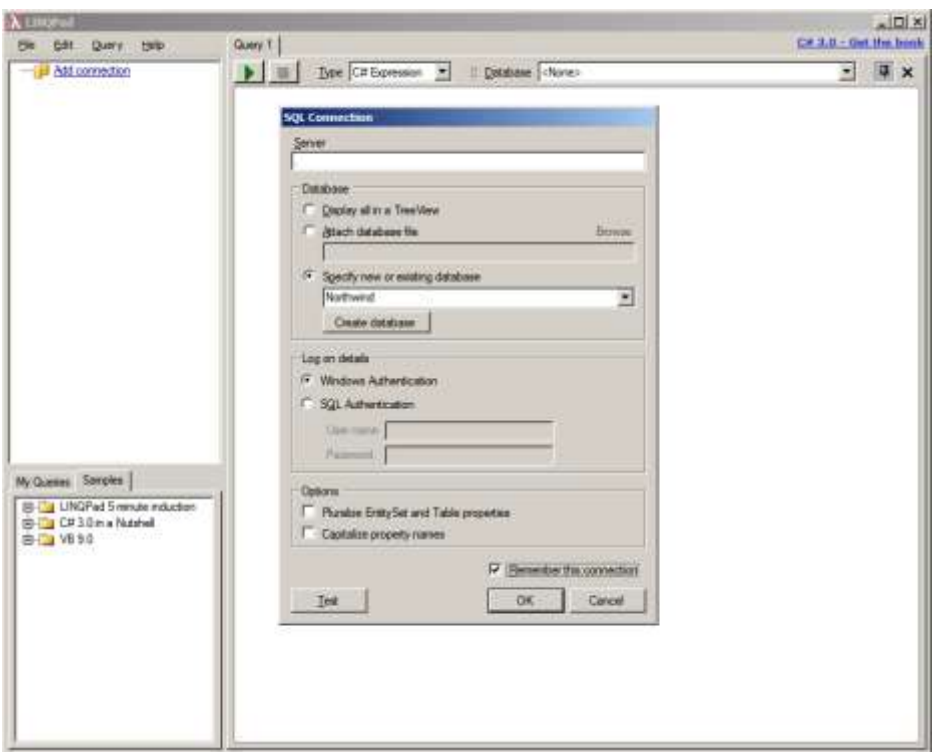
LINQ (Language INtegrated Query): Jazyk, který lze v .NET použít pro dotazy a aktualizace dat bez ohledu na to, odkud data pocházejí. LINQ To SQL interně používá ADO.NET. *Překlad pro foxaře:* Umožňuje klást dotazy psané stejným jazykem jak na data na SQL Serveru, tak na data v XML souboru, tak i na procesy běžící ve Windows apod.

Microsoft v současnosti více prosazuje pro práci se SQL Serverem a jinými databázemi svůj Entity Framework, který je složitější než LINQ, a slibuje, že ve verzi 2.0, na které pracuje, bude možno pracovat s databázemi stejně snadno jako v LINQ To SQL. Uvidíme, jak to dopadne, nicméně i v Entity Frameworku můžete používat LINQ To Entities… Navíc existují i konkurenční projekty jako nHibernate (s LINQ to nHibernate), LLBLGen Pro a další ORM (Object/Relation mapper).

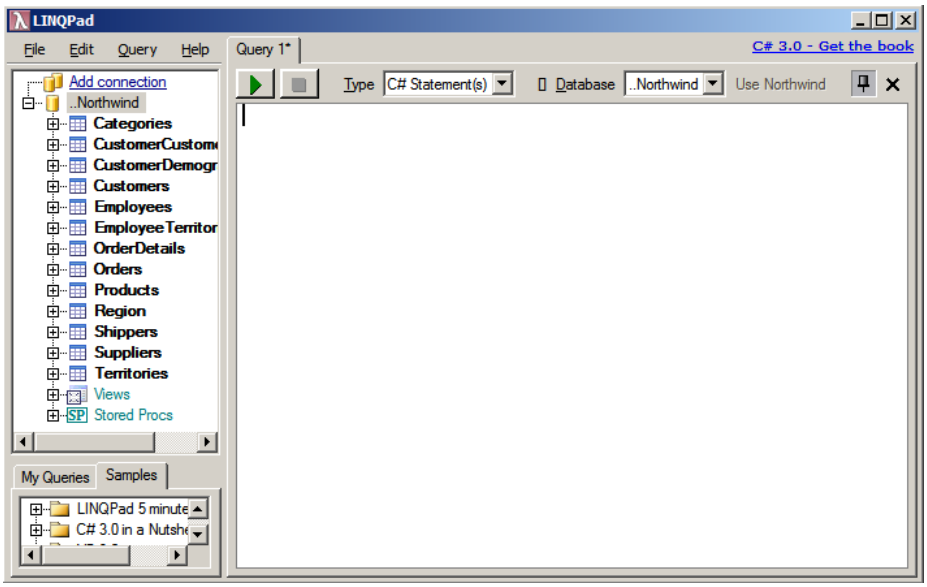
LINQPad

LINQPad je skvělý malý (2 MB) program, který je zcela zdarma. Dají se v něm snadno zkoušet kousky programů, aniž bychom museli mít nainstalované Visual Studio. To bude pro nás výhodné v prvních kapitolách, protože nebudeme muset psát „omáčku“ kolem testovaného kousku programu.
V závěrečné kapitole si ukážeme výhody Visual Studia na skutečných malých prográmcích.

Nejdůležitější je zadat správně připojení k SQL Serveru. Později už si bude LINQPad toto připojení pamatovat. Spusťte tedy LINQPad, klikněte na „Add connection“ a vyplňte dialogové okno tak, jak je uvedeno (tečka v Server znamená, že se jedná o server na tomto PC - je to stejné jako „(local)“):

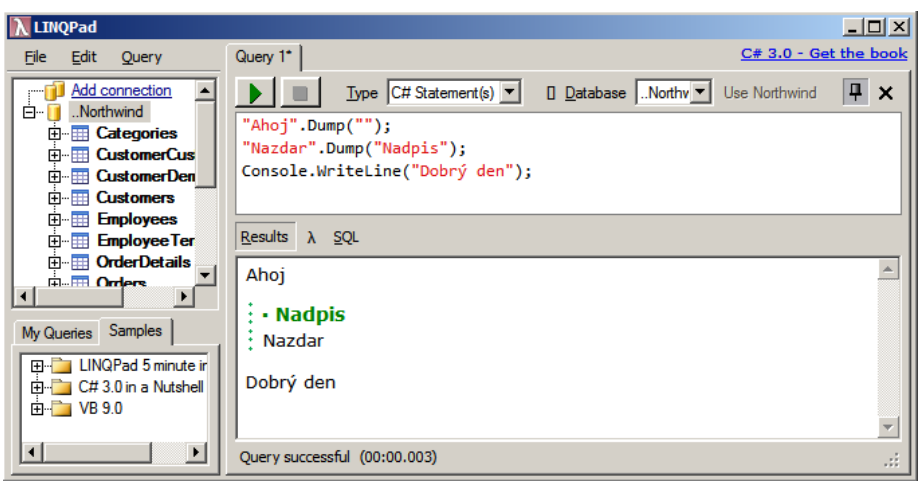


Změňte potom ještě Type na „C# Statement(s)“ a Database na „..Northwind“ a jste připraveni programovat.



Příkazy končí v C# středníkem. Pro jejich spuštění klikneme na zelenou šipku nebo stiskneme F5. Pro pomocné výpisy budeme používat dva způsoby:

- Prakticky za cokoli lze (pozor - pouze v LINQPadu!) připsat .Dump() nebo .Dump("Nadpis"). LINQPad pak zobrazí vše, co o daném objektu zjistí – případně včetně nadpisu (všimněte si svislé čáry, která spojuje nadpis a výpis).
- Console.WriteLine() nám umožňuje něco podobného jako „?“ ve Foxce.



Od této chvíle už budu ve screenshotech ukazovat pouze pravou stranu LINQPadu.

Mimochodem, LINQPad napsali autoři knihy „C# 3.0 in a nutshell“, kterou vám velmi doporučuji (viz „Použitá literatura“).

Malá poznámka: V současnosti je na webu novější verze LINQPadu než ta, se kterou jsem vytvářel tyto screenshoty.

ADO.NET

ADO.NET zhruba odpovídá tomu, jak je se SQL Serverem schopna pracovat Visual FoxPro. Nebudeme tedy dlouho teoretizovat a postupně si ukážeme, co je a co není v ADO.NET možné.

ADO.NET – Připojení

Připojení k SQL Serveru

Ve Visual FoxPro bychom k připojení použili

```
connString = ;
 "DRIVER=SQL Server;SERVER=(local);Trusted_Connection=Yes;Database= Northwind;"
nHandle = SQLSTRINGCONNECT(connString)
```

a k odpojení

```
SQLDISCONNECT(nHandle)
```

Vzhledem k tomu, že proces otevírání trvá dlouho, patrně otevřete spojení na začátku programu a zavřete na konci.

V .NET se otevírání a zavírání zapíše podobně:

```
string connString =
 "Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();    // Otevřeme spojení
    conn.Dump();    // Pracujeme. Zde si místo toho vypíšeme vlastnosti
    conn.Close();   // A zavřeme spojení!
```

}

```
string connString = "Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();    // otevřeme spojení
    conn.Dump();    // pracujeme... zde si místo toho vypíšeme vlastnosti
    conn.Close();   // a zavřeme spojení!
}
```

Results λ SQL

SqlConnection

System.Data.SqlClient.SqlConnection

StatisticsEnabled	False
ConnectionString	Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;
ConnectionTimeout	15
Database	Northwind
DataSource	(local)
PacketSize	8000
ServerVersion	09.00.2047
WorkstationId	GILGAMES
FireInfoMessageEventOnUserErrors	False
State	Open
Site	*null*
Container	*null*

Jediný rozdíl je tedy v tom, že se snažíme spojení uzavřít co nejrychleji. Proč? Protože se ve skutečnosti neuzavře ☺, Close() znamená pouze informaci Frameworku o tom, že toto spojení už může použít někdo jiný. Dochází tak k minimalizaci počtu připojení a zároveň k jejich efektivnímu využívání.

Poznámka: Od této chvíle už nebudu ve screenshotech ukazovat horní část, ale pouze část s výsledky běhu programu.

ADO.NET – základní stavební prvky

V následujících kapitolách budeme používat následující základní stavební kameny ADO.NET:

- SqlDataReader – z pohledu foxaře je to SQLExec, který umí stáhnout data – ale lze pouze jednou provést SCAN…ENDSCAN (toto procházení se samozřejmě nemusí provádět ručně). Obrovská výhoda tohoto přístupu je, že není potřeba mít v paměti všechna data, ale pouze jednu větu, takže paměťové nároky jsou minimální. Vzhledem k tomu, že po dobu zpracování je otevřené spojení na SQL Server, je potřeba zpracování provést co nejrychleji.
- DataSet a DataTable – z pohledu foxaře jsou to vylepšené vzdálené pohledy (pokud je plníme daty ze SQL Serveru pomocí SqlDataAdapter) nebo lokální pohledy (DBC a DBF – můžeme je vytvářet programově, ale existují pouze v paměti; i když se dají uložit na disk a načíst zpátky, nejsou k tomu primárně určeny).
- SqlDataAdapter – je vlastně implementace vzdálených pohledů – zajišťuje přenos dat mezi DataSety a SQL Serverem a jejich aktualizaci.

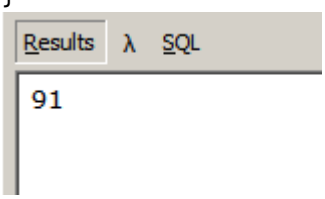
ADO.NET – SqlDataReader

Dotaz, který vrací 1 hodnotu

Ve Visual FoxPro nic takového neexistuje, SQLExec nám vždy vrací tabulku. Zde nám ADO.NET může vrátit přímo jednu hodnotu (v příkladu dále je to číslo). Postup:

- Pro dané připojení vytvoříme SQLCommand (ekvivalent SQLExec ve VFP)
- Zadáme příkaz, který chceme provést
- Zavoláme metodu ExecuteScalar() která nám vrátí jednu hodnotu

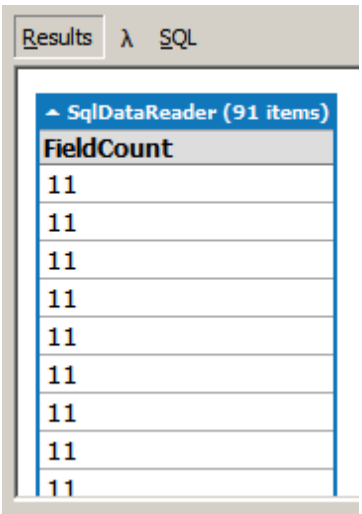
```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();

    SqlCommand cmd = conn.CreateCommand(); // budeme posílat příkaz po připojení
    cmd.CommandText = "SELECT COUNT(*) FROM dbo.Customers"; // zadáme SQL
    cmd.ExecuteScalar().Dump();            // provedeme

    conn.Close();
}
```



Dotaz, který vrací 1 tabulku (ale ne úplně)

A teď to, na co jistě čekáte – nebo vám to alespoň připadá ☺. Místo ExecuteScalar budeme chtít vrátit více dat – zavoláme tedy metodu ExecuteReader. Podívejte se na následující kód:

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();

    SqlCommand cmd = conn.CreateCommand();
    cmd.CommandText = "SELECT Customers.* FROM dbo.Customers";
    using (SqlDataReader rdr = cmd.ExecuteReader())
    {
        rdr.Dump();
    }

    conn.Close();
}
```



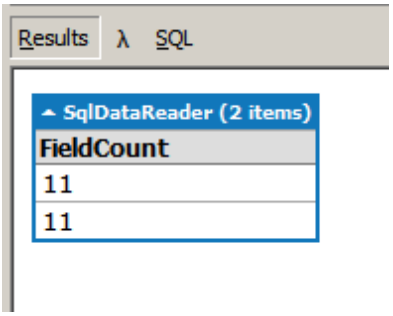
Výsledek tedy není to, co jsme očekávali (tabulka) – místo tabulky vidíme pouze počet vrácených položek v každé větě. SqlDataReader totiž umožňuje efektivnější zpracování vrácených dat – místo toho, aby naplnil celou tabulku, (což může zaplnit hodně MB), dovolí vám postupně procházet (SCAN…ENDSCAN) větu za větou a postupně všechny věty a jejich položky zpracovat. Dříve než si ukážeme, jak SqlReader používat, ukážeme si parametrizaci dotazu.

Dotaz s parametry

Parametr se v textu SQL příkazu označí pomocí „@“ (ve FoxPro to bylo pomocí „?“) a hodnota parametru se zadá například voláním metody AddWithValue() (tedy „přidej parametr a jeho hodnotu“) u kolekce všech existujících parametrů dotazu cmd.Parameters:

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();

    SqlCommand cmd = conn.CreateCommand();
    cmd.CommandText =
       "SELECT Customers.* FROM dbo.Customers WHERE Customers.Country = @Country";
    cmd.Parameters.AddWithValue("@Country", "SWEDEN");
    using (SqlDataReader rdr = cmd.ExecuteReader())
    {
        rdr.Dump();
    }

    conn.Close();
```

```
}
```



Vidíme, že se skutečně počet vybraných vět změnil (z 91 na 2).

Vlastní zpracování dotazu

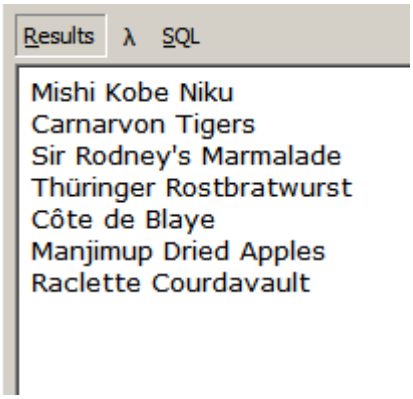
Máme tedy k dispozici SqlReader a potřebujeme projít všechny věty a z nich získat hodnoty jednotlivých položek.

Nejprve musíme znát mechanismus, jak postupně projít všechny věty. K tomu slouží volání metody rdr.Read() (přesně odpovídá SKIP 1 ve Foxce). Rozdíly jsou pouze v tom, že na začátku je SqlReader PŘED první větou (tedy na BOF) a že rdr.Read() vrací logickou hodnotu (zda se mu povedl SKIP – je to tedy i testování na EOF).

Nejjednodušším způsobem pro získání hodnot položek je napsat za rdr do hranatých závorek jméno položky (používáme takzvaný indexer). Pokud potřebujeme získat hodnotu položky ProductName v dané větě, napíšeme rdr["ProductName"]. Pokud ale potřebujeme pracovat s hodnotou sloupečku UnitPrice jako s číslem, musíme uvést informaci o tom, v jakém formátu data jsou – např. (decimal)rdr["UnitPrice"] (to znamená „převeď hodnotu rdr["UnitPrice"] na typ decimal“):

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();

    SqlCommand cmd = conn.CreateCommand();
    cmd.CommandText = "SELECT * FROM dbo.Products";
    using (SqlDataReader rdr = cmd.ExecuteReader())
    {
        while (rdr.Read())
        {
            if ((decimal)rdr["UnitPrice"] > 50M) // 50M je to samé jako ve VFP $50
            {
                Console.WriteLine(rdr["ProductName"]);
            }
        }
    }
    conn.Close();
```

}



Je jasné, že tento způsob není nejrychlejší – při každém volání musí C# hledat, který ze sloupců je vlastně ten správný. Efektivnější tedy bude provést toto hledání pouze jednou a využít nalezené číslo sloupce opakovaně:

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();

    SqlCommand cmd = conn.CreateCommand();
    cmd.CommandText = "SELECT * FROM dbo.Products";
    using (SqlDataReader rdr = cmd.ExecuteReader())
    {
        int ordUnitPrice = rdr.GetOrdinal("UnitPrice");     // číslo sloupce
        int ordProductName = rdr.GetOrdinal("ProductName"); // číslo sloupce
        while (rdr.Read())
        {
            if (rdr.GetDecimal(ordUnitPrice) > 50M) // a použijeme číslo sloupce
            {
                Console.WriteLine(rdr.GetString(ordProductName));
            }
        }
    }
    conn.Close();
```

}

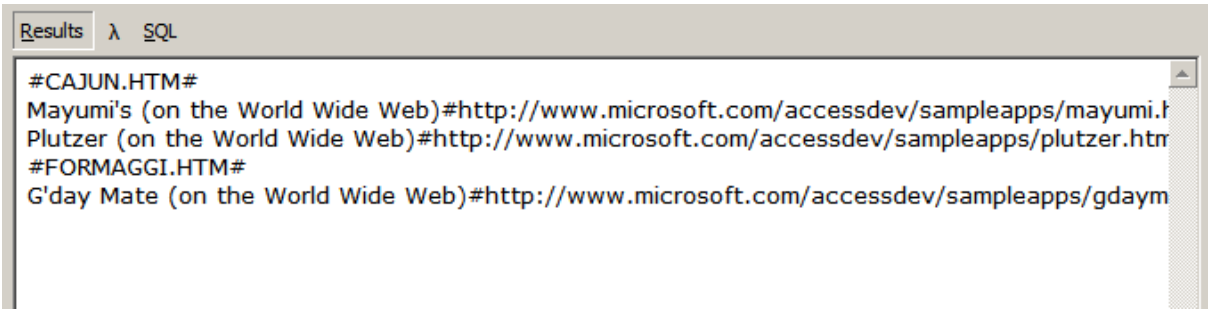
Pomocí metody GetOrdinal("UnitPrice") zjistíme pořadové číslo sloupce UnitPrice – a potom už můžeme předat toto číslo sloupce specializovaným metodám (které vědí, jaký typ dat mají vrátit - zda string, decimal, …).

Hodnoty NULL

Zpracování neexistujících hodnot vždy přináší komplikace ☺. Jinak to není ani tady. Asi bychom očekávali, že se test bude provádět na úrovni sloupce. Ale není to tak a je to pochopitelné. Test používá metodu IsDBNull() SqlDataReaderu. Takže pokud chceme vybrat všechny adresy, které nejsou NULL, postupujeme takto:

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();

    SqlCommand cmd = conn.CreateCommand();
    cmd.CommandText = "SELECT * FROM dbo.Suppliers";
    using (SqlDataReader rdr = cmd.ExecuteReader())
    {
        int ordHomePage = rdr.GetOrdinal("HomePage");
        int ordCompanyName = rdr.GetOrdinal("CompanyName");
        while (rdr.Read())
        {
            if (!rdr.IsDBNull(ordHomePage))
            {
                Console.WriteLine(rdr.GetString(ordHomePage));
            }
        }
    }

    conn.Close();
```

```
}
```

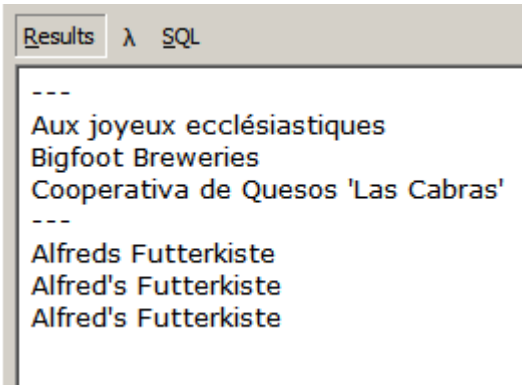


Dotaz vracející více výsledků

Postup je zde zcela stejný, jako je uvedeno výše, pouze poté, co v první tabulce dojdeme na konec, zavoláme metodu rdr.NextResult(), která způsobí skok na další vrácenou tabulku. Pokud už žádná další tabulka neexistuje, vrátí rdr.NextResult() false.

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();

    SqlCommand cmd = conn.CreateCommand();
    cmd.CommandText = @"
SELECT TOP 3 Suppliers.CompanyName FROM dbo.Suppliers ORDER BY 1;
SELECT TOP 3 Orders.ShipName FROM dbo.Orders ORDER BY 1";
    using (SqlDataReader rdr = cmd.ExecuteReader())
    {
        do
        {
            Console.WriteLine("---");
            while (rdr.Read())
            {
                Console.WriteLine(rdr[0]); // nemusí se jméno, lze i číslo sloupce
            }
        } while (rdr.NextResult());
    }

    conn.Close();
```

```
}
```

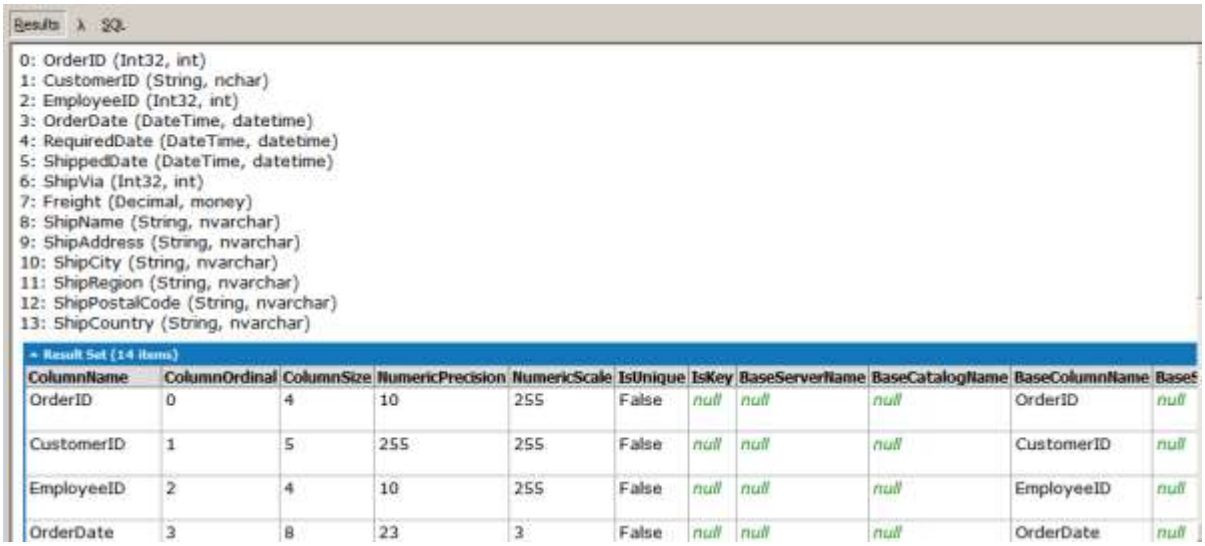


Struktura vrácené tabulky

Pokud potřebujeme zjistit strukturu vrácené tabulky (obdoba Visual FoxPro funkce SQLTables()), zadáme metodě ExecuteReader() volitelný parametr, který říká, že má pouze vrátit informace o datech, nikoli data – a tyto informace pak normálně zpracujeme:

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();

    SqlCommand cmd = conn.CreateCommand();
    cmd.CommandText = "SELECT * FROM dbo.Orders";
    using (SqlDataReader rdr = cmd.ExecuteReader(CommandBehavior.SchemaOnly))
    {
        // Vypíšeme po jedné položce
        for (int i = 0; i < rdr.FieldCount; i++)
        {
            Console.WriteLine(i.ToString()+": "+
            rdr.GetName(i)+" ("+rdr.GetFieldType(i).Name+", "+
            rdr.GetDataTypeName(i)+")");
        }
        // nebo zpracujeme tabulku, která vše obsahuje
        rdr.GetSchemaTable().Dump();
    }

    conn.Close();
```

```
}
```



Results λ SQL

```
0: OrderID (Int32, int)
1: CustomerID (String, nchar)
2: EmployeeID (Int32, int)
3: OrderDate (DateTime, datetime)
4: RequiredDate (DateTime, datetime)
5: ShippedDate (DateTime, datetime)
6: ShipVia (Int32, int)
7: Freight (Decimal, money)
8: ShipName (String, nvarchar)
9: ShipAddress (String, nvarchar)
10: ShipCity (String, nvarchar)
11: ShipRegion (String, nvarchar)
12: ShipPostalCode (String, nvarchar)
13: ShipCountry (String, nvarchar)
```

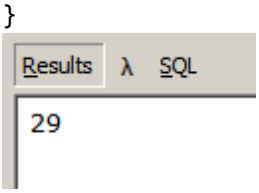
Result Set (14 items)

ColumnName	ColumnOrdinal	ColumnSize	NumericPrecision	NumericScale	IsUnique	IsKey	BaseServerName	BaseCatalogName	BaseColumnName	BaseS
OrderID	0	4	10	255	False	null	null	null	OrderID	null
CustomerID	1	5	255	255	False	null	null	null	CustomerID	null
EmployeeID	2	4	10	255	False	null	null	null	EmployeeID	null
OrderDate	3	8	23	3	False	null	null	null	OrderDate	null

Dotaz, který nevrací tabulku

Aktualizační příkazy nevracejí žádné informace – k jejich provedení slouží speciální metoda ExecuteNonQuery(), která vrací jako funkční hodnotu počet zpracovaných vět:

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();

    SqlCommand cmd = conn.CreateCommand();
    cmd.CommandText = "UPDATE dbo.Suppliers SET Country = RTRIM(Country)";
    int rowsAffected = cmd.ExecuteNonQuery();
    Console.WriteLine(rowsAffected.ToString());

    conn.Close();
}
```



Results λ SQL

```
29
```

ADO.NET – DataSet, DataTable a další

Nyní budeme mít konečně pevnější půdu pod nohama – tato část má opravdu mnoho společného s Visual FoxPro. Takže vzhůru do práce!

Nejdříve si musíme přeložit terminologii:

- „DataSet“ = něco jako foxovská database DBC, ale existuje pouze v paměti.
- „DataTable“ = něco jako foxovská tabulka DBF, ale také existuje pouze v paměti. Může existovat buď samostatně, nebo uvnitř DataSetu.

Vzhledem k tomu, že vše existuje pouze v paměti, musíme si dataset a datatable vytvářet vždy znova a znova. Máme na výběr 2 cesty – buď si je vytvoříme ručně, nebo necháme ADO.NET, aby je automaticky vytvořil podle dat stažených z SQL Serveru (tento způsob bude pro nás asi jednodušší). Nejdříve se ale naučíme pracovat „ručně“.

Vytvoření DataSetu a DataTable

Nejprve vytvoříme DataSet (pro foxaře DBC pomocí CREATE DATABASE) a potom 2 DataTable (pro foxaře 2 DBF pomocí CREATE TABLE) s informacemi o odděleních a zaměstnancích. Nastavíme také relaci mezi nimi – pro foxaře „persistent relation“ v DBC (trvalou vazbu). Všimněte si, že neustále vše ukládáme do proměnných. S nimi pak můžeme dále pracovat.

```
// Vytvoříme prázdný DataSet
DataSet dsPam = new DataSet("PAM");

// Začínáme vytvářet DataTable jako object.
DataTable dtOddeleni = new DataTable("Oddeleni");
// Přidáme sloupec oddId typu integer.
DataColumn colOddId = dtOddeleni.Columns.Add("oddId", typeof(int));
// Přidáme sloupec oddNazev typu string
DataColumn colOddNazev = dtOddeleni.Columns.Add("oddNazev", typeof(string));
// a hned nastavíme délku na maximálně 50 znaků.
colOddNazev.MaxLength = 50;
// Tohle je trochu složitější - nastavujeme primární klíč pro tabulku.
// Uvedeme tedy seznam sloupců, které tvoří primární klíč.
// Část new DataColumn[] znamená předání jako pole sloupečků o jednom prvku.
dtOddeleni.PrimaryKey = new DataColumn[] { colOddId };

// Stejně jako výše
DataTable dtZamestnanci = new DataTable("Zamestnanci");
DataColumn colZamId = dtZamestnanci.Columns.Add("zamId", typeof(int));
DataColumn colZamOddId = dtZamestnanci.Columns.Add("zamOddId", typeof(int));
DataColumn colZamJmeno = dtZamestnanci.Columns.Add("zamJmeno", typeof(string));
colOddNazev.MaxLength = 150;
DataColumn colZamPlat = dtZamestnanci.Columns.Add("zamPlat", typeof(decimal));
dtZamestnanci.PrimaryKey = new DataColumn[] { colZamId };

// A nyní vytvoříme trvalou vazbu (persistent relation) jako v DBC
// - podle čísla oddělení.
dtZamestnanci.Constraints.Add("FK_Oddeleni_Zamestnanci", colOddId, colZamOddId);

// A nakonec do DataSetu přidáme obě DataTable
dsPam.Tables.Add(dtOddeleni);
dsPam.Tables.Add(dtZamestnanci);
```

Hotovo, datové struktury jsou vytvořeny. Nyní budeme měnit data v těchto tabulkách.

Vkládání dat do DataTable

Vzhledem k tomu, že DataSet a DataTable existuje pouze v paměti, pro testy v LINQPadu musíte následující řádky přidávat za program pro jejich vytvoření (viz výše). Následuje jednoduchý program

na vložení několika vět. V programu je dostatek komentářů pro pochopení, pouze za programem následuje diskuze k práci s RowState.

```
Console.WriteLine("\r\n--- INSERT ---\r\n");
// INSERT

// 1. metoda - nejprve vytvoříme samostatně prázdný řádek, ten nejprve naplníme
// daty a pak ho přidáme do DataTable (nic takového Visual FoxPro nezná)
DataRow row = dtOddeleni.NewRow();
row["oddId"] = 1;
row["oddNazev"] = "Vedení";
dtOddeleni.Rows.Add(row);
dtOddeleni.Dump("dtOddeleni");

// 2. metoda - přímo přidáváme řádky do tabulky (jako INSERT INTO …)
dtZamestnanci.Rows.Add(1, 1, "Adam", 20000M);
dtZamestnanci.Rows.Add(2, 1, "Eva", 23000M);
dtZamestnanci.Rows.Add(3, 1, "Kain", 30000M);
dtZamestnanci.Rows.Add(4, 1, "Abel", 18000M);

// Vypíšeme obsah tabulky zaměstnanců.
dtZamestnanci.Dump("dtZamestnanci");
// A nyní ještě jednou vypíšeme obsah tabulky zaměstnanců - tentokrát po řádcích
// !!! Zde jsou důležité 2 věci:
// - je možné pracovat s libovolným řádkem tabulky bez jakéhokoli posouvání -
//   zápis je jako s array (VFP nic takového nezná)
//   dtZamestnanci.Rows[j] znamená použití (j+1)-ního řádku tabulky
//   POZOR! Čísluje se od 0!
//   Takže dtZamestnanci.Rows[1]["zamJmeno"] nám umožňuje získat hodnotu položky
//    zamJmeno pro 2. řádek tabulky
// - pomocí dtZamestnanci.Rows[j].RowState zjistíme stav řádku - diskuze následuje
for (int j = 0; j < dtZamestnanci.Rows.Count; j++)
    Console.WriteLine(string.Format("{0}.row: {1} = {2}", j,
        dtZamestnanci.Rows[j]["zamJmeno"], dtZamestnanci.Rows[j].RowState));

// AcceptChanges() pošle změny na SQL Server (odpovídá tedy TableUpdate() ve VFP)
dtZamestnanci.AcceptChanges();
Console.WriteLine("\r\ndtZamestnanci.AcceptChanges();\r\n");

dtZamestnanci.Dump("dtZamestnanci");
for (int j = 0; j < dtZamestnanci.Rows.Count; j++)
    Console.WriteLine(string.Format("{0}.row: {1} = {2}", j,
```

```
        dtZamestnanci.Rows[j]["zamJmeno"], dtZamestnanci.Rows[j].RowState));
```

Results λ SQL

```
--- INSERT ---
```

▪ dtOddeleni

Result Set (1 item)

oddId	oddNazev
1	Vedení

▪ dtZamestnanci

Result Set (4 items)

zamId	zamOddId	zamJmeno	zamPlat
1	1	Adam	20000
2	1	Eva	23000
3	1	Kain	30000
4	1	Abel	18000

```
0.row: Adam = Added
1.row: Eva = Added
2.row: Kain = Added
3.row: Abel = Added

dtZamestnanci.AcceptChanges();
```

▪ dtZamestnanci

Result Set (4 items)

zamId	zamOddId	zamJmeno	zamPlat
1	1	Adam	20000
2	1	Eva	23000
3	1	Kain	30000
4	1	Abel	18000

```
0.row: Adam = Unchanged
1.row: Eva = Unchanged
2.row: Kain = Unchanged
3.row: Abel = Unchanged
```

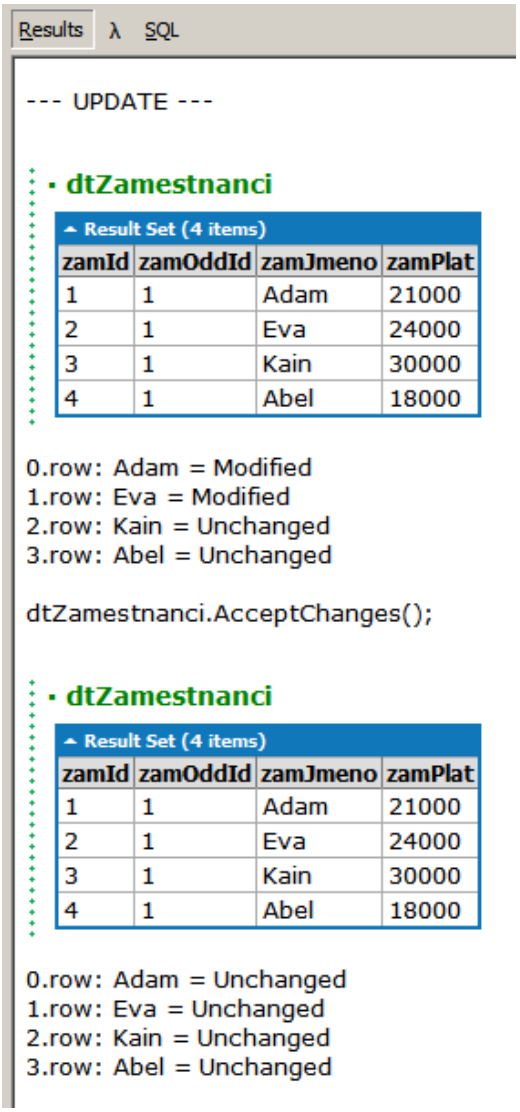
V předchozím programu vidíte použití nové vlastnosti – RowState. Při tomto použití `dtZamestnanci.Rows[j].RowState` nám umožňuje otestovat v jakém stavu je daný (j-tý) řádek tabulky (přidaný, zrušený, změněný atd.) – foxařům to určitě připomene GetFldState().

Změna dat v DataTable

Změna dat je velmi jednoduchá. Nejdříve je potřeba najít větu, kterou chceme modifikovat, a pak ji změnit. Opět využijeme RowState jako v předchozím příkladě, abychom zjistili stav dat.

```
Console.WriteLine("\r\n--- UPDATE ---\r\n");
// UPDATE

// 1. metoda - podle primárního klíče
//    - najdeme řádek pomocí primárního klíče
```

```
//    - referenci na tento řádek si uložíme do proměnné!
//    - pak už normálně data změníme
DataRow rowFind = dtZamestnanci.Rows.Find(2);
rowFind["zamPlat"] = 24000M;
// 2. metoda - podle indexu (foxovsky je to podle RecNo())
dtZamestnanci.Rows[0]["zamPlat"] = 21000M;

dtZamestnanci.Dump("dtZamestnanci");
for (int j = 0; j < dtZamestnanci.Rows.Count; j++)
    Console.WriteLine(string.Format("{0}.row: {1} = {2}", j,
        dtZamestnanci.Rows[j]["zamJmeno"], dtZamestnanci.Rows[j].RowState));

// AcceptChanges() pošle změny na SQL Server (odpovídá tedy TableUpdate() ve VFP)
dtZamestnanci.AcceptChanges();
Console.WriteLine("\r\ndtZamestnanci.AcceptChanges();\r\n");

dtZamestnanci.Dump("dtZamestnanci");
for (int j = 0; j < dtZamestnanci.Rows.Count; j++)
    Console.WriteLine(string.Format("{0}.row: {1} = {2}", j,
        dtZamestnanci.Rows[j]["zamJmeno"], dtZamestnanci.Rows[j].RowState));
```

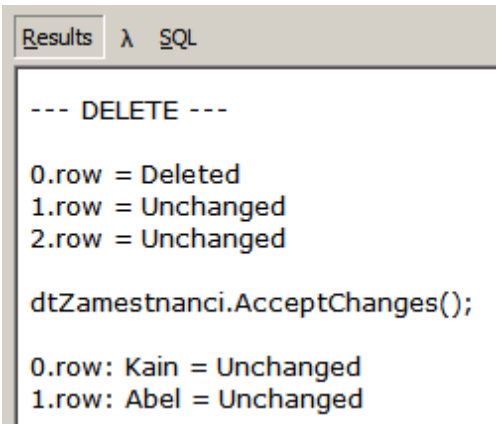


Rušení dat v DataTable

Rušení dat funguje stejně jako ve Visual FoxPro: Věta se nejprve označí ke zrušení a během AcceptChanges() (pro foxaře po TableUpdate()) se fyzicky zruší na SQL Serveru:

```
// DELETE - označí ke zrušení na serveru – její RowState bude Deleted
dtZamestnanci.Rows[1].Delete();
// REMOVE – ihned zruší v DataTable – vůbec nebude ve výpisu
dtZamestnanci.Rows.Remove(dtZamestnanci.Rows[0]);

// vzhledem k tomu že je věta zrušena, nemůžeme vypisovat obsah sloupečků
for (int j = 0; j < dtZamestnanci.Rows.Count; j++)
    Console.WriteLine(string.Format("{0}.row = {1}", j,
        dtZamestnanci.Rows[j].RowState));

// AcceptChanges() pošle změny na SQL Server (odpovídá tedy TableUpdate() ve VFP)
dtZamestnanci.AcceptChanges();
Console.WriteLine("\r\ndtZamestnanci.AcceptChanges();\r\n");
```

```
for (int j = 0; j < dtZamestnanci.Rows.Count; j++)
    Console.WriteLine(string.Format("{0}.row: {1} = {2}", j,
        dtZamestnanci.Rows[j]["zamJmeno"], dtZamestnanci.Rows[j].RowState));
```



Naplnění DataTable nebo DataSetu daty ze SQL Serveru pomocí SqlDataReaderu

Rozhodně není potřeba pokaždé ručně vytvářet strukturu DataTable, stejně jako to nemusíme dělat ve Visual FoxPro, kde nám SQLExec kursor automaticky vytvoří. Zde k tomu slouží metody Load() u DataTable a DataSet. V následujícím příkladě použijeme SqlDataReader (který už známe) dvakrát. Nejdříve naplníme samostatnou DataTable (tedy něco jako volnou DBF ve VFP), a potom DataTable v DataSet (odpovídá DBF v DBC).

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

// budeme plnit 2x - jednou do DataTable, která není v DataSetu, jednou DataSet
DataSet ds = new DataSet();
DataTable dt = new DataTable();

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();

    SqlCommand cmd = conn.CreateCommand();
    cmd.CommandText = @"
SELECT C.CustomerID, C.CompanyName, C.ContactName, C.Country
FROM dbo.Customers C WHERE C.Country = 'SWEDEN'";

    using (SqlDataReader rdr = cmd.ExecuteReader())
    {
// pomocí SqlDataReaderu naplníme přímo DataTable
        dt.Load(rdr);
    }

    using (SqlDataReader rdr = cmd.ExecuteReader())
    {
```

```
// pomocí SqlDataReaderu naplníme DataTable v DataSetu
// - pomocí parametru řekneme, jak se má jmenovat tabulka, kterou vytvoří
        ds.Load(rdr, LoadOption.PreserveChanges, "Vyber");
    }

    conn.Close();
}

dt.Dump("Výpis osamocené DataTable");
ds.Tables["Vyber"].Dump("Výpis DataTable z DataSetu");
```

Results λ SQL

Výpis osamocené DataTable

Result Set (2 items)

CustomerID	CompanyName	ContactName	Country
BERGS	Berglunds snabbköp	Christina Berglund	Sweden
FOLKO	Folk och fä HB	Maria Larsson	Sweden

Výpis DataTable z DataSetu

Result Set (2 items)

CustomerID	CompanyName	ContactName	Country
BERGS	Berglunds snabbköp	Christina Berglund	Sweden
FOLKO	Folk och fä HB	Maria Larsson	Sweden

Naplnění DataTable nebo DataSetu daty ze SQL Serveru pomocí SqlDataAdapteru

Nejprve si ukážeme pouhé naplnění DataSet a DataTable daty ze SQL Serveru pomocí SqlDataAdapteru bez zajištění aktualizace. SqlDataAdapter je prostředníkem mezi SQL Serverem a DataTable. Při vytváření mu tedy musíme říci, kam se má připojit (connection string) a jaký příkaz má provést (SQL řetězec). Metoda Fill provede vlastní naplnění tabulky daty (tabulka se předá jako parametr této metodě):

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

DataTable dt = new DataTable();

string sql = @"
SELECT C.CustomerID, C.CompanyName, C.ContactName, C.Country
FROM dbo.Customers C WHERE C.Country = 'SWEDEN'";

// Tentokrát vytvoříme SqlConnection samostatně
SqlConnection conn = new SqlConnection(connString);
// SqlDataAdapter vytvoříme z tohoto připojení a ze SQL řetězce
SqlDataAdapter da = new SqlDataAdapter(sql, conn);

// 1. Řekneme DataAdapteru da, aby naplnil DataTable dt, a je hotovo :)
```

```
da.Fill(dt);
dt.Dump("dt");

// 2. Naplníme DataSet a navíc přejmenujeme sloupečky
DataSet ds = new DataSet();
// následující 3 řádky provedou přejmenování
var tableMap = da.TableMappings.Add("Table", "MyCustomers");
tableMap.ColumnMappings.Add("CustomerID", "custId");
tableMap.ColumnMappings.Add("Country", "CompanyCountry");
// a naplníme DataSet
da.Fill(ds);
ds.Dump("ds");

// 3. Metoda Fill() má další parametry - zde se například omezíme pouze na 1 větu
DataTable dt2 = new DataTable();
// upřímně řečeno to není žádná výhra - výběr se provádí až na klientu ☹
da.Fill(0, 1, dt2); // v C# se indexuje od 0!
dt2.Dump("dt2");
```

Results λ SQL

dt

Result Set (2 items)

CustomerID	CompanyName	ContactName	Country
BERGS	Berglunds snabbköp	Christina Berglund	Sweden
FOLKO	Folk och fä HB	Maria Larsson	Sweden

ds

Result Set (2 items)

custId	CompanyName	ContactName	CompanyCountry
BERGS	Berglunds snabbköp	Christina Berglund	Sweden
FOLKO	Folk och fä HB	Maria Larsson	Sweden

dt2

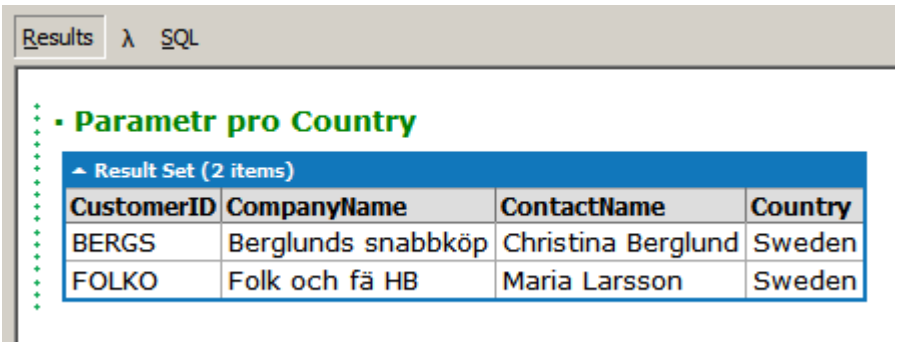
Result Set (1 item)

CustomerID	CompanyName	ContactName	Country
BERGS	Berglunds snabbköp	Christina Berglund	Sweden

Stejně jako výše je možné i v SqlDataAdapteru používat parametry:

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

DataTable dt = new DataTable();

string sql = @"
SELECT C.CustomerID, C.CompanyName, C.ContactName, C.Country
```

```
FROM dbo.Customers C WHERE C.Country = @Country";
SqlConnection conn = new SqlConnection(connString);
SqlDataAdapter da = new SqlDataAdapter(sql, conn);
da.SelectCommand.Parameters.Add("@Country", SqlDbType.NVarChar).Value = "SWEDEN";
da.Fill(dt);
dt.Dump("Parametr pro Country");
```



CustomerID	CompanyName	ContactName	Country
BERGS	Berglunds snabbköp	Christina Berglund	Sweden
FOLKO	Folk och fä HB	Maria Larsson	Sweden

Aktualizace dat na SQL Serveru pomocí SqlDataAdapteru

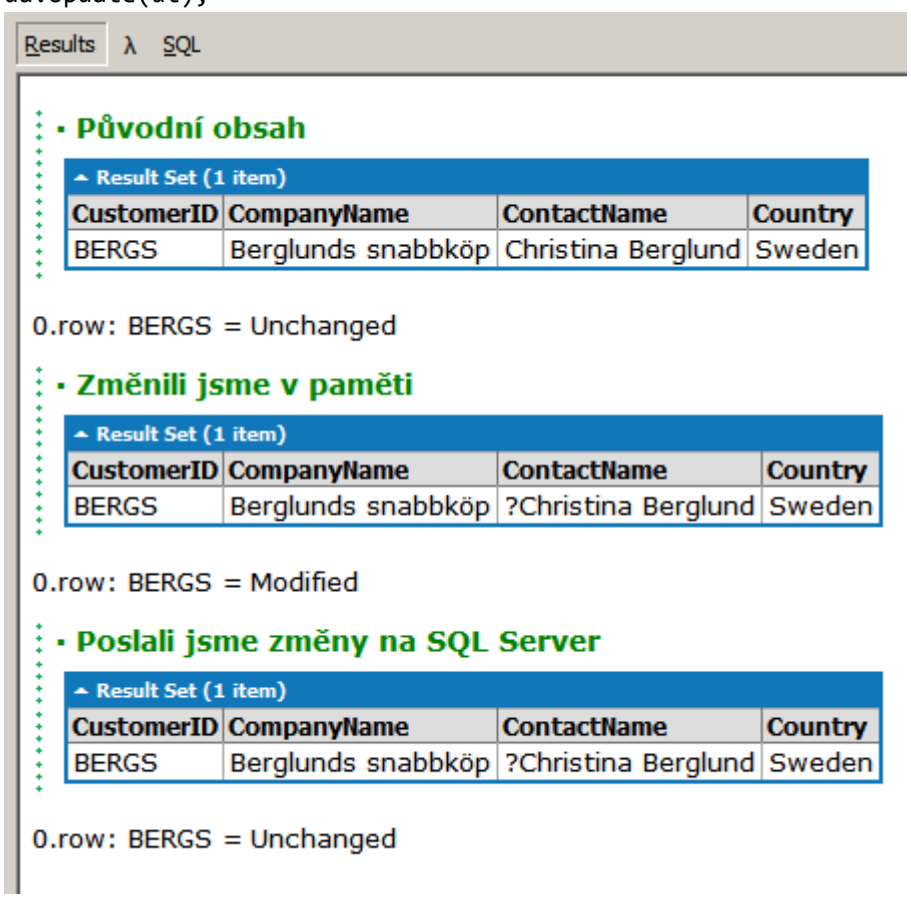
Stejně jako můžeme vzdálené pohledy použít k aktualizaci dat na SQL Serveru, můžeme pro aktualizaci použít i SqlDataAdapter. A uvidíte, že to není o nic složitější! Automatické vygenerování příkazů pro aktualizaci (které ve Foxce provedete zaškrtnutím „Send SQL Updates“ v návrháři vzdálených pohledů) zajistíte tím, že pro SqlDataAdapter vytvoříte nový objekt (SqlCommandBuilder) pomocí příkazu

```
SqlCommandBuilder cb = new SqlCommandBuilder(da);
```

Tím zároveň připojíte tento nový SqlCommandBuilder cb k SqlDataAdapter da. Potom už stačí zavolat da.Update(dt); (ekvivalent foxovského TableUpdate()):

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

DataTable dt = new DataTable();

string sql = @"
SELECT TOP 1 C.CustomerID, C.CompanyName, C.ContactName, C.Country
FROM dbo.Customers C WHERE C.Country = @Country ORDER BY C.CustomerID";
SqlConnection conn = new SqlConnection(connString);
SqlDataAdapter da = new SqlDataAdapter(sql, conn);
SqlCommandBuilder cb = new SqlCommandBuilder(da);

da.SelectCommand.Parameters.Add("@Country", SqlDbType.NVarChar).Value = "SWEDEN";
// Naplníme DataTable daty
da.Fill(dt);

dt.Dump("Původní obsah");
for (int j = 0; j < dt.Rows.Count; j++)
    Console.WriteLine(string.Format("{0}.row: {1} = {2}", j,
        dt.Rows[j]["CustomerID"], dt.Rows[j].RowState));
```

```
// data změníme stejně jako výše - zde přidáme v první větě před ContactName "?"
dt.Rows[0]["ContactName"] = "?"+(string)dt.Rows[0]["ContactName"];

dt.Dump("Změnili jsme v paměti");
for (int j = 0; j < dt.Rows.Count; j++)
    Console.WriteLine(string.Format("{0}.row: {1} = {2}", j,
        dt.Rows[j]["CustomerID"], dt.Rows[j].RowState));

// provedeme TableUpdate() (foxovsky řečeno)
da.Update(dt);

dt.Dump("Poslali jsme změny na SQL Server");
for (int j = 0; j < dt.Rows.Count; j++)
    Console.WriteLine(string.Format("{0}.row: {1} = {2}", j,
        dt.Rows[j]["CustomerID"], dt.Rows[j].RowState));

// zbytek slouží k tomu, abychom uvedli data do původního stavu:
dt.Rows[0]["ContactName"] = ((string)dt.Rows[0]["ContactName"]).Substring(1);
da.Update(dt);
```

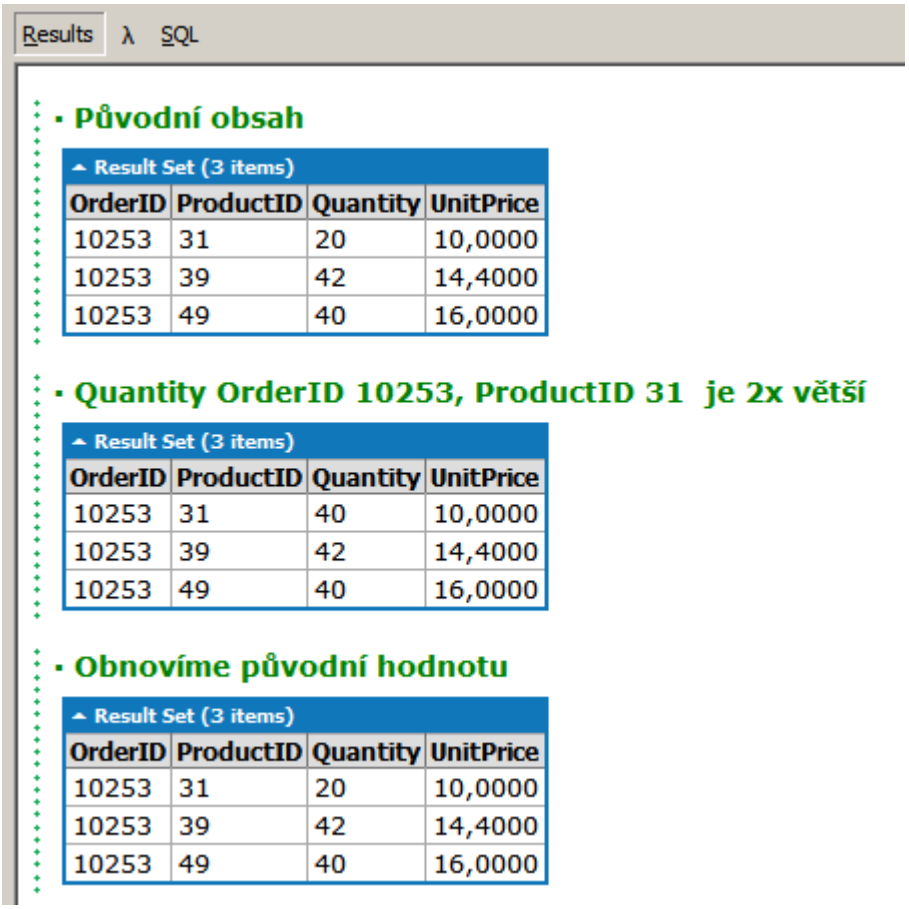


Samozřejmě je možné nastavit si i vlastní aktualizační logiku místo standardně vygenerované (můžete třeba provádět změny v datech pomocí uložených procedur apod.). Není to, jak se můžete přesvědčit na následujícím příkladě, úplně jednoduché. Nebudu v programu uvádět podrobné komentáře. Idea je jednoduchá. Ručně definujeme, jak má SqlDataAdapter provádět insert, update a delete:

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

DataTable dt = new DataTable();
string sql = @"
SELECT OrderID, ProductID, Quantity, UnitPrice
FROM [Order Details] WHERE OrderID = @OrderID
ORDER BY ProductID";
SqlConnection conn = new SqlConnection(connString);
SqlDataAdapter da = new SqlDataAdapter(sql, conn);
da.SelectCommand.Parameters.AddWithValue("@OrderID", 10253);
da.Fill(dt);

dt.Dump("Původní obsah");

SqlParameterCollection pc;
SqlParameter p;

// InsertCommand
string strSQL = @"
INSERT INTO [Order Details] (OrderID, ProductID, Quantity, UnitPrice)
VALUES (@OrderID, @ProductID, @Quantity, @UnitPrice)";
da.InsertCommand = new SqlCommand(strSQL, conn);
pc = da.InsertCommand.Parameters;
pc.Add("@OrderID", SqlDbType.Int, 0, "OrderID");
pc.Add("@ProductID", SqlDbType.Int, 0, "ProductID");
pc.Add("@Quantity", SqlDbType.SmallInt, 0, "Quantity");
pc.Add("@UnitPrice", SqlDbType.Money, 0, "UnitPrice");

// UpdateCommand
strSQL = @"
UPDATE [Order Details]
SET OrderID   = @OrderID_New
  , ProductID = @ProductID_New
  , Quantity  = @Quantity_New
  , UnitPrice = @UnitPrice_New
WHERE OrderID   = @OrderID_Old
  AND ProductID = @ProductID_Old
  AND Quantity  = @Quantity_Old
  AND UnitPrice = @UnitPrice_Old";
da.UpdateCommand = new SqlCommand(strSQL, conn);
pc = da.UpdateCommand.Parameters;
pc.Add("@OrderID_New", SqlDbType.Int, 0, "OrderID");
pc.Add("@ProductID_New", SqlDbType.Int, 0, "ProductID");
pc.Add("@Quantity_New", SqlDbType.SmallInt, 0, "Quantity");
pc.Add("@UnitPrice_New", SqlDbType.Money, 0, "UnitPrice");
p = pc.Add("@OrderID_Old", SqlDbType.Int, 0, "OrderID");
p.SourceVersion = DataRowVersion.Original;
p = pc.Add("@ProductID_Old", SqlDbType.Int, 0, "ProductID");
```

```
p.SourceVersion = DataRowVersion.Original;
p = pc.Add("@Quantity_Old", SqlDbType.SmallInt, 0, "Quantity");
p.SourceVersion = DataRowVersion.Original;
p = pc.Add("@UnitPrice_Old", SqlDbType.Money, 0, "UnitPrice");
p.SourceVersion = DataRowVersion.Original;

// DeleteCommand
strSQL = @"
DELETE [Order Details]
WHERE OrderID   = @OrderID
  AND ProductID = @ProductID
  AND Quantity  = @Quantity
  AND UnitPrice = @UnitPrice";
da.DeleteCommand = new SqlCommand(strSQL, conn);
pc = da.DeleteCommand.Parameters;
p = pc.Add("@OrderID", SqlDbType.Int, 0, "OrderID");
p.SourceVersion = DataRowVersion.Original;
p = pc.Add("@ProductID", SqlDbType.Int, 0, "ProductID");
p.SourceVersion = DataRowVersion.Original;
p = pc.Add("@Quantity", SqlDbType.SmallInt, 0, "Quantity");
p.SourceVersion = DataRowVersion.Original;
p = pc.Add("@UnitPrice", SqlDbType.Money, 0, "UnitPrice");
p.SourceVersion = DataRowVersion.Original;

dt.Rows[0]["Quantity"] = (short)(2 * ((short)dt.Rows[0]["Quantity"]));
da.Update(dt);

DataTable dt2 = new DataTable();
da.Fill(dt2);

dt2.Dump("Quantity OrderID 10253, ProductID 31  je 2x větší");


dt.Rows[0]["Quantity"] = ((short)dt.Rows[0]["Quantity"]) / 2;
da.Update(dt);

DataTable dt3 = new DataTable();
da.Fill(dt3);
```

```
dt3.Dump("Obnovíme původní hodnotu");
```



Idea je tedy jednoduchá, ale ruční práce je zde hodně. Vše se však dá usnadnit, a to pomocí metod Get____Command(true).CommandText():

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

DataTable dt = new DataTable();
string sql = @"
SELECT TOP 1 Customers.CustomerID, Customers.CompanyName, Customers.ContactName,
Customers.Country
FROM dbo.Customers WHERE Customers.Country = @Country ORDER BY
Customers.CustomerID";
SqlConnection conn = new SqlConnection(connString);
SqlDataAdapter da = new SqlDataAdapter(sql, conn);
da.SelectCommand.Parameters.Add("@Country", SqlDbType.NVarChar).Value = "SWEDEN";
da.Fill(dt);

SqlCommandBuilder cb = new SqlCommandBuilder(da);

cb.GetInsertCommand(true).CommandText.Dump("GetInsertCommand");
cb.GetUpdateCommand(true).CommandText.Dump("GetUpdateCommand");
cb.GetDeleteCommand(true).CommandText.Dump("GetDeleteCommand");
```

```
// v Northwind bohužel není tabulka s Timestamp :( - jinak by ji použil
cb.ConflictOption = ConflictOption.CompareRowVersion;

cb.GetInsertCommand(true).CommandText.Dump(
                                        "CompareRowVersion a GetInsertCommand");
cb.GetUpdateCommand(true).CommandText.Dump(
                                        "CompareRowVersion a GetUpdateCommand");
cb.GetDeleteCommand(true).CommandText.Dump(
                                        "CompareRowVersion a GetDeleteCommand");
```

Takto získáte SQL příkazy, které si můžete dále upravit podle potřeby.

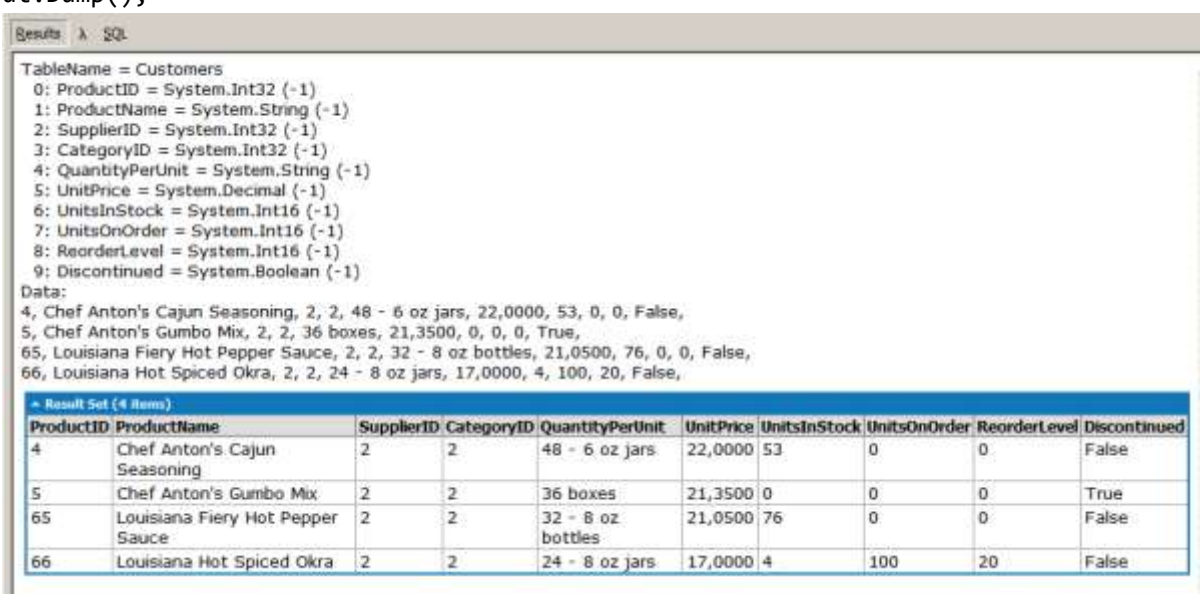
DataTable – práce s řádky a sloupci

A konečně se dostáváme k práci s daty.

Nejprve si ukážeme triviální program na zjištění struktury tabulky ze SQL Serveru a postupný výpis všech hodnot (a pro kontrolu ještě druhý výpis pomocí .Dump()).

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

DataTable dt = new DataTable("Customers");

string sql = @"SELECT Products.* FROM dbo.Products WHERE Products.SupplierID =
@SupplierID";
SqlConnection conn = new SqlConnection(connString);
SqlDataAdapter da = new SqlDataAdapter(sql, conn);

da.SelectCommand.Parameters.Add("@SupplierID", SqlDbType.Int).Value = 2;
// Naplníme DataTable daty
da.Fill(dt);
```

```
// Zjištění jména tabulky
Console.WriteLine(string.Format("TableName = {0}", dt.TableName));
for (int i = 0; i < dt.Columns.Count; i++)
{
    Console.WriteLine(string.Format("  {0}: {1} = {2} ({3})"
// Zjištění parametrů sloupečků
        , dt.Columns[i].Ordinal
        , dt.Columns[i].ColumnName
        , dt.Columns[i].DataType
        , dt.Columns[i].MaxLength
        ));
}
Console.WriteLine("Data:");
// Zjištění počtu řádků (RecCount())
for (int j = 0; j < dt.Rows.Count; j++)
{
    for (int i = 0; i < dt.Columns.Count; i++)
    {
// Zjištění hodnot všech sloupečků
        Console.Write(string.Format("{0}, ", dt.Rows[j][i]));
    }
    Console.WriteLine();
}
// Kontrola :)
dt.Dump();
```



Results λ SQL

```
TableName = Customers
  0: ProductID = System.Int32 (-1)
  1: ProductName = System.String (-1)
  2: SupplierID = System.Int32 (-1)
  3: CategoryID = System.Int32 (-1)
  4: QuantityPerUnit = System.String (-1)
  5: UnitPrice = System.Decimal (-1)
  6: UnitsInStock = System.Int16 (-1)
  7: UnitsOnOrder = System.Int16 (-1)
  8: ReorderLevel = System.Int16 (-1)
  9: Discontinued = System.Boolean (-1)
Data:
4, Chef Anton's Cajun Seasoning, 2, 2, 48 - 6 oz jars, 22,0000, 53, 0, 0, False,
5, Chef Anton's Gumbo Mix, 2, 2, 36 boxes, 21,3500, 0, 0, 0, True,
65, Louisiana Fiery Hot Pepper Sauce, 2, 2, 32 - 8 oz bottles, 21,0500, 76, 0, 0, False,
66, Louisiana Hot Spiced Okra, 2, 2, 24 - 8 oz jars, 17,0000, 4, 100, 20, False,
```

Result Set (4 items)

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22,0000	53	0	0	False
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21,3500	0	0	0	True
65	Louisiana Fiery Hot Pepper Sauce	2	2	32 - 8 oz bottles	21,0500	76	0	0	False
66	Louisiana Hot Spiced Okra	2	2	24 - 8 oz jars	17,0000	4	100	20	False

DataTable – třídění, hledání a filtrování

Tato kapitola popisuje dvě základní metody pro práci s tabulkami:

- Vyhledání podle primárního klíče pomocí metody Find(). Odpovídá to hledání pomocí funkce SEEK() podle primárního klíče ve FoxPro.

- Výběr dat a případné třídění pomocí metody Select(). To vypadá podobně jako SET FILTER a volitelně SET ORDER najednou – ale POZOR! Pokud .NET během práce zjistí, že potřebuje pomocný index, tak ho vytvoří, použije a ihned zahodí. Ale vydržte, bude lépe…

Všimněte si také, že výsledkem je samostatný řádek/skupina řádků, které ukazují zpátky do tabulky, ze které jsme provedli výběr.

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

DataTable dt = new DataTable("Customers");

string sql = @"SELECT Products.* FROM dbo.Products WHERE Products.SupplierID =
@SupplierID";
SqlConnection conn = new SqlConnection(connString);
SqlDataAdapter da = new SqlDataAdapter(sql, conn);

da.SelectCommand.Parameters.Add("@SupplierID", SqlDbType.Int).Value = 2;
// Naplníme DataTable daty
da.Fill(dt);

// Hledání podle primárního klíče:
Console.WriteLine("Hledání podle primárního klíče:");
dt.PrimaryKey = new DataColumn[] { dt.Columns["ProductID"]};
DataRow row = dt.Rows.Find(65);
if (row == null)
    Console.WriteLine("Neexistuje");
else
    Console.WriteLine((string)row["ProductName"]);
row.Dump();
```

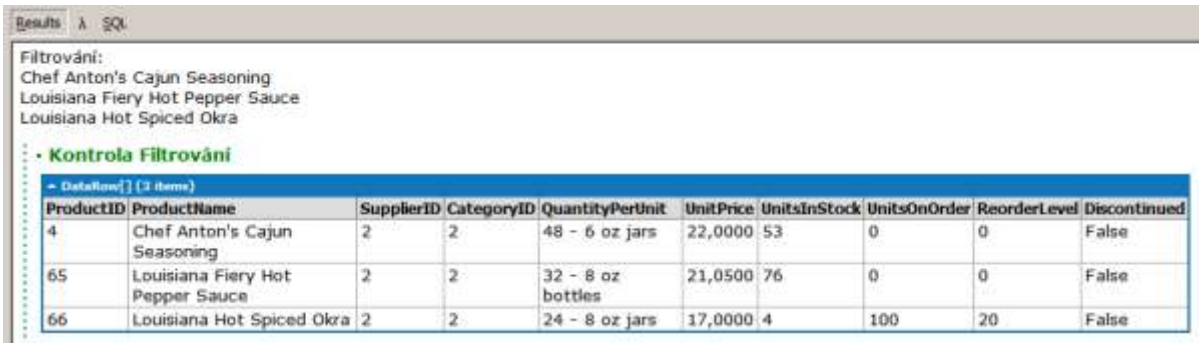
Results λ SQL

Hledání podle primárního klíče:
Louisiana Fiery Hot Pepper Sauce

DataRow

ProductID	65
ProductName	Louisiana Fiery Hot Pepper Sauce
SupplierID	2
CategoryID	2
QuantityPerUnit	32 - 8 oz bottles
UnitPrice	21,0500
UnitsInStock	76
UnitsOnOrder	0
ReorderLevel	0
Discontinued	False

```
// Filtrování:
Console.WriteLine("\r\nFiltrování:");
foreach (DataRow row2 in dt.Select("UnitPrice>=22 OR ProductName LIKE '%Hot%'"))
    Console.WriteLine((string)row2["ProductName"]);
dt.Select("UnitPrice>=22 OR ProductName LIKE '%Hot%'").Dump("Kontrola
Filtrování");
```



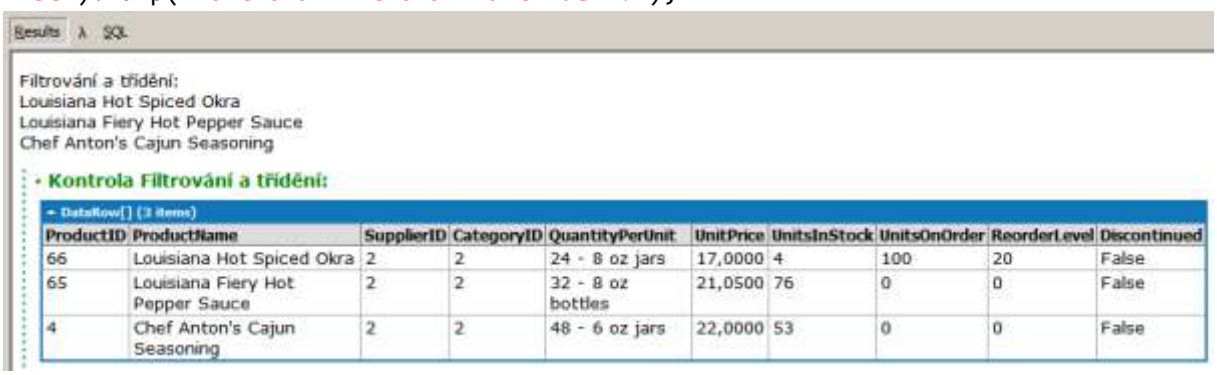
Filtrování:
Chef Anton's Cajun Seasoning
Louisiana Fiery Hot Pepper Sauce
Louisiana Hot Spiced Okra

• Kontrola Filtrování

DataRow[] (3 items)

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22,0000	53	0	0	False
65	Louisiana Fiery Hot Pepper Sauce	2	2	32 - 8 oz bottles	21,0500	76	0	0	False
66	Louisiana Hot Spiced Okra	2	2	24 - 8 oz jars	17,0000	4	100	20	False

```
// Filtrování a třídění:
Console.WriteLine("\r\nFiltrování a třídění:");
foreach (DataRow row2 in dt.Select("UnitPrice>=22 OR ProductName LIKE '%Hot%'",
"ProductName DESC"))
    Console.WriteLine((string)row2["ProductName"]);
dt.Select("UnitPrice>=22 OR ProductName LIKE '%Hot%'", "ProductName
DESC").Dump("Kontrola Filtrování a třídění:");
```



Filtrování a třídění:
Louisiana Hot Spiced Okra
Louisiana Fiery Hot Pepper Sauce
Chef Anton's Cajun Seasoning

• Kontrola Filtrování a třídění:

DataRow[] (3 items)

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
66	Louisiana Hot Spiced Okra	2	2	24 - 8 oz jars	17,0000	4	100	20	False
65	Louisiana Fiery Hot Pepper Sauce	2	2	32 - 8 oz bottles	21,0500	76	0	0	False
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22,0000	53	0	0	False

DataTable a DataView – aneb to nejdůležitější!

A teď přichází ta nejzajímavější kapitola o ADO.NET (aspoň doufám ☺) – naučíme se INDEXOVAT! Ke každé tabulce DataTable si můžete vytvořit tolik datových pohledů DataView, kolik chcete. U každého z nich můžete zadat třídění a/nebo výběrovou podmínku. K datům původní tabulky pak můžete přistupovat (modifikovat data atd.) přes tento pohled a tím používat indexy. Každá definice pohledu znamená, že ADO.NET vytvoří přesně stejný index, jako si vytváří i Visual FoxPro. V podstatě je to ještě výhodnější situace než ve Visual FoxPro. V jednom okamžiku nemusí být aktivní pouze jeden index, protože najednou můžete používat libovolný počet pohledů.

Pohled přináší další důležitou vlastnost. Z předchozího známe RowState (u řádku). Navíc máme DataRowVersion. To je ekvivalent foxovského OLDVAL/CURVAL/EVAL. A podstatně rozšířený. Oba pak můžete použít nejen při filtrování (takže si například můžete vybrat, zda chcete pracovat s původním obsahem řádku, s modifikovaným obsahem řádku atd.), ale také při vlastní práci s hodnotami. Abych zde zdůraznil, jaké to má výhody, uvedu příkaz z programu:

```
  DataView view8 = new DataView(dt, "UnitPrice > 20"
            , "SupplierID DESC, UnitPrice DESC", DataViewRowState.OriginalRows);
```

to znamená: vyfiltruj ty věty, které mají UnitPrice větší než 20, setřiď je sestupně podle SupplierID a UnitPrice, výsledné řádky chci vidět ve tvaru před modifikacemi.

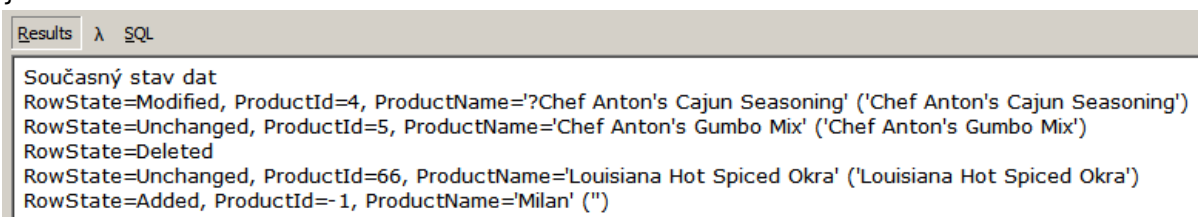
V programu si také povšimněte, že u normálních tabulek se k hodnotám OLDVAL/CURVAL/EVAL dostaneme také, a to pomocí této syntaxe:

```
dt.Rows[0]["ProductName", DataRowVersion.Original]
```

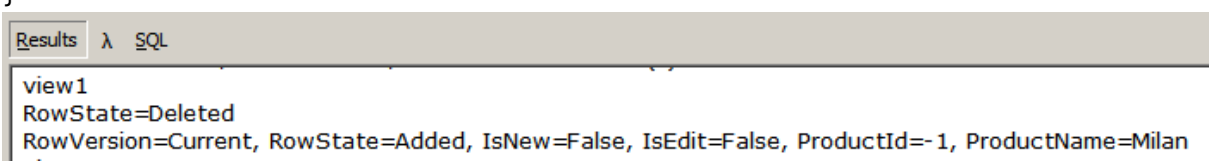
Poznámka: Metoda .Dump() LINQPadu není schopna korektně pracovat se zrušenými větami, proto je v tomto příkladě použit vlastní jednoduchý program pro výpis. Abych ho nemusel pořád opakovat (v LINQPadu nelze vytvořit funkci), je uveden pouze u první definice pohledu, u dalších už ale uveden není.

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

DataTable dt = new DataTable("Customers");

string sql = @"SELECT Products.* FROM dbo.Products WHERE Products.SupplierID =
@SupplierID";
SqlConnection conn = new SqlConnection(connString);
SqlDataAdapter da = new SqlDataAdapter(sql, conn);

da.SelectCommand.Parameters.Add("@SupplierID", SqlDbType.Int).Value = 2;
// Naplníme DataTable daty
da.Fill(dt);

// Připravíme si data pro další práci
// - změníme jednu větu
dt.Rows[0]["ProductName"] = "?" + (string)dt.Rows[0]["ProductName"];
// - zrušíme jednu větu
dt.Rows[2].Delete();
// - přidáme jednu větu
DataRow row = dt.NewRow();
row["ProductID"] = -1;
row["ProductName"] = "Milan";
row["UnitPrice"] = 100;
row["UnitsInStock"] = 0;
dt.Rows.Add(row);

// a vypíšeme data, abychom mohli další práci kontrolovat
Console.WriteLine("Současný stav dat");
foreach(DataRow r in dt.Rows)
{
    if (r.RowState!=DataRowState.Deleted)
        Console.WriteLine(string.Format(
              "RowState={0}, ProductId={1}, ProductName='{2}' ('{3}')"
            , r.RowState
            , r["ProductId"]
            , r["ProductName", DataRowVersion.Current]
            , (r.RowState!=DataRowState.Added
                ? r["ProductName", DataRowVersion.Original] : "")));
    else
```

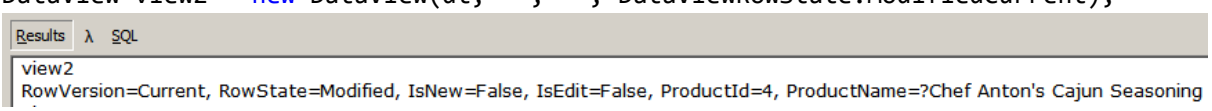
```
        Console.WriteLine(string.Format("RowState={0}", r.RowState));
}
```



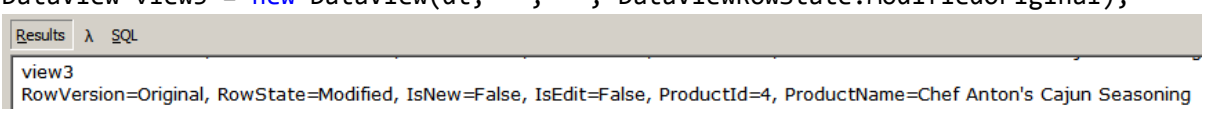

```
// vybereme pouze přidané nebo zrušené řádky
DataView view1 = new DataView(dt, "", "",
                              DataViewRowState.Added | DataViewRowState.Deleted);
Console.WriteLine("view1");
foreach(DataRowView r in view1)
{
    if (r.Row.RowState!=DataRowState.Deleted)
    Console.WriteLine(string.Format("RowVersion={0}, RowState={1}, IsNew={2},
IsEdit={3}, ProductId={4}, ProductName={5}"
        , r.RowVersion, r.Row.RowState, r.IsNew, r.IsEdit
        , r["ProductId"], r["ProductName"], r["ProductName"]));
    else
        Console.WriteLine(string.Format("RowState={0}", r.Row.RowState));
}
```



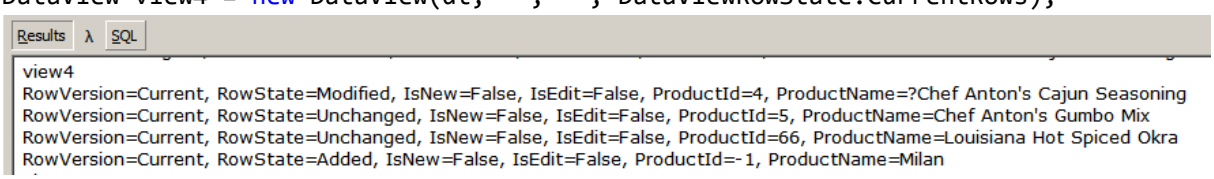

```
// vybereme pouze modifikované a chceme vidět jejich současný (modifikovaný) stav
DataView view2 = new DataView(dt, "", "", DataViewRowState.ModifiedCurrent);
```



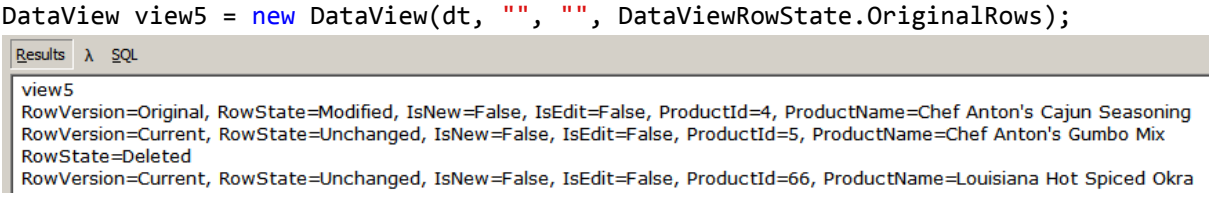

```
// vybereme pouze modifikované a chceme vidět jejich původní (nemodifikovaný) stav
DataView view3 = new DataView(dt, "", "", DataViewRowState.ModifiedOriginal);
```




```
DataView view4 = new DataView(dt, "", "", DataViewRowState.CurrentRows);
```




```
DataView view5 = new DataView(dt, "", "", DataViewRowState.OriginalRows);
```




```
DataView view6 = new DataView(dt);
view6.Sort = "UnitPrice";
```

```
Results λ SQL
view6
RowVersion=Current, RowState=Unchanged, IsNew=False, IsEdit=False, ProductId=66, ProductName=Louisiana Hot Spiced Okra
RowVersion=Current, RowState=Unchanged, IsNew=False, IsEdit=False, ProductId=5, ProductName=Chef Anton's Gumbo Mix
RowVersion=Current, RowState=Modified, IsNew=False, IsEdit=False, ProductId=4, ProductName=?Chef Anton's Cajun Seasoning
RowVersion=Current, RowState=Added, IsNew=False, IsEdit=False, ProductId=-1, ProductName=Milan
```

```
DataView view7 = new DataView(dt);
view7.RowFilter = "UnitPrice < 22";
Console.WriteLine("view7");
```

```
Results λ SQL
view7
RowVersion=Current, RowState=Unchanged, IsNew=False, IsEdit=False, ProductId=5, ProductName=Chef Anton's Gumbo Mix
RowVersion=Current, RowState=Unchanged, IsNew=False, IsEdit=False, ProductId=66, ProductName=Louisiana Hot Spiced Okra
```

```
DataView view8 = new DataView(dt, "UnitPrice > 20",
                "SupplierID DESC, UnitPrice DESC", DataViewRowState.OriginalRows);
```

```
Results λ SQL
view8
RowVersion=Original, RowState=Modified, IsNew=False, IsEdit=False, ProductId=4, ProductName=Chef Anton's Cajun Seasoning
RowVersion=Current, RowState=Unchanged, IsNew=False, IsEdit=False, ProductId=5, ProductName=Chef Anton's Gumbo Mix
RowState=Deleted
```

```
DataView view9 = new DataView(dt);
view9.Sort = "ProductID";
int ind = view9.Find(66);
Console.WriteLine(string.Format("view9.Find(66) = {0}", ind));
```

```
Results λ SQL
view9.Find(66) = 3
```

```
DataView view10 = new DataView(dt);
view10.Sort = "SupplierID";
DataRowView[] aRows = view10.FindRows(2);
if (aRows.Length == 0)
    Console.WriteLine("Neexistuje!");
else
    foreach (DataRowView rowv in aRows)
        Console.WriteLine(string.Format("'{0}'", rowv["ProductName"]));
```

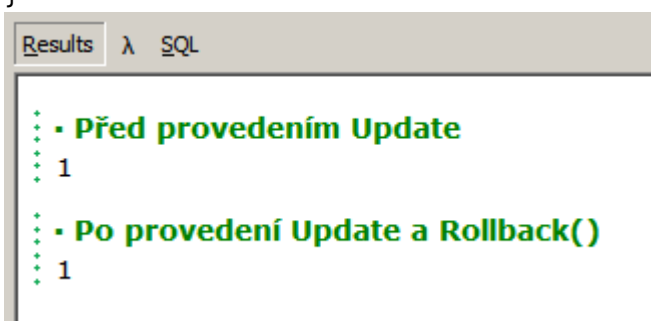
```
Results λ SQL
'?Chef Anton's Cajun Seasoning'
'Chef Anton's Gumbo Mix'
'Louisiana Hot Spiced Okra'
```

ADO.NET – transakce

Práce s transakcemi v režimu online je podobná jako ve Visual FoxPro – otevřeme připojení k SQL Serveru, nastavíme na tomto připojení transakci a pokračujeme dál v práci:

```
string connString =
"Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();
```

```
    SqlCommand cmdDisplay = conn.CreateCommand();
    cmdDisplay.CommandText = @"
SELECT COUNT(*)
FROM dbo.Customers
WHERE Customers.CustomerID = 'ALFKI' AND Customers.Country='Germany'";

    cmdDisplay.ExecuteScalar().Dump("Před provedením Update");
// Vytvoříme transakci
    using (SqlTransaction txn = conn.BeginTransaction())
    {
        SqlCommand cmd = conn.CreateCommand();
        cmd.CommandText = "UPDATE dbo.Customers SET Country = 'SWEDEN' WHERE
CustomerID = 'ALFKI'";
        cmd.Transaction = txn;
// provedeme update
        cmd.ExecuteNonQuery();
// ABYCHOM VIDĚLI, ŽE TRANSAKCE FUNGUJÍ, VRÁTÍME VŠE ZPÁTKY!
        txn.Rollback();
    };

    cmdDisplay.ExecuteScalar().Dump("Po provedení Update a Rollback()");
}
```



ADO.NET – „Typed DataSet“

Můj vztah k typovaným datasetům není nejlepší ☺, dovolím si vás před nimi varovat (i když uznávám, že jsou určitě lidé, kteří s nimi bez problémů pracují).

LINQ

LINQ Vám dovoluje psát dotazy (včetně aktualizačních) na různé datové zdroje jednotným způsobem. Nezáleží tedy na tom, jestli se budete ptát na data uložená na SQL Serveru, v XML souboru, na jména souborů v adresářích, na jména běžících procesů v operačním systému atd. Důležitá je univerzálnost a nezávislost na typu dat. (Poznámka: LINQ je zkratka z „Language INtegrated Query“.)

Zároveň LINQ řeší jeden velký problém ADO.NET. Před chvílí jsme si napsali jako příklad práce s daty v ADO.NET:

```
row["ProductName"] = "Milan";
```

Co se ale stane, pokud přejmenujeme sloupeček na ProductFullName? C# to, stejně jako Visual FoxPro, nepozná a chybu vám oznámí až uživatel telefonem. Bylo by tedy výhodné, kdyby už v okamžiku zápisu programu vám pomáhala IntelliSense a při překladu byla prováděna kontrola na platnost jmen všech sloupečků, počtu a jmen parametrů uložených procedur apod. A tohle přesně vám poskytne LINQ.

Pro nás může být velkým zadostiučiněním, že LINQ vymyslel programátor, který vzpomínal na Visual FoxPro. Z jeho blogu: „It started out as a humble Visual Studio project on my desktop machine way back in the fall of 2003, … I needed expression trees and a basic SQL translator to get ready for the day when I would get my hands on the C# codebase and turn it into language that could actually do something interesting for a change; a language that would know how to query. [Cue the FoxPro fans]…“

Nemáme zde dost místa na to, abychom se mohli věnovat LINQu ze všech stran, a proto si pouze řekneme, že existují dvě varianty LINQu (syntaxe je stejná, ale interně se liší):

- LINQ, který pracuje s objekty, které má všechny k dispozici v paměti a přístup k nim je tedy bez režie (IEnumerable <T>)
- LINQ, který pracuje s objekty, na které se musí teprve ptát a ke kterým se dá optimalizovat přístup (IQueryable<T>). Typicky jde o data na SQL Serveru, neboť každý dotaz na data má podstatně větší režii než předchozí případ.

My se omezíme pouze na práci se SQL Serverem.

Přehled výhod LINQu

Dovolím si uvést pouhý výčet výhod jazyka LINQ v bodech:

1) Snadno lze převádět data do/z objektů
2) Jedna syntaxe pro práci s různými druhy dat
3) Silně typovaný kód
4) Lze jednoduše kombinovat data z různých zdrojů
5) Zjednodušení práce
6) Kupodivu má LINQ vysoký výkon

7) Bezpečný přístup k datům SQL Serveru (vše přes parametry, nehrozí tedy útok typu „parameter injection“)
8) LINQ je deklarativní

Generování zdrojového programu pro LINQ to SQL a SQLMetal

Aby mohl LINQ kontrolovat platnost všech jmen sloupečků atd., musí se vygenerovat zdrojový program, který bude pro každý objekt v databázi (tabulku, uloženou proceduru, …) obsahovat speciální třídu. Jelikož používáme LINQPad, máme usnadněnou práci – tyto třídy pro nás generuje LINQPad dynamicky v okamžiku, když jsou potřeba, takže se o tyto třídy nemusíme vůbec starat. Při normálním programování je ale potřeba je vygenerovat. K tomu slouží (kromě návrhářů, které osobně nemám příliš rád ☺) i program SQLMetal.exe (který se schovává v adresáři C:\Program Files\Microsoft SDKs\Windows\v6.0A\Bin\SqlMetal.exe). Pro vygenerování tříd ho stačí spustit s těmito parametry:

```
SQLMetal /server:local /database:Northwind /views /sprocs /functions
                                        /serialization:unidirectional /code:Northwind.cs
```

Výsledkem bude program v C#, který bude obsahovat například následující třídu (která je zde uvedena ve výrazně zkráceném tvaru, většinu příkazů jsem pro přehlednost smazal):

```
// Tato třída odpovídá tabulce Products
[Table(Name="dbo.Products")]
public partial class Products : INotifyPropertyChanging, INotifyPropertyChanged
{
    …
// ProductID odpovídá v tabulce položka se stejným jménem,
// je to primární klíč, typu INT
    [Column(Storage="_ProductID", AutoSync=AutoSync.OnInsert,
        DbType="Int NOT NULL IDENTITY", IsPrimaryKey=true, IsDbGenerated=true)]
    [DataMember(Order=1)]
    public int ProductID
    { … }
// ProductName je v tabulce položka se stejným jménem, nesmí být NULL,
// typu NVARCHAR(40)
    [Column(Storage="_ProductName", DbType="NVarChar(40) NOT NULL",
      CanBeNull=false)]
    [DataMember(Order=2)]
    public string ProductName
    { … }
    …
// Jeden objekt třídy Product může být obsažen ve více detailech objednávek
// (vazba 1:N)
    [Association(Name="FK_Order_Details_Products", Storage="_OrderDetails",
                ThisKey="ProductID", OtherKey="ProductID", DeleteRule="NO ACTION")]
    [DataMember(Order=11, EmitDefaultValue=false)]
    public EntitySet<OrderDetails> OrderDetails
    { … }
// Jeden objekt třídy Product může být obsažen maximálně v jedné kategorii
```

```
// (vazba 1:1)
    [Association(Name="FK_Products_Categories", Storage="_Categories",
                 ThisKey="CategoryID", OtherKey="CategoryID", IsForeignKey=true)]
    public Categories Categories
    { … }
    …
}
```

Díky popisu vztahu třídy a tabulky na SQL Serveru pomocí atributů lze v LINQu dělat skutečně zajímavé věci.

Znovu připomínám, že my nemusíme používat v LINQPadu SQLMetal, protože LINQPad dělá to samé pro nás automaticky.

DataContext

Dříve než se pustíme do prvního příkladu, musíme si něco povědět o datovém kontextu. Datový kontext (DataContext) se stará v LINQu o vše: zpřístupní vám třídy odpovídající tabulce na SQL Serveru, umožní vám zadat výběrové podmínky, bude sledovat všechny změny v datech a dokáže je později poslat ve správném formátu na SQL Server. Nejprve si tedy musíme datový kontext vytvořit:

```
string connString =
  "Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";
DataContext dataContext = new DataContext(connString);
```

a pak ho používat. Například v následujících příkladech chceme pracovat s tabulkou Products:

```
dataContext.GetTable<Products>();
```

nebo v

```
dataContext.Products.Dump();
```

LINQPad před námi opět datový kontext schová, takže pokud s ním chcete přece jenom pracovat (protože v normálním programu s ním pracovat musíte), musíte si zde pomoci malým trikem:

```
var dataContext = this;
```

a ihned máte správný kontext k dispozici. Uveďme si příklady na práci s kontextem:

```
string connString =
  "Data Source=(local);Initial Catalog=Northwind;Integrated Security=True;";

DataContext dataContext = new DataContext(connString);
dataContext.GetTable<Products>().Dump("dataContext.GetTable<Products>()");
// tohle v LINQPadu nefunguje, ale v programu ano
//   dataContext.Products.Dump();
```

Results λ SQL

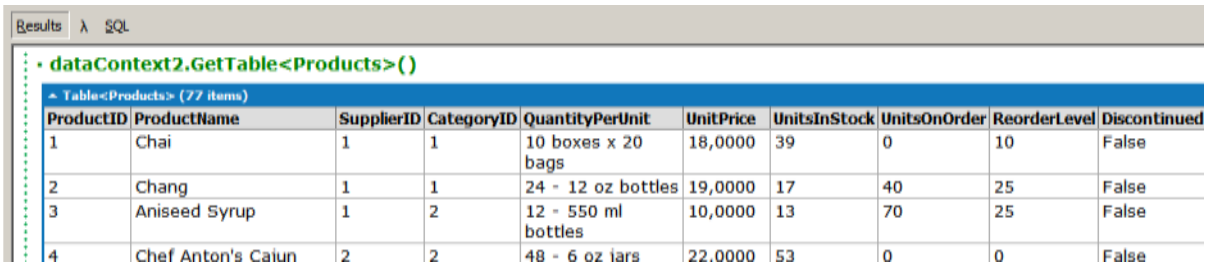
dataContext.GetTable<Products>()

Table<Products> (77 items)

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
1	Chai	1	1	10 boxes x 20 bags	18,0000	39	0	10	False
2	Chang	1	1	24 - 12 oz bottles	19,0000	17	40	25	False
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10,0000	13	70	25	False
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22,0000	53	0	0	False
5	Chef Anton's Gumbo	2	2	36 boxes	21,3500	0	0	0	True

…

```
var dataContext2 = this;
dataContext2.GetTable<Products>().Dump("dataContext2.GetTable<Products>()");
// tohle už v LINQPadu funguje
dataContext2.Products.Dump("dataContext2.Products");
```



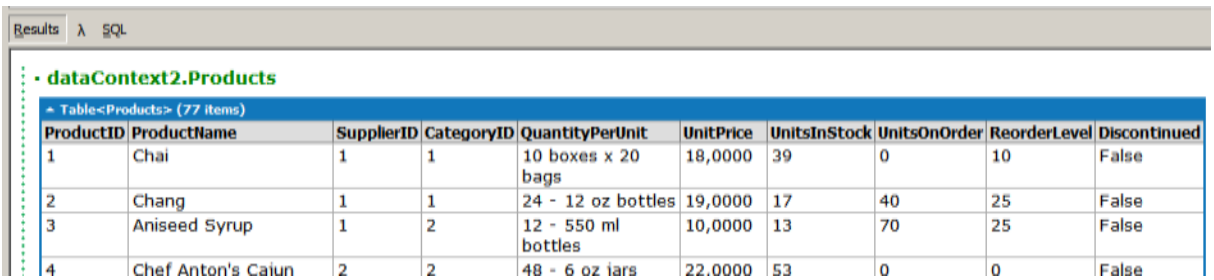
Results λ SQL

dataContext2.GetTable<Products>()

Table<Products> (77 items)

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
1	Chai	1	1	10 boxes x 20 bags	18,0000	39	0	10	False
2	Chang	1	1	24 - 12 oz bottles	19,0000	17	40	25	False
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10,0000	13	70	25	False
4	Chef Anton's Cajun	2	2	48 - 6 oz jars	22,0000	53	0	0	False

…



Results λ SQL

dataContext2.Products

Table<Products> (77 items)

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
1	Chai	1	1	10 boxes x 20 bags	18,0000	39	0	10	False
2	Chang	1	1	24 - 12 oz bottles	19,0000	17	40	25	False
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10,0000	13	70	25	False
4	Chef Anton's Cajun	2	2	48 - 6 oz jars	22,0000	53	0	0	False

První příklad na LINQ to SQL

Postupně si zde uvedeme několik příkladů na LINQ dotazy. Všimněte si, že nejprve se dotaz definuje a teprve pak se provádí.

```
var dataContext = this;

// definujeme dotaz - POZOR! ZATÍM SE NEPROVÁDÍ DOTAZ NA SQL Server!!!
var qry1 = from p in dataContext.Products
    where p.UnitPrice>60
    select p;
```

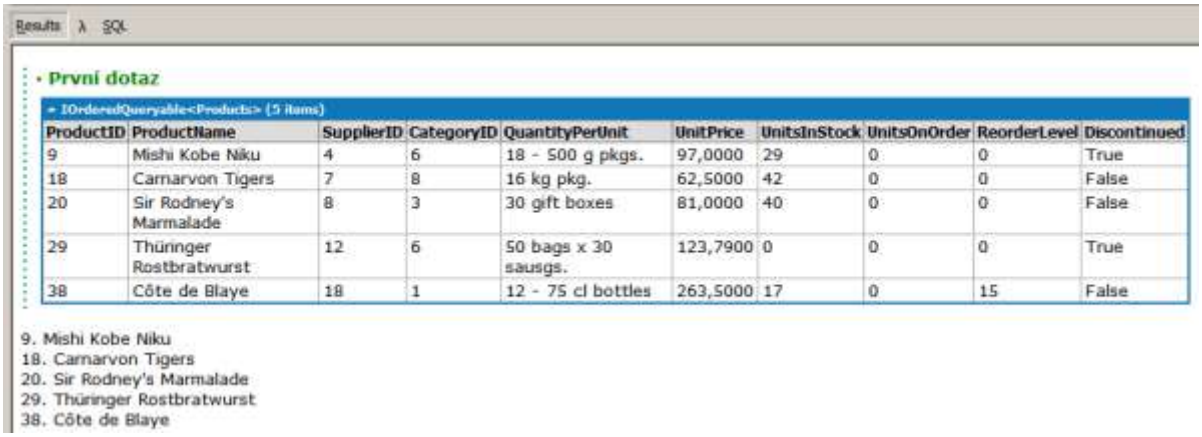
Zatím máme pouze definovaný jednoduchý dotaz, ještě jsme ho ale neprovedli. Co si na něm můžeme všimnout:

- Syntaxe trochu připomíná SELECT jazyka SQL, ale i zde jsou vidět odlišnosti. První z nich je, že v C# se musí začínat klauzulí „from“, a nikoli „select“. Na další rozdíly narazíme později.
- Data vybíráme z dataContext.*JménoTabulky*.
- V klauzuli „where“ se provádí kontrola na jméno sloupečku. Pokud bychom se spletli, program by nebylo možné přeložit.

Nyní tento dotaz 2x provedeme (tedy se 2x pošle dotaz na SQL Server).

```
// My budeme provádět výpisy pomocí .Dump()
// Teprve teď se dotaz na SQL Server provádí!
qry1.Dump("První dotaz");
// Ale v programu bychom provedli vlastní zpracování třeba takto
// (a teď se dotaz na SQL Server provádí znova!)
foreach(var p in qry1)
```

```
    Console.WriteLine(p.ProductID.ToString() + ". " + p.ProductName);
```



Results λ SQL

• První dotaz

IOrderedQueryable<Products> (5 items)

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97,0000	29	0	0	True
18	Carnarvon Tigers	7	8	16 kg pkg.	62,5000	42	0	0	False
20	Sir Rodney's Marmalade	8	3	30 gift boxes	81,0000	40	0	0	False
29	Thüringer Rostbratwurst	12	6	50 bags x 30 sausgs.	123,7900	0	0	0	True
38	Côte de Blaye	18	1	12 - 75 cl bottles	263,5000	17	0	15	False

9. Mishi Kobe Niku
18. Carnarvon Tigers
20. Sir Rodney's Marmalade
29. Thüringer Rostbratwurst
38. Côte de Blaye

Znovu bych chtěl zdůraznit, že se dotaz posílá na SQL Server znova. Pokud bychom tomu chtěli zabránit, stačí jednoduše převést výsledek do List<Products> (o tom zde ale nebudeme mluvit).

Mohlo by nás také zajímat, jaký příkaz SELECT jazyka SQL vygeneroval LINQ to SQL, což zjistíme pomocí GetCommand():

```
// Jak se dozvíme, co se vlastně provedlo?
this.GetCommand(qry1).CommandText.Dump("SQL pro první dotaz");
// celý SQL se nevešel
```

Results λ SQL

• SQL pro první dotaz

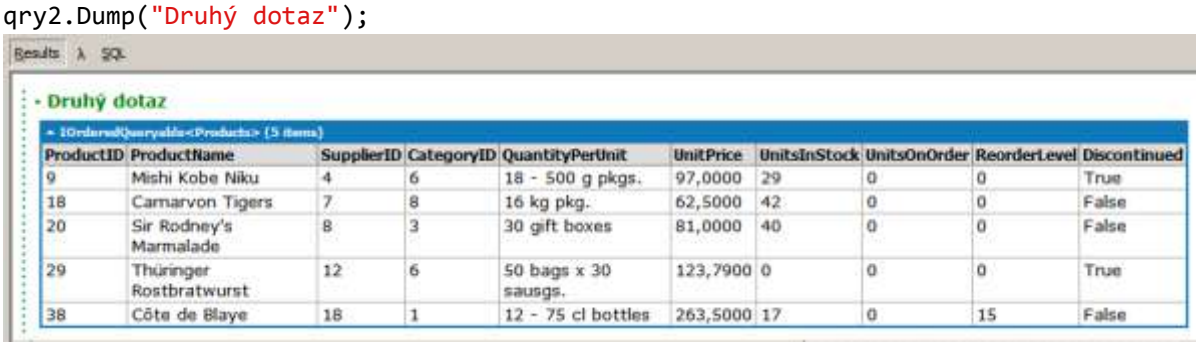
```
SELECT [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID], [t0].[CategoryID], [t0].[QuantityPerUnit], [t0].[UnitPrice], [t0].[UnitsInS
FROM [Products] AS [t0]
WHERE [t0].[UnitPrice] > @p0
```

Vidíme, že se dotaz parametrizuje, aby SQL Server mohl použít prováděcí plán opakovaně.

Možná si teď říkáte, že je to jenom trochu pozměněný SELECT a že to o moc víc neumí. Tak to tedy rozhodně není pravda. Existuje ještě druhý zápis – pomocí takzvaných lambda funkcí. A pomocí něj dokážete napsat věci, které v „normálním“ zápisu nedokážete (a zase naopak pro něco je výhodnější „normální“ zápis, takže je dobré oba kombinovat podle potřeby).

Následuje stejný dotaz zapsaný pomocí lambda funkcí:

```
var qry2 = dataContext.Products
    .Where(p => p.UnitPrice>60)
    .Select(p => p);
qry2.Dump("Druhý dotaz");
```



Results λ SQL

• Druhý dotaz

IOrderedQueryable<Products> (5 items)

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97,0000	29	0	0	True
18	Carnarvon Tigers	7	8	16 kg pkg.	62,5000	42	0	0	False
20	Sir Rodney's Marmalade	8	3	30 gift boxes	81,0000	40	0	0	False
29	Thüringer Rostbratwurst	12	6	50 bags x 30 sausgs.	123,7900	0	0	0	True
38	Côte de Blaye	18	1	12 - 75 cl bottles	263,5000	17	0	15	False

V čem se tento zápis odlišuje:

- Po jménu tabulky postupně voláme metody (proto *.Metoda(parametry)*) jako Where() a Select().
- Parametry metod jsou zadávány pomocí lambda funkce:
  - před šipkou (zapsanou pomocí rovnítka a větší než) je argument nebo seznam argumentů v závorkách,
  - za šipkou je výraz, který je speciální pro každou metodu, takže například:
    - pro Where je to výběrová podmínka
    - pro Select je to to, co má SELECT vrátit (seznam sloupečků).
- Každý zápis pomocí „normálního“ způsobu lze převést na lambda zápis, opačně to není vždy možné

Osobně spíše preferuji lambda zápis, ale vy si musíte vybrat, s čím se vám bude pracovat lépe. I v příkladech budu uvádět jeden nebo druhý způsob, abyste si zvykli na oba (nekamenovat prosím ☺).

Přehled lambda operátorů

V následující tabulce vidíte přehled všech operátorů použitelných v lambda zápisu.

Kategorie	***Operátory***
Filtering	Where, Distinct, Take, TakeWhile, Skip, SkipWhile
Projecting	Select, SelectMany
Joining	Join, GroupJoin
Ordering	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy
Set	Concat, Union, Intersect, Except
Conversion (import)	OfType, Cast
Conversion (export)	ToArray, ToList, ToDictionary, ToLookup, AsEnumerable, AsQueryable
Element	First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault, ElementAt, ElementAtOrDefault, DefaultIfEmpty
Aggregation	Aggregate, Average, Count, LongCount, Sum, Max, Min
Quantifiers	All, Any, Contains, SequenceEqual
Generation	Empty, Range, Repeat

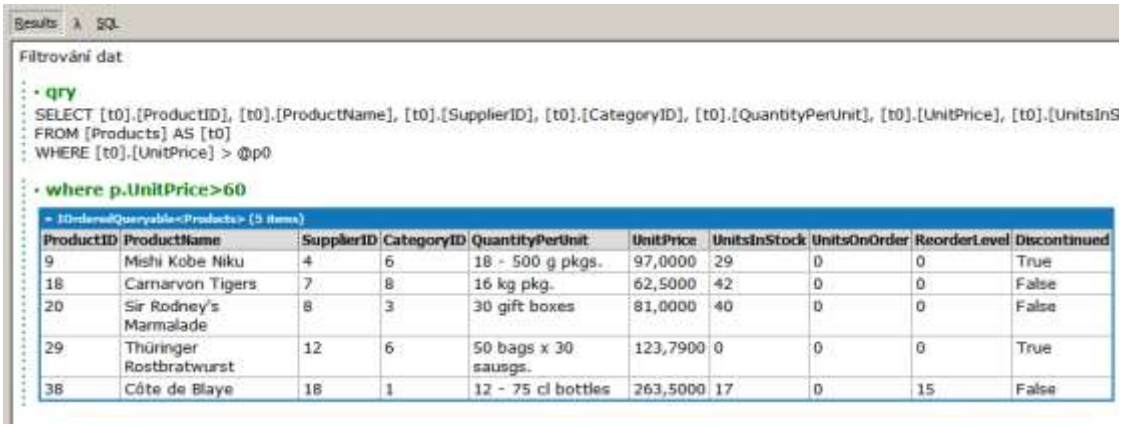
Vidíte sami, že nabídka je skutečně bohatá. Ale nelekejte se, my si jich dál v textu ukážeme pouze několik vybraných.

Filtrování dat

Filtrování dat jsme si ukázali už před chvílí, takže první příklad bude pouze pro zopakování, ale druhý už bude zajímavější. Ukážeme si, že se v podmínce můžeme přímo odkazovat na tabulku svázanou s hlavní tabulkou pomocí relace (vazba 1:n). Zde budeme vybírat pouze ty objednávky, kde je počet položek větší než 5. Abychom viděli, že to není úplně jednoduché, vypíšeme si i SQL příkazy posílané na SQL Server:

```
var dataContext = this;

"Filtrování dat".Dump();
```

```
// Vybereme výrobky se cenou >60
var qry1 = from p in dataContext.Products
    where p.UnitPrice>60        // jednoduchý výběr
    select p;
this.GetCommand(qry1).CommandText.Dump("qry1");
qry1.Dump("where p.UnitPrice>60");
```



Filtrování dat

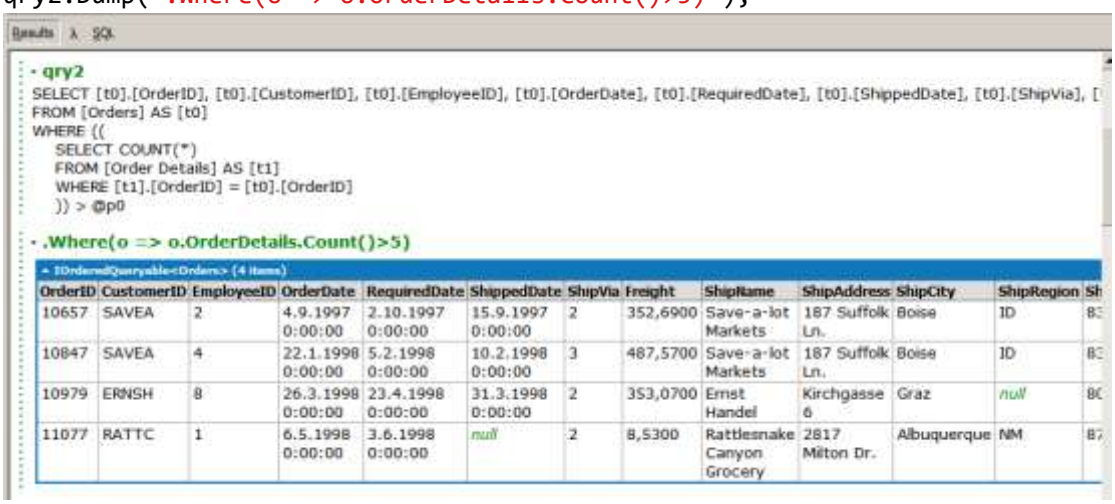
- qry

```
SELECT [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID], [t0].[CategoryID], [t0].[QuantityPerUnit], [t0].[UnitPrice], [t0].[UnitsInS
FROM [Products] AS [t0]
WHERE [t0].[UnitPrice] > @p0
```

- where p.UnitPrice>60

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97,0000	29	0	0	True
18	Carnarvon Tigers	7	8	16 kg pkg.	62,5000	42	0	0	False
20	Sir Rodney's Marmalade	8	3	30 gift boxes	81,0000	40	0	0	False
29	Thüringer Rostbratwurst	12	6	50 bags x 30 sausgs.	123,7900	0	0	0	True
38	Côte de Blaye	18	1	12 - 75 cl bottles	263,5000	17	0	15	False

```
// Vybereme objednávky s počtem položek >5
var qry2 = dataContext.Orders
// POZOR! Odkazujeme se přes relaci do podřízené tabulky OrderDetails
// A pomocí .Count() zjistíme počet řádků navázaných přes relaci
    .Where(o => o.OrderDetails.Count()>5)
// Zápis o=>o znamená: „vyber všechno“
    .Select(o => o);
this.GetCommand(qry2).CommandText.Dump("qry2");
qry2.Dump(".Where(o => o.OrderDetails.Count()>5)");
```



- qry2

```
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate], [t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [
FROM [Orders] AS [t0]
WHERE ((
    SELECT COUNT(*)
    FROM [Order Details] AS [t1]
    WHERE [t1].[OrderID] = [t0].[OrderID]
    )) > @p0
```

- .Where(o => o.OrderDetails.Count()>5)

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress	ShipCity	ShipRegion	Sh
10657	SAVEA	2	4.9.1997 0:00:00	2.10.1997 0:00:00	15.9.1997 0:00:00	2	352,6900	Save-a-lot Markets	187 Suffolk Ln.	Boise	ID	8
10847	SAVEA	4	22.1.1998 0:00:00	5.2.1998 0:00:00	10.2.1998 0:00:00	3	487,5700	Save-a-lot Markets	187 Suffolk Ln.	Boise	ID	8
10979	ERNSH	8	26.3.1998 0:00:00	23.4.1998 0:00:00	31.3.1998 0:00:00	2	353,0700	Ernst Handel	Kirchgasse 6	Graz	null	8
11077	RATTC	1	6.5.1998 0:00:00	3.6.1998 0:00:00	null	2	8,5300	Rattlesnake Canyon Grocery	2817 Milton Dr.	Albuquerque	NM	8

Projekce

Nelekejte se, projekce znamená pouze to, že si ukážeme, jak lze specifikovat sloupečky výsledku. I když… Možná budete překvapeni, co všechno se dá udělat.

Nejprve použijeme výběr pomocí Select. Ukážeme si 3 příklady:

1. Jednoduchý výběr sloupečků ProductID, ProductName, UnitPrice z tabulky Products. Stačí použít konstrukci „new {*seznam vybraných položek*}“.
2. Druhý příklad je podobný, navíc přidáme sloupeček s počtem položek v navázané tabulce položek objednávek. Všimněte si, jak je zápis jednoduchý.
3. A ve třetím příkladě máme jen tři sloupečky! Ano, skutečně tři! První sloupeček totiž obsahuje celý řádek tabulky Orders, ve druhém je počet řádků položek objednávky a ve třetím sloupečku je celková cena (bez slevy) objednávky. O SQL dotazu ani nebudu mluvit…

```
var dataContext = this;

"Projekce".Dump();

var qry1 = from p in dataContext.Products
    where p.UnitPrice>60
// Výsledkem mají být 3 sloupečky
    select new {p.ProductID, p.ProductName, p.UnitPrice};
this.GetCommand(qry1).CommandText.Dump("qry1");
qry1.Dump("select new {p.ProductID, p.ProductName, p.UnitPrice}");
```

ProductID	ProductName	UnitPrice
9	Mishi Kobe Niku	97,0000
18	Carnarvon Tigers	62,5000
20	Sir Rodney's Marmalade	81,0000
29	Thüringer Rostbratwurst	123,7900
38	Côte de Blaye	263,5000

```
var qry2 = dataContext.Orders
    .Where(o => o.OrderDetails.Count()>5)
// Výsledkem mají být 4 sloupečky
    .Select(o => new {
                    o.OrderID,
                    o.CustomerID,
                    o.ShipName,
// počet vět v podřízené tabulce
                    Cnt = o.OrderDetails.Count()});
this.GetCommand(qry2).CommandText.Dump("qry2");
qry2.Dump(".Select(o => new {o.OrderID, o.CustomerID, o.ShipName, "
                                        +"Cnt= o.OrderDetails.Count()})");
```

```
qry2
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[ShipName], (
    SELECT COUNT(*)
    FROM [Order Details] AS [t2]
    WHERE [t2].[OrderID] = [t0].[OrderID]
    ) AS [Cnt]
FROM [Orders] AS [t0]
WHERE ((
    SELECT COUNT(*)
    FROM [Order Details] AS [t1]
    WHERE [t1].[OrderID] = [t0].[OrderID]
    )) > @p0

.Select(o => new {o.OrderID, o.CustomerID, o.ShipName, Cnt= o.OrderDetails.Count()})
```

IOrderedQueryable<> (4 items)

OrderID	CustomerID	ShipName	Cnt
10657	SAVEA	Save-a-lot Markets	6
10847	SAVEA	Save-a-lot Markets	6
10979	ERNSH	Ernst Handel	6
11077	RATTC	Rattlesnake Canyon Grocery	25

```
var qry3 = dataContext.Orders
    .Where(o => o.OrderDetails.Count()>5)
    .Select(o => new {
// sloupeček obsahuje vnořený celý objekt Orders
                    o,
// počet vět v podřízené tabulce
                    Cnt = o.OrderDetails.Count(),
// součet všech cena*množství pro všechny položky objednávky
                    Sum = o.OrderDetails.Sum(od=>od.UnitPrice*od.Quantity)});
this.GetCommand(qry3).CommandText.Dump("qry3");
qry3.Dump(".Select(o => new {o, Cnt= o.OrderDetails.Count(),"
                    +"Sum=o.OrderDetails.Sum(od=>od.UnitPrice*od.Quantity)}");
```

```
qry3
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate], [t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia
    SELECT COUNT(*)
    FROM [Order Details] AS [t2]
    WHERE [t2].[OrderID] = [t0].[OrderID]
    ) AS [Cnt], (
    SELECT SUM([t4].[value])
    FROM (
        SELECT [t3].[UnitPrice] * (CONVERT(Decimal(29,4),[t3].[Quantity])) AS [value], [t3].[OrderID]
        FROM [Order Details] AS [t3]
        ) AS [t4]
    WHERE [t4].[OrderID] = [t0].[OrderID]
    ) AS [Sum]
FROM [Orders] AS [t0]
WHERE ((
    SELECT COUNT(*)
    FROM [Order Details] AS [t1]
    WHERE [t1].[OrderID] = [t0].[OrderID]
    )) > @p0

.Select(o => new {o, Cnt= o.OrderDetails.Count(), Sum=o.OrderDetails.Sum(od=>od.UnitPrice*od.Quantity)}
```

IOrderedQueryable<> (4 items)

o	Cnt	Sum
Orders LINQPad.User.Orders OrderID: 10657 CustomerID: SAVEA EmployeeID: 2 OrderDate: 4.9.1997 0:00:00 RequiredDate: 2.10.1997 0:00:00 ShippedDate: 15.9.1997 0:00:00 ShipVia: 2 Freight: 352,6900 ShipName: Save-a-lot Markets ShipAddress: 187 Suffolk Ln. ShipCity: Boise ShipRegion: ID ShipPostalCode: 83720 ShipCountry: USA	6	4371,600000
Orders LINQPad.User.Orders OrderID: 10847 CustomerID: SAVEA EmployeeID: 4	6	6164,900000

A teď se dostáváme k SelectMany. Normální Select má v mnoha případech nevýhodu, že začne vnořovat jeden objekt do druhého, takže za chvilku se nám to nemusí líbit. Tady přichází na pomoc SelectMany, který vše převede na normální nevnořenou strukturu. Příklad ukáže víc. Chtěl bych poznamenat, že druhý a třetí příklad jsou stejné, pouze zapsané jako lambda výraz nebo jako „normální“ příkaz:

```
var dataContext = this;

"Projekce SelectMany".Dump();

var qry1 = dataContext.Orders
    .Where(o => o.OrderDetails.Count()>15)
    .Select(o => new {
// sloupeček obsahuje vnořený celý objekt Orders
                    o,
// sloupeček obsahuje vnořenou kolekci VŠECH detailních položek OrderDetails
                    o.OrderDetails});
qry1.Dump(".Select(o => new {o, o.OrderDetails});");
```

Results λ SQL

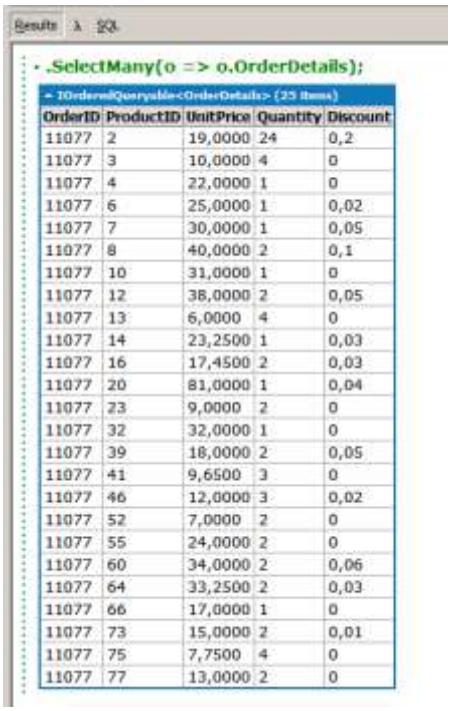
Projekce SelectMany

.Select(o => new {o, o.OrderDetails});

IOrderedQueryable<> (1 item)

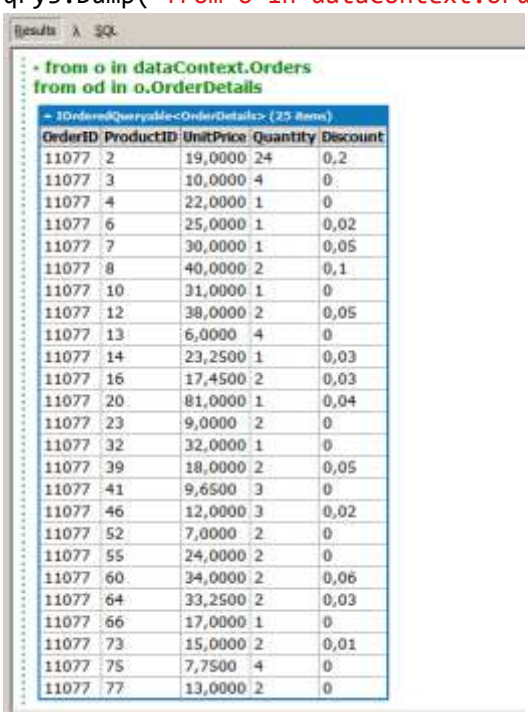
o		OrderDetails				
Orders		EntitySet<OrderDetails> (25 items)				
LINQPad.User.Orders		OrderID	ProductID	UnitPrice	Quantity	Discount
OrderID	11077	11077	2	19,0000	24	0,2
CustomerID	RATTC	11077	3	10,0000	4	0
EmployeeID	1	11077	4	22,0000	1	0
OrderDate	6.5.1998 0:00:00	11077	6	25,0000	1	0,02
RequiredDate	3.6.1998 0:00:00	11077	7	30,0000	1	0,05
ShippedDate	null	11077	8	40,0000	2	0,1
ShipVia	2	11077	10	31,0000	1	0
Freight	8,5300	11077	12	38,0000	2	0,05
ShipName	Rattlesnake Canyon Grocery	11077	13	6,0000	4	0
ShipAddress	2817 Milton Dr.	11077	14	23,2500	1	0,03
ShipCity	Albuquerque	11077	16	17,4500	2	0,03
ShipRegion	NM	11077	20	81,0000	1	0,04
ShipPostalCode	87110	11077	23	9,0000	2	0
ShipCountry	USA	11077	32	32,0000	1	0
		11077	39	18,0000	2	0,05
		11077	41	9,6500	3	0
		11077	46	12,0000	3	0,02
		11077	52	7,0000	2	0
		11077	55	24,0000	2	0
		11077	60	34,0000	2	0,06
		11077	64	33,2500	2	0,03
		11077	66	17,0000	1	0
		11077	73	15,0000	2	0,01
		11077	75	7,7500	4	0
		11077	77	13,0000	2	0

```
var qry2 = dataContext.Orders
    .Where(o => o.OrderDetails.Count()>15)
    .SelectMany(o => o.OrderDetails);
qry2.Dump(".SelectMany(o => o.OrderDetails);");
```



OrderID	ProductID	UnitPrice	Quantity	Discount
11077	2	19,0000	24	0,2
11077	3	10,0000	4	0
11077	4	22,0000	1	0
11077	6	25,0000	1	0,02
11077	7	30,0000	1	0,05
11077	8	40,0000	2	0,1
11077	10	31,0000	1	0
11077	12	38,0000	2	0,05
11077	13	6,0000	4	0
11077	14	23,2500	1	0,03
11077	16	17,4500	2	0,03
11077	20	81,0000	1	0,04
11077	23	9,0000	2	0
11077	32	32,0000	1	0
11077	39	18,0000	2	0,05
11077	41	9,6500	3	0
11077	46	12,0000	3	0,02
11077	52	7,0000	2	0
11077	55	24,0000	2	0
11077	60	34,0000	2	0,06
11077	64	33,2500	2	0,03
11077	66	17,0000	1	0
11077	73	15,0000	2	0,01
11077	75	7,7500	4	0
11077	77	13,0000	2	0

```
var qry3 =
    from o in dataContext.Orders
    from od in o.OrderDetails
    where o.OrderDetails.Count()>15
    select od;
qry3.Dump("from o in dataContext.Orders\r\nfrom od in o.OrderDetails");
```



OrderID	ProductID	UnitPrice	Quantity	Discount
11077	2	19,0000	24	0,2
11077	3	10,0000	4	0
11077	4	22,0000	1	0
11077	6	25,0000	1	0,02
11077	7	30,0000	1	0,05
11077	8	40,0000	2	0,1
11077	10	31,0000	1	0
11077	12	38,0000	2	0,05
11077	13	6,0000	4	0
11077	14	23,2500	1	0,03
11077	16	17,4500	2	0,03
11077	20	81,0000	1	0,04
11077	23	9,0000	2	0
11077	32	32,0000	1	0
11077	39	18,0000	2	0,05
11077	41	9,6500	3	0
11077	46	12,0000	3	0,02
11077	52	7,0000	2	0
11077	55	24,0000	2	0
11077	60	34,0000	2	0,06
11077	64	33,2500	2	0,03
11077	66	17,0000	1	0
11077	73	15,0000	2	0,01
11077	75	7,7500	4	0
11077	77	13,0000	2	0

Spojování více tabulek

Spojování tabulek odpovídá normálnímu JOIN…ON. V tomto případě je výhodnější nepoužívat lambda výrazy, ale „normální“ syntaxi. Ani tak to není úplně jednoduché, a proto se nenechte odradit (a případně tuto kapitolku pro začátek ignorujte ☺).

Ukážeme si tři příklady na spojení zákazníků a objednávek:

1. Normální join – všechna data o zákaznících a jejich objednávkách budou normálně v řádcích, takže nemáme žádné informace o zákaznících bez objednávek.
2. Left outer join – řádky budou obsahovat data o zákaznících, případné objednávky budou jako vnořený objekt. Máme informace o zákaznících bez objednávek, ale musíme pracovat s vnořenými objekty
3. Jako 2. příklad – ale navíc převod do normální tabulky.

```
var dataContext = this;

"Join - INNER join a bez vnořených objektů".Dump();
var qry1 =
    from c in dataContext.Customers
    where c.City == "Madrid"
    join p in dataContext.Orders
      on c.CustomerID equals p.CustomerID
    select new {c.CompanyName, p.OrderID};
this.GetCommand(qry1).CommandText.Dump("qry1");
qry1.Dump();
```

Results λ SQL

Join - INNER join a bez vnořených objektů

• qry1

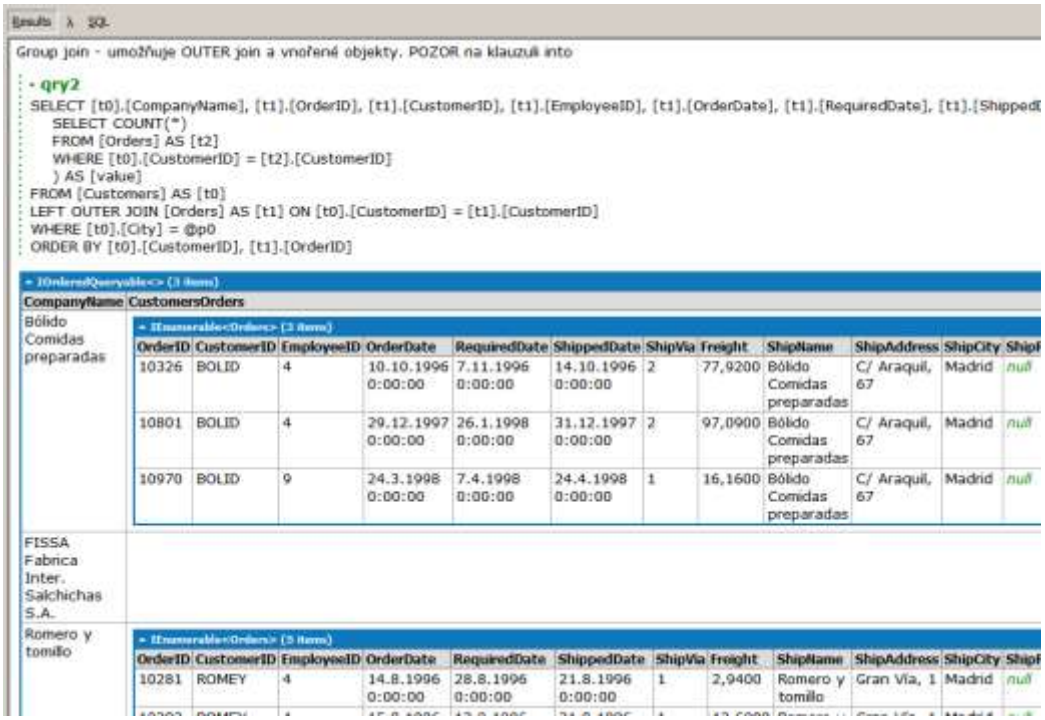
```
SELECT [t0].[CompanyName], [t1].[OrderID]
FROM [Customers] AS [t0]
INNER JOIN [Orders] AS [t1] ON [t0].[CustomerID] = [t1].[CustomerID]
WHERE [t0].[City] = @p0
```

IOrderedQueryable<> (8 items)

CompanyName	OrderID
Bólido Comidas preparadas	10326
Bólido Comidas preparadas	10801
Bólido Comidas preparadas	10970
Romero y tomillo	10281
Romero y tomillo	10282
Romero y tomillo	10306
Romero y tomillo	10917
Romero y tomillo	11013

```
"Group join - umožňuje OUTER join a vnořené objekty. POZOR na klauzuli
into".Dump();
var qry2 =
    from c in dataContext.Customers
    where c.City == "Madrid"
    join p in dataContext.Orders
      on c.CustomerID equals p.CustomerID
    into CustomersOrders
    select new {c.CompanyName, CustomersOrders};
this.GetCommand(qry2).CommandText.Dump("qry2");
```

```
qry2.Dump();
```


Group join - umožňuje OUTER join a vnořené objekty. POZOR na klauzuli into

- qry2

```
SELECT [t0].[CompanyName], [t1].[OrderID], [t1].[CustomerID], [t1].[EmployeeID], [t1].[OrderDate], [t1].[RequiredDate], [t1].[Shipped
    SELECT COUNT(*)
    FROM [Orders] AS [t2]
    WHERE [t0].[CustomerID] = [t2].[CustomerID]
    ) AS [value]
FROM [Customers] AS [t0]
LEFT OUTER JOIN [Orders] AS [t1] ON [t0].[CustomerID] = [t1].[CustomerID]
WHERE [t0].[City] = @p0
ORDER BY [t0].[CustomerID], [t1].[OrderID]
```

CompanyName	CustomersOrders

Bólido Comidas preparadas

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress	ShipCity	ShipR
10326	BOLID	4	10.10.1996 0:00:00	7.11.1996 0:00:00	14.10.1996 0:00:00	2	77,9200	Bólido Comidas preparadas	C/ Araquil, 67	Madrid	null
10801	BOLID	4	29.12.1997 0:00:00	26.1.1998 0:00:00	31.12.1997 0:00:00	2	97,0900	Bólido Comidas preparadas	C/ Araquil, 67	Madrid	null
10970	BOLID	9	24.3.1998 0:00:00	7.4.1998 0:00:00	24.4.1998 0:00:00	1	16,1600	Bólido Comidas preparadas	C/ Araquil, 67	Madrid	null

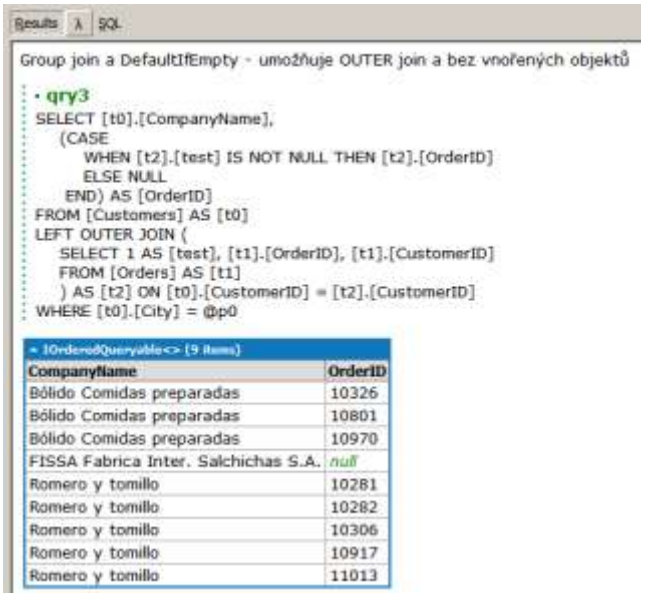
FISSA Fabrica Inter. Salchichas S.A.

Romero y tomillo

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress	ShipCity	ShipR
10281	ROMEY	4	14.8.1996 0:00:00	28.8.1996 0:00:00	21.8.1996 0:00:00	1	2,9400	Romero y tomillo	Gran Vía, 1	Madrid	null

```
"Group join a DefaultIfEmpty - umožňuje OUTER join a bez vnořených objektů".Dump();
var qry3 =
    from c in dataContext.Customers
    where c.City == "Madrid"
    join p in dataContext.Orders
      on c.CustomerID equals p.CustomerID
    into CustomersOrders
    from co in CustomersOrders.DefaultIfEmpty()
// Nelekejte se ☺. Klauzuli:
//    co!=null?(int?)co.OrderID:null
// si můžete přeložit jako:
//    IIF(co<>NULL, co.OrderID, NULL)
    select new {c.CompanyName, OrderID = co!=null?(int?)co.OrderID:null};
this.GetCommand(qry3).CommandText.Dump("qry3");
```

```
qry3.Dump();
```



Třídění

Třídění nás skutečně nemá čím překvapit, takže jenom jeden příklad:

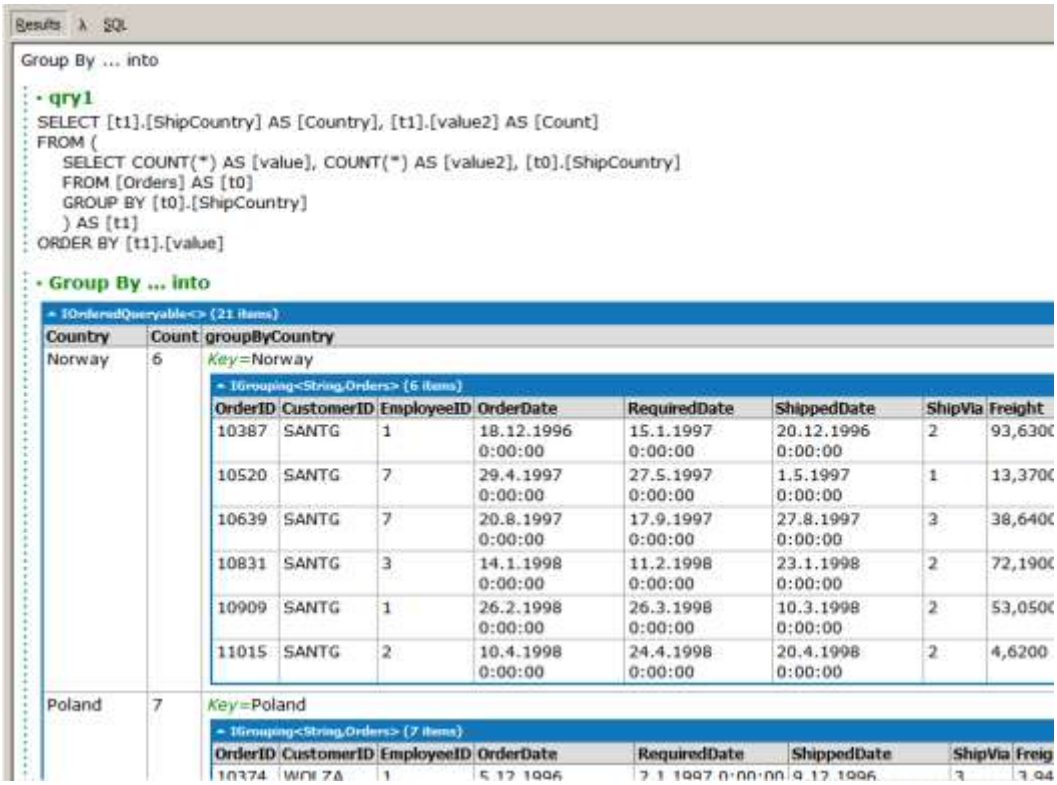
```
var dataContext = this;

"Order By".Dump();
var qry1 =
    from c in dataContext.Customers
    where c.City == "Madrid"
    orderby c.Address descending, c.CompanyName
    select c;
qry1.Dump();
```

Agregace (seskupování)

Pro vytváření skupin a práce s těmito skupinami jsem připravil jednoduchý příklad rozdělení objednávek podle zemí a navíc zjištění počtu objednávek:

```
var dataContext = this;

"Group By ... into".Dump();
var qry1 =
    from p in dataContext.Orders
    group p by p.ShipCountry into groupByCountry
    orderby groupByCountry.Count()
```

```
    select new {
      Country = groupByCountry.Key,
      Count = groupByCountry.Count(),
      groupByCountry};
this.GetCommand(qry1).CommandText.Dump("qry1");
qry1.Dump("Group By ... into");
```



Group By ... into

• qry1

```
SELECT [t1].[ShipCountry] AS [Country], [t1].[value2] AS [Count]
FROM (
    SELECT COUNT(*) AS [value], COUNT(*) AS [value2], [t0].[ShipCountry]
    FROM [Orders] AS [t0]
    GROUP BY [t0].[ShipCountry]
    ) AS [t1]
ORDER BY [t1].[value]
```

• Group By ... into

Country	Count	groupByCountry
Norway	6	Key=Norway

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight
10387	SANTG	1	18.12.1996 0:00:00	15.1.1997 0:00:00	20.12.1996 0:00:00	2	93,630
10520	SANTG	7	29.4.1997 0:00:00	27.5.1997 0:00:00	1.5.1997 0:00:00	1	13,370
10639	SANTG	7	20.8.1997 0:00:00	17.9.1997 0:00:00	27.8.1997 0:00:00	3	38,640
10831	SANTG	3	14.1.1998 0:00:00	11.2.1998 0:00:00	23.1.1998 0:00:00	2	72,190
10909	SANTG	1	26.2.1998 0:00:00	26.3.1998 0:00:00	10.3.1998 0:00:00	2	53,050
11015	SANTG	2	10.4.1998 0:00:00	24.4.1998 0:00:00	20.4.1998 0:00:00	2	4,6200

Poland	7	Key=Poland

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freig
10374	WOLZA	1	5.12.1996	2.1.1997 0:00:00	9.12.1996	3	3.94

Další příklad

V dalším příkladě si ukážeme několik nových věci. Nejdříve to bude možnost pracovat s tabulkami, na které se daná tabulka odkazuje přes relace, aniž bychom museli psát nějaké joiny. V našem příkladě to bude odkaz z tabulky zákazníků Customers na tabulku objednávek Orders. Budeme vybírat pouze takové zákazníky, kteří mají aspoň 15 objednávek a kde průměrná hodnota položky Freight bude větší než 10 (ukážeme si tedy jak pracovat s agregáty COUNT() a AVG jazyka SQL). Výsledek budeme chtít ještě setřídit.

```
var dataContext = this;

"Aggregates".Dump();
var qry1 =
    from c in dataContext.Customers
    where c.Orders.Count()>15
    where c.Orders.Average(n=>n.Freight)>10
    orderby c.Orders.Average(n=>n.Freight) descending
    select new {c.CustomerID, Cnt=c.Orders.Count(),
Avg=c.Orders.Average(n=>n.Freight)};
this.GetCommand(qry1).CommandText.Dump("qry1");
qry1.Dump("Aggregates");
```

Results λ SQL

Results

Aggregates

▪ qry1

```
SELECT [t0].[CustomerID], (
    SELECT COUNT(*)
    FROM [Orders] AS [t4]
    WHERE [t4].[CustomerID] = [t0].[CustomerID]
    ) AS [Cnt], (
    SELECT AVG([t5].[Freight])
    FROM [Orders] AS [t5]
    WHERE [t5].[CustomerID] = [t0].[CustomerID]
    ) AS [Avg]
FROM [Customers] AS [t0]
WHERE (((
    SELECT AVG([t1].[Freight])
    FROM [Orders] AS [t1]
    WHERE [t1].[CustomerID] = [t0].[CustomerID]
    )) > @p0) AND (((
    SELECT COUNT(*)
    FROM [Orders] AS [t2]
    WHERE [t2].[CustomerID] = [t0].[CustomerID]
    )) > @p1)
ORDER BY (
    SELECT AVG([t3].[Freight])
    FROM [Orders] AS [t3]
    WHERE [t3].[CustomerID] = [t0].[CustomerID]
    ) DESC
```

Results λ SQL

▪ Aggregates

▲ IOrderedQueryable<> (9 items)

CustomerID	Cnt	Avg
SAVEA	31	215,6032
ERNSH	30	206,8463
QUICK	28	200,2010
HUNGO	19	145,0126
RATTC	18	118,5672
FOLKO	19	88,3200
BERGS	18	86,6400
BONAP	17	79,8747
HILAA	18	69,9533

Aktualizace dat

V příkladech dotazů jazyka LINQ bychom mohli ještě dlouho pokračovat, ale my se teď zaměříme na další velmi důležitou oblast, na aktualizaci dat pomocí LINQ. Postupně si ukážeme uložení věty na SQL Server, její aktualizaci a nakonec její zrušení. Zároveň si ukážeme jednu věc, která vás patrně ze začátku vyděsí, ale uvidíte, že vše má logické vysvětlení.

Základem jsou následující poznatky:

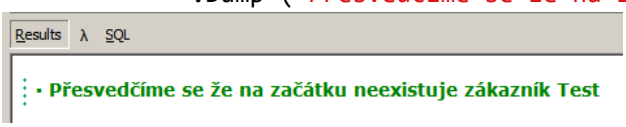
- Větu označíme jako větu k vložení pomocí metody dataContext.InsertOnSubmit(). Vlastní poslání věty na server se provede až v okamžiku volání metody dataContext.SubmitChanges() (je to ekvivalent foxovské TableUpdate()).
- Větu změníme pomocí přiřazovacího příkazu. Změna se na SQL Server opět pošle až po volání metody dataContext.SubmitChanges().
  POZOR! Zde vás musím upozornit na jednu zdánlivou anomálii (viz příklad dále). Uvedli jsme, že se věta změní na SQL Serveru až po SubmitChanges(). Pokud ale znova provedeme dotaz pomocí primárního klíče (v našem příkladě je to CustomerID), LINQ ihned zjistí, že hledáme větu s takovým primárním klíčem, který odpovídá primárnímu klíči věty, kterou jsme změnili,

a namísto toho, aby poslal dotaz na SQL Server, vrátí větu tak, jak jsme ji my změnili, a nikoli tvar, který je na SQL Serveru. Pokud nás zajímá, co je skutečně na SQL Serveru, musíme pro dotaz použít jiný dataContext než ten, ve kterém je provedena změna. Vzhledem k tomu, že v LINQPadu nemohu vytvořit jiný kontext, vybírám data pomocí dotazu na CompanyName (to není primární klíč).

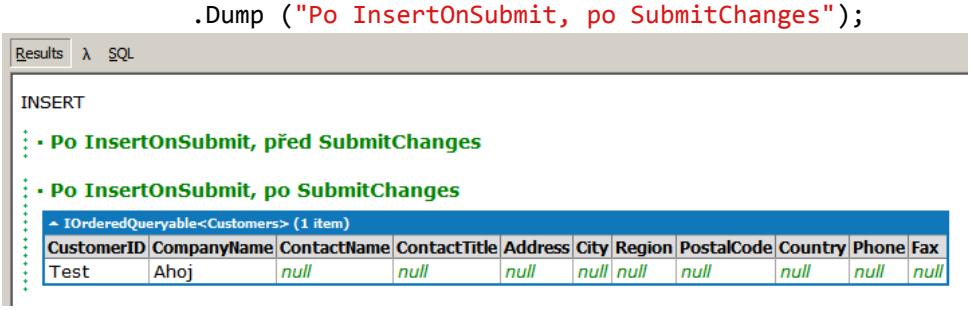
- Větu označíme jako větu ke zrušení pomocí metody dataContext.DeleteOnSubmit(). Vlastní zrušení věty na serveru se provede až v okamžiku volání metody dataContext.SubmitChanges().

Příklad je dostatečně komentován:

```
var dataContext = this;

// Přesvědčíme se, že na začátku neexistuje zákazník Test.
dataContext.Customers.Where(c=>c.CustomerID=="Test")
             .Dump ("Přesvědčíme se že na začátku neexistuje zákazník Test");
```

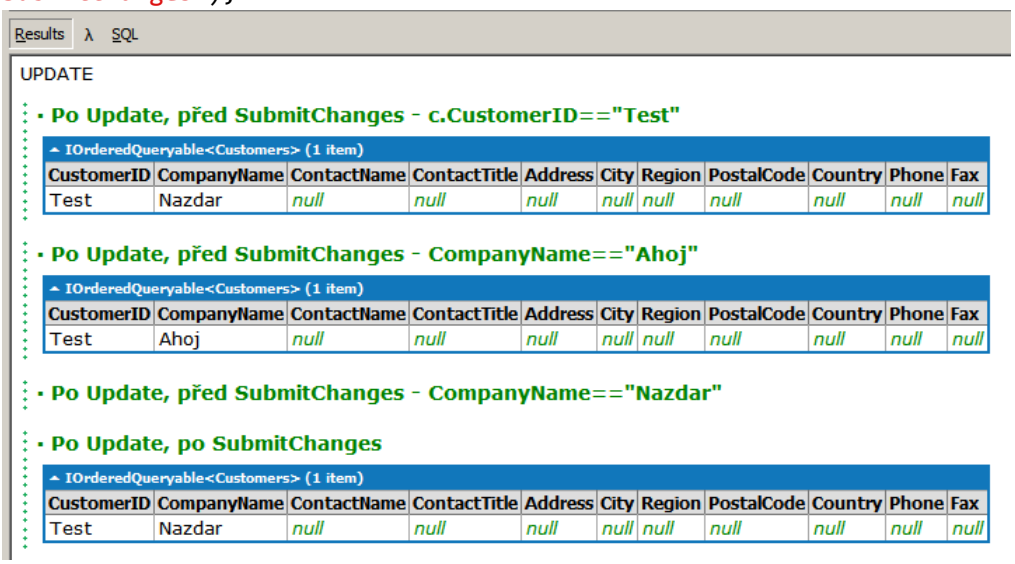



```
"INSERT".Dump();
// Vytvoříme normální objekt.
Customers cust = new Customers { CustomerID="Test", CompanyName="Ahoj" };
// Stejné jako INSERT INTO ...
dataContext.Customers.InsertOnSubmit (cust);
// Ještě jsme změny nezapsali na server.
dataContext.Customers.Where(c=>c.CustomerID=="Test")
             .Dump ("Po InsertOnSubmit, před SubmitChanges");
// Stejné jako TableUpdate()
dataContext.SubmitChanges();

// Přesvědčíme se, že již existuje zákazník Test.
dataContext.Customers.Where(c=>c.CustomerID=="Test")
             .Dump ("Po InsertOnSubmit, po SubmitChanges");
```




```
"UPDATE".Dump();
cust.CompanyName = "Nazdar";
// POZOR! Viz text výše.
dataContext.Customers.Where(c=>c.CustomerID=="Test")
             .Dump ("Po Update, před SubmitChanges - c.CustomerID==\"Test\"");
dataContext.Customers.Where(c=>c.CompanyName=="Ahoj")
```

```
            .Dump ("Po Update, před SubmitChanges - CompanyName==\"Ahoj\"");
dataContext.Customers.Where(c=>c.CompanyName=="Nazdar")
            .Dump ("Po Update, před SubmitChanges - CompanyName==\"Nazdar\"");
dataContext.SubmitChanges();

// Přesvědčíme se, že zákazník Test změnil jméno.
dataContext.Customers.Where(c=>c.CustomerID=="Test").Dump ("Po Update, po
SubmitChanges");
```



Results λ SQL

UPDATE

· Po Update, před SubmitChanges - c.CustomerID=="Test"

IOrderedQueryable<Customers> (1 item)

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
Test	Nazdar	null	null	null	null	null	null	null	null	null

· Po Update, před SubmitChanges - CompanyName=="Ahoj"

IOrderedQueryable<Customers> (1 item)

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
Test	Ahoj	null	null	null	null	null	null	null	null	null

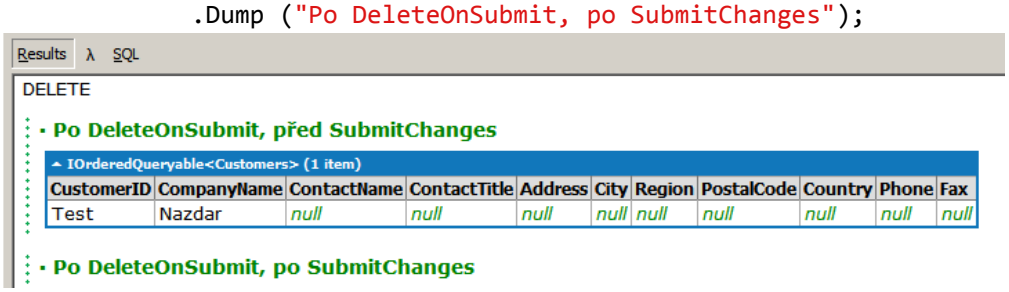
· Po Update, před SubmitChanges - CompanyName=="Nazdar"

· Po Update, po SubmitChanges

IOrderedQueryable<Customers> (1 item)

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
Test	Nazdar	null	null	null	null	null	null	null	null	null

```
"DELETE".Dump();
dataContext.Customers.DeleteOnSubmit (cust);
// Ještě jsme změny nezapsali na server.
dataContext.Customers.Where(c=>c.CustomerID=="Test")
            .Dump ("Po DeleteOnSubmit, před SubmitChanges");
dataContext.SubmitChanges();

// Přesvědčíme se, že již neexistuje zákazník Test.
dataContext.Customers.Where(c=>c.CustomerID=="Test")
            .Dump ("Po DeleteOnSubmit, po SubmitChanges");
```



Results λ SQL

DELETE

· Po DeleteOnSubmit, před SubmitChanges

IOrderedQueryable<Customers> (1 item)

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax
Test	Nazdar	null	null	null	null	null	null	null	null	null

· Po DeleteOnSubmit, po SubmitChanges

Transakce

Situace s transakcemi je podobná jako ve Visual FoxPro. Pokud nic neuděláme, celé volání metody SubmitChanges() je zabaleno do jedné transakce, takže buď se provedou všechny změny, nebo žádná. Pokud si ale chceme řídit transakce sami, můžeme si předem otevřít připojení pro datový kontext a začít na tomto připojení transakci předem. Potom už můžeme s touto transakcí normálně pracovat. Uveďme si pouze kratičký příklad:

```
var dataContext = this;

try
{
// Ručně otevřeme připojení na SQL Server.
    dataContext.Connection.Open();
// Na tomto připojení nastartujeme transakci.
    dataContext.Transaction = dataContext.Connection.BeginTransaction();
    dataContext.SubmitChanges();
// Pokud je vše ok - potvrdíme transakci.
    dataContext.Transaction.Commit();
}
catch (ChangeConflictException)
{
// Pokud došlo k chybě, provedeme Rollback.
    dataContext.Transaction.Rollback();
}
```

Ošetření chyb při aktualizaci

LINQ používá (stejně jako Visual FoxPro při práci se SQL Serverem) optimistický přístup k řešení konfliktů. Předpokládá tedy, že to dopadne dobře, a pokud ne, tak se s tím musí vypořádat. Připomínám, že i v tomto příkladu použijeme ruční řízení transakcí, což zde není nutné, ale z pedagogických důvodů si to uvedeme.

```
var dataContext = this;

// Tady normálně pracujeme…
try
{
// Ručně otevřeme připojení na SQL Server.
    dataContext.Connection.Open();
// Na tomto připojení nastartujeme transakci.
    dataContext.Transaction = dataContext.Connection.BeginTransaction();
    dataContext.SubmitChanges(ConflictMode.ContinueOnConflict);
// Pokud je vše ok, potvrdíme transakci.
    dataContext.Transaction.Commit();
}
catch (ChangeConflictException)
{
// Pokud došlo k chybě, zkoušíme vyřešit konflikt.
// Takto se dá provést výpis důvodů, kde se co nepodařilo - opět pomocí LINQu.
    var exceptionDetail =
    from conflict in dataContext.ChangeConflicts
    from member in conflict.MemberConflicts
    select new  {
        TableName = dataContext.GetTable(conflict.Object.GetType())
      , MemberName = member.Member.Name
      , CurrentValue = member.CurrentValue.ToString()
      , DatabaseValue = member.DatabaseValue.ToString()
```

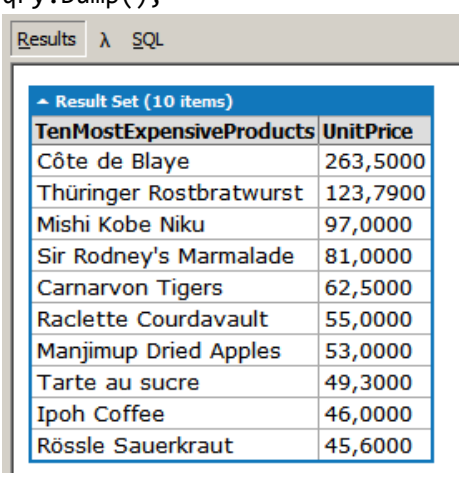
```
      , OriginalValue = member.OriginalValue.ToString()
      };
    exceptionDetail.Dump();
    try
    {
        dataContext.SubmitChanges(ConflictMode.ContinueOnConflict);
// Pokud je aspoň nyní vše ok, potvrdíme transakci.
        dataContext.Transaction.Commit();
    }
    catch (ChangeConflictException)
    {
// Pokud znovu došlo k chybě, provedeme Rollback.
        dataContext.Transaction.Rollback();
    }
  }
```

Více se ale nebudeme o řešení konfliktů zajímat.

Uložené procedury

Práce s uloženými procedurami už snad ani nemůže být jednodušší. Voláme metody, které se jmenují stejně jako uložené procedury (nebo uživatelem definované funkce), předáme parametry a dostaneme zpátky výsledek (nebudeme zde probírat, jakého typu). Takže příklad nám ukáže, jak je to jednoduché:

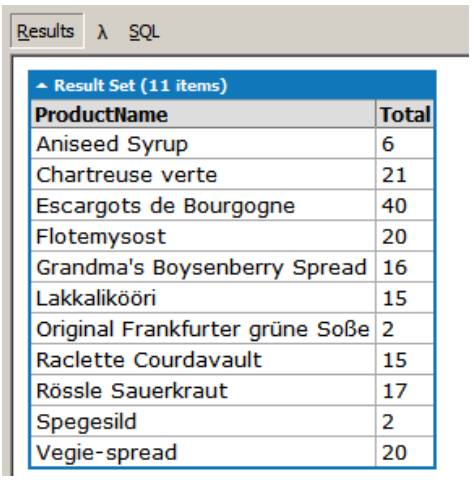
```
var dataContext = this;

// Uložená procedura bez parametrů
var qry = dataContext.TenMostExpensiveProducts();
qry.Dump();
```



Result Set (10 items)

TenMostExpensiveProducts	UnitPrice
Côte de Blaye	263,5000
Thüringer Rostbratwurst	123,7900
Mishi Kobe Niku	97,0000
Sir Rodney's Marmalade	81,0000
Carnarvon Tigers	62,5000
Raclette Courdavault	55,0000
Manjimup Dried Apples	53,0000
Tarte au sucre	49,3000
Ipoh Coffee	46,0000
Rössle Sauerkraut	45,6000

```
// Uložená procedura s 1 parametrem
var qry2 = dataContext.CustOrderHist("ALFKI");
qry2.Dump();
```


Results λ SQL

Result Set (11 items)

ProductName	Total
Aniseed Syrup	6
Chartreuse verte	21
Escargots de Bourgogne	40
Flotemysost	20
Grandma's Boysenberry Spread	16
Lakkalikööri	15
Original Frankfurter grüne Soße	2
Raclette Courdavault	15
Rössle Sauerkraut	17
Spegesild	2
Vegie-spread	20

Vytvoření malé aplikace

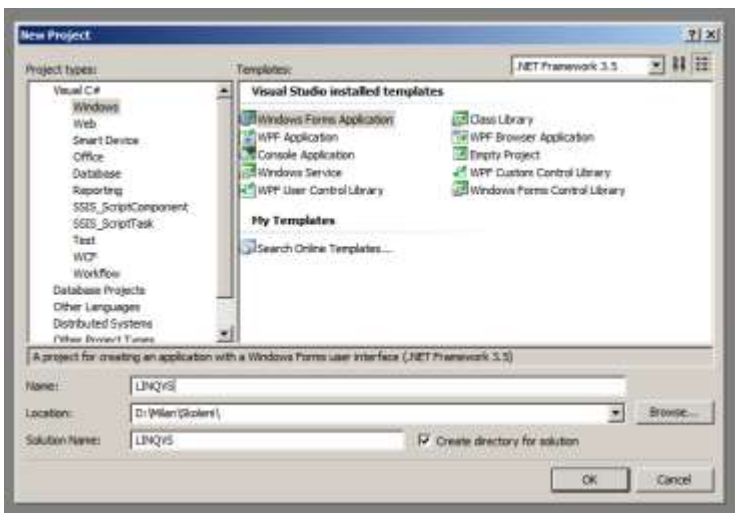
Konečně nastal čas na to, abychom vytvořili několik skutečně miniaturních aplikací. Bude pro nás důležité to, jak snadno lze vytvořit aplikaci ve Visual Studiu, nechceme si zde (na rozdíl od předchozích kapitol) ukazovat složitější konstrukce (aktualizaci dat apod.).

Vytvoření stejné aplikace ve Visual FoxPro a v C# s využitím LINQ

Naším cílem je vytvořit co nejjednodušší spustitelný program. Uvidíme, že jednotlivé kroky v C# odpovídají krokům ve VFP. Takže vzhůru do práce! Spusťte vedle sebe Visual FoxPro a Visual Studio a můžete tak vytvářet podobný program najednou ve VFP i ve VS.

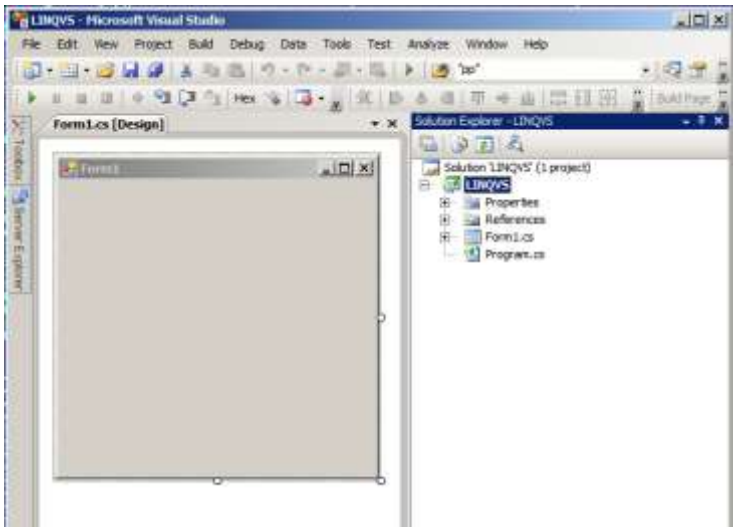
1. Vytvoření projektu

VFP: Menu File/New, vybrat Project a zadat jeho jméno LINQVFP (nebo `CREATE PROJECT LINQVFP`)
VS: Menu File/New/Project… a zadat jméno projektu (LINQVS) a adresář, ve kterém bude tento projekt uložen (u mne D:\Milan\Skoleni)



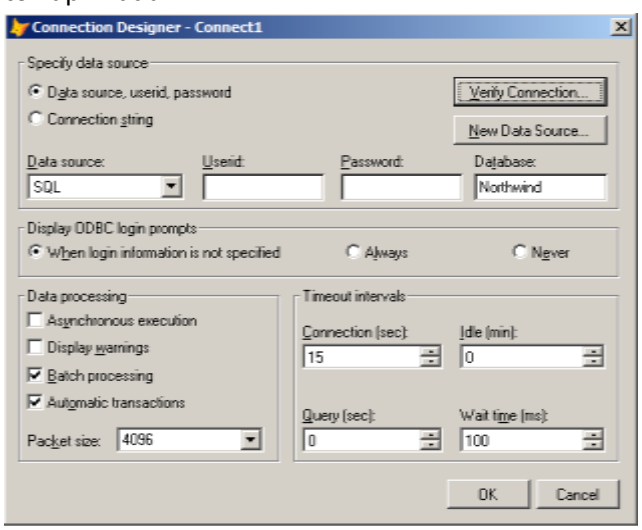
Vznikne tím solution (můžete totiž uložit několik projektů dohromady do jednoho solution – u nás to bude pouze jeden projekt), který se zobrazí v Solution Exploreru (ekvivalent VFP Project) vpravo. Vlevo už vidíme návrhář formuláře (nemusíme tedy dělat to, co děláme ve VFP: vybrat

Documents/Forms a kliknout na New) :



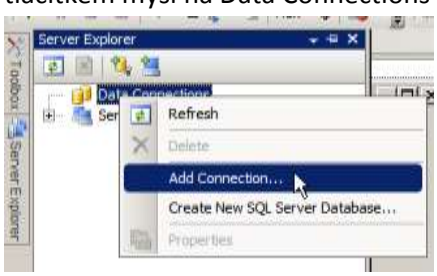
2. Vytvoření vazby na data v databázi

VFP: V projektu vybereme záložku Data/Database/New… a zadáme, že chceme vytvořit databázi Northwind.dbc. V Database Designeru klikneme pravým tlačítkem myši a vybereme Connections…/New. Dialogové okno vyplníme, jak je potřeba pro správné připojení – na mém PC je to kupříkladu:



Otestujeme, zda je možné se připojit k datům a uložíme připojení pod názvem Northwind.

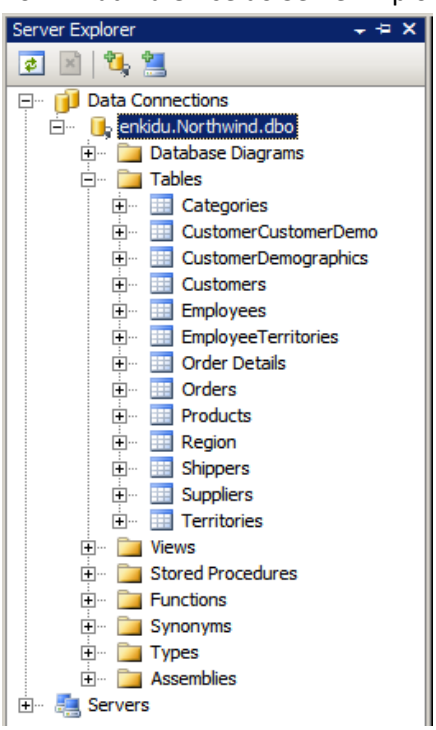
VS: Otevřeme okno Server Exploreru (pomocí menu View/Server Explorer), klikneme pravým tlačítkem myši na Data Connections a vybereme Add Connection…



V následujícím okně vybereme Microsoft SQL Server a Continue

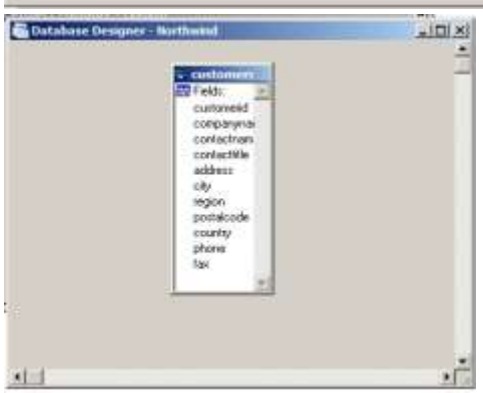
a vyplníme jméno serveru a jméno databáze:

Po kliknutí na OK se do Server Exploreru přidá toto připojení:

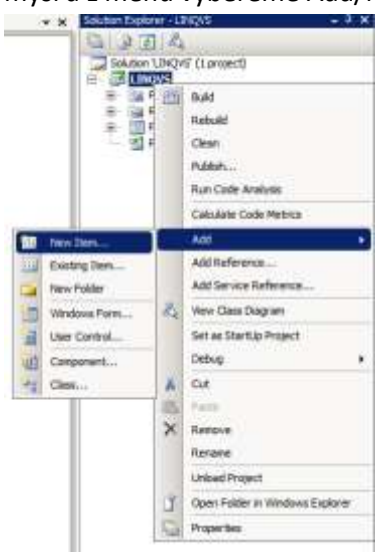


3. Vytvoření pohledů na data

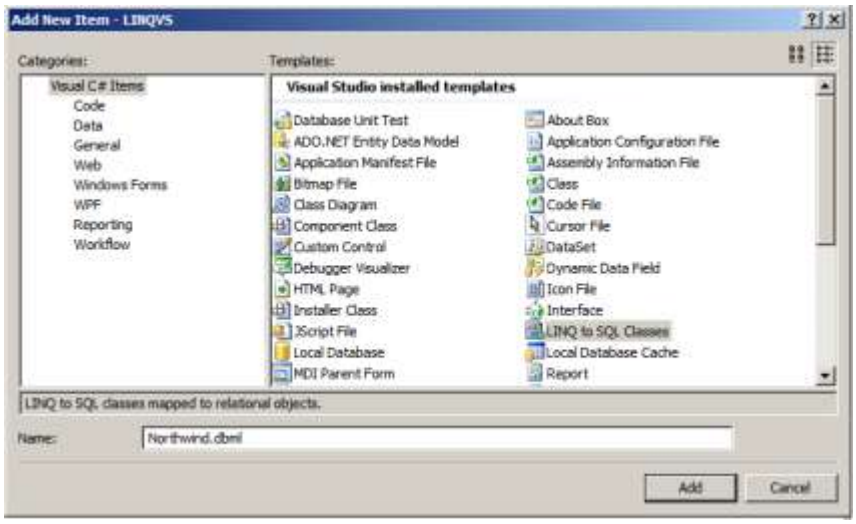
VFP: Vzdálený pohled vytvoříme snadno: Klikneme pravým tlačítkem myši na plochu Database Designer, vybereme volbu New Remote View…, potvrdíme připojení pomocí connection Northwind, vybereme všechny sloupečky z tabulky Customers a uložíme pod jménem Customers.



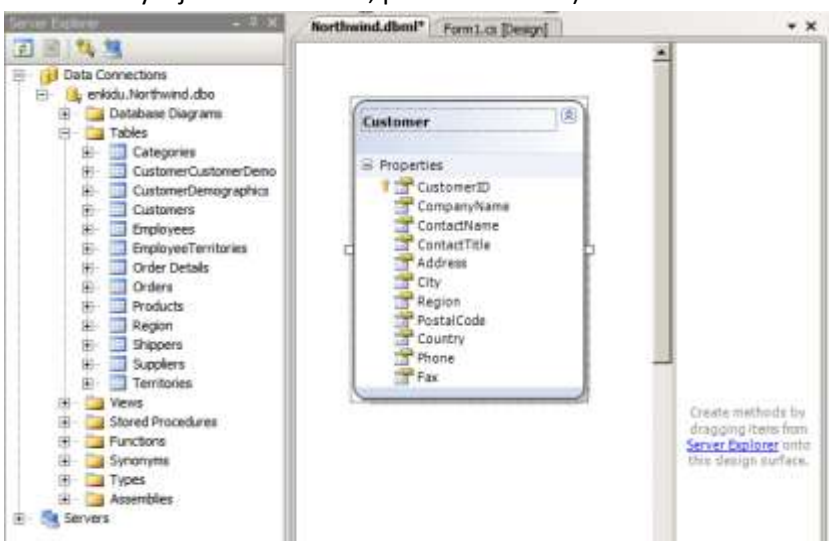
VS: Musíme vygenerovat třídy pro LINQ. V kapitole o LINQu jsme si uvedli pomocný program SQLMetal. Nyní si vytvoříme třídy interaktivně. V Solution Exploreru tedy klikneme pravým tlačítkem myši a z menu vybereme Add/New Item…



V následujícím okně vybereme nejprve v nabídce nahoře LINQ To SQL Classes (protože budeme vytvářet třídy pro LINQ To SQL) a potom zadáme jméno vytvářeného souboru, například Northwind:



Na prázdnou plochu návrháře přetáhněte ze Server Exploreru tabulku Customers (jméno třídy je automaticky v jednotném čísle, proto Customer):

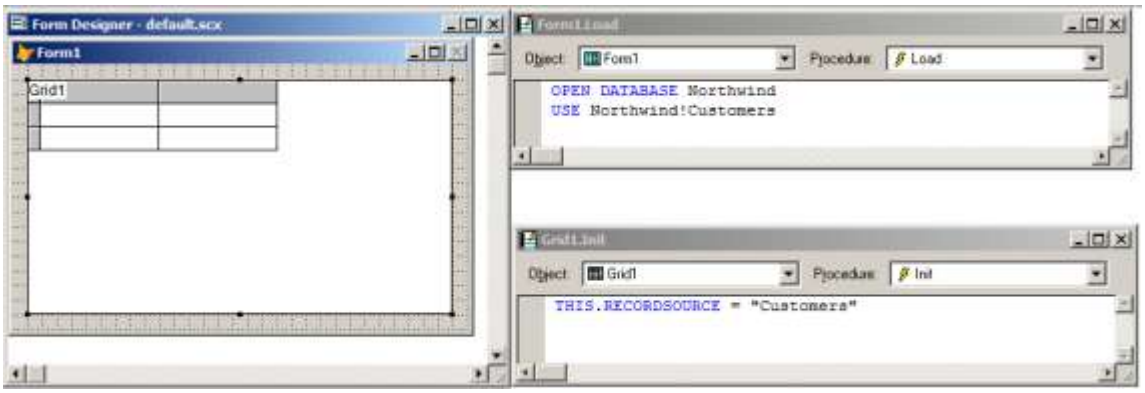


4. Na závěr musíme vytvořit jednoduchý formulář, který bude zobrazovat data. Samozřejmě bychom mohli jak ve VFP, tak i ve VS použít drag&drop, ale většinou budeme přeci jen pracovat s kódem, takže budeme programovat i my :

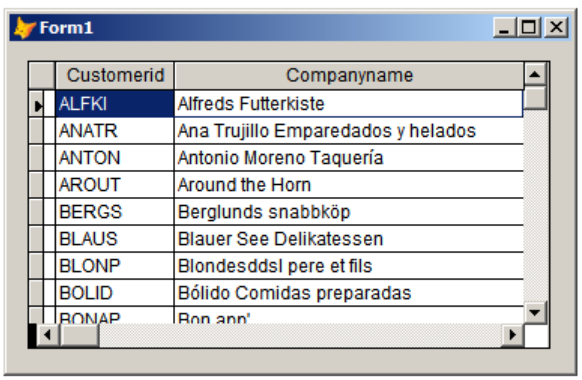
VFP: Otevřeme návrhář formuláře

```
MODIFY FORM Default.SCX
```

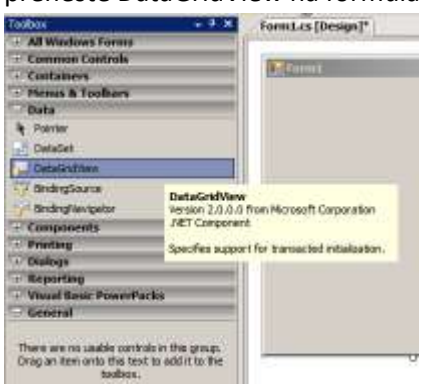
a na formulář umístíme grid. V metodě Load() otevřeme vzdálený pohled a v metodě Init() gridu nastavíme zdroj dat gridu na vzdálený pohled Customers:



A formulář můžeme spustit.

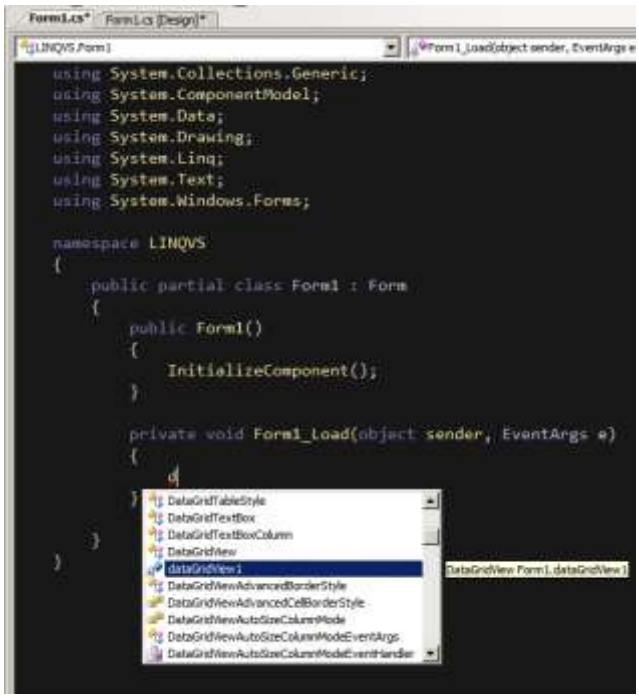


VS: Nejprve najděte v okně Toolbox (menu View/Toolbox) část Data a pomocí drag&drop přeneste DataGridView na formulář



a dostanete

Okénko průvodce DataGridView Tasks potřebovat nebudeme a zavřeme ho. Nyní poklepejte na volnou část formuláře (ne na DataGridView!). Otevře se vám editor kódu v metodě Load(). Na následujících screenshotech se nedivte černému pozadí – mně osobně toto zobrazení vyhovuje podstatně více než standardní.



Začal jsem psát referenci na náš DataGridView1. Všimněte si, jak nám IntelliSense pomáhá. Napíšeme následující kód (můžete použít jak lambda zápis, tak i normální):

```
NorthwindDataContext db = new NorthwindDataContext();
private void Form1_Load(object sender, EventArgs e)
{
    // dataGridView1.DataSource = db.Customers.Select(cust=>cust);
    dataGridView1.DataSource = from cust in db.Customers
                               select cust;
}
```

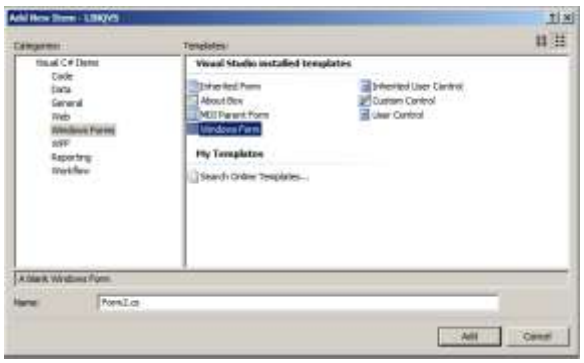
Musíme tedy nejprve vytvořit kontext db. Do DataGridView.DataSource (ekvivalent RecordSource gridu VFP) uložíme výsledek LINQ dotazu – v našem případě jednoduše vybereme všechny zákazníky. Nyní již můžeme program spustit, a to nejčastěji klávesou F5 nebo z menu Debug/Start with (without) Debugging nebo kliknutím na ikonu v toolbaru:

CustomerID	CompanyName	ContactName
ALFKI	Alfreds Futterkiste	Maria Anders
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo
ANTON	Antonio Moreno Taquería	Antonio Moreno
AROUT	Around the Horn	Thomas Hardy
BERGS	Berglunds snabbköp	Christina Berglund
BLAUS	Blauer See Delikatessen	Hanna Moos
BLONP	Blondesddsl pere et fils	Frédérique Citeaux

Pokud si odmyslíte ikonu v titulku, formuláře od sebe nepoznáte. Doufám, že vás příklad přesvědčil o tom, že vývoj v C# je stejně (ne)snadný jako ve VFP ☺.

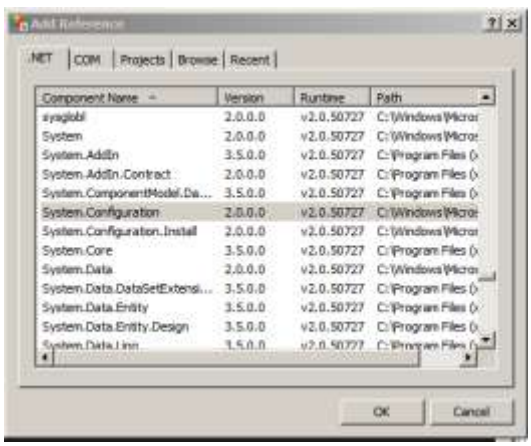
Stejná aplikace v C# s využitím ADO.NET

Do projektu přidáme ještě jeden formulář pomocí menu Project/Add New Item… Vybereme (pokud není vybrána) položku Windows Form a vytvoříme formulář Form2:



Naším cílem bude tentokrát pro práci použít ADO.NET. Jak víte z předchozích kapitol, při práci s ADO.NET musíme znát connection string, který definuje parametry připojení. Během vytváření předchozího formuláře se connection string již vytvořil a uložil na to správné místo, tedy do XML souboru app.config. U Exe se potom bude jmenovat *jménoexe*.config – tedy například Test.exe.config. To umožňuje jednoduché nastavení parametrů připojení u zákazníka, kde stačí změnit tento textový soubor.

Abychom mohli v programu pracovat s údaji z konfiguračního souboru, musíme k projektu přidat referenci na knihovnu implementující potřebné metody. K tomu je třeba v projektu kliknout pravým tlačítkem myši na References, vybrat volbu Add Reference… a v okně vybrat System.Configuration:



Navíc musíme na začátek programu ještě přidat odkaz na tuto knihovnu (a rovnou ještě na další pro práci se SQL Serverem). Začátek programu bude potom vypadat takto:

```
using System.Configuration;
using System.Data.SqlClient;

namespace LINQVS
```

Potom již můžeme bez problému uložit connection string do proměnné connString při spouštění formuláře:

```
        string connString;
        public Form2()
        {
```

```
            InitializeComponent();
            connString = ConfigurationManager // nevejde se na 1 řádek
  .ConnectionStrings["LINQVS.Properties.Settings.NorthwindConnectionString"]
                        .ConnectionString;
        }
```

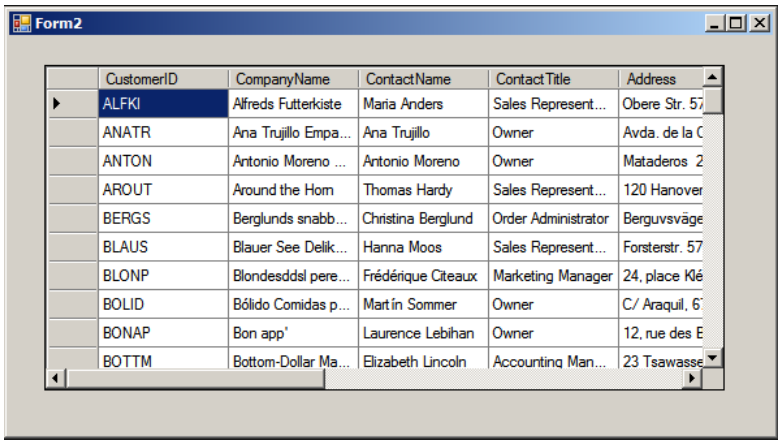
Stejně tak jako v předchozím příkladě umístíme DataGridView na formulář. Pro naplnění použijeme stejný postup jako v kapitole o ADO.NET (nejprve naplníme DataTable) a potom nastavíme DataSource (ekvivalent foxovského RecordSource) na tuto tabulku:

```
        private void Form2_Load(object sender, EventArgs e)
        {
            DataTable dt = new DataTable();
            using (SqlConnection conn = new SqlConnection(connString))
            {
                conn.Open();
                SqlCommand cmd = conn.CreateCommand();
                cmd.CommandText = "SELECT Customers.* FROM dbo.Customers";
                using (SqlDataReader rdr = cmd.ExecuteReader())
                {
                    dt.Load(rdr);
                }
                conn.Close();
            }
            dataGridView1.DataSource = dt;
        }
```

Chtěli bychom nyní spustit tento formulář pomocí klávesové zkratky F5, ale VS bude neustále spouštět Form1. Proč? Stejně jako VFP má i VS projekt nastaven hlavní formulář. Otevřeme tedy hlavní program (Program.cs) a změníme jméno spouštěného formuláře z Form1 na Form2:

```
using …

namespace LINQVS
{
    static class Program
    {
        /// <summary>

        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form2());
        }
    }
}
```

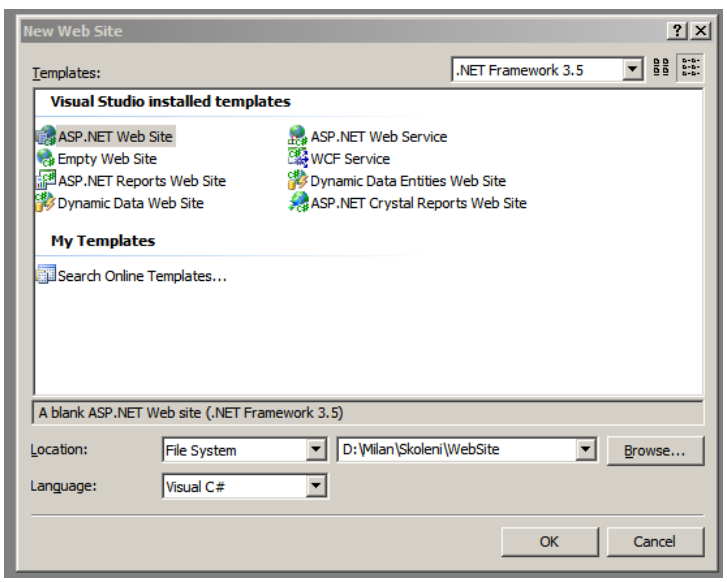
A můžeme spustit formulář Form2 pomocí klávesy F5:



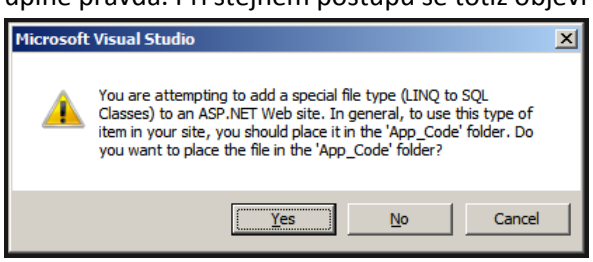
Vytvoření webové aplikace (Web Forms) v C# s využitím LINQ

Posledním příkladem bude vytvoření jednoduché webové aplikace v C# pomocí LINQ. Vzhledem k tomu, že nepředpokládám, že by se vám chtělo instalovat a konfigurovat webový server IIS, otestujeme korektnost aplikace pomocí interního webového serveru Visual Studia.

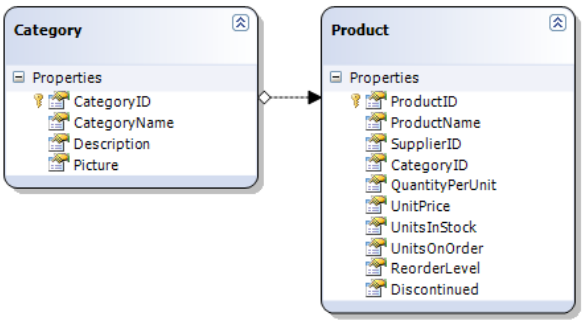
Hned na začátku musíme vytvořit website pomocí menu File/New/Web Site… Jediné důležité zde je zadání adresáře:



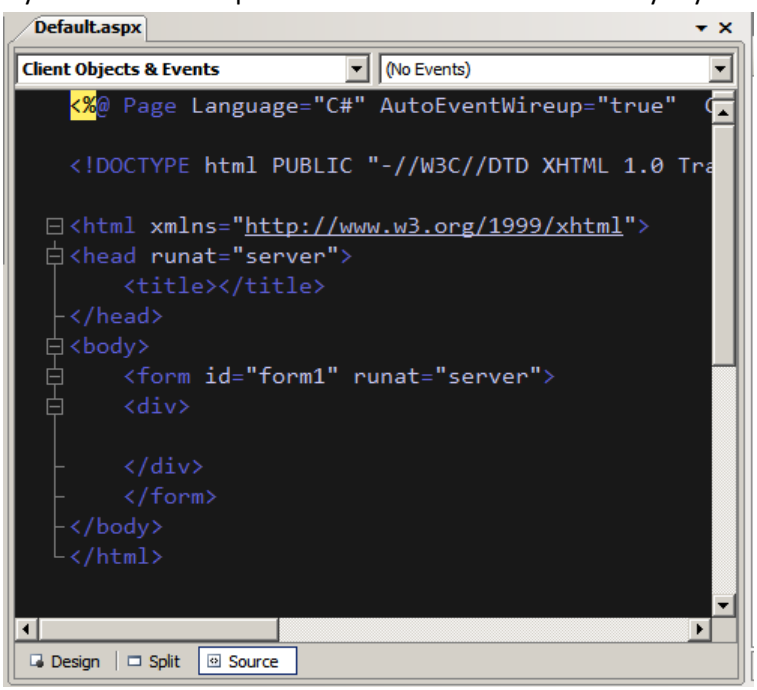
Nejprve stejně jako v LINQ příkladu vytvoříme třídy pro tabulky z databáze Northwind. Vzhledem k tomu, že je postup uveden výše, zdálo by se, že je postup úplně stejný. Ale asi tušíte, že to není tak úplně pravda. Při stejném postupu se totiž objeví následující dotaz:



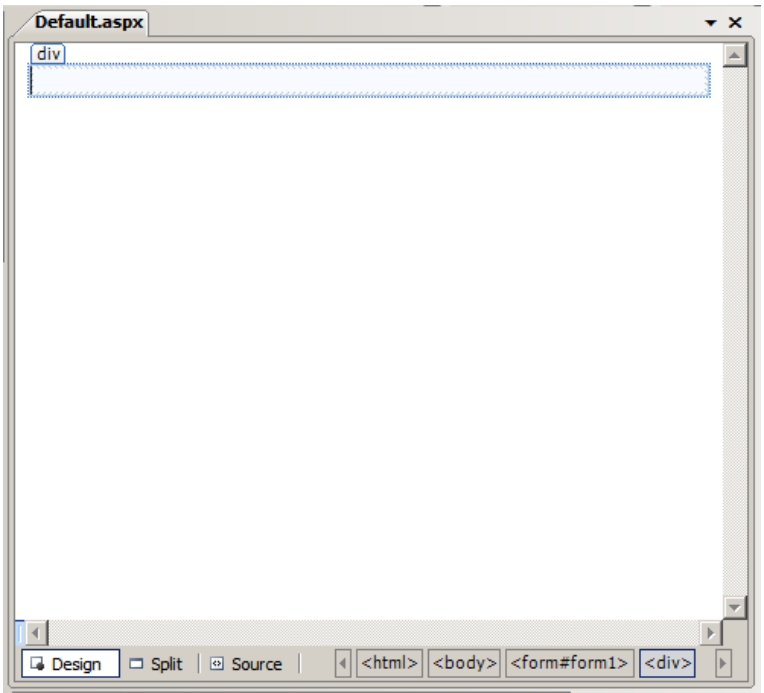
Podobné třídy je v ASP.NET potřeba vytvářet ve speciálním adresáři App_Code (z důvodů, které nejsou pro naše povídání důležité). Odpovězte tedy Yes. Pomocí drag&drop přeneste tabulky Categories a Products (a všimněte si, že se jména tabulek automaticky převedla z množného čísla do jednotného)



Nyní obrátíme naši pozornost k návrhu webové stránky. Výchozí zobrazení může být HTML kód:

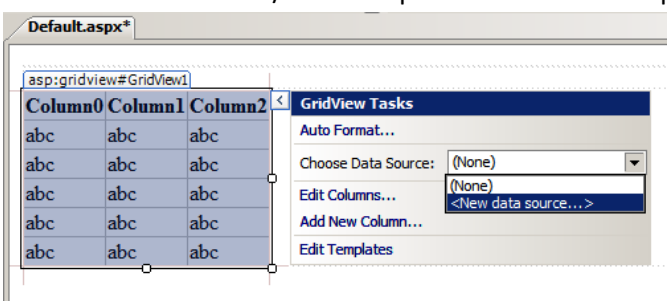


nebo vizuální navrhář:

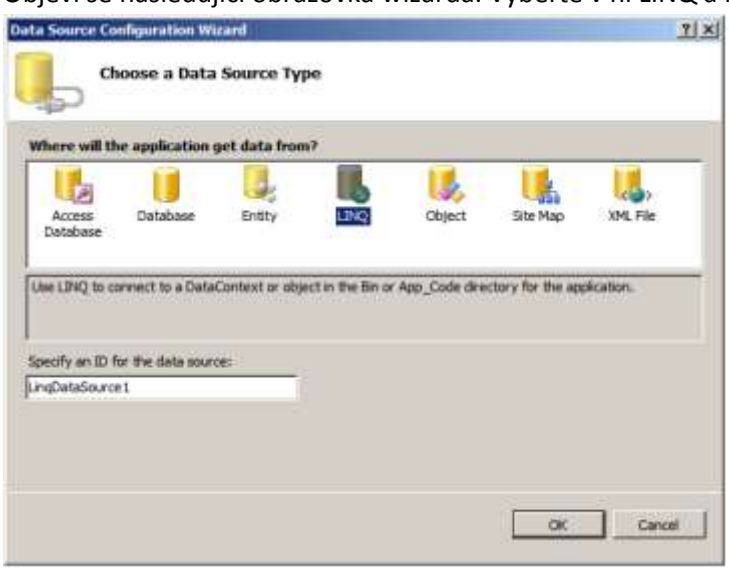


Přepínat se mezi nimi můžeme pomocí tlačítek Design/Source ve spodní části obrazovky. My budeme nyní používat Design (v praxi je to naopak).

Přetáhněte z Toolboxu/Data komponentu GridView na plochu návrháře:

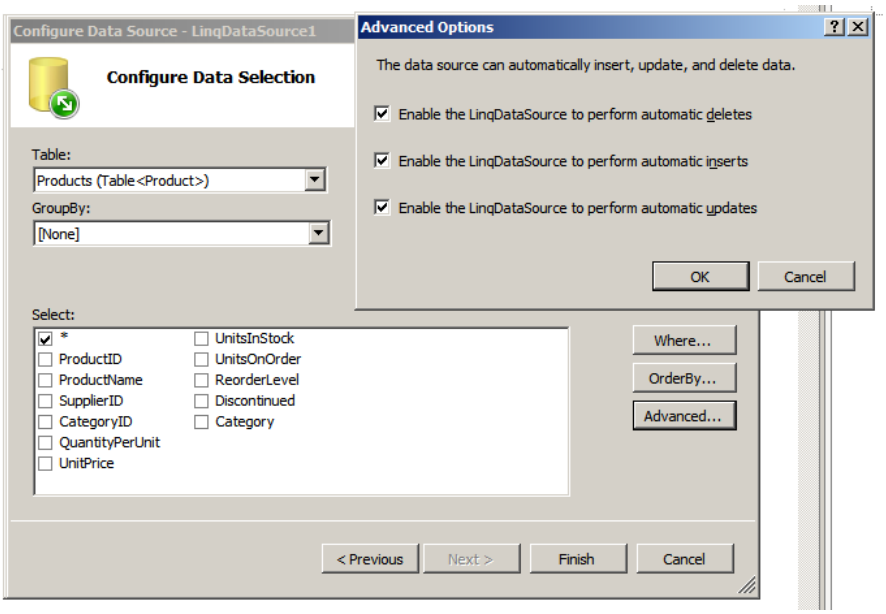


a vyberte volbu Choose Data Source: <New data source… >, protože nyní chceme zadat zdroj dat. Objeví se následující obrazovka wizarda. Vyberte v ní LINQ a klikněte na OK.

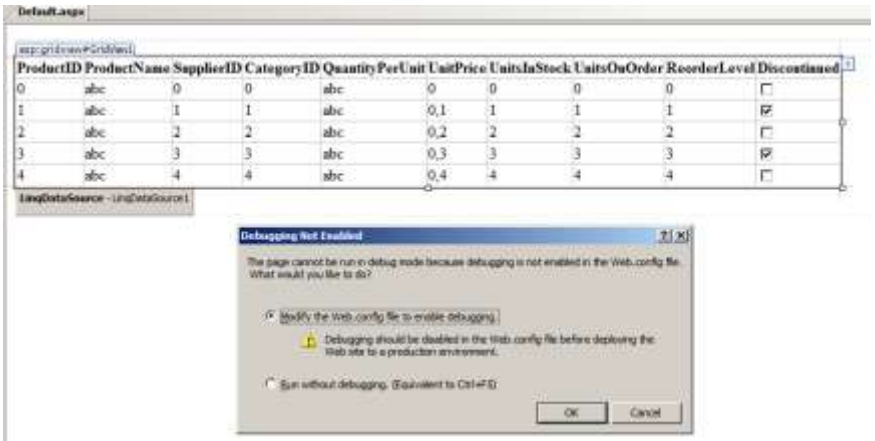


Tím se automaticky vytvořil LinqDataSource. V dalším okně potvrďte, že chcete pracovat s NorthwindDataContext.

V následujícím okně pak vyberte tabulku Products. Where i OrderBy (význam je jasný) necháme být a klikneme na Advanced… Zde povolíme podporu pro všechny aktualizace.



Po kliknutí na Finish jsme hotovi a můžeme aplikaci spustit pomocí F5 nebo menu Debug/Start Debugging. Kupodivu se nám nepodaří spustit naši stránku a zobrazí se tento dotaz:

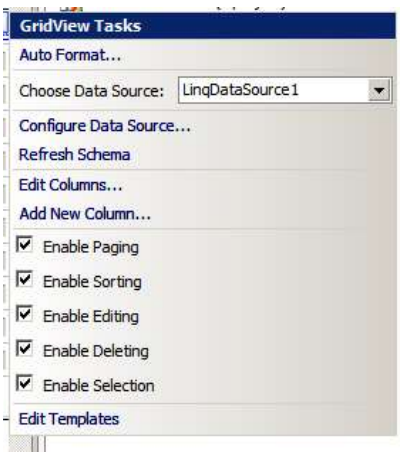


Nemáme totiž v konfiguračním souboru povoleno ladění. Klikneme na OK a konečně se zobrazí tato stránka:

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
1	Chai	1	1	10 boxes x 20 bags	18,0000	39	0	10	☐
2	Chang	1	1	24 - 12 oz bottles	19,0000	17	40	25	☐
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10,0000	13	70	25	☐
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22,0000	53	0	0	☐
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21,3500	0	0	0	☑
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25,0000	120	0	25	☐
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30,0000	15	0	10	☐
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40,0000	6	0	0	☐
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97,0000	29	0	0	☑
10	Ikura	4	8	12 - 200 ml jars	31,0000	31	0	0	☐
11	Queso Cabrales	5	4	1 kg pkg.	21,0000	22	30	30	☐
12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	38,0000	86	0	0	☐
13	Konbu	6	8	2 kg box	6,0000	24	0	5	☐
14	Tofu	6	7	40 - 100 g pkgs.	23,2500	35	0	0	☐
15	Genen Shouyu	6	2	24 - 250 ml bottles	15,5000	39	0	5	☐
16	Pavlova	7	3	32 - 500 g boxes	17,4500	29	0	10	☐
17	Alice Mutton	7	6	20 - 1 kg tins	39,0000	0	0	0	☑
18	Carnarvon Tigers	7	8	16 kg pkg.	62,5000	42	0	0	☐

Vidíme ale, že zde nemůžeme měnit data. Povolili jsme totiž změnu dat pouze v LinqDataSource, ale nikoli v GridView! Klikneme tedy na GridView (abychom ho vybrali) a pak klikneme na šipku vpravo nahoře (

). Tím se zobrazí menu, ve kterém zaškrtneme všechno, co jde.



Navíc ještě pomocí volby Auto Format… změníme trochu vzhled:

Nyní již lze data měnit (klikněte na Edit), rušit, třídit (kliknutím na nadpis sloupce), stránkovat…

Samozřejmě bychom mohli webovou stránku dál vylepšovat, ale asi to pro vytvoření představy stačí.

Závěr

Věřte, že bychom si mohli ještě dlouho povídat o všech možnostech ADO.NETu a LINQu. A to jsem se ani slovem nezmínil o Entity Frameworku, nHibernate atd. (a měl bych poznamenat, že jsem se nezmínil, protože se vůbec neodvažuji na toto téma něco napsat ☺). Ale nabídka je bezesporu dostatečná.

Mým cílem při psaní této brožury bylo poskytnout foxařům představu o tom, co je čeká v .NETu – a hlavně je přesvědčit, že možnosti práce s daty v .NET jsou dostatečné. Snad se mi to aspoň trochu podařilo.

Použitá literatura

Kromě různých zdrojů na internetu jsem čerpal převážně z následujících knih, které považuji za velmi dobré a dovoluji si je vám doporučit:

- Programming Microsoft® ADO.NET 2.0 Core Reference, David Sceppa, Microsoft Press; 2nd edition (August 30, 2006)
- LINQ in Action, Fabrice Marguerie, Steve Eichert, Jim Wooley, Manning Publications (February 4, 2008)
- C# 3.0 in a Nutshell: A Desktop Quick Reference, Joseph Albahari, Ben Albahari, O'Reilly Media, Inc.; 3rd edition (September 26, 2007)
- ASP.NET 2.0 Website Programming: Problem - Design - Solution, Marco Bellinaso, Wrox; 1th edition (May 8, 2006)