|  |  |
| --- | --- |
| [Výsledek obrázku pro database](https://www.google.cz/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwif0MSwwc_XAhXIbxQKHaXpDgAQjRwIBw&url=https://www.cbronline.com/what-is/what-is-a-database-4917209/&psig=AOvVaw3XeEjiVHL3t4gvZgpcNA6e&ust=1511348300838903)  TVORBA DATABÁZOVé aplikace sklad  Ing. Petr Hořejší, Ph.D. | ANOTACE  Tento cvičný materiál slouží k vytvoření databázové aplikace, která byla tvořena na cvičeních předmětu KPV/DBC. V rámci textu bude popsána tvorba databázové aplikace realizované v C# za pomocí technologie Entity Framework. Aplikace bude ostylována pomocí stylu Metro.  **Poděkování:**  Děkuji doc. Ing. Pavlu Kopečkovi, CSc. za základní ideu programu, doc. Ing. Zdeňkovi Ulrychovi, Ph.D. za původní zpracování ve Visual Basic a Ing. Martinu Strapkovi za cennou pomoc při přípravě těchto podkladů. |

# Informace ke kurzu

Tento cvičný materiál slouží k vytvoření databázové aplikace, která byla tvořena na cvičeních předmětu DBC. Věnujte v textu pozornost zvýrazněným částem a tučnému písmu, jelikož se v odstavcích mohou nacházet důležité informace.

Jelikož se verze aplikace, na kterou byl tento návod sepsán, a finální (rozšířená) verze v některých částech liší, není zaručená 100% shoda obou verzí aplikace. Z důvodu rozídlů mezi oběma verzemi aplikací doporučujeme stáhnout finální verzi aplikace, kterou najdete na Courseware v sekci cvičení na odkazu FTP. Soubor s názvem SkladFINAL.zip. Tuto aplikaci můžete využít pro kopírování kódů a stavbě vlastní aplikace.

**Abyste mohli tuto aplikaci spustit, je potřeba změnit ConnectionString v souboru app.config – viz dále. (Pokud, chcete spustit aplikaci již nyní, je možné na FTP nalézt pod adresářem Podklady/App.config\_proLocalDB soubor app.config. Tímto souborem je možno nahradit soubor z archivu Sklad/Sklad/App.config. )**

**Dále je lepší smazat adresář packages.**

**(V úložišti je možno nalézt i soubor SkladFINALLocal.zip, kde vše zmiňované bylo zařízeno.)**

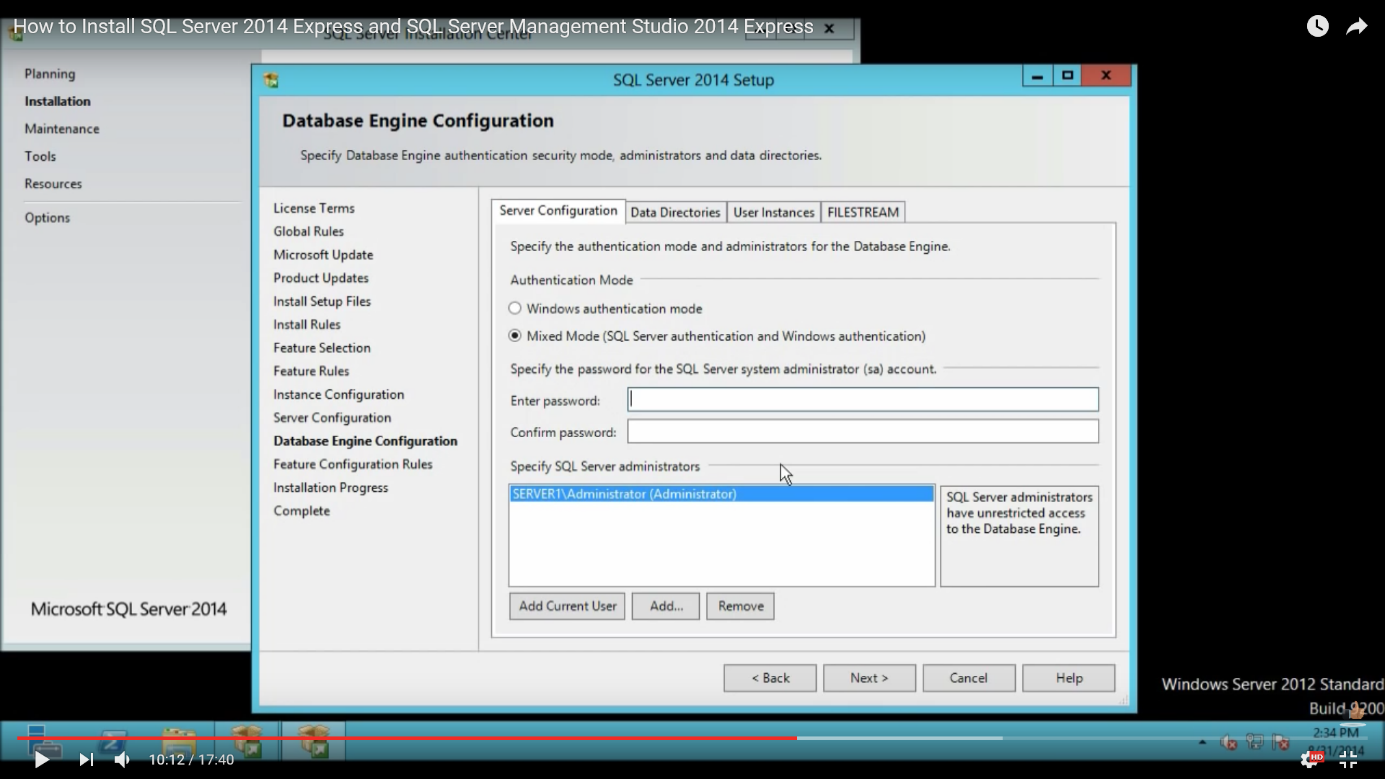
Pokud najdete zásadní nedostatky v tomto materiálu, případně máte návrhy na zlepšení, rádi přivítáme zpětnou vazbu pro zlepšení materiálů.

V rámci kurzu budeme pracovat s integrovaným SQL Serverem. V rámci cvičení se pak můžeme podívat i jako pracovat s „ostrým“ SQL Serverem – tato část je v textu popsaná jako ALTERNATIVA – písmo je zelené. Pokud nebude pracovat s SQL Serverem, témto částem nemusíte věnovat pozornost.

# Instalace

Z prostředí Imagine si stáhněte a nainstalujte produkt Visual Studio 2017 Enterprise. Během instalace zvolte, že nebudete potřebovat prostředí Xamarin (vývoj pro mobily – zabírá hodně místa na disku). Zkontrolujte, zda se bude instalovat rozšíření SQL Server Data Tools. Po dokončení instalace zkontrolujte, že máte instalovány nejnovější aktualizace – v aplikaci Visual Studio - Tools, Extensions and Updates (kategorie Updates) – zde je možno dodatečně doinstalovat SQL Server Data Tools (což je integrovaný databázový SQL Server).

ALTERNATIVA: Alternativním krokem je instalace MS SQL Server Express. Tento balík je zdarma i pro komerční účely do 50-ti uživatelů. Stáhněte si buď starší verzi 2014, lépe však novější 2016. Během instalace proběhne celá řada otázek, vše lze přeskočit. Pouze je třeba vložit administrátorské heslo. U dotazu Server Configuration, Autentification Mode zvolte Mixed Mode a zadejte heslo dbc1234.



Obr. 1 Konfigurace instalace SQL Serveru

# Popis uživatelského prostředí Visual Studio

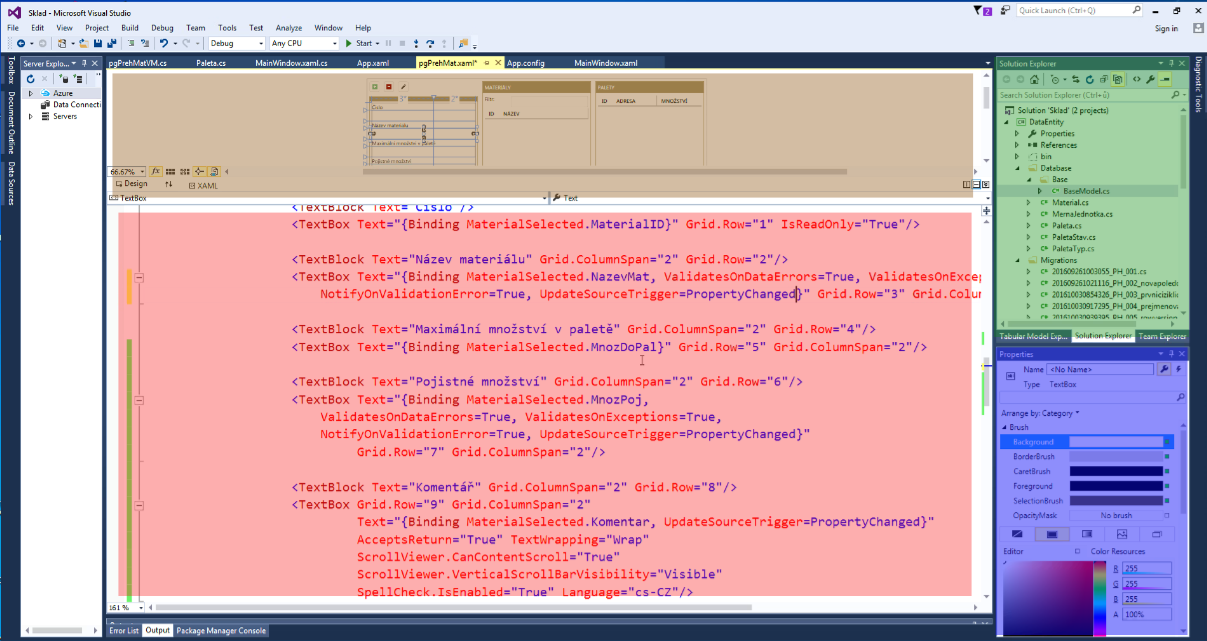
V této kapitole si v krátkosti představíme uživatelské prostředí a ukážeme, kde se nacházejí jednotlivé sekce, které budou v materiálech používány. Na obrázku – viz Obr. 2 - vidíme prostředí VisualStudio, ve kterém budeme pracovat. Prostředí je rozděleno na 4 sekce:

Editorové okno (červená) – v editorovém okně probíhá veškerý zápis kódů

Okno vlastností a událostí (modrá) – v tomto okně se nachází seznam vlastností, které je možné nastavit jednotlivým prvkům, a také seznam událostí (ikonka blesku), který budeme často používat

SolutionExplorer (zelená) – zde je zobrazeno struktura projektu. Do tohoto okna se vkládají nové adresáře a soubory.

Grafický editor (hnědá) – v rámci designu prezentační vrstvy se zde zobrazuje průběžná grafika. Můžeme zde vybírat jednotivlé objekty, které budeme postupně vkládat a pracovat s nimi.



Obr. Uživatelské prostředí VisualStudio

# Trocha teorie na začátek

Cílem tohoto kurzu je představit moderní způsob tvorby aplikací pomocí moderních tehnologií. Veskrze se jedná jen o úvod, nicméně po absolvování tohoto kurzu byste měli být schopni vytvořit jednoduchou databázovou aplikaci. Pokud vás téma zaujme, můžete se třeba vrhnout i na aplikaci složitější, pak je však potřeba dohledávát další informace z jiných zdrojů (vzhledem k tomu, že budeme používat známé a používané technologie, je na Internetu celá řada informací).

**Využívané technologie:**

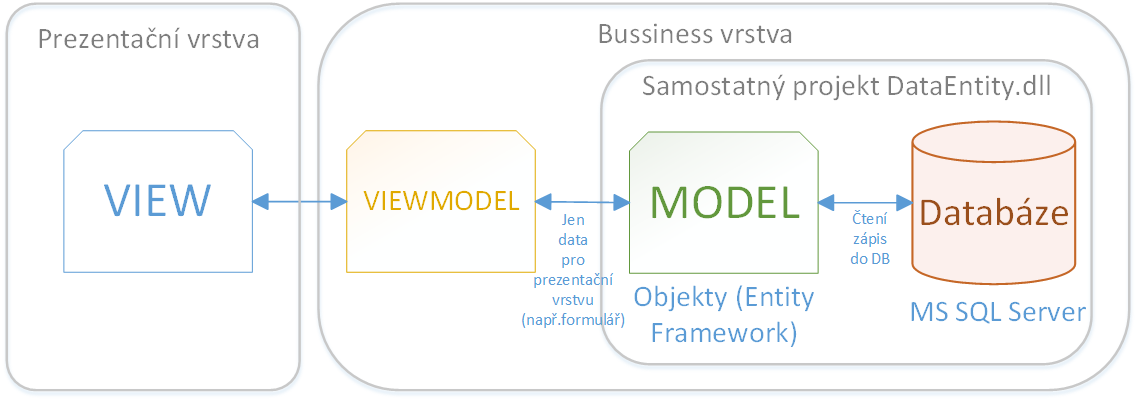
**Jazyk C#** - jedná se o moderní příjemný objektový jazyk. Pokud jste pracovali v jiném jazyku, s tímto nebudete mít problémy. Syntaxe je příjemná a přehledná, jazyk se navíc stále vyvíjí. Doporučujeme zopakovat a znát pojmy: **objekt** (nebo také třída), **vlastnost** (property) objektu, **metoda**, **událost**, **konstruktor**

**WPF (Windows Presentation Foundation)** – náhrada (alternativa) za starší Windows Forms. Technologie pro správu grafického prostředí aplikace. Formuláře se definují pomocí značkovacího jazyka XAML (dnes běžné téměř u všech aplikaci včetně mobilních)

**Entity Framework (EF) –** Microsoftem doporučovaný balík (tzv. NuGet balík) pro tvorbu databázových aplikací. Filosofie práce spočívá v převedení relační databáze na objekty. EF „vytahuje“ z databáze data dle potřeby (on demand - tzv. lazy loading). Představte si, že máme jednu tabulku Třidy a druhou Studenti. V rámci aplikace si uživatel chce prohlížet třdy – EF tedy naplní tabuku Třidy. Uživatel chce vidět studenty ve třídě (studijní skupině): v programu se objeví něco jako TatoTrida.Studenti – to je pokyn pro EF, že má spustit dotaz (představme si SELECT \* FROM Studenti WHERE TridaID=TatoTridaID) a naplnit takto seznam studentů v konkrétní třídě. O spouštění tohoto dotazu a naplnění objektu (třídy) se programátor vůbec nemusí starat!

Existují dva přístupy. První (ten nebudeme používat) je tzv. Database First – z existující databáze vytváříme objekty, druhý pak je ***Code First***, ten budeme využívat. Tento přístup spočívá ve vytvoření objektů, ze kterých se pak automaticky vytváří databáze. Přístup Code First má celou řadu výhod: je možné jednoduše udělat celou řadu poměrně komplexních kontrol (validací), přímá kontrola nad zdrojovými objekty, možnost tzv. migrací (verzování databáze).

A na závěr úvodu ještě je třeba zmínit využívanou architekturu **MVVM (Model-View-ViewModel):**



Obr. MVVM (včetně přístupu do databáze)

Na obrázku - viz Obr. 3 - je vidět architektura MVVM včetně napojení na databázi. Logické vrstvě, která obstarává funkčnost aplikace, říkáme **Bussiness vrstva.** Částem aplikace, které obstarávají komunikaci s uživatelem ve formě grafické prezentace, říkáme **Prezentační vrstva**. Bussiness vrstva bývá částo komplexnější nežli prezentační vrstva.

V rámci tvorby našeho solution (hlavní „nadprojekt“, který obsahuje projekty) začneme vytvářet projekt, který bude obstarávat přímou komunikaci s databází (my jej nazveme projekt DataEntity – viz dále). Oddělení tzv. objektového modelu je poměrně šikovné, už jen proto, že jej lze použít opakovaně pro jiné (třeba i mobilní nebo webové) aplikace.

Viewmodel a View souvisí spolu a vyvíjejí se najednou. View je libovolný grafický prvek (např. formulář, page (viz dále), usercontrol), který umožňuje uživateli pracovat s daty. „Pod tímto grafickým prvkem“ si připravíme datovou základnu jen pro tento prvek - ViewModel, která jednak může obsahovat vhodně připravená data modelu (z projektu DataEntity) a současně i další data pro správu tohoto prvku (takto si lze například ve vrstvě viewmodelu držet informace, na jaké je uživatel straně, který prvek ze seznamu zvolit, v jakém stavu je přepínací tlačítko atd. – vše ve formě vlastností, tzv. properties). Ve Viewmodelu se obecně také obsluhují akce při změně hodnot těchto vlastností (pomocí metod).

Každá změna této vlastnosti bude automaticky reflektována do View (např. změním vlastnost aktuálního čísla zvolené záložky na formuláři a jen na základě změny hodnoty dojde i ke grafické změně). To se neděje samo sebou – je více možností, jak to zařídit – nám pomůže NuGet package Fody (viz dále).

**Než začneme vlastní práci na „velkém“ projektu, podívejte se na kód z Dodatku B – Generické seznamy. Hotový projekt Seznamy.zip je na FTP.**

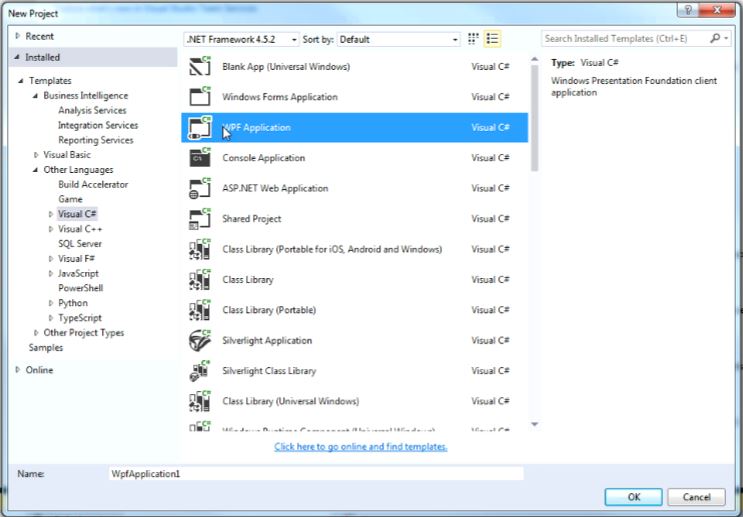
# Nový projekt

V této kapitole se naučíme zakládat novou databázi, ukládat ji a provádět prvotní nastavení. Toto téma ukázané na příkladu můžete prostudovat i zde:

https://msdn.microsoft.com/en-us/library/jj193542(v=vs.113).aspx

## Založení nového projektu

Nový projekt založíme otevřením záložky **File** v panelu nástrojů a zvolením položky **New Project**. Tím se dostaneme do okna – viz Obr. 4. V levém sloupci zvolíme jazyk **Visual C#** (pokud není v nabídce, najdeme ho pod záložkou **Others**). Z prostřední nabídky následně zvolíme typ **WPF Application** a zadáme název projektu („Sklad“). Je velmi důležité zadat název v tomto kroku! Poté potvrdíme.



Obr. Založení nového projektu

## Ukládání projektu

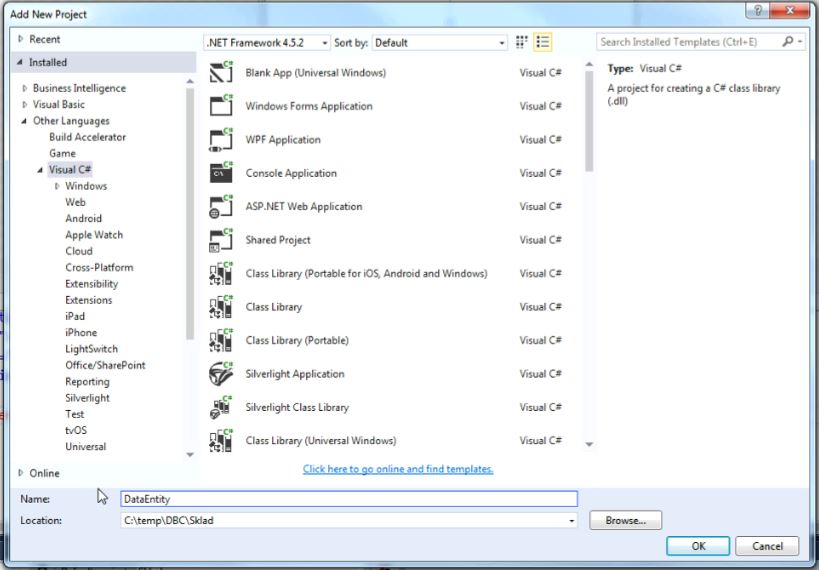
Projekt ukládáme položkou **Save all** v nabídce **File**. V nabídce - viz Obr. 5 - zvolíme cestu pro ukládání souboru. Pokud chceme s projektem dále pracovat, nezaškrtáváme možnost **Add to Source Control** (cloudové sdílení práce více vývojářů). Při práci na projektu ukládáme průběžně. Povšimněte si, že s novým projektem se založilo i nové Solution (panel Solution Explorer).



Obr. Ukládání projektu

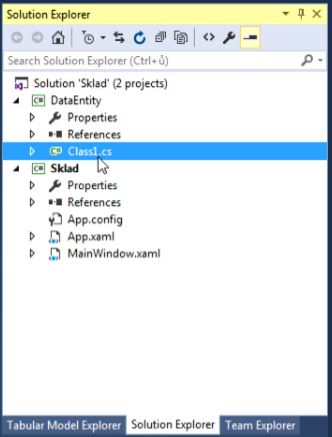
## Přidání knihovny tříd

V dalším kroku je nutné přidat knihovny tříd. Zvolíme záložku **File** z panelu nástrojů a dále **Add** -> **New Project**. V následujícím okně - viz Obr. 6 - v levé nabídce opět zvolíme jazyk **Visual C#** a vybereme typ **Class Library**. Následně zadáme název **DataEntity** a potvrdíme. Nyní budeme pracovat se dvěma projekty.



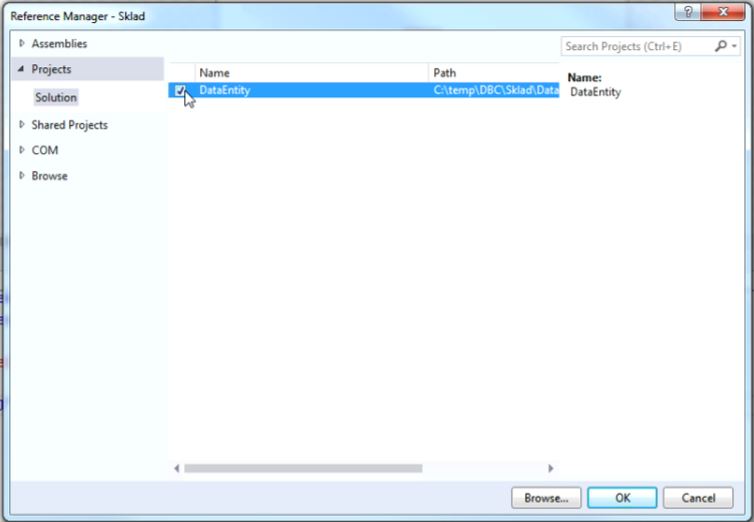
Obr. Přidání knihovny tříd

V okně **Solution Explorer** - viz Obr. 7 (pokud v uživatelském prostředí okno Solution Explorer není, otevřeme ho v záložce **View** v hlavním panelu) - následně odstraníme položku **Class1.cs** z DataEntity (brzy založíme vlastní třídy).



Obr. Solution explorer

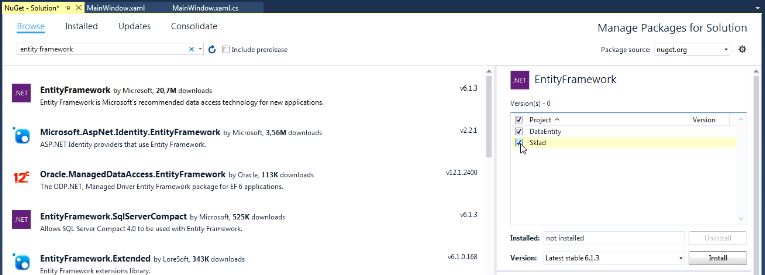
V témže okně **Solution Explorer** nastavíme reference (spojení projektů – tak, aby projekt Sklad „věděl“ o novém projektu). Pravým tlačítkem rozbalíme u **References** pod nadřazeným projektem **Sklad(!)** a zvolíme možnost **Add reference.** V nově otevřeném okně - viz Obr. 8 - rozbalíme nabídku Projects a zvolíme Solutions. Poté zaškrtneme políčko u DataEntity a potvrdíme.



Obr. Nastavení referencí

## Přidání Entity Framework do projektu

Dalším krokem je přidání nutných rozšíření (tzv. NuGetů) do našeho projektu. V panelu nástrojů zvolíme záložku **Tools** a poté **NuGet Package Manager** -> **Manage NuGet Package for Solution.** V nově otevřeném okně - viz Obr. 9 - pod záložkou **Browse** nebo zadáním názvu EntityFramework do vyhledávače zvolíme „rozšíření“ **EntityFramework** (verze v6.1.3). V pravé části pak zaškrtneme oba projekty - políčka Project**: DataEntity a Sklad(!)** a zvolíme Install.

****

Obr. Instalace rozšíření

## Přidání třídy Materiál

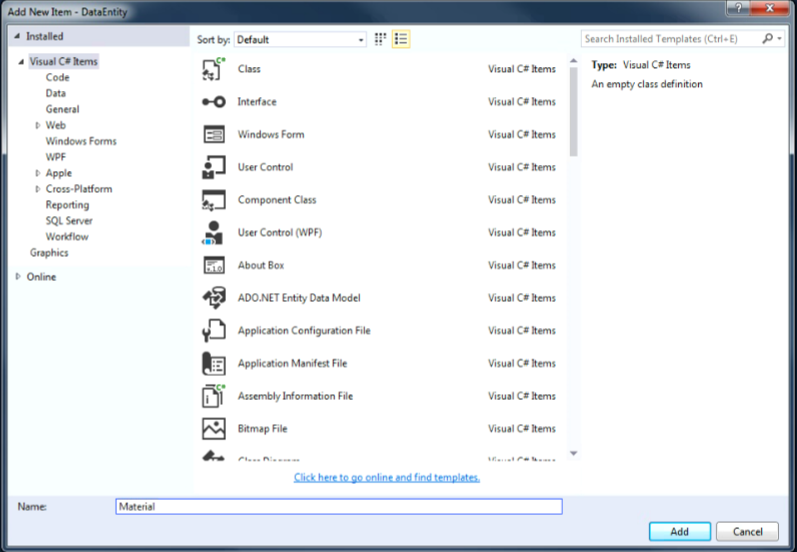
*Nyní začneme vytvářet databázovou strukturu. To provedeme tak, že vytvoříte tzv. objektový model (nebo jen MODEL), ze kterého budeme aktualizovat databázi na SQL serveru pomocí tzv. Migrací.*

*Zjednodušeně řečeno třída (čili objekt) na úrovni MODELu = tabulka v databázi. Vlastnost třídy na úrovni MODELu = atribut tabulky (sloupec).*

*Pozn.:Budeme-li hovořit o odkazu na jeden záznam, budeme používat jednotné číslo pro název vlastnosti – např. Material (bez české diakritiky). Pokud se bude jednat o seznam záznamů (o tzv. kolekci), budeme používat množné číslo – např. Materialy.*

Nyní začneme vytvářet Model, který bude přímo komunikovat s databází. Na základě vytvořených tříd, bude založen struktura databáze. Začneme třídou Material.

Novou třídu do databáze přidáme kliknutím na pravé tlačítko myši na složku **Database** pod knihovnou tříd **DataEntity** v **Solution Explorer**. Z nabídky pokračujeme **Add -> Class**. Zkontrolujeme, zda je zvolen jazyk C#, vybereme typ **Class** a pojmenujeme (Material).



Obr. Přidání nové třídy do databáze

*Každá třída je uložena v tzv. namespacu (pojmenované místo), to si lze představit, jako adresářovou strukturu pro zjednodušení hledání jednotlivých tříd. Standardní názvy namespaců jsou tvořeny ze jména projektu+stuktura adresářů v daném projektu. Když pak chcete v jiném projektu používat třídy z daného namespacu, musíte tento namespace v této třídě využít (pomocí klíčového slovíčka using). My sice zakládáme novou třídu v adresáři Database (standarní namespace se bude jmenovat DataEntity.Database), ale nechceme, aby pro využití této třídy bylo nutné využívat namespace DataEntity a ještě k tomu DataEntity.Database . Nejsnažší bude v rámci projektu DataEntity definovat vše do namespacu DataEntity.*

V nově otevřeném okně editoru smažeme část ***Database*** v řádku **namespace DataEntity. Database**. Tento krok provedeme pokaždé, kdy přidáme novou třídu.

## Nastavení třídy, vkládání nových properties tj. nových sloupců do budoucí tabulky

*Nyní definujeme vlatsnosti třídy Material, tyto vlastnosti jsou současně i budoucími položkami databáze. V rámci této třídy jsou definovány i některé další vlastnosti uvedené hranatými závorkami [..] – těmto dalším vlastnostem se říká data annotations (datové anotace) – ty je možno definovat buď u třídy, metody nebo vlastnosti. Před samotnou třídou je takto možno např. uvést, jak se bude ve skutečnosti jmenovat tabulka v databázi.*

*U samotných jednotlivých vlatsností je možno dodefinovat další datové anotace:*

*[Key] – primární klíč, je šikovné nazývat primární klíč ve formátu Nazev****Id*** *– logika EF. Je možné pomocí datové anotace zavést i kombinovaný primární klíč, to však není v rámci EF doporučované. Standardně je vhodné vytvořit tzv. autoinkrementační primární klíč – číslo, které se automaticky pro každý nový záznam zvětší o 1. (autoinkrementace není závazná, je možno primární klíč ukládat ručně, pak se však musíme postarat o kontroly na unikátnost)*

*[Column(Order=x)] – udává v jakém pořadí budou jednotlivé sloupce v databázi*

*[Required] – povinné pole*

*[StringLength (x)] – omezení textu na počet znaků*

*Dále je možné do těchto datových anotací přímo uvést text chyby, který se objeví v případě, že omezení není splněno.*

*Více o datových anotacích: https://msdn.microsoft.com/en-us/library/jj591583%28v=vs.113%29.aspx?f=255&MSPPError=-2147217396*

V editorovém okně provedeme následující zápis:

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

//zde jsou vynechány standardní using namespaces; - dále vynecháváme

namespace DataEntity

{

[Table("Materialy")] //pojmenování tabulky

public partial class Material

{

[Key] // přiřazení primárního klíče

[Column(Order = 1)] //zajištění, aby sloupec MaterialID byl v tabulce na prvním místě

public int MaterialId { get; set; } //založení sloupce MaterialID v tabulce, který je datového typu Integer

[Required (ErrorMessage = "Název materiálu je povinné pole")] //zajištění, že ve sloupci nesmí být prázdná hodnota, při zadání nulové hodnoty vyskočí chybová hláška

[StringLength(255)] //datový typ String může mít maximálně 255 znaků

public string NazevMat { get; set; } //get; set; píšeme při každém založení nové vlastnosti/sloupce - tzv. standardní getter a setter

}

}

Pokud se při zadávání nového namespace (např. [StringLength(255)]) červeně podtrhne (znamená to, že naše třída „nezná“ odkaz na třídu s definicemi datových anotací), klikneme na ikonu žárovky vlevo od příkazu (pomocník) a z nabídky zvolíme **using System.ComponentModel.DataAnnotations;** (pomocník nám inteligentně navrhl, kde by se tato chybějící třída mohla nacházet). Tuto úpravu budeme provádět i dále (pokud chybějící třída existuje, mělo by se jedna o první volbu v seznamu using ….).

**POZOR (ČASTÁ CHYBA): Při přidávání chybějících namespaců nabízí pomocník – žárovka ještě další možnosti, jak například založení nové chybějící třídy do daného projektu. Chyba pak zmizí, ale samozřejmě v tomto případě datová anotace nebude funkční – respektive bude využívat novou prázdnou třídu.**

## Vytvoření kontextové třídy

*Nyní založíme tzv. Kontextovou třídu – to je hlavní správcovská třída, která drží odkazy na seznamy (DbSet) všeho, co je v databázi (může také obsahovat různé definice chování – jaké akce mají nastat při ukládání dat, lze definovat speciální chování databáze pomocí tzv. Fluent API, např. kaskádové mazání a aktualizace, atp.). Seznam (DbSet) je de facto budoucí tabulka, tj. V kontextové třídě obsahuje seznamy dílčích tříd (Material, Paleta, atd.) v budoucí databáze je to pak tabulka a jednotlivé řádky v ní. Založení a aktualizace struktury databáze nemůže fungovat, pokud alespoň neuvedeme, že se v databázi nachází seznam materiálů.*

V **Solution Explorer** klikneme pravým tlačítkem na třídu **DataEntity** **-> Add -> New item**. Zvolíme typ **Class** a nazveme ji **SkladContext**. Následně provedeme zápis:

using System.Data.Entity;

namespace DataEntity

{

public partial class SkladContext : DbContext

{

public virtual DbSet<Material> Materialy { get; set; }

//SkladContext je jméno standardního connection stringu

//pokud byste chtěli změnit jméno connectionstringu, je možno odkomentovat

//následující konstruktor a přejmenovat SkladContext

//public SkladContext () : base ("name = SkladContext")

//{

//}

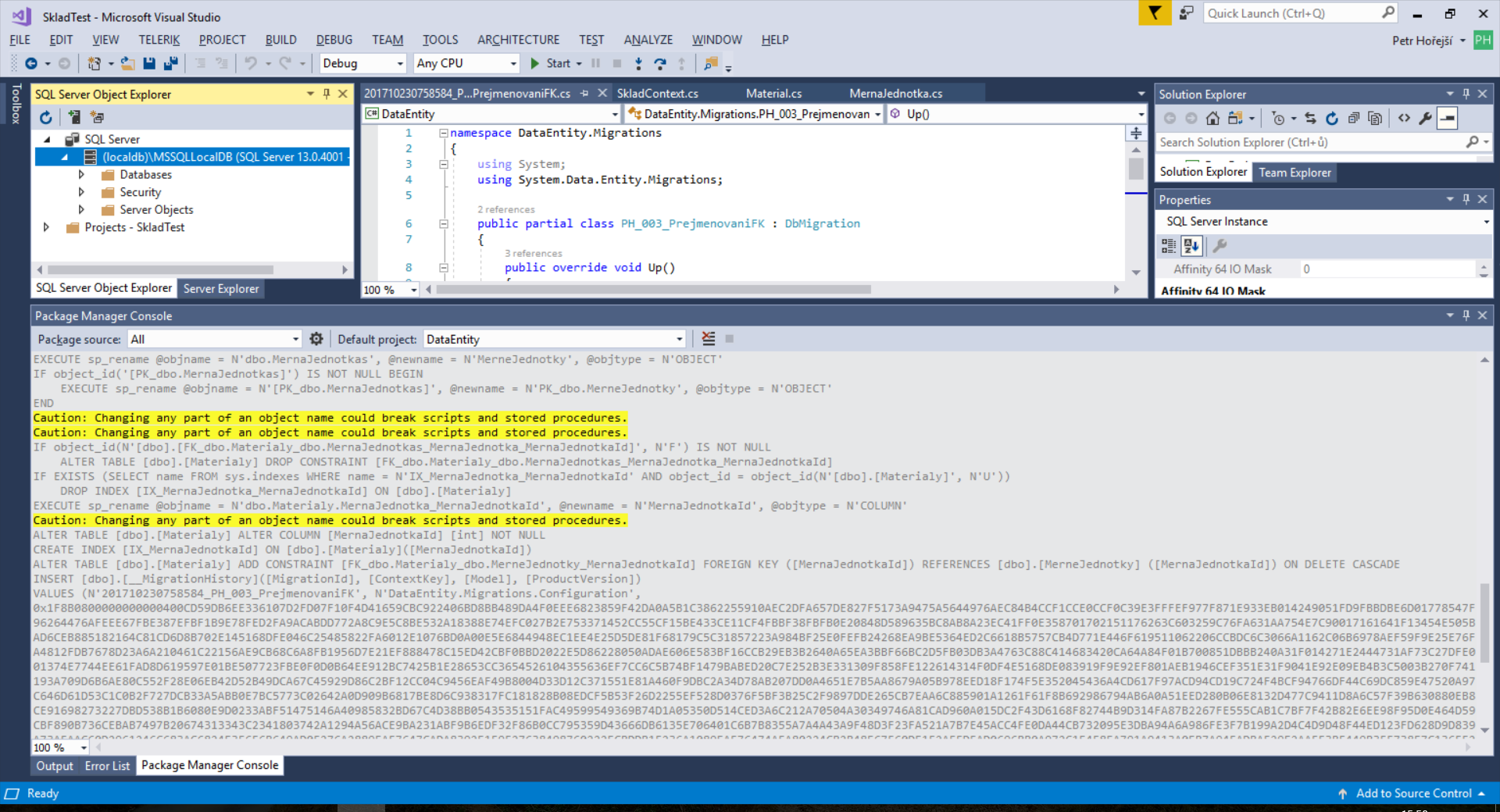
}

}

## Příprava a konfigurace SQL Serveru

*Nyní se pokusíme tuto jednu jedinou tabulku vytvořit na SQL serveru (na základě třidy Material a kontextu SkladContext).*

Nejprve zkontrolujeme, zda běží lokální SQL server: Zobrazíme plovoucí okno pro správu SQL připojení – **View, SQL Server Object Explorer.** Zde by měl být vidět spuštěný lokální SQL server (localdb)\MSSQLLocalDB (viz Obr. 11). Pokud není server „viditelný“, je potřeba jej spustit pomocí návodu v Příloze 3.



Obr. SQL Server Object Explorer

Nyní nastavíme připojení na SQL server, abychom mohli založit strukturu databáze na tento server. Musíme tedy specifikovat tzv. ConnectionString.

Zkopírujeme tento kód:

<connectionStrings>

<add name="SkladContext" connectionString="Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Sklad;Integrated Security=True" providerName="System.Data.SqlClient" />

</connectionStrings>

*V ConnectionStringu definujeme:*

*Name – takto je pojmenován connectionstring v aplikaci*

*Data Source – cesta k SQL serveru (může být buď jméno serveru, IP adresa, popř. web alias pro cloud SQL)*

*Initial Catalog – jméno databáze*

*User Id – jméno uživatele na SQL serveru (lze zakládat a měnit na SQL serveru). Uživatel sa znamená superadmin – standardní uživatel – který má maximální práva. V praxi se zakládají uživatelé s přístupy jen k dílčím databázím*

*Password – heslo pro uživatele*

Dále v **SolutionExplorer** v projektu **Sklad** otevřeme **AppConfig** a do tohoto souboru překopírujeme kód (za příkaz **</startup>** viz Obr. 12). Zkontrolujeme, zda je **InitialCatalog = Sklad**. (název databáze na serveru)



Obr. Zkopírování kódu ConnectionString

ALTERNATIVA: V případě využívání „ostrého“ SQL Serveru vše provedme takto:

<connectionStrings>

<add name="SkladContext" connectionString="Data Source=jmenoinstanceSQL;Initial Catalog=Sklad;MultipleActiveResultSets=true;Integrated Security=false;User Id=sa;Password=dbc1234" providerName="System.Data.SqlClient"/>

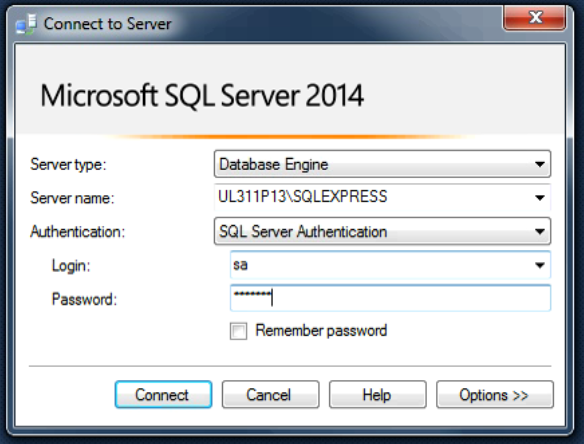
</connectionStrings>

Otevřeme program **SQL Server Management Studio**, kterým jsme nainstalovali spolu s MS SQL Server. Přihlašovací okno nastavíme - viz Obr. 13. - Výchozí přihlašovací údaje pro laboratoře jsou **Login: sa Heslo: dbc1234** (pokud jste instalovat SQL server doma, možná jste si nastavili jiné výchozí heslo)**.** Z tohoto okna zkopírujeme název serveru, zde **UL311P3\SQLEXPRESS** (název serveru bude u každého počítače jiný). Tento název překopírujeme do kódu **AppConfig**, kde jím nahradíme **Data Source = jmenoinstanceSQL**. Finální podoba konfiguračního kódu je pak např:

<connectionStrings>

<add name="SkladContext" connectionString="Data Source=UL311P03\SQLEXPRESS;Initial Catalog=Sklad;MultipleActiveResultSets=true;Integrated Security=false;User Id=sa;Password=dbc1234" providerName="System.Data.SqlClient"/>

</connectionStrings>



Obr. 13 Přihlašovací okno MS SQL Server

# Spuštění první migrace

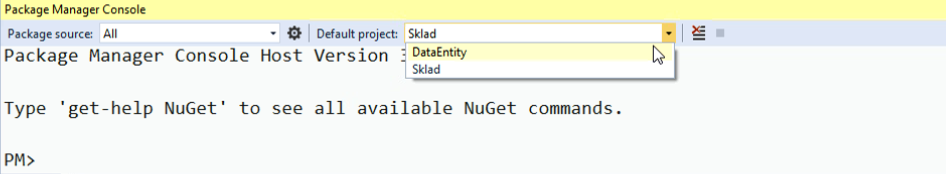
*Migrace představují v rámci Entity Framework Code First nástroj pro komunikaci objektový model (MODEL) vs. Databáze. Migrace je programový předpis pro aktualizaci databáze a je generována anutomaticky na základě změn v MODELu.*

*Pozn.: Migrace jsou také užitečný nástroj pro udržování databáze aktuální: Pokud distribuujeme aplikaci a vytvoříme novou verzi, zbývající migrace (aktualizace databáze) se automaticky spouští dle potřeby (pokud je vše správně nastavenou).*

*Vždy je možno pomocí migrací aktualizovat nahoru nebo i dolu (vracet se ke starším verzím databáze.*

V této kapitole se naučíme používat migrace pomocí NuGet konzole a základních migračních příkazů a budeme jimi schopni aktualizovat databázi.

Pomocí komunikační konzole vytváříme postupné aktualizace na serveru. Konzoli spustíme pomocí **Tools -> NuGet Package Manager -> Package Manager Console.** Konzole se otevře ve spodní části uživatelského prostředí. Pokaždé, když zapneme konzoli, je nutné vždy přepnout projekt na **DataEntity(!) -** viz Obr. 14. (s databází pracujeme na úrovni projektu DataEntity a nikoliv Sklad)



Obr. Komunikační konzole

## Migrace – povolení migrací (příkaz Enable-Migrations)

Migrace jsou části kódu, kterými se může aktualizovat databáze do dílčí verze nebo kterými lze provedené změny vrátit zpět do původního stavu. Nejprve je nutné migrace povolit. Do komunikační konzole vložíme příkaz PM**> Enable-Migrations.** Při zadávání příkazů do konzole je nutné dodržet jejich formát (velká a malá písmena, pomlčky)! Poté příkaz potvrdíme Enter. Pokud jsme postupovali správně, vytvoří se nám v **SolutionExplorer** nový adresář **Migrations**.

Pokud vše proběhne správně, bude v Solution Exploreru vytvořen v rámci projektu DataEntity nový adresář Migrations se třídou Configurations.cs.

### Důležité - Časté chyby:

1. *Pokud k jeho vytvoření nedošlo, je nutné ověřit, zda se podařilo projekt přeložit (tj. vytvořit tzv. Build – exe soubor a ostatní potřebné soubory). Pokud jsou v projektu syntaktické chyby, samozřejmě projekt přeložit nepůjde. Seznam chyb nalezneme v okně Error List (zobrazíme pomocí View – Error List). Poklepáním na chybu se přesuneme k části kódu, kde chyba vznikla a opravíme ji (většinou nesprávná syntaxe). V této fázi lze předpokládat spíše chyby vzniknuvší překlepy, nicméně v případě jiné neznámé chyby doporučuji hledant dílčí chybu na Internetu.*

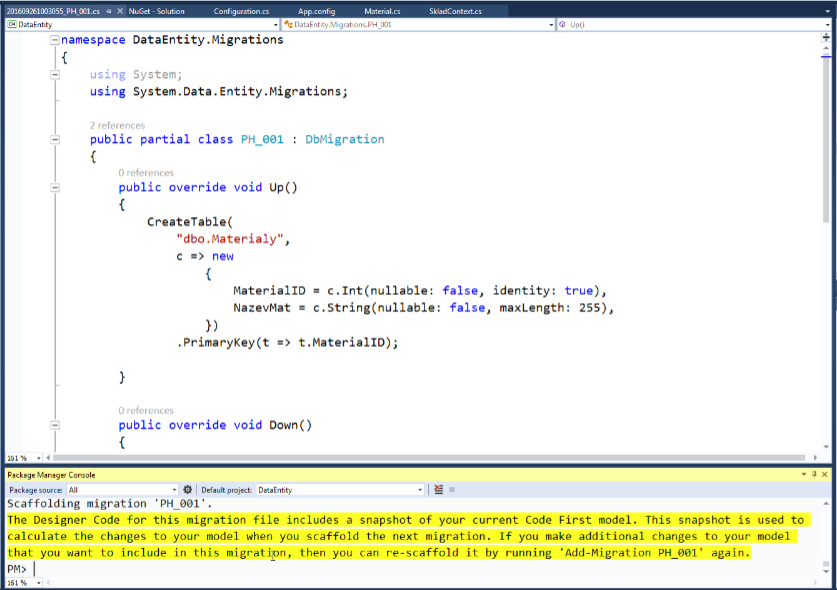
*Po odladění je možno zkusit přeložit a sestavit projekt(y) pomocí* ***Build*** *–* ***Build Solution****. Pokud se zobrazí dialogové okno, je něco špatně a je potřebaznovu nahlédlout do okna Error List.*

1. *Pokud jsou stávající kódy bez chyby a konzole stále nefunguje, je dalším možným řešením přeinstalace EntityFramework (toto se děje víceméně jen v našich laboratořích kvůli zajtím nezjištěné chybě, která bude nejspíše souviset s uživatelskými právy). V panelu nástrojů zvolíme záložku* ***Tools*** *a poté* ***NuGet Package Manager*** *->* ***Manage NuGet Package for Solution.*** *V novém okně zvolíme balíček EntityFramework. V pravé části pak zaškrtneme políčka* ***Projec****t****, DataEntity a Sklad*** *a zvolíme* ***Uninstall.*** *Po jejich odinstalaci je opět znovu nainstalujeme jako v 1.4. Po instalaci restartujeme Visual Studio.*

## Vytvoření migrace (příkaz Add-Migration název migrace)

*Nyní vytvoříme první migraci (pojmenovanou aktualizaci), pomocí které bude možno provádět upgrade (nebo downgrade) databáze. Každá migrace musí být pojmenovaná – je dobrým zvykem uvádět pojmenování migrací svými iniciály (kvůli práci v týmu).*

Migraci vytvoříme v NuGet konzoli příkazem PM**>Add-Migration PH\_001 *(název migrace zadáme libovolně).*** Potvrdíme Enter a vygeneruje se nám kód – viz Obr. 15.



Obr. Kód vygenerovaný pomocí migrace

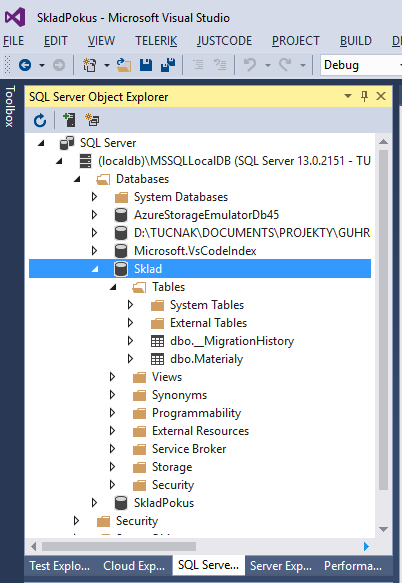
Tento automaticky generovaný kód říká, že se v rámci databáze vytvoří tabulka s názvem Materiály, která bude obsahovat atributy (sloupce) MaterialID, který bude obsahovat datový typ Integer *(c.Int*), není povolena prázdná hodnota *(nullable: false)* a je povolena autoinkrementace *(identity: true).* Dalším atributem tabulky bude NazevMat s datovým typem string, s nepovolenou hodnotou null (prázdnou hodnotou) a maximální délkou záznamu 255 znaků. Primární klíč je nastaven MaterialID.

## Aktualizace databáze (příkaz Update-Database)

Příkaz slouží k tomu, aby se námi provedené změny projevily v databázi. Do konzole zapíšeme příkaz PM**> Update-Database** a potvrdíme Enter. Pokud proběhne aktualizace databáze, máme ověřeno, že předešlé kroky a konfigurace byly správně provedeny.

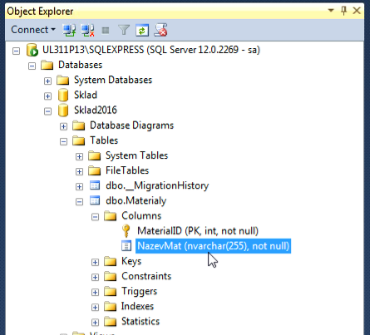
*Pozn.: Pokud neproběhne aktualizace, je pravděpodobné, že je chyba v ConnectionString z App.config. Pokud i po kontrole ConnectionStringu nelze databázi aktualizovat, vyzkoušejte, zda jste schopni připojit se k SQL serveru – to můžete vyzkoušet např. v Excelu, viz Dodatek A, pro SQL Server pak Data – Načíst externí data, Z jiných zdrojů, SQL server. Zde si můžete snadno otestovat uživatele a hesla. Pokud se stále nelze připojit s SQL serveru – je možné, že není správně nastaven SQL server (povolit SQL Autentification Mode, povolení přihlásit uživatele, uživatelské heslo – viz Internet nebo vyučující)*

Nýní ověříme, zda databáze vznikla. Pro testovací účely využíváme lokální SQL server, který je součástí Visual Studia. Otevřeme panel SQL Server Object Explorer (pokud není zobrazen, je možné jej zobrazit pomocí View, SQL Server Object Explorer), ve kterém nalezneme pod názvem Serveru (výchozí je (localdb)\MSSQLLocalDB) a složkou Databases naší databázi.



Obr. Databáze Sklad na interním testovacím serveru

ALTERNATIVA SQL Server: Ověření změn zkontrolujeme přihlášením do SQL Server Management Studio – viz kapitola 1.7. Zde po připojení vidíme strukturu - viz Obr. 17. V Databázi Sklad2016 (popř. Sklad) vidíme vytvořené tabulky dbo.Materialy (dbo je standardní namespace v SQL serveru – můžete ignorovat) s námi vytvořenými sloupci a vlastnostmi a také tabulku dbo.MigrationHistory, kam se ukládá historie provedených migrací – tu si také prohlédněte, je to užitečná zpětná vazba, abychom věděli v jaké je databáze verzi.



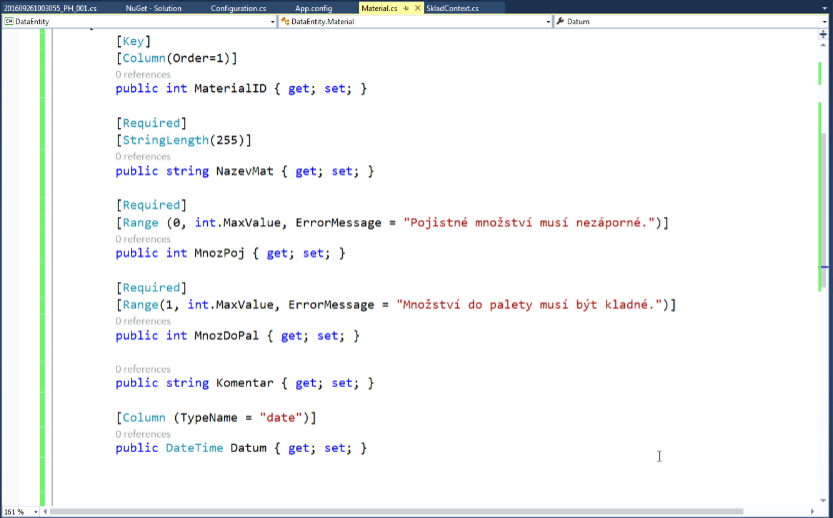
Obr. Struktura databáze po migraci a aktualizaci

# Úprava tabulky Materiály

V této kapitole vytvoříme kompletní tabulku Materialy pro cvičnou databázi za pomocí migrací

## Přidání atributů Pojistné množství, Množství do palety, Komentář, Datum a čas

V MS Visual Studio se přepneme do okna editoru třídy **Material**. Zde vložíme nové sloupce pomocí kódů - viz Obr. 18.



Obr. Kód pro vytvoření sloupců v tabulce Material

### Založení sloupce Pojistné množství:

[Required] //atribut musí mít hodnotu

[Range (0, int.MaxValue, ErrorMessage = "Pojistné množství musí být kladné celé číslo.")] //zadaná hodnota musí být v rozmezí 0 - maximální hodnota datového typu Integer, pokud je zadaná hodnota jiná, zobrazí se chybová hláška

public int MnozPoj { get; set; } //vytvoření sloupce MnozPoj datového typu Integer

### Založení sloupce Množství do palety:

[Required] //atribut musí mít hodnotu

[Range(0, int.MaxValue, ErrorMessage = "Množství do palety musí být kladné celé číslo.")] //zadaná hodnota musí být v rozmezí 1 - maximální hodnota datového typu Integer

public int MnozDoPal { get; set; } //vytvoření sloupce MnozDoPal datového typu Integer

### Založení sloupce Komentář:

public string Komentar { get; set; }

### Založení sloupce Datum a čas:

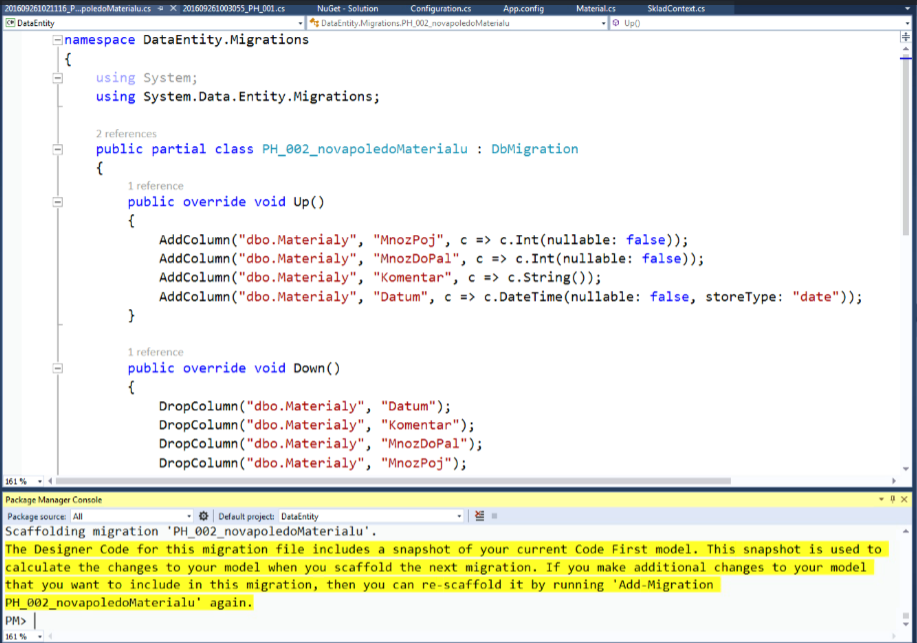
[Column(TypeName = "date")] //jméno typu date

public DateTime Datum { get; set;} //vytvoření sloupce Datum datového typu DateTime

Po provedených změnách vytvoříme v konzoli novou migraci například PM**>Add-Migration PH\_002.** *(pozn. U názvu migrace můžeme rovněž zvolit popis provedených změn, tedy např. Add-Migration PH\_002\_PridaniSloupcuDoMaterial.)* Potvrdíme Enter a vygeneruje se nám kód – viz Obr. 19 - kde vidíme provedené změny (upgrade kód pro přidání sloupců s jejich datovými typy a dalšími vlastnostmi a také downgrade kód DropColumn pro případné vrácení do verze databáze před touto migrací).

Změny ve struktuře vidíme na panleu SQL Server Object Explorer.

Pokud otevřeme databázi v SQL Server Management Studio, ve struktuře uvidíme provedené změny (přidání nových sloupců v tabulce Materiály a tabulka s provedenou migrací PH002)



Obr. Kód vygenerovaný po provedení migrace pro přidání nových sloupců do tabulky Material

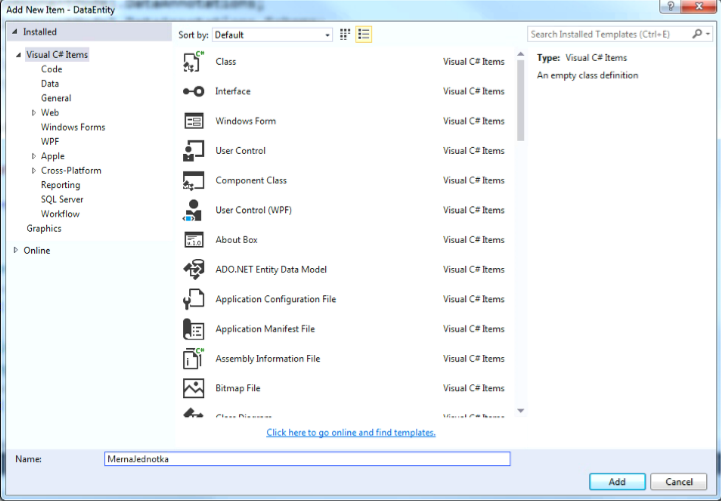
# Tabulka Měrné Jednotky

V této kapitole se založí tabulka měrné jednotky a vysvětlí se zakládání cizích klíčů.

*Zatím jsme tabulky ještě nepropojovali, nyní založíme tabulku měrné jednotky, která bude propojena s tabulkou Materiály vazbou 1:N. Tabulka MerneJednotky je seznam různých měrných jednotek (kg, ks, l, …). V tabulce Materialy založíme „odkaz“ na jednu měrnou jednotu. V tabulce MerneJednotky bude zase seznam Materiálů, tj. bude možno pro danou měrnou jednotku snadno nalézt seznam materiálů, ke kterým je přiřazena.*

## Založení tabulky MernaJednotka

V Solution Explorer pod projektem **DataEntity** a adresářem **Database** založíme novou třídu (**Add – New item**). Zvolíme typ **Class** a pojmenujeme **MernaJednotka**.



Obr. Založení třídy MernaJednotka

## Nastavení třídy MernaJednotka – nastavení cizího klíče

V okně editoru **MernaJednotka** následně smažeme **namespace DataEntity~~.Database~~**(jak bylo zmíněno dříve, odstranění Database provádíme u každé nově založené třídy). Následně provedeme editaci:

*Pozn. Důvodem je, aby vše bylo ve stejném namespace. Pokud bychom tak neučili, museli bychom ze všech tříd, kde bychom datový model využívali, uvádět using DataEntity a ještě např. using DataEntity.Database.*

namespace DataEntity

{

// [Table("MerneJednotky")] – tuto část odkomentujeme v další kapitole

public partial class MernaJednotka // : BaseModel

{

[Key]

public int MernaJednotkaId { get; set; } //založení primárního klíče (pokud pojmenujeme třídu Id, bude docházet k automatické inkrementaci této hodnoty)

[StringLength(50)] //nastavení délky datového typu String

public string Popis { get; set; } //sloupec Popis datoého typu String

public virtual ObservableCollection<Material> Materialy { get; set; } //vytvoření N ve vazbě 1:N, která je mezi entitami Material a Merna Jednotka

}

}

Pomocí ikony se žárovkou doplníme odkaz na knihovnu, kde se nachází seznam ObservableCollection - using System.Collections.ObjectModel.

Poté musíme ještě zajistit druhou část vazby 1:N ve třídě **Materialy**, kam vložíme:

public virtual MernaJednotka MernaJednotka { get; set; }

*Pozn. Slovo* ***virtual*** *je zde použito kvůli aktivaci tzv. lazy loading pro tyto vlastnosti. To znamená, že související data budou z databáze načítána vždy v okamžiku potřeby. Jinými slovy, pokud bude v budoucím kódu např. MernaJednotka mj = TentoMaterial.MernaJednotka; provede Entity Framework SQL příkaz SELECT \* FROM MerneJednotky WHERE MernaJednotkaId = TentoMaterial.MernaJednotkaId a dle výsledku tohoto dotazu naplní jednak TentoMaterial.MernaJednotka a současně i referenci mj – vše se děje pak automaticky a programátor se nemusí o nic starat!*

*Poprvé se zde setkáváme se seznamem tříd, tzv. Kolekcí – druhů takových seznamů může být v C# celá řada (nejtypičtější je typ List nebo IEnumerable), my budeme všude používat* ***ObservableCollection****, neboť tuto kolekci lze velmi snadno využívat na WPF formulářích, neboť „umí“ reagovat na změny směrem do grafického prostředí).*

Dále otevřeme třídu **SkladContext**, do které přidáme novou vlastnost:

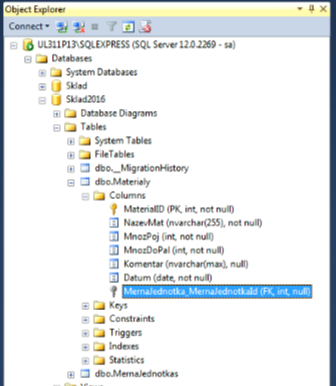
public virtual DbSet<MernaJednotka> MerneJednotky { get; set; }

Tato vlastnost umožní přistupovat k Měrným jednotkám.

## Vytvoření migrace

Po provedení změn vytvoříme migraci (jak vytvořit migraci viz kapitola 2.2). Důležité je opět změnit **Default Project** ze **Sklad** na **DataEntity**! Poté zadáme příkaz k vytvoření migrace **PM> Add-Migration PH\_003\_PrvniCiziKlic** a potvrdíme Enter (pokud nastane chyba, zkuste některé z řešení – viz kapitola 2.1 Časté chyby).

Poté aktualizujeme databázi příkazem **PM> Update-Database**. Otevřeme si SQL Object Explorer nebo SQL Server 2014 Management Studio (náhled je obdobný) a na Obr. 21 vidíme, že do databáze byla přidána tabulka **MernaJednotkas** (pojmenování je z důvodu výchozího nastavení prostředí Entity Framework převedeno na množné číslo (tzv. pluralizace), což dobře funguje v angličtině, v češtině však vznikají „podivné“ názvy – později upravíme) a v tabulce **Materialy** přibyl cizí klíč s výchozím jménem **MernaJednotka\_MernaJednotkaID** (také brzy přejmenujeme).



Obr. Struktura databáze

## Přejmenování tabulky a přidělení jména cizímu klíči

V předchozím kroku jsme vytvořili tabulku a cizí klíč, jejichž názvy nejsou zcela v pořádku. Tabulku **MernaJednotkas** přejmenujeme tak, že do třídy **MernaJednotka** vložíme nad definici třídy **MernaJednotka** tuto datovou anotaci – v předchozím kódu okomentováno:

[Table("MerneJednotky")]

Příkaz se označí jako chybný, vyřešíme opět pomocí ikonky žárovky (přidání using).

Následně provedeme zápis do třídy **Material** podle Obr. 22 :

public int MernaJednotkaId { get; set; }

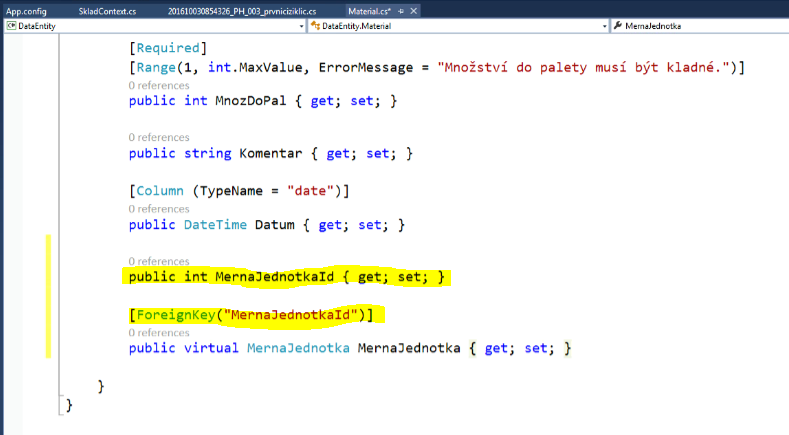
[ForeignKey("MernaJednotkaId")]

public virtual MernaJednotka MernaJednotka { get; set; }

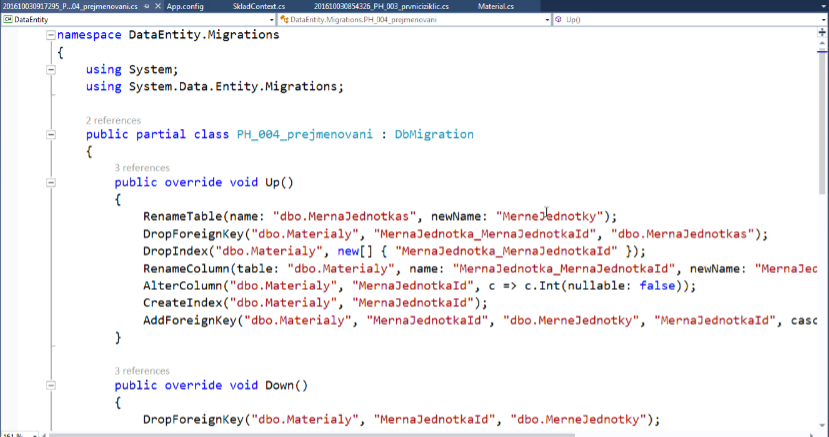
vytvoříme novou vlastnost MernaJednotkaId a ke stávajícímu odkazu připojíme datovou anotaci, která jednoznačně spojí definici objektu a nové vlastnosti s cizím klíčem (text uvnitř anotace ForeignKey se musí shodovat s názvem nové vlastnosti).

*Shrnutí: Vztah mezi materiálem a měrnými jednotkami je 1:N. Na straně Materiálu je jedna jediná měrná jednotka, tj. Ve třídě materiál je odkaz na třídu MernaJednotka. Na straně měrných jednotek je možno ke každé měrné jednotce přiřadit více materiálů, tj. na straně měrných jednotek je seznam materiálů (kolekce ObservableCollection).*

Na provedené změny vytvoříme migraci **PM> PH\_004\_prejmenovani** a potvrdíme. Vygeneroval se nám kód na Obr. 23, kde vidíme přejmenování tabulky, odstranění klíče, přejmenování sloupce a opětovné přidání klíče s novým názvem. Pokud si databázi otevřeme v SQL Server Object Exploreru nebo SQL Server Management Studio, uvidíme provedené změny (přejmenování tabulky **MerneJednotky** a přejmenování cizího klíče v tabulce **Materialy**)



Obr. Přejmenování cizího klíče



Obr. Kód po migraci přejmenování tabulky a cizího klíče

# Base třída a naplnění tabulek

Base (bazická) třída slouží k dědění vlastností. Veškeré vlastnosti, které definujeme v této třídě se převedou (dědí) na potomky, kterými jsou v tomto případě jednotlivé třídy **Material**, **Palety**, atd. Nemusíme tedy tyto vlastnosti psát pro každou třídu zvlášť.

*To je v rámci MODELu obzvláště užitečné, pokud chceme definovat společnou funkčnost nebo vlastnosti. Pokud bychom např. chtěli v každé tabulce mít informace o uživateli, který založil záznam, popř. kdy jej založil, jednoduše založíme tyto vlastnosti v bazické třídě a následně budou k dipozici ve všech tabulkách.*

## Založení třídy

Pod adresářem **Database** v **Solution Explorer** vytvoříme novou složku (**Add – New Folder**) a pojmenujeme jí **Base**. V této složce poté vytvoříme novou třídu (**Add – New Item**) **Class** a pojmenujeme ji **BaseModel.** Opět smažeme namespace DataEntity.~~Database.Base.~~

BaseModel.cs je možno buď si nahrát z hotového projektu, popř. zkopírujte následující kód:

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Linq.Expressions;

using System.Runtime.CompilerServices;

using System.Text;

using System.Threading.Tasks;

namespace DataEntity

{

public abstract class BaseModel : IDataErrorInfo

{

#region "validace"

string IDataErrorInfo.Error

{

get

{

//return null;

throw new NotSupportedException("IDataErrorInfo.Error is not supported, use IDataErrorInfo.this[propertyName] instead.");

}

}

string IDataErrorInfo.this[string propertyName]

{

get

{

// return null;

return OnValidate(propertyName);

}

}

protected virtual string OnValidate(string propertyName)

{

if (string.IsNullOrEmpty(propertyName))

throw new ArgumentException("Invalid property name", propertyName);

string error = string.Empty;

var value = this.GetType().GetProperty(propertyName).GetValue(this, null);

var results = new List<ValidationResult>(1);

var context = new ValidationContext(this, null, null) { MemberName = propertyName };

var result = Validator.TryValidateProperty(value, context, results);

if (!result)

{

var validationResult = results.First();

error = validationResult.ErrorMessage;

}

return error;

}

#endregion

[Timestamp]

public byte[] RowVersion { get; set; }

}

}

*Tento kód se zdá být trochu složitější, nicméně asi není potřeba mu kompletně rozumět. Bázická třída bude přidávat ke všem ostatním třídám, které tuto třídu oddědí následující funkčnost:*

1. *Díky implementaci rozhraní IDataErrorInfo a přidání metody OnValidate fungují všechna ověření (která byla již přidána) pomocí datových anotací*
2. *Díky vlastnosti RowVersion, která v databázi bude typu TimeStamp (časová značka) budeme vědět pro každou řádku i její verzi – to je klíčové pro řízení víceuživatelského přístupu (zde uvedeno pro zajímavost, dále se nejspíše víceuživatelským řízením nebudeme zabývat, nicméně je ukázáno ve finálním projektu SkladFINAL)*

Dále je nutné do všech tříd, které budou dědit třídu **BaseModel** přidat odkaz na tuto třídu. V definici každé další „databázové“ třídy (Material, MerneJednotky) přidáme **BaseModel**, tedy

public partial class Material : BaseModel

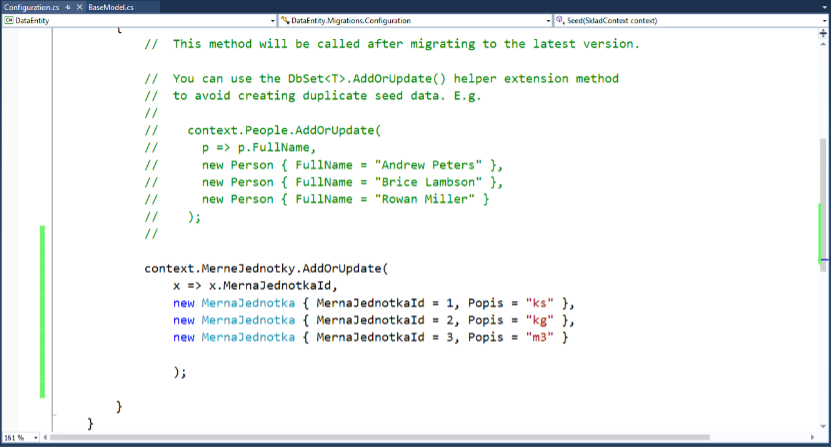
public partial class MernaJednotka : BaseModel

## Naplnění tabulky MerneJednotky daty

Jednou z možností, jak plnit vytvořené tabulky daty je pomocí SQL Server Object Explorer popř. pomocí SQL Server Management Studio, kde si můžeme otevřít např. tabulku Měrné jednotky, pravým tlačítkem vybereme možnost View Data, popř. Edit Top 200 Rows a můžeme plnit daty – to je výhodné provést, v okamžiku, kdy máme hotovou vstupní datovou strukturu.

Pro distribuci databáze na více počítačů je však vhodnější plnit tabulky daty pomocí metody Seed, která se zavolá po každé aktualizaci databáze pomocí migrace – to se týká číselníkových tabulek, o kterých předpokládáme, že by se jejich obsah mohl na různých instalacích shodovat.. Tady je výhodné využít (tzv. extenzní) metodu AddOrUpdate, která data buď přidá, v případě, že v tabulce nejsou nebo stávající data aktualizuje.

Otevřeme soubor **Configuratin** v **Solution Explorer** v adresáři **Migrations** a vyhledáme jeho část na Obr. 24. Zde máme pomocí komentáře vytvořenou „předlohu“ pro vyplňování dat.



Obr. Předloha v metodě Seed

Zadáme do metody Seed nový kód:

context.MerneJednotky.AddOrUpdate(x => x.Popis,

new MernaJednotka { Popis = "ks" },

new MernaJednotka { Popis = "kg" },

new MernaJednotka { Popis = "m3" });

*//každým zápisem přidáme nový záznam a uvedeme jeho popis v tabulce, pokud již daná hodnota v tabulce existuje, nebude přidána*

Po úpravách opět provedeme migraci **PM> Add-Migration PH\_005\_rowversion** a aktualizujeme databázi **PM> Update-Database**. Následně v SQL Server Object Explorer, popř. SQL Server Management Studio otevřeme pravým tlačítkem tabulku Měrné jednotky a přes **View Data**, popř. **Select first 1000 rows** tabulku zobrazíme. Měla by být naplněna zadanými údaji, tedy ID 1-3 a ks, kg, m3.

Tabulku Materialy bude výhodnější naplnit až později

# Injekce kódu (Fody)

*Všechny třídy bude nyní potřeba připravit pro využití ve WPF. Potřebujeme zařídít, aby třídy fungovaly tak, aby se každá změna v hodnotě jakékoliv vlastnosti (property) vyvolala i změnu v grafice (pokud je to relevantní). To se dělá pomocí implementace INotifyPropertyChanged do bazické třídy (v našem případě by to byla třida BaseModel), kde se definuje událost OnPropertyChanged a ta se pak musí volat naprosto všude, kde dochází ke změně jakékoliv hodnoty, např. takto:*

*private int \_mnozPa;*

*public int MnozPa*

*{*

*get*

*{*

*return \_mnozPa;*

*}*

*set*

*{*

*\_mnozPa = value;*

*OnPropertyChanged(() => MnozPa);*

*}*

*}*

*Pracovat tímto „starším“ zbůsobem dá velkou práci a vzniká spousta kódu (tento způsob je naznačen ve finální projektu SkladFINAL – kde je toto vidět v bazické třídě a ve vlastnosti MnozPa).*

*Jelikož je programátor obecně tvor líný, existují různé zkratky: Nejsnazší je použít tzv. injekci kódu pomocí balíku Fody. Pak není potřeba nic implementovat a Fody pak sám při překladu z např. jednoduchého kódu se standardními getry a setry:*

*public int MnozPa { get; set; }*

*při překladu vytvoří kód výše uvedený (provede tzv. injekci kódu). Programátor se tedy nemusí téměř o nic starat – jen musí Fody využít (dále)*

## Nainstalování rozšíření

Stejně jako jsme v 1.4 instalovali rozšíření Entity Framework (**Tools** - **NuGet Package Manager** - **Manage NuGet Package for Solution**), tak nyní přidáme nejprve rozšíření **Fody** a potom **PropertyChanged.Fody**. Při jejich instalaci nezapomene označit pole **DataEntity i Sklad.**

Po jejich instalaci do všech tříd (Material, MerneJednotky…) přidáme za namespace DataEntity příkaz

[ImplementPropertyChanged]

*Pozn.: Touto datovou anotací zajistíme nastavení standardních příkazů get; set; na nestandardní, které umožní zobrazení změn kódu v prezentačním rozhraní. Tato injekce představuje nebývale obrovské ulehčení práce.*

***Důležité!: v září 2017 došlo k aktualizaci – je třeba využívat anotaci:***

**[AddINotifyPropertyChangedInterface]**

**Jinak vše funguje standardně.**

Tuto datovou anotaci bude uvádět jak u všech (databázových) tříd z projektu (knihovny) DataEntity tak u všech ViewModelů (viz úvod, viz dále).

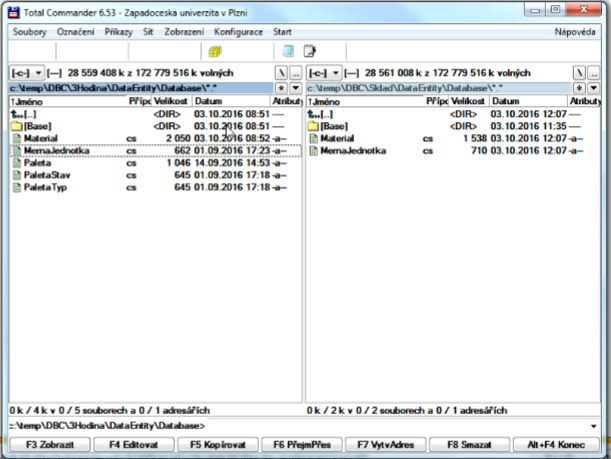
# Přidání tříd Paleta, PaletaTyp, PaletyStav

Posledním krokem k vytvoření modelu je přidání zbývajících tříd. Pro jejich přidání bychom postupovali postupem dříve zmiňovaným v kapitolách 1.5 (přidání nové třídy), 2 (tvorba migrací), 3.1(přidání atributů do tabulky), 4.2 (tvorba cizích klíčů), 5 (vložení odkazu na BaseModel) a 6 (vložení příkazu na injekci kódu).

## Přidání nových souborů do adresáře

Pro urychlení práce je možné stáhnout si z **Courseware -DBC – Cvičení – FTP – Podklady – TridyDBSklad (TridyDBSklad.zip)** soubory, ve kterých jsou tyto třídy již vytvořené. V adresáři **DataEntity – Database** nalezneme soubory **Paleta, PaletaStav a PaletaTyp**, které překopírujeme do kořenového adresáře databáze **(Sklad – DataEntity – Database)** viz. Obr. 25.

Jakmile se po zkopírování nových souborů vrátíme do Visual Studio, objeví se zpráva o zjištěné změně souborů. Povolíme změny možností **Yes to All.**



Obr. Kopírování tříd do adresáře (verze pro Total Commander)

Nově přidané soubory se zatím v Solution Explorer nezobrazují. Pro jejich zobrazení zvolíme možnost **Show All Files** z pásu karet Solution Explorer (Obr. 26). Poté klikneme pravým tlačítkem na nově zobrazené třídy a zvolíme možnost **Include in Project**.

Zde vidíme kód pro vytvoření tabulky PaletyStavy. Jsou zde použity všechny příkazy jako v předchozích případech pro vytvoření tabulky:

namespace DataEntity //smazání .Database

{

[AddINotifyPropertyChangedInterface] //zajištění injekce kódu

[Table("PaletyStavy")] // pojmenování tabulky

public partial class PaletaStav: BaseModel //vytvoření veřejné třídy a připojení na BaseModel

{

[Key]

public int PaletaStavId { get; set; }

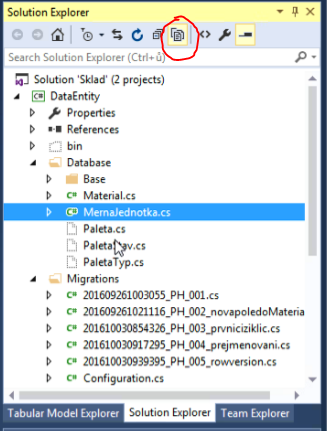
[StringLength(50)]

public string Popis { get; set; }

public ObservableCollection<Paleta> Palety { get; set; }

}

}



Obr. Zobrazení nově přidaných souborů

V našem případě je ještě nutné do třídy **Material** vložit vazbu na tabulku Palety – to aby bylo možno zobrazovat všechny palety, ve kterých je materiál:

public virtual ObservableCollection<Paleta> Palety { get; set; }

*Pozn.: Raději zkontrolujme, zda jsou všechny vazby v pořádku, tj. cizí klíče – (dle Obr. 22), např.:*

public int nazev klíče (např. MaterialID) { get; set;}

[ForeignKey("MaterialID")]

Dále je nutné upravit třídu **SkladContext** dle:

public SkladContext () : base ("name = SkladContext")

{

}

public partial class SkladContext : DbContext

{

public virtual DbSet<Material> Materialy { get; set; }

public virtual DbSet<Paleta> Palety { get; set; }

public virtual DbSet<PaletaTyp> PaletyTypy { get; set; }

public virtual DbSet<PaletaStav> PaletyStavy { get; set; }

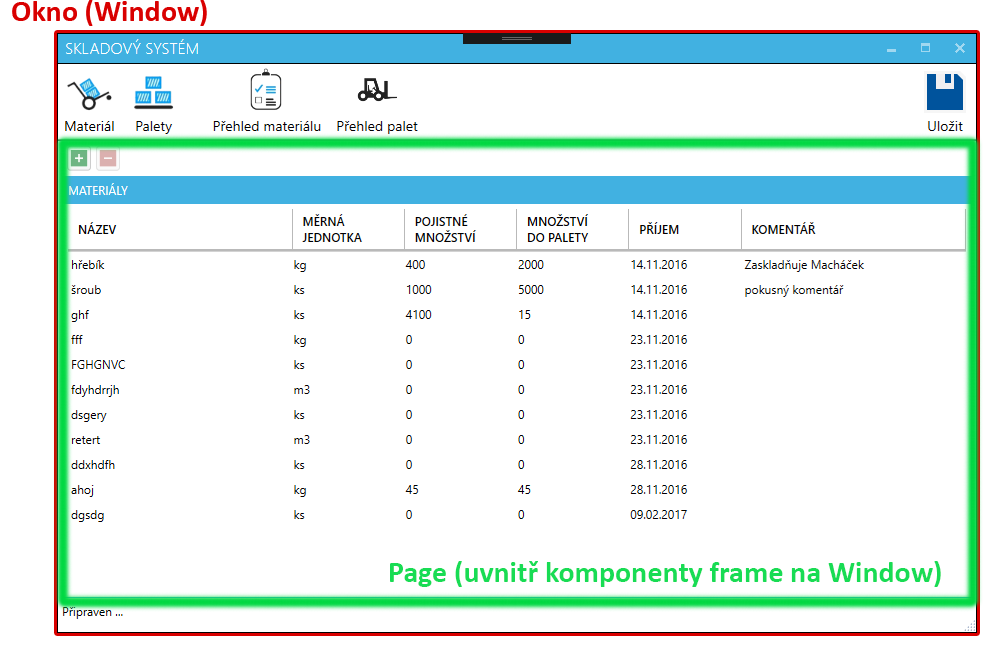
public virtual DbSet<MernaJednotka> MerneJednotky { get; set; }

}

Po provedení všech změn zbývá vytvořit migraci **PM> Add-Migration PH\_006** a aktualizujeme databázi **PM> Update-Database**

# Prezentační vrstva

V této kapitole se naučíme vytvářit grafické rozhraní v rámci tzv. prezentační vrstvy vytvářené aplikace. Za tímto účelem se naučíme používat XAML značkovací jazyk (markup language), který vychází z XML. Tento značkovací jazyk budeme využívat pro tvorbu GUI – grafického uživatelského prostředí. Tento jazyk je dnes standardem. Pokud jste někdy vytvářeli webové stránky pomocí HTML, vězte, že z hlediska struktury je XAML podobný.



Obr. MDI logika aplikace

*Pro naší prezentační vrstvu budeme využívat známou logiku tvorby aplikací – MDI (Multiple Document Interface). V našem případ to znamená, že budeme v rámci jednoho okna – Window moci zobrazovat více různých podformulářů (které budou v našem případě typu Page). Tyto podformuláře se budou v rámci hlavního okna* ***winHlavni*** *(****MainWindow)*** *zobrazovat v komponentě typu frame – rámeček (zelená oblast na Obr. 27). Uvnitř tohoto rámečku budeme zobrazovat různé stránky (Pages) – dle pořadí, ve kterém je budeme vytvářet:*

* ***pgPrehMat*** *– Přehled materiálu – seznam materálů s pasivní informací, kde jsou v paletách uložené*
* ***pgPalety*** *– Jednoduché zakládání palet do systému*
* ***pgMaterial*** *– Zakládání materiálů (viz Obr. 27)*
* ***pgPrehPalet*** *– Poměrně komplexní formulář pro přiřazování materiálů do palet, správu, rezervace atd.*

*Dva poslední formuláře budou předvedeny jen přehledově.*

## Startup okno

Standardně se hlavní spouštěcí okno jmenuje MainWindow a nachází se v rootu projektu. Jelikož budeme chtít mít v projektu trochu „pořádek“, bude se toto okno jmenovat winHlavni a bude v adresáři Presentation.

Klikneme pravým tlačítkem na název projektu **Sklad (nikoliv DataEntity)** a vytvoříme nový adresář (**Add – New Folder).** Tento adresář pojmenujeme **Presentation**. Do tohoto adresáře budeme ukládat všechny kódy, které obstarávají grafickou vrstvu aplikace.

Klikneme pravým tlačítkem na nový adresář Presentation a založíme nové okno: **Add – New Item**. Z nabídky vybereme možnost **Window (WPF)** a jako název vyplníme winHlavni.xaml (pozor na velká a malá písmena).

Nyní jsme přidali nové okno. Teď je potřeba zvolit, že toto okno bude startovací (hlavní): najdeme si v rootu projektu Sklad konfigurační soubor projektu App.xaml, kde změníme vlatsnost StartupUri takto (včetně adresáře):

<Application x:Class="Sklad.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:Sklad"

StartupUri="Presentation/winHlavni.xaml">

Nyní je možné „staré“ MainWindow.xaml smazat (pravé tlačítko myši na název MainWindow.xaml a Delete).

## Přidání nových souborů do projektu

Pro usnadnění práce byly vytvořeny soubory potřebné pro práci. Na **Courseware** předmětu DBC, v sekci cvičení na odkazu **FTP** najdeme pracovní podklady. Zde stáhneme soubor **TridyDBSklad.zip.** V tomto souboru nalezneme adresář **Sklad**, jehož obsah nahrajeme do adresáře Sklad v našem projektu.

Ve VisualStudio poté označíme v SolutionExplorer projekt Sklad a zvolíme Show all files jako na Obr. 26.

Následně připojíme do projektu následující adresáře: **Dictionary, Globals, Resources a Validations**. Soubory připojíme pravým klepnutím na soubor a zvolení Include in Project.

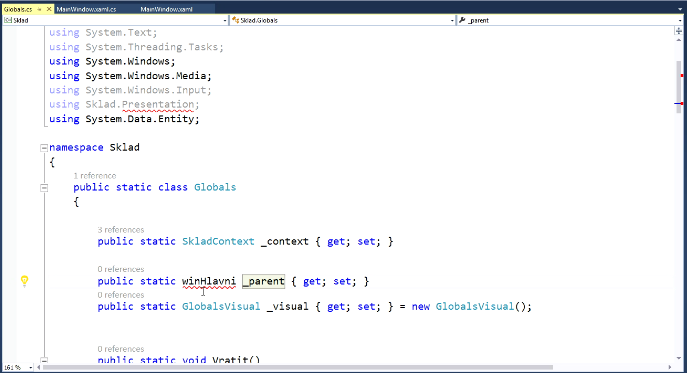
Nový adresář Dictionary obsahuje seznam stylů, kde je předdefinováno, jaké má být formátování komponent jako TextBox, ComboBox atd.

Adresář Globals obsahuje například rutiny, které umožňují vrácení k předchozímu stavu. V Globals otevřeme třídu Globals.cs viz. Obr. 28 a opravíme chyby. V prvním řádku using Sklad.Presentation; tento řádek **okomentujeme** – buďto napíšeme před řádek // a nebo jej označíme a kombinací kláves Ctrl + K + C okomentujeme. Komentovaný obsah můžeme odkomentovat Ctrl + K + U.

Stejně tak okomentujte řádku:

// public static GlobalsVisual \_visual { get; set; } = new GlobalsVisual();

Třídu GlobalsVisual vyloučíme z projektu – pravým tlačítkem na ní v Solution Explorer a Exclude from Project. (tato třída slouží pro ovládání vizuální stylů oken z prostředí Metro – viz dále – není nutno ji používat).



Obr. Globals.cs

Adresář Resource obsahuje obrázky. V tomto adresáři najdeme obrázky, které budeme používat pro tvorbu uživatelského prostředí. Pokud najedeme myší na nějaký obrázek v adresáři, zobrazí se jeho náhled.

## Hlavní okno

V SolutionExplorer otevřeme **winHlavni.xaml v adresáři Presentation**. Pro otevření kódu v novém okně (to šikovné pro práci na dvou a více monitorech) můžeme kliknout pravým tlačítkem na **winHlavni.xaml**, dále zvolit **Open with..** a **Source Code Editor**. Tím se nám xaml kód otevře ve větším okně nebo ho můžeme přetáhnout na druhou obrazovku. Kód vidíme na **Chyba! Nenalezen zdroj odkazů.**.

V XAML se pracuje pomocí tzv. tagů, které označují začátek a konec nějaké části nebo prvku. Začátek části se vždy označuje <nazev> a konec se označuje </nazev>. Na obrázku vidíme například **<Window>** a **</Window>,** což jsou tagy, které ohraničují celou oblast okna. Ve Window je vnořen například **<Button>,** který ohraničuje část pro tlačítko. Další možností je provést například takovýto zápis <Button nastavení vlastností />. Tento zápis funguje stejně jako dvojice tagů <> </> bez nutnosti ukončení bloku.

**Některé úvodní poznámky k XAML jsou v příloze D – doporučeno prostudovat.**

Podívejme se na vygenerovaný kód winHlavni. V tomto případě bylo do formuláře (okna) vloženo tlačítko:

<Window x:Class="Sklad.Presentation.winHlavni"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Sklad.Presentation"

mc:Ignorable="d"

Title="winHlavni" Height ="350" Width="525">

<Grid Name="grMain">

<Button Content="Ahoj" Width="100" Height="30"/>

</Grid>

</Window>

Kód je možné měnit přímo zápisem, nebo pokud v grafickém rozhraní označíme položku (například Okno, ikonu, tlačítko…) zobrazí se nám pod SolutionExplorer okno Properties, kde můžeme měnit vlastnosti objektu a provedené změny se automaticky projeví v XAML kódu.

V kódu vidíme zápis Title =“winHlavni“ Height =“350“ Width=“525“. Tento zápis vytvoří na našem okně titulek winHlavni a toto okno bude mít rozměry 350x525. (Praxí se osvědčila velikost okna 600x920) Veškeré změny, které v kódu provedeme, se okamžitě zobrazují v grafickém rozhraní.

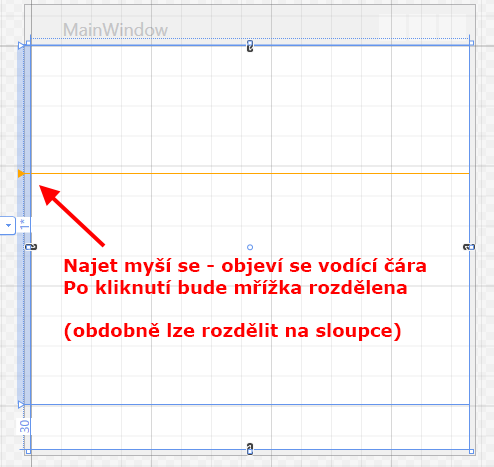
## Mřížky (grid)

Pro formátování budoucích komponent na grafickém rozhraní formulářů se používají např. a nejčastěji mřížky (grid). Pomocí těchto mřížek si okno formuláře rozdělíme na oblasti, do kterých budeme následně vkládat další komponenty. V XAML kódu je vytvořena oblast **<grid>.** Pokud v grafické rozhraní najedene myší na okraj okna, naznačí se nám oblast mřížky. Kliknutím pak tuto oblast vytvoříme. Rozdělíme si tedy oblast okna na tři horizontální oblasti dle Obr. 29. Dále pak vidíme XAML kód po vytvoření těchto oblastí.

### Nastavení výšek oblastí

Řádky **RowDefinition Height** určují výšky těchto oblastí. Existují tři možnosti zadaných hodnot:

* **Absolutní číselná hodnota**: Čísla uvádějí poměry výšek. Můžeme tedy u všech oblastí nastavit konkrétní hodnotu.
* **Stejné poměry**: Pokud bychom všechny výšky nahradili **“\*“**, tedy <RowDefinition Height =“\*“/>, každá z oblastí bude pokrývat právě třetinu celého okna. Pokud bychom například hodnotu první oblasti změnily na “2“, tak tato oblast pokryje 2/4 okna a zbývající dvě oblasti pak po ¼ okna.
* **Přizpůsobení se obsahu**: Pokud zadáme hodnotu **“auto“,** tak se velikost okna bude přizpůsobovat obsahu, například po vložení textu se velikost upraví, po změně výšky tohoto textu se opět oblast automaticky upraví



Obr. Tvorba mřížky

V našem případě potřebujem v prvním řádku mít menu (automatické přizpůsobení), v druhém řádnu bude Page-stránka (tj. \* - doplnění velikosti dle ostatních komponent) a stavový řádek na posledním řádku – pevná velikost:

<Window x:Class="Sklad.Presentation.winHlavni"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Sklad"

mc:Ignorable="d"

Title="Skladový systém" Height="600" Width="920">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

<RowDefinition Height="30"/>

</Grid.RowDefinitions>

</Grid>

</Window>

### Přiřazování komponent do jednotlivých řádků mřížky

V tomto bodě máme formulář rozdělen do tří řádků. Pokud bychom nyní do tohoto gridu vložili například tlačítko nebo rámeček, automaticky se nám vloží do prvního řádku. Kdybych tak nastavovali například odsazení či zarovnání, budou se tato nastavení vztahovat k tomuto řádku ,nikoliv celému formuláři. Pokud chceme vloženou komponentu přemístit do druhého či třetího řádku, použili bychom vlastnost komponenty **Grid.Row=“číslo řádku mřížky“**. Například zápis:

<Border Grid.Row=“1“ BorderBrush=“Black“ BorderThickness=“5“/>

na Obr. 30 vytvoří rámeček černé barvy a tloušťky 5px a přiřadí ho druhého řádku. Na první řádek se odkazujeme pomocí Grid.Row=“0“, další řádek má pak index vždy o 1 větší (1,2,3…). Pokud Grid.Row nebude u komponenty uvedena, automaticky se předpokládá první (nultý) řádek.

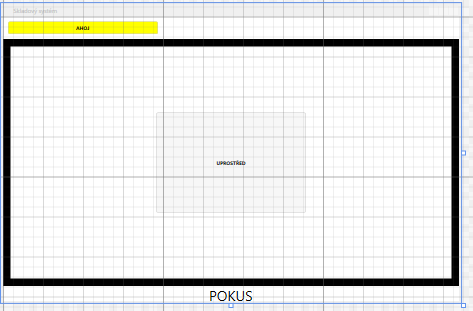
## Fiktivní ukázka přiřazení jednoduchých komponent do vytvořené mřížky

Zde si ukážeme, jaký způsobem by bylo možné vytvořit layout, který je vidět na Obr. 30.

1. V prvním řadku je tlačítko (Button), které obsahuje text AHOJ (vlastnost Content), pozadí (Background) je žluté, je odsazené vlevo (HorizontalAlignment) a má bílý okraj kolem sebe (Margin) o velikosti 10. Všimněte si, že vlastnost Grid.Row není uvedena – je to to samé, jako kdybychom uvedli Grid.Row=“0“
2. V druhém řádku – Grid.Row=“1“ jsou dvě komponenty – okraj/orámování o tlouštce 15 a tlačítko
3. Ve spodním řádku je textový blok (pro výpis textu na formulář), který je zarovnaný doprosřed

Mnoho vlastností komponent se překrývá (např. odsazení pro tlačítko nebo textbox), mnoho vlastností má každá komponenta unikátních.

*Pozn.: Komponenty lze vytvářet i vlastní – buď je možné vyjít z existující komponenty, nebo je možno vytvořit vše úplně od začátku. To však přesahuje rámec tohoto kurzu.*



Obr. Přiřazování komponent do jednotlivých řádků mřížky

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

<RowDefinition Height="30"/>

</Grid.RowDefinitions>

<Button Width="300" Content="AHOJ" Background="Yellow" HorizontalAlignment="Left" Margin="10"/>

<Border BorderBrush="Black" BorderThickness="15" Grid.Row="1"/>

<Button Width="300" Height="200" Content="UPROSTŘED" Grid.Row="1"/>

<TextBlock Text="POKUS" FontSize="28" Grid.Row="2" HorizontalAlignment="Center"/>

</Grid>

**Toto ukázka je pouze fiktivní. Vytvořené komponenty nebudeme pro další vývoj potřebovat. Je tedy možné definice všech tlačítek, okraje a textového bloku smazat.**

*Pozn.: Pokud byste potřebovali v rámci jedné buňky Gridu organizovat dílčí komponenty za sebou nebo vedle sebe, je potřeba použít <StackPanel> nebo <WrapPanel> (viz dále).*

## Založení stavového řádku

Dále založíme stavový řádek. Do gridu v XAML kódu vložíme:

<TextBlock Margin=“10,5,0,0“ Text=“Připraven…“ Grid.Row=“2“ FontSize=“14“ FontWeight=“SemiBold“/>

čímž vytvoříme textové pole, které bude odsazeno 10 bodů zleva, 5 bodů shora a 0 bodů zprava a zdola s textem Připraven… a přiřadíme ho do spodní oblasti. Text bude napsán velikostí znaku 14 a středně tučně. Oblast stavového řádku se následně automaticky přizpůsobí obsahu (to je definováno ve vlastnosti Gridu).

## Založení menu

Do první řádky Gridu vložíme položku Menu (MenuItem). Obecně lze ke každé vlastnosti přistupovat uvnitř tagu komponenty pomocí tečkové notace (zde MenuItem.Header nebo Grid.RowDefinitions). My pořebujeme změnit vlastnost Header (hlavička) a uložit do ní složitější strukturu – obrázek a nápis. Následující kód ukazuje, jak založit jednu položku menu s ikonou:

<Window x:Class="Sklad.Presentation.winHlavni"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Sklad.Presentation"

xmlns:Controls="clr-namespace:MahApps.Metro.Controls;assembly=MahApps.Metro"

mc:Ignorable="d"

Title="Sklad " Height="600" Width="920">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="100"/>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Menu>

<MenuItem Name="miPrehMat">

<MenuItem.Header>

<StackPanel>

<Image Width="48" Height="48" Source="..\Resources\MainMenu\prehledmat.png"/>

<TextBlock Text="Přehled materiálu" HorizontalAlignment="Center"/>

</StackPanel>

</MenuItem.Header>

</MenuItem>

</Menu>

</Window>

Rozepišme si v této a následující kapitole tvorbu tohoto kódu:

V XAML kódu si založíme oblast <Menu> </Menu>. Do tohoto tagu dále vložíme další tag <MenuItem> </MenuItem>. MenuItem pojmenujeme <MenuItem Name =“miPrehMat“>. Do MenuItem dále vložíme <MenuItem.Header> </MenuItem.Header> což nám zpřístupní možnost nastavení vlastního obsahu menu.

### Vložení obrázku do menu

Obrázky vkládáme pomocí tagu <Image Source=“cesta ke zdroji obrázku“/>. Mezy uvozovky musíme vložit odkaz na obrázek, který chceme vložit. V SolutionExplorer pod projektem **Sklad** otevřeme adresář **Resource – Icons – MainMenu.** Zde si vybereme obrázek a přetáhneme ho do XAML kódu do oblasti uvozovek. Tím se nám vytvoří cesta pro obrázek na konkrétním počítači. Abych zajistili kompatibilitu i na jíných počítačích, musíme vytvořit relativní cestu, smažeme tedy část odkazu a ponecháme pouze **Resources/MainMenu/prehledmat.png**.

*Pozn.: Alternativně je možné vybrat cestu v plovoucím okně Properties (označit obrázek a vybrat ze seznamu obrázků ve vlastnosti Source)*

### Vložení popisku ikon

Typicky je možné do komponenty, popř. do buňky gridu atp. vkládat jen jednu dílčí komponentu, pokud jich potřebujeme vkládat více, je tyto uzařít možné do panelu WrapPanel (standardně horizontální panel) nebo StackPanel (standardně vertikální panel). Abychom tedy mohli k ikonám vkládat i popisky, musíme založit tag <StackPanel> a do něho vložit jednak <Image>, kde musíme upravit rozměry obrázku a dále přidat <TextBlock Text =“Přehled materiálu“/>

Tímto máme hotový předpis pro první ikonu. Nyní celý tag <MenuItem> zkopírujeme a přepíšeme zde hodnoty pro další ikonu.

Kód je pak:

<Window x:Class="Sklad.Presentation.winHlavni"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Sklad.Presentation"

xmlns:Controls="clr-namespace:MahApps.Metro.Controls;assembly=MahApps.Metro"

mc:Ignorable="d"

Title="Sklad " Height="600" Width="920">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="100"/>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Menu>

<MenuItem Name="miPrehMat">

<MenuItem.Header>

<StackPanel>

<Image Width="48" Height="48" Source="..\Resources\MainMenu\prehledmat.png"/>

<TextBlock Text="Přehled materiálu" HorizontalAlignment="Center"/>

</StackPanel>

</MenuItem.Header>

</MenuItem>

<MenuItem Name="miPrehPal">

<MenuItem.Header>

<StackPanel>

<Image Width="48" Height="48" Source="..\Resources\MainMenu\prehledpal.png"/>

<TextBlock Text="Přehled palet" HorizontalAlignment="Center"/>

</StackPanel>

</MenuItem.Header>

</MenuItem>

</Menu>

</Window>

*Pozn.: je také možné vytvořit uživatelskou komponentu (UserControl) pro každou položku Menu – opět nad rámec tohoto kurzu*

## Příprava pro vložení podformuláře (Page) do framu formuláře

Dále budeme potřebovat do okna hlavního formuláře vložit podformulář (materiály, palety, přehled, … - dle toho, co vybere uživatel v menu). Pokud bychom potřebovali tyto podformuláře nějak nastavovat z hlavního formuláře, nebo zajistit jejich vzájemnou komunikaci, je lepší, aby každý tento podformulář byl typu UserControl (uživatelská komponenta). Vzhledem k tomu, že tuto funkčnost nepotřebujeme využívat, použijeme typ Page – strana. Tu budeme vkládat do tzv. rámce (Frame).

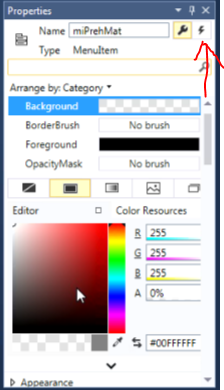
Do <Grid> v XAML za ukončovací tag </Menu> vložíme tag pro rámec – Frame, do které se budou v budoucnu vkládat podformuláře:

<Frame x:Name="frame" Grid.Row="1" NavigationUIVisibility="Hidden"/>

Frame je původně urče pro prohlížení dokumentů nebo webových stránek. Skryjeme tedy šipky pro přesun dopředu a dozadu (UIVisibility).

V SolutionExplorer klikneme pravým tlačítkem na **Sklad** a vložíme novou složku **Presentation**, do které budeme ukládat všechny soubory prezentační vrstvy. Do této složky vložíme **New item**. Zvolíme typ **Page** a pojmenujeme **pgPrehMat**.

Poté si ve Visual Studiu otevřeme **winHlavni** a v XAML kódu pod grafickým prostředím (nikoliv ve vytaženém podle 8.3) klikneme na MenuItem Name=“miPrehMat“. Tím se nám pod SolutionExplorer otevřou **Properties** této ikony. V okně Properties klikneme na ikonku blesku viz. Obr. 31 a otevře se nám seznam událostí. Zde dvakrát klikneme na událost **OnClick**. Tím se nám otevře okno události.



Obr. Události

V tomto okně provedeme zápis (opět musíme přidat namespace)

private void miPrehMat\_Click(object sender, RoutedEventArgs e)

{

if (!(frame.Content is pgPrehMat))

{

frame.Navigate(new pgPrehMat());

}

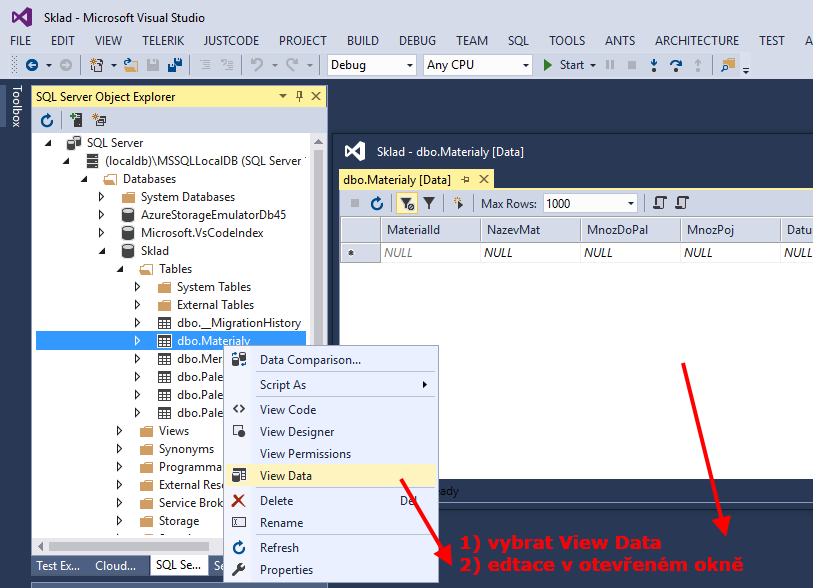
}

Po kliknutí na položku menu Přehled materiálu se spustí událost, která zkontroluje, zda ve Framu již není otevřen Přehled materiálů, pokud není, otevřeme do framu tento obsah.

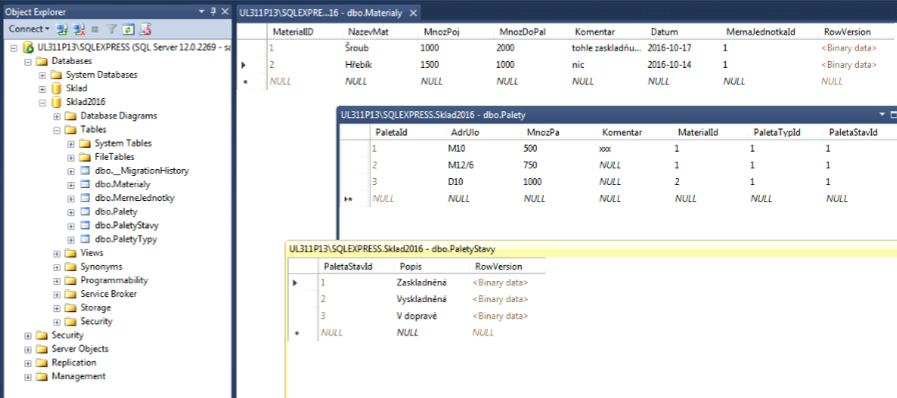
Po spuštění aplikace (tlačítko Start v hlavním pásu karet VisualStudio) a po kliknutí na ikonu Přehled materiálu se nám v prostředním okně otevře formulář pgPrehMat (který je momentálně prázdný).

## Naplnění databáze cvičnými daty

Otevřeme si databázi v SQL Server Object Inspectoru nebo v  SQL Server Management Studio a zde naplníme tabulky cvičnými daty. Pravým kliknutím na tabulku (cesta k tabulce je **Databases – Sklad (popř. Sklad2016) – Tables**) zvolíme View Data nebo **Edit top 200 rows**. Tento krok provedeme u tabulek **PaletyStavy, Materialy, Palety** (zachovat pořadí). Do tabulky PaletyStavy vložíme data **Zaskladněná, Vyskladněná, V Dopravě**. Sloupec ID u tabulek nevyplňujeme, číslo se generuje automaticky. Dále do tabulky materiály vložíme dva materiály **Šroub** a **Hřebík** a naplníme daty viz.Obr. 32, popř Obr. 33. Stejně tak podle obrázku naplníme tabulku palety. Veškerá vložená data se automaticky uloží do databáze.



Obr. Editace dat pomocí SQL Server Object Explorer



Obr. Testovací data v SQL Management Studio

# View Model a globální instance na SkladContext

**View Model** je mezivrstva (interface) mezi grafickým prostředím (GUI) a databází, pomocí které grafické prostředí komunikuje s databází (viz Obr. 3). Zavádí se z důvodu oddělení dat formuláře od ostatních a z toho vyplývající lepší přehlednosti, současně zde vznikají další možnosti – např. dědičnost z „pra-viewmodelu“ – tj. sdílené vlastnosti formulářů.

## Založení View Model

V **Solution Explorer** pod projektem **Sklad** založíme nový adresář **ViewModels**, do kterého budeme ukládat všechny tyto modely. Do adresáře vložíme první model **(Add – New Item – Class**) a pojmenujeme ho **pgPrehMatVM**.

Kód této třídy dle:

using DataEntity;

using PropertyChanged;

using System;

using System.Collections.Generic;

using System.Collections.ObjectModel;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Data;

namespace Sklad.ViewModels

{

[AddINotifyPropertyChangedInterface] //automatické obnovení grafiky po změně kódu

public class pgPrehMatVM

{

private ObservableCollection<Material> \_materialyCol; // vytvoření uzavřené vlastnosti

public ObservableCollection<Material> MaterialyCol

{ //vytvoření kolekce materiálu

get //vytvoření vlastního get set

{

if (\_materialyCol == null)//pokud \_materialyCol je prázdná (== porovnává hodnoty)

\_materialyCol = new ObservableCollection<Material>(Globals.\_context.Materialy); // naplní \_materialcol novou kolekcí z databáze(= nastavuje hodnotu)

return\_materialyCol;

}

set

{

\_materialyCol = value;

}

}

public Material MaterialSelected { get; set; } //vrácení vybraného materiálu4

public bool JeOznacenMaterial

{

get

{

return MaterialSelected != null;

}

}

}

}

Ve viewmodelu je definována kolekce (seznam) materiálů MaterialyCol. Při načítání je zkontrolováno, zda již jsou data naplněna (test na null) a pokud nejsou, dojde k jejich načtení z databáze. Do MaterialSelected bude ukládán aktuální materiál, který je označen (se kterým se pracuje). JeOznacen materiál je logická property (vlastnost), která vrací true, pokud je označen nějaký materiál.

## Nastavení odkazu na globální instanci na context

*Nyní je potřeba moci se odkazovat na globální instanci SkladContext, který spravuje všechny „databázové třídy“ respektive tabulky. Odkaz na ní je již připraven v třídě Globals, jedná se o řádku:*

*public static SkladContext \_context { get; set; }*

*slovíčko static určuje tzv. globální proměnnou platnou napříč všemi třídami.*

*Pozn.: V rámci našeho projektu zavádíme jediný globální context: to má své výhody, např. sdílený context umožňuje např. pracovat ve více oknech nad stejnými daty. Na druhou stranu, při využití tohoto přístupu množství dat, které se udržují v paměti „kyne“ a někdy jednoduše nechceme sdílená data pro dílčí formuláře. Pak je možné (a je to i doporučované pro větší projekty) např. ke každému formuláři zakládat novou instanci kontextové třidy (nový kontext)..*

V **SolutionExplorer** otevřeme kód (tzv. CodeBehind) pro **winHlavni** (View Code nebo klávesa F7). Poté provedeme zápis dle následujícího kódu, čímž naplníme hodnotu globální proměnné Globals.\_context pomocí vytvoření nového Contextu – tato řádka ve skutečnosti připojí databázi. Pomocí Globals.\_context budeme komunikovat s databází. Také vytvoříme globální odkaz na rodičovský formulář.

public partial class winHlavni : Window

{

public winHlavni()

{

Globals.\_context = new SkladContext();

Globals.\_parent = this;

InitializeComponent();

}

}

Dále si otevřeme kód pgPrehMat (F7) a nastavíme odkazy na tento globální context:

public partial class pgPrehMat : Page

{

public pgPrehMatVM \_datacontext;

private SkladContext \_context = Globals.\_context;

public pgPrehMat()

{

InitializeComponent();

\_datacontext = new pgPrehMatVM();

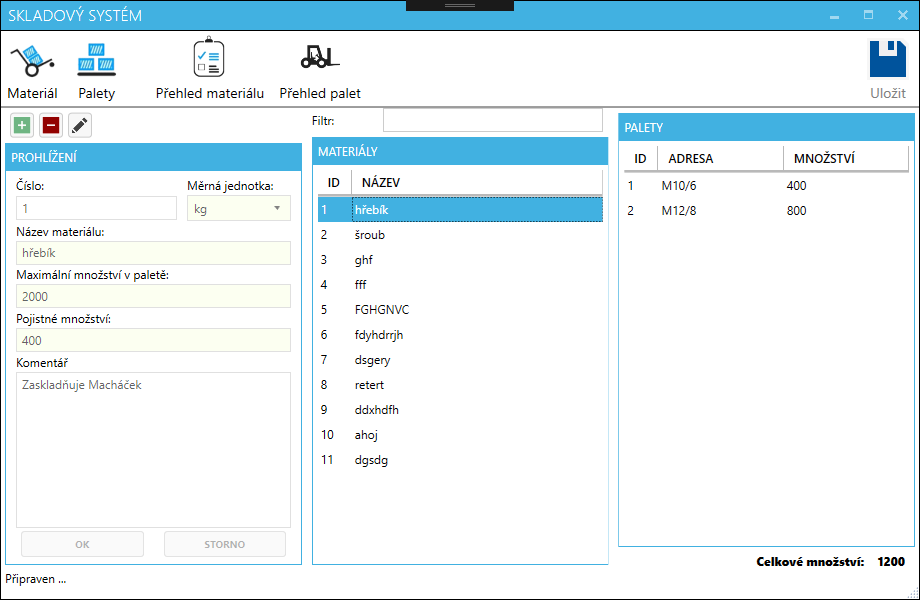
this.DataContext = \_datacontext;

}

Ve skutečnosti bychom mohli všude používat Globals.\_context – vzhledem k pohodlnosti jsme provedli „jakési přejmenování“ na \_context. Důležitým zápisem je pak spojení formuláře s příslušným viewmodelem. To se provede pomocí zavedení proměnné \_datacontext. Do ní se naplní nový viewmodel (často se používá i globální viewmodel, kde se volá konstruktér viewmodelu jen jednou – formuláře si pak neustále „pamatují“ poslední stav – tomuto přístupu se říká SingleTon). Pak je nutné říci, že viewmodel je tzv. datacontext formuláře. Tato syntaxe bude na všech formulářích obdobná.

# Grafická úprava formuláře Přehled materiálu I.

V této kapitole vytvoříme grafické prostředí formuláře Přehled materiálu včetně napojení na data.



Obr. Finální grafická podobá pgPrehMat (spodní část formuláře - page)

## Vytvoření seznamu materiálů

*Začneme tvorbou seznamu materiálů v prostředním sloupci formuláře. Obdélník s titulkem Materiály je tzv. GroupBox (ten se využívá pro grafické oddělení skupin komponent - controls), Uvnitř je pak mřížka – datagrid.*

Začneme vlastním Page: Otevřeme si XAML kód pro formulář **pgPrehMat**. Šířku a výšku celého formuláře nastavíme na **Height =“600“** a **Width=“800“.** Dále si celý formulář pomocí grid rozdělíme na tři sloupce (viz. 8.4) a jejich šířku nastavíme na stejný poměr (\*).

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

</Grid>

Dále si uvnitř <Grid> vytvoříme **<GroupBox>.** Do tohoto GroupBoxu vložíme hlavičku a vnoříme ho do prostředního sloupce, tedy:

<GroupBox Header="Materiály" Grid.Column="1">

</GroupBox>

Do GroupBoxu dále vložíme **<DataGrid>,** což je mřížka, která obsahuje data:

<DataGrid Name="dgMaterialy" ItemsSource="{Binding MaterialyCol}" HeadersVisibility="Column" SelectionMode="Single"

AutoGenerateColumns="False"  
IsReadOnly="True" SelectedValue="{Binding MaterialSelected}">

Tento zápis vytvoří datovou mřížku s názvem **dgMaterialy**, připojí se na zdroj položek **MaterialyCol** (již připravené v prehMatVM – všechna data se čerpají z DataContextu, který jsme nastavili na ViewModel). DataGrid se nastaví pouze pro čtení (IsReadOnly) a zamezí se automatickému generování sloupců a výpisu všech hodnot v databázi (AutoGenerateColumn – zkuste si nastavit true). Dále musíme vložit pouze ty hodnoty z databáze, které potřebujeme vypisovat v tomto datagridu, v tomto případě ID materiálu a jeho název. Do **DataGrid** vložíme další vlastnost **<DataGrid.Columns>,** do kterého nadefinujeme jednotlivé sloupce:

<DataGrid.Columns>

<DataGridTextColumn Header="ID" Width="Auto" Binding="{Binding MaterialId}"/>

<DataGridTextColumn Header="Název" Width="\*" Binding="{Binding NazevMat}"/>

</DataGrid.Columns>

Tímto jsme vytvořili sloupce s hlavičkou ID a Název, nastavili jsme jim šířku, která si přizpůsobí obsahu, a připojili jsme k nim data **MaterialID** a **NazevMat** z tabulky materiály.

Celý XAML kód je pak:

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<GroupBox Header="Materiály" Grid.Column="1">

<DataGrid Name="dgMaterialy" ItemsSource="{Binding MaterialyCol}" HeadersVisibility="Column" SelectionMode="Single"

AutoGenerateColumns="False" IsReadOnly="True" SelectedValue="{Binding MaterialSelected}">

<DataGrid.Columns>

<DataGridTextColumn Header="ID" Width="Auto" Binding="{Binding MaterialId}"/>

<DataGridTextColumn Header="Název" Width="\*" Binding="{Binding NazevMat}"/>

</DataGrid.Columns>

</DataGrid>

</GroupBox>

</Grid>

## Vytvoření seznamu palet s vybraným materiálem

V další části formuláře se bude nacházet tabulka, kde se vypíší palety, ve kterých se nachází materiál, který vybereme z předchozího seznamu. Celý předchozí **GroupBox**, ve kterém jsme tvořili seznam materiálů, překopírujeme do stejného **gridu** a provedeme vyznačené změny. Přejmenujeme hlavičku, umístíme grid do třetího sloupce formuláře, přejmenujeme grid, změníme zdroj položek, smažeme vybranou hodnotu, přejmenujeme názvy sloupců a jejich zdroje a přidáme sloupec. Zejména při zadávání zdrojů **(Binding)** je potřeba dodržovat velká a malá písmena uložená v databázi (case sensitive). Výsledný XAML kód bude:

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<GroupBox Header="Materiály" Grid.Column="1">

<DataGrid Name="dgMaterialy" ItemsSource="{Binding MaterialyCol}" HeadersVisibility="Column" SelectionMode="Single"

AutoGenerateColumns="False" IsReadOnly="True" SelectedValue="{Binding MaterialSelected}">

<DataGrid.Columns>

<DataGridTextColumn Header="ID" Width="Auto" Binding="{Binding MaterialId}"/>

<DataGridTextColumn Header="Název" Width="\*" Binding="{Binding NazevMat}"/>

</DataGrid.Columns>

</DataGrid>

</GroupBox>

<GroupBox Header="Palety" Grid.Column="2">

<DataGrid Name="dgMaterialyPalety" ItemsSource="{Binding MaterialSelected.Palety}" HeadersVisibility="Column" SelectionMode="Single"

AutoGenerateColumns="False" IsReadOnly="True">

<DataGrid.Columns>

<DataGridTextColumn Header="ID" Width="Auto" Binding="{Binding PaletaId}"/>

<DataGridTextColumn Header="Adresa" Width="\*" Binding="{Binding AdrUlo}"/>

<DataGridTextColumn Header="Množství" Width="Auto" Binding="{Binding MnozPa}"/>

</DataGrid.Columns>

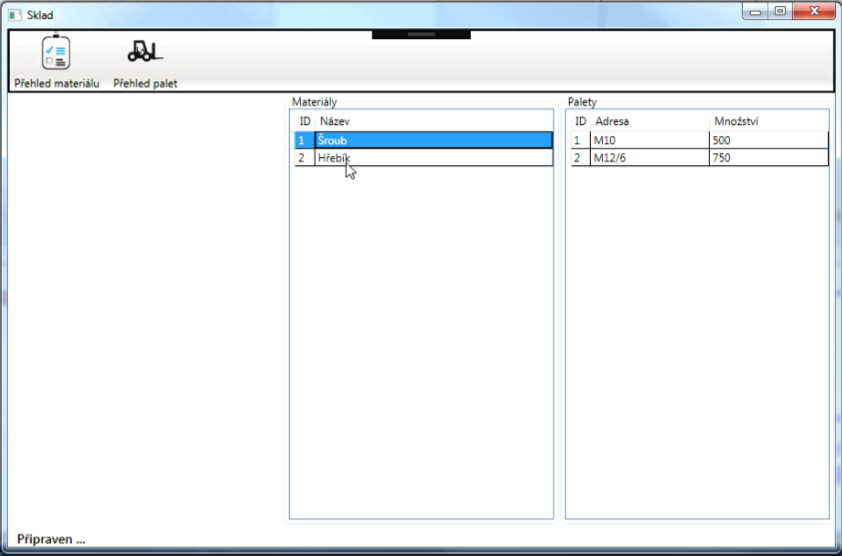
</DataGrid>

</GroupBox>

</Grid>

*Pozn.: Povšimněte si zejména vlastnosti ItemSource u nového DataGridu – ta je nastavena na hodnotu MaterialSelected.Palety – to znamená – jako zdroj pro tento datagrid bude seznam všech palet, ve kterých je právě označený materiál (pro palety není nutné vytvářet speciální property ve viewmodelu).*

Průběžná aplikace by pak měla vypadat dle Obr. 35



Obr. Aplikace po provedených změnách

*Seznam palet se mění dle označeného materiálu. Takovému formuláři říkáme formulář 1:N.*

## Počítání celkového množství materiálu

Dále do pravého sloupce umístíme pole, které bude počítat celkové množství určitého materiálu ve všech paletách.

*Trocha teorie:*

*Vlastní výpočet (který se také dal provést například voláním SQL dotazu) provedeme pomocí tzv. extenzních metod a za použití tzv. anonymních funkcí. Všechny seznamy, které používáme, jsou typu ObservableCollection. ObservableCollection je potomkem třídy List – cokoliv, co je tohoto typu, nebo typu potomek této třídy je tzv. generický seznam (generický – může pracovat s libovolným datovým typem – v našem případě Material). V rámci gerických listů, je možné nad nimi provádět různé akce (příkaz Where, Count, Sum atd.). Abychom si vše zjednodušili, využívá se tzv. anonymních funkcí (zde je to volání x=>x.neco v následujícím kódu. Vše je vysvětleno v Dodatku B.*

Otevřeme si třídu **Material (v projektu DataEntity)** a zapíšeme novou počítanou (jen get bez set) property (vlastnost):

public int CelkoveMnozstvi

{

get

{

if(Palety == null)

{

return 0;

}

else

{

return Palety.Sum(x => x.MnozPa);

}

}

}

Tato vlastnost nejprve zkontroluje, zda pole palety není prázdné. Pokud ano, vrátí hodnotu 0, pokud není prázdné, vrátí hodnotu agregační funkce **SUM**.

Další možný zápis je:

public int CelkoveMnozstvi

{

get

{

return Palety==null ? 0 : Palety.Sum(x => x.MnozPa);

}

}

který funguje stejně jako předchozí. **Otazník je tzv. ternární operátor a ve spojení s :**  nahrazuje funkci **If – Then**.

Od verze jazyka C# 6 je možno vše ještě zjednodušit (pro properties jen s get bez set):

public int CelkoveMnozstvi => Palety == null ? 0 : Palety.Sum(x => x.MnozPa);

Výsledek výpočtu celkového množství chceme zobrazit v posledním sloupci pod DataGrid dgMaterialyPalety v TextBoxu. Otevřeme XAML kód formuláře **PrehMat.** Budeme potřebovat již existující GroupBox uzavřít do nového Gridu, který bude obsahovat DataGrid a nový TextBox. Za definice **GroupBoxu Palety** vložíme nový **<Grid>.** V XAML kódu klikneme za **GroupBox palety** a kódujeme nový Grid:

<Grid Grid.Column="2">

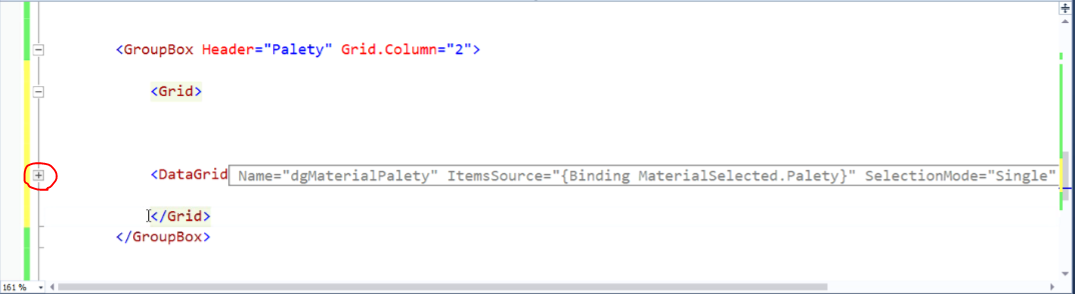
<Grid.RowDefinitions>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

Ukončení tagu **</Grid>** umístíme až za konec tagu **<DataGrid> v GroupBoxu palety**. V XAML editoru je užitečná možnost zabalení části kódu, zejména ve fázích, kdy pracujeme s velmi dlouhým kódem. Na Obr. 36 vidíme označené políčko, kterým jsme zabalili celou část kódu v <DataGrid>. Tuto část je možné kdykoliv znovu rozbalit a nebo zabalení použít na jiné části kódu. Na obrázku rovněž vidíme, kam je potřebné umístit tag <Grid> </Grid>.



Obr. Zabalení/rozbalení části kódu

Do tohoto gridu vložíme:

<WrapPanel Grid.Row="1" Margin="0,0,10,0" HorizontalAlignment="Right">

<TextBlock Text="Celkové množství: " />

<TextBlock Text="{Binding MaterialSelected.CelkoveMnozstvi}" />

</WrapPanel>

*Pozn.: WrapPanel a StackPanel jsou další možnosti řízení layoutu formuláře (kromě Grid). Pokud chcete, aby se komponenty přidávaly za sebe, zvolte WrapPanel, pokud pod sebe, zvolte StackPanel. (je možné zařídit, aby se WrapPanel choval jako StackPanel a opačně)*

Finální část kódu, včetně DataGrid, je pak:

<Grid Grid.Column="2">

<Grid.RowDefinitions>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<GroupBox Header="Palety" >

<DataGrid Name="dgMaterialyPalety" ItemsSource="{Binding MaterialSelected.Palety}" HeadersVisibility="Column" SelectionMode="Single"

AutoGenerateColumns="False" IsReadOnly="True">

<DataGrid.Columns>

<DataGridTextColumn Header="ID" Width="Auto" Binding="{Binding PaletaId}"/>

<DataGridTextColumn Header="Adresa" Width="\*" Binding="{Binding AdrUlo}"/>

<DataGridTextColumn Header="Množství" Width="\*" Binding="{Binding MnozPa}"/>

</DataGrid.Columns>

</DataGrid>

</GroupBox>

<WrapPanel Grid.Row="1" Margin="0,0,10,0" HorizontalAlignment="Right">

<TextBlock Text="Celkové množství: " />

<TextBlock Text="{Binding MaterialSelected.CelkoveMnozstvi}" />

</WrapPanel>

</Grid>

V aplikaci se pak na formuláři Přehled materiálů ve třetím sloupci dole nachází vytvořené počítadlo množství materiálu.

## Vytvoření filtru materiálů

Nyní si vytvoříme v prostředním poli filtr, pomocí kterého budeme vyhledávat materiály. Nejprve si otevřeme z adresáře **ViewModels pgPrehMatVM**. Zde provedeme zápis kódu:

namespace Sklad.ViewModels

{

[AddINotifyPropertyChangedInterface]

public class pgPrehMatVM

{

public pgPrehMatVM ()

{

CollectionView view = (CollectionView)CollectionViewSource.GetDefaultView(MaterialyCol);

view.Filter = UserFilter;

}

private bool UserFilter (object item)

{

if (String.IsNullOrEmpty(FiltrMaterialNazev))

return true;

else

return ((item as Material).NazevMat.IndexOf(FiltrMaterialNazev, StringComparison.OrdinalIgnoreCase) >= 0);

}

private string \_filtrMaterialNazev;

public string FiltrMaterialNazev

{

get

{

return \_filtrMaterialNazev;

}

set

{

\_filtrMaterialNazev = value;

CollectionViewSource.GetDefaultView(MaterialyCol).Refresh();

}

}

}

}

Jelikož je CollectionView z nového namespace, přidáme pomocí ikony se žárovkou, tj. **using System.Windows.Data**.

*Je zde vytořeno metoda UserFilter, která se stará o testování každé položky (Materiálu) proti kritériu (v našem případě nachází se kdekoliv v textu hledaný podtext – IndexOf… s ignorování malých a velkých písmen). Tato metoda je předána do vlastnosti Filter u nově vzniknuvšího CollectionView. CollectionView může „obalit“ libovolný seznam a umí navíc držet informaci o řazení, filtru a aktuálním vybraném záznamu – to obyčený list nesvede.*

Nyní potřebujeme, aby se v prostředním sloupci, kde se nachází seznam materiálů, objevil nad nimi grafický prvek pro filtrování (Obr. 37). V XAML kódu formuláře **pgPrehMat** založíme nový grid. Zabalíme si kód pro **dgMaterialy** a vložíme tag pro grid. Začátek **<Grid>** vložíme mezi **GroupBox Materialy** a zabalenou **dgMaterialy**, konec **</¨Grid>** vložíme za **dgMaterialy**. Poté vytvoříme v grafickém rozhraní mřížku o dvou sloupcích (u filtru bude text a textové pole) a dvou řádcích. Nastavíme následující rozměry:

<Grid Grid.Column="1">

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="3\*"/>

</Grid.ColumnDefinitions>

<GroupBox …………………..

</GroupBox>

</Grid>

GroupBox materiály posuneme na řádku gridu s indexem 1 (druhou řádku) a ještě přidáme informace, že má zabírat dva sloupce (ColumnSpan – zkuste nejprve bez této vlastnosti) :

<GroupBox Header="Materiály" Grid.Row="1" Grid.ColumnSpan="2">

Nad **GroupBox Materialy** poté vložíme grafiku k filtraci:

<TextBlock Text="Filtr:"/>

<TextBox Text="{Binding FiltrMaterialNazev, UpdateSourceTrigger=PropertyChanged}" Grid.Column="1" Margin="0,0,10,0"/>

*Zde je důležité u vlastnosti spojení na viewmodel – Binding uvést UpdateSourceTrigger=PropertyChanged – to totiž zajistí, že se bude spojená vlastnost FiltrMaterialNazev měnit neustále při jakékoliv změně textu. Standardní vlastnost je LostFocus,tj. opustil jsem komponentu.*

Tímto kódem tedy vytvoříme textblock s hodnotou Filtr a TextBox, do kterého můžeme psát, který je datově napojen na FiltrMaterialNazev a výsledky filtrování se mění okamžitě po změně (během psaní).

Celý XAML kód pro „prostřední sloupec“ s filtrem je pak:

<Grid Grid.Column="1">

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="3\*"/>

</Grid.ColumnDefinitions>

<TextBlock Text="Filtr:"/>

<TextBox Text="{Binding FiltrMaterialNazev, UpdateSourceTrigger=PropertyChanged}" Grid.Column="1" Margin="0,0,10,0"/>

<GroupBox Header="Materiály" Grid.Row="1" Grid.ColumnSpan="2">

<DataGrid Name="dgMaterialy" ItemsSource="{Binding MaterialyCol}" HeadersVisibility="Column" SelectionMode="Single"

AutoGenerateColumns="False" IsReadOnly="True" SelectedValue="{Binding MaterialSelected}">

<DataGrid.Columns>

<DataGridTextColumn Header="ID" Width="Auto" Binding="{Binding MaterialId}"/>

<DataGridTextColumn Header="Název" Width="\*" Binding="{Binding NazevMat}"/>

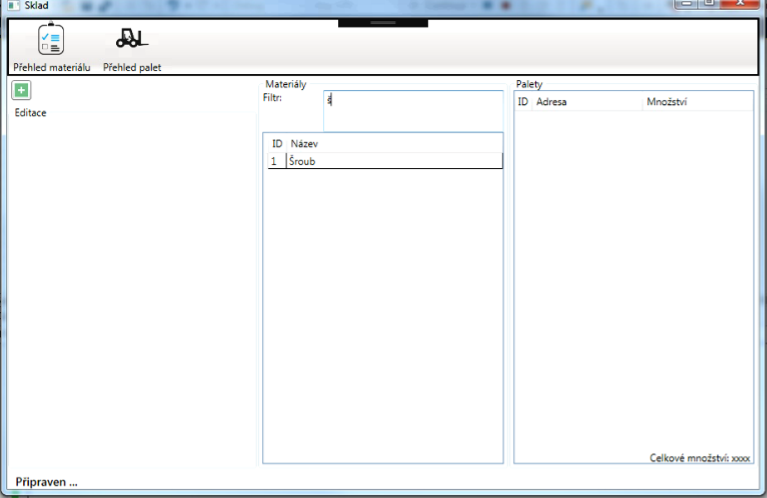
</DataGrid.Columns>

</DataGrid>

</GroupBox>

</Grid>

Ve vytvořené aplikaci pak máme filtr viz Obr. 37, pomocí kterého můžeme vybírat pouze data, ve kterých se nachází zadané znaky (např. pří zadání Š se zobrazí pouze šrouby)



Obr. Filtr materiálu

## Vytvoření stavů

Stavy slouží k tomu, abychom rozdělili práci s formulářem do několika stavů, kdy v každém z nich bude možné provádět jen určité operace: prohlížení, přidávání nového záznamu, editace. Otevřeme si **pgPrehMatVM** a zapíšeme:

public enum Rezimy { Prohlizeni, Pridavani, Editace }

private Rezimy \_aktualniRezim = Rezimy.Prohlizeni;

public Rezimy AktualniRezim {

get

{

return \_aktualniRezim;

}

set

{

\_aktualniRezim = value;

//následující řádka - příprava na to, aby nebylo možno používat menu při editaci

//Globals.\_parentvm.LzePouzivatMenu = \_aktualniRezim == Rezimy.Prohlizeni;

}

}

public string AktualniRezimString

{

get

{

switch (AktualniRezim)

{

case Rezimy.Pridavani: return "Přidávání";

case Rezimy.Editace: return "Editace";

}

return "Prohlížení";

}

}

public bool RezimPridavaniNeboEditace

{

get

{

return AktualniRezim != Rezimy.Prohlizeni;

}

}

Formulář může být ve třech stavech (referenční obázek - Obr. 34):

1. Prohlížení – není možné využívat levý panel s textovými boxy (nelze editovat aktuální záznam), je však možné kliknout na tlačítka +,-, editace
2. Přidávání nebo Eidtace – je možné využívat textové boxy pro editaci záznamu, je možné kliknout na potvrzovací nebo strono tlačítko. Není však možné klikat na tlačítka +,-,editace (nebo dokonce využívat hlavní menu). Režim přidávání nebo editace se pak liší v reakci na potvrzení záznamu – tlačítko OK.

Povšimněte si, že se stavem je možno pracovat na úrovni výčtového typu (AktualniRezim). Dle hodnoty tohoto výčtového typu je pak vracen i textový popis, který se objeví na formuláři (AktualniRezimString).

## Vytvoření prohlížeče

Posledním panelem v tomto formuláři je první panel prohlížení. Do hlavního **gridu** za **</Grid.ColumnDefinition>** vložíme nový **<GroupBox>** a do něj **<Grid>** pro první panel. Kliknutím do grafického editoru vytvoříme mřížku. XAML poté upravíme dle:

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<WrapPanel Margin="10,0,0,0">

<Button Padding="3" Grid.Column="1" Name="btnAddMat" Width="24" Height="24" Margin="0,5,5,0" 23 >

<Image Source="..\Resources\Icons\Add.png"></Image>

</Button>

<WrapPanel IsEnabled="{Binding JeOznacenMaterial}">

<Button Padding="3" Grid.Column="1" Name="btnDelMat" Width="24" Height="24" Margin="0,5,5,0" >

<Image Source="..\Resources\Icons\Delete.png"></Image>

</Button>

<Button Padding="3" x:Name="btnEditMat" Width="24" Height="24" Margin="0,5,5,0" >

<Image Source="..\Resources\Icons\Edit.png"/>

</Button>

</WrapPanel>

</WrapPanel>

<GroupBox Header="{Binding AktualniRezimString}" Grid.Row="1" IsEnabled="{Binding RezimPridavaniNeboEditace}" Name="gbLeva">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="3\*"/>

<ColumnDefinition Width="2\*"/>

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<TextBlock Text="Číslo:"/>

<TextBlock Text="Měrná jednotka:" Grid.Column="1"/>

<TextBox Text="{Binding MaterialSelected.MaterialId}" IsReadOnly="True" Grid.Row="1"/>

<ComboBox ItemsSource="{Binding MerneJednotkyCol}" SelectedValue="{Binding MaterialSelected.MernaJednotka, NotifyOnValidationError=True, ValidatesOnDataErrors=True}" DisplayMemberPath="Popis" Grid.Row="1" Grid.Column="1" Background="{DynamicResource PovinnePole}"/>

<TextBlock Text="Název materiálu:" Grid.Row="2"/>

<TextBox Text="{Binding MaterialSelected.NazevMat, UpdateSourceTrigger=PropertyChanged, NotifyOnValidationError=True, ValidatesOnDataErrors=True}" Grid.Row="3" Grid.ColumnSpan="2" Background="{DynamicResource PovinnePole}"/>

<TextBlock Text="Maximální množství v paletě:" Grid.Row="4" Grid.ColumnSpan="2"/>

<TextBox Text="{Binding MaterialSelected.MnozDoPal, UpdateSourceTrigger=PropertyChanged, ValidatesOnDataErrors=True, NotifyOnValidationError=True}" Grid.Row="5" Grid.ColumnSpan="2" Background="{DynamicResource PovinnePole}"/>

<TextBlock Text="Pojistné množství:" Grid.Row="6" Grid.ColumnSpan="2"/>

<TextBox Text="{Binding MaterialSelected.MnozPoj, ValidatesOnDataErrors=True, NotifyOnValidationError=True, UpdateSourceTrigger=PropertyChanged}" Grid.Row="7" Grid.ColumnSpan="2" Background="{DynamicResource PovinnePole}"/>

<TextBlock Text="Komentář" Grid.Row="8"/>

<TextBox Grid.Row="9" Grid.ColumnSpan="2" ScrollViewer.HorizontalScrollBarVisibility="Auto"

ScrollViewer.VerticalScrollBarVisibility="Auto"

ScrollViewer.CanContentScroll="True" AcceptsReturn="True" TextWrapping="Wrap" SpellCheck.IsEnabled="True" Language="cs-CZ" VerticalAlignment="Stretch" Text="{Binding MaterialSelected.Komentar, UpdateSourceTrigger=PropertyChanged}" />

</Grid>

</GroupBox>

</Grid>

Nyní bylo vloženo mnoho XAML kódu. Tento kód vytvoří celý levý panel. Který už bude dynamicky reagovat na stav formuláře a dle toho bude upravovat chování grafické vrstvy.

*Na tomto místě bychom se měli zastavit u vlastností Bindingu – spojení na datacontext (v našem případě viewodel). Za slovo Binding můžeme napsat nějakou property z viewmodelu, např. JeOznacenMaterial, MaterialCol, atp. Tak, kde je to relevantní (typ třída, záznam, …), tam je možno jeětě přistupvat k dalším podobjektům vlatsnosti definované ve viewmodulu, např. MaterialSelected.Palety, MaterialSelected.PojistneMnozstvi. Dále je možno nastavovat další vlastnosti spojení:*

*UpdateSourceTrigger=PropertyChanged – ke změně v datech dochází neustále (celkové množství je dopočítáváno okamžitě, pokud je v textboxu dílčího množství měněno číslo, a to i např. pokud je číslo dopisováno, pokud bych do takového textboxu dopisoval např. číslo 50. Dojde k výpočtu všech souvisejících polí pro 5 a pak pro 50. Tyto vlatsnost máme nastavenou u textboxu s filtrem.*

*UpdateSourceTrigger=LostFocus – k přepočtům dochází až po opuštění komponenty*

*ValidatesOnDataError=True – zapíná kontrolu chyb. Probíhá volání příslušných metod, které jsou implementoványrozhraním INotifyError (to máme vytvořen u všech databázových tříd)*

*NotifyOnDataError=True – zapíná grafické zobrazování chybových hlášek, v případě, kdy validace neproběhne v pořádku.*

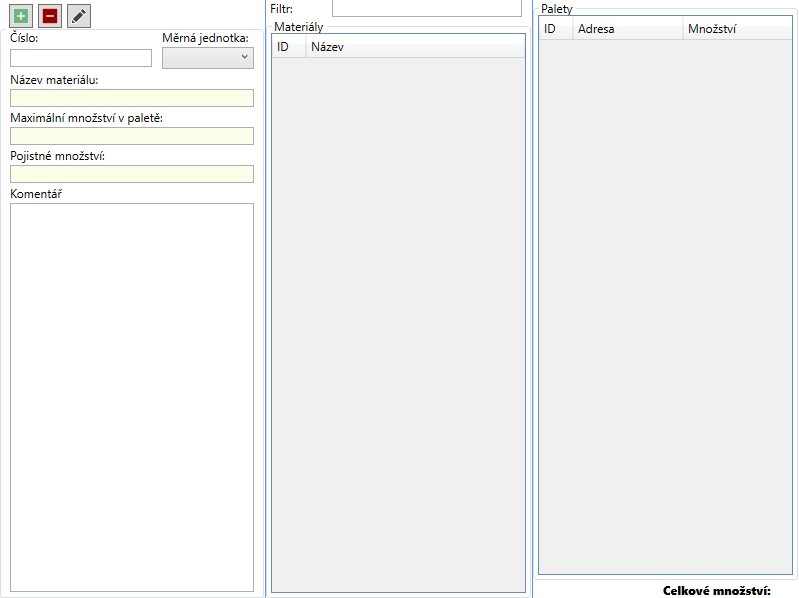
**Důležité!!!!: Formulář ještě bude hlásit chyby – nemáme vytvořena zdrojová data pro Combobox měrné jednotky.**

**Často se vám (alespoň ze začátku) stane, že uvedete chybný nebo neexistující název propojení, např. místo MaterialId, uvedete Materialid. Na tomto formuláři je tento problém demontrován. Combobox měrné jednotky nefunguje, neboť není připravena kolekce, která se jmenuje MerneJednotkyCol. V XAMLu je tento Binding připraven. Tuto chybu snadno odhalíte buď v okně Immediate Window nebo Output, jedná se o chybovou hlášku:**

*System.Windows.Data Error: 40 : BindingExpression path error: 'MerneJednotkyCol' property not found on 'object' ''pgPrehMatVM' (HashCode=11236435)'. BindingExpression:Path=MerneJednotkyCol; DataItem='pgPrehMatVM' (HashCode=11236435); target element is 'ComboBox' (Name=''); target property is 'ItemsSource' (type 'IEnumerable')*

**Okno Immediate Window (nebo Output) používejte při ladění!**

Page nyní vypadá takto - viz. Obr. 38. Vytvořili jsme tlačítka pro přidání, editaci a úpravu materiálu, které se zpřístupní, pouze pokud je vybrán materiál z prostředního sloupce. Dále jsme v prohlížeči vytvořili textboxy pro editaci nebo prohlížení názvu materiálu a maximální množství v paletě, pojistného množství a komentáře které je možné upravovat. Dále je na page i číslo materiálu (jen pro čtení) a připravený ComboBox pro měrné jednotky, který zatím nefunguje.



Obr. Grafika PrehMat

## Vyzkoušní režimu editace

Formulář nyní „umí“ být jen v režimu prohlížení. Kód k tlačítkům realizujeme později, nyní vyzkoušíme jen přepnout formulář do režimu Editace.

Na formuláři pak klikneme na tlačítko pro editaci (tužka). Otevřeme si seznam událostí a poklepeme na událost Click. Do kódu události pak napíšeme:

private void btnEditMat\_Click(object sender, RoutedEventArgs e)

{

\_datacontext.AktualniRezim = pgPrehMatVM.Rezimy.Editace;

}

Nyní je možné po označení materiálu použít tlačítko Editace a přednout formulář do tohoto režimu.

Povšimněte si:

* Díky vlastnosti spojení UpdateSourceTrigger=PropertyChanged u textboxu s názvem materiálu dochází automaticky ke změně názvu i v tabulce
* Po vyplnění nevalidních údajů (název s nulovou délou, množství je text nebo záprné, atp.) dochází ke kontrole a komponenta se červeně orámuje. To zařídily vlastnosti propojení ValidatesOnDataErrors=True, NotifyOnValidationError=True a implementace rozhraní IDataErrorInfo ve třídě basemodel. Jsou reflektovány validace z datavého modelu vytvořené pomocí datových anotací.
* Zatím se nikde neobjevují chybové hlášky při nevalidních datech – není definován styl. To napravíme v další kapitole

# Styl aplikace

V této kapitole si ukážeme možnosti nastavení jednotného grafického stylu pro celou aplikaci, nutná globální nastavení a možnosti jejího použití. Možností pro vytvoření „lepšího“ grafického stylu je celá řada – počínaje vlastním nastylování, přes jednoduché stylovací balíky až po velká komerční řešení. My půjdeme cestou jednoduchého stylovacího balíku. V našem případě se jmenuje MahAppsMetro.

Nejprve nainstalujeme další NuGet balíček podle 1.4 **MahAppsMetro**. Toto rozšíření nainstalujeme pouze do projektu **Sklad**. Po jeho instalaci se zobrazí nové okno, které nezavíráme, nýbrž ho využijeme pro nastavení a také slouží jako podrobný návod pro veškeré možnosti a použití aplikace.

V otevřeném okně s návodem si nalezneme sekci **Modifying the XAML file**. Poté si otevřeme XAML kód pro **winHlavni**. Na úplném začátku XAML kódu přepíšeme **Window: na Controls: MetroWindow** podle:

<Controls:MetroWindow x:Class="Sklad.Presentation.winHlavni"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Sklad.Presentation"

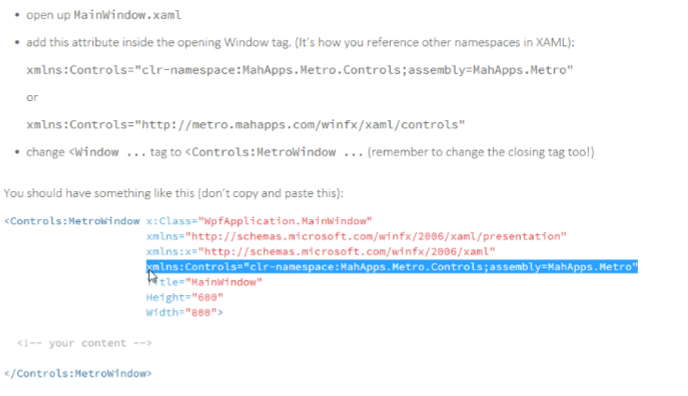
xmlns:Controls="clr-namespace:MahApps.Metro.Controls;assembly=MahApps.Metro"

mc:Ignorable="d"

Title="Skladový systém" Height="600" Width="920">

Následně zkontrolujeme, zda se na posledním řádku XAML kódu nachází uzavření tagu **</controls:MetroWindow>.** Dále přidáme do hlavičky XAML winHlavni „namespace“ podle návodu viz. Obr. 39 (a viz výše):

xmlns:Controls="clr-namespace:MahApps.Metro.Controls;assembly=MahApps.Metro"

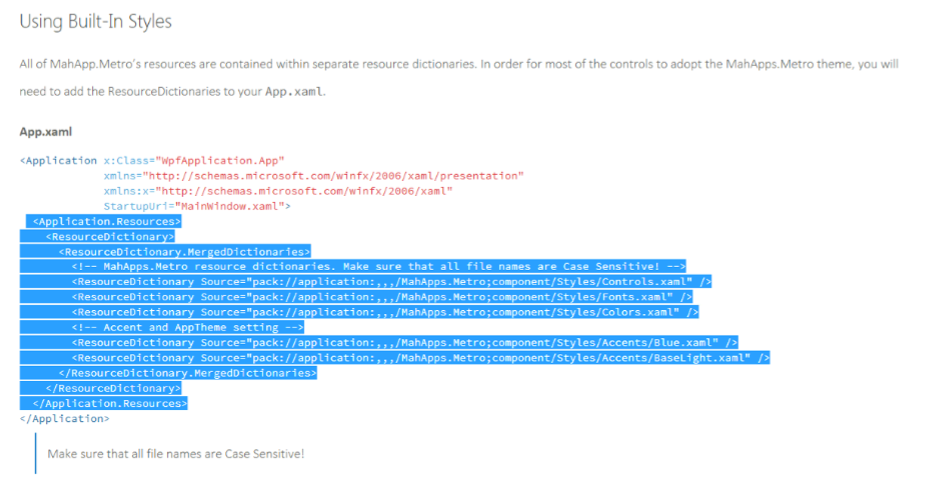


Obr. Namespace pro XAML kód

Poté musíme upravit kód v CodeBehind pro **winHlavni**, který otevřeme pomocí F7. Zde přepíšeme v definici třídy předka z Window na MetroWindow:

public partial class winHlavni : MetroWindow

a pomocí ikonky žárovky přidáme příslušný namespace. Následně otevřeme v SolutionExplorer soubor **App.xaml** a do sekce **<Application.Resources>** překopírujeme kód z nápovědy aplikace viz. Obr. 40.



Obr. Kopírovaný kód z nápovědy aplikace

Výsledná podoba části kódu pro **App.xaml** je tedy:

<Application.Resources>

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

<!-- MahApps.Metro resource dictionaries. Make sure that all file names are Case Sensitive! -->

<ResourceDictionary Source="pack://application:,,,/MahApps.Metro;component/Styles/Controls.xaml" />

<ResourceDictionary Source="pack://application:,,,/MahApps.Metro;component/Styles/Fonts.xaml" />

<ResourceDictionary Source="pack://application:,,,/MahApps.Metro;component/Styles/Colors.xaml" />

<ResourceDictionary Source="pack://application:,,,/MahApps.Metro;component/Styles/Accents/Blue.xaml" />

<ResourceDictionary Source="pack://application:,,,/MahApps.Metro;component/Styles/Accents/BaseLight.xaml" />

<ResourceDictionary Source="Dictionaries/StylyProZadavani.xaml" />

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

</Application.Resources>

Tímto krokem jsou přidány globální styly pro všechny komponenty z MahAppsMetro. Tohoto kroku využíváme pro přidání našeho vlastního stylu, který pro komponenty jako TextBox, ComboBox a TextBlock definuje výchozí rozložení (Margin, FontSize, …) – styl Dictionaries/StylyProZadavani.xaml. (tyto styly byly přidány přidáním balíku z FTP).

Abychom mohli styly z Dictionaries/StylyProZadavani.xaml mohli využít v rámci naší page, je třeba přidat k definici Page následující:

<Page x:Class="Sklad.Presentation.pgPrehMat"

…………….

<Page.Resources>

<Style TargetType="TextBlock" BasedOn="{StaticResource TextBlockStyleProZadavani}"/>

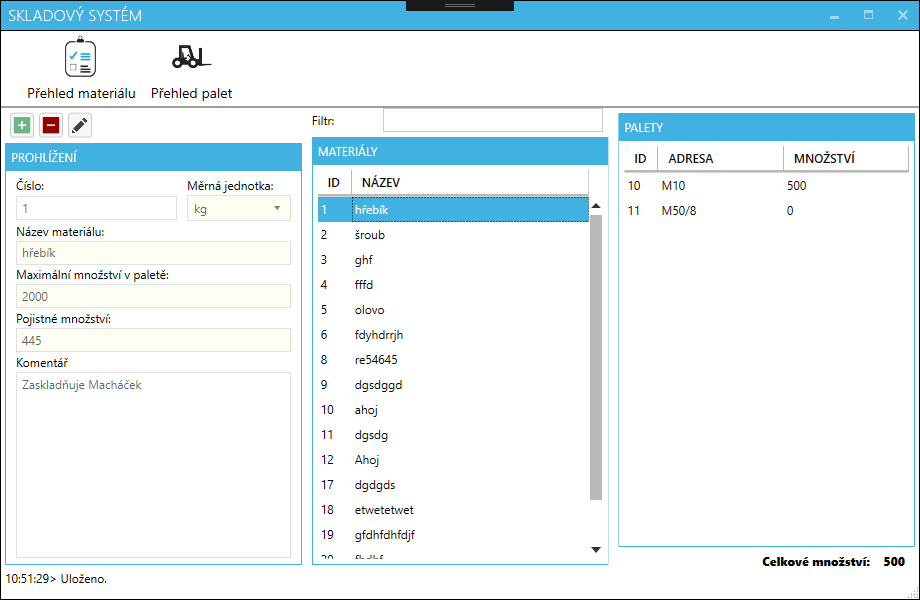
<Style TargetType="ComboBox" BasedOn="{StaticResource ComboBoxStyleProZadavani}"/>

<Style TargetType="TextBox" BasedOn="{StaticResource TextBoxStyleProZadavani}"/>

<Style TargetType="CheckBox" BasedOn="{StaticResource CheckBoxStyleProZadavani}"/>

</Page.Resources>

Tímto nám vznikne grafická podoba aplikace na Obr. 41. Podívejte se do Dictionaries/StylyProZadavani.xaml – povšimňete si, jak tento styl ovlivnil výslednou podobu page.



Obr. Aplikace s nastaveným stylem

**Vyzkoušejte: Validace pro textová pole fungují včetně zobrazování chybových hlášek. To zařídil nový styl.**Grafická úprava formuláře Přehled materiálu II.

V této kapitole vytvoříme zbývající komponenty formuláře Přehled materiálu.

## Nastavení rámečku okna formuláře

Do hlavičky XAML kódu **winHlavni** vložíme vlastnost **BorderThickness =“1“** a dále **GlowBrush =“Black“**, čímž vytvoříme kolem okna černý rámeček. Dále můžeme vložit vlastnost **ResizeMode=“CanResizeWithGrip“,** která do pravého dolního rohu okna vloží možnost roztahování okna, WindowTransitionsEnabled zruší animace a ještě můžeme řici, že okno má být na strtu uprostřed:

<Controls:MetroWindow x:Class="Sklad.Presentation.winHlavni"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Sklad.Presentation"

xmlns:Controls="clr-namespace:MahApps.Metro.Controls;assembly=MahApps.Metro"

mc:Ignorable="d"

BorderThickness="1"

GlowBrush="Black"

ResizeMode="CanResizeWithGrip"

WindowTransitionsEnabled="False"

WindowStartupLocation="CenterScreen"

Title="Skladový systém" Height="600" Width="920">

## Vytvoření dat pro ComboBox Měrná jednotka

ComboBox byl již vytvořen. Bohužel zatím nefunguje a pak si po spuštění programu zobrazíte Immediate Window (někde v Output), bude hlásit chyby propojení (Binding Errors).

***Pozn.: Immediate Window je VELMI důležitý pomocník pro ladění aplikace. Odhalí vám chyby propojení na datový kontext!***

Pro ComboBox je tedy nutné zajistit data. Veškerá data by měl obsahovat viewmodel. Vzhledem k tomu, že seznam s měrnými jednotkami budeme v budoucnu potřebovat i v jiných Pages, vytvoříme tato data společná pro všechny viewmodely těchto Pages:

Data pro combobox tedy připravíme v novém rodičovském (bázickém) viewmodelu, kterým tak může být společný pro všechny další viewmodely. V **SolutionExplorer** pod adresářem **ViewModels** založíme nový **ViewModel (Add – New Item),** který pojmenujeme **BaseVM**. Tento ViewModel bude sloužit jako prapředek pro ostatní ViewModels (můžeme z něho tedy dědit vlastnosti, které pak nasdílíme ve více ViewModels). Otevřeme si **BaseVM** a zapíšeme (kód můžeme zkopírovat z **pgPrehMatVM** a poté upravit):

using DataEntity;

using PropertyChanged;

using System;

using System.Collections.Generic;

using System.Collections.ObjectModel;

using System.ComponentModel;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

namespace Sklad.ViewModels

{

[AddINotifyPropertyChangedInterface]

public class BaseVM : BaseModel

{

private ObservableCollection<MernaJednotka> \_merneJednotkyCol;

public ObservableCollection<MernaJednotka> MerneJednotkyCol

{

get

{

if (\_merneJednotkyCol == null)

\_merneJednotkyCol = new ObservableCollection<MernaJednotka>(Globals.\_context.MerneJednotky);

return \_merneJednotkyCol;

}

set

{

\_merneJednotkyCol = value;

}

}

}

}

Tato bázická třída dědí bazickou třídu, kterou jsme připravili v projektu DataEntity. To děláme proto, abychom mohli i na property ve viewmodelu vyuřívat vlastmí kontrolu chyb (v tomto kurzu není využito).

Poté otevřeme **pgPrehMatVM** a upravíme public class pgPrehMatVM : BaseVM, čímž zajistíme dědičnost po BaseVM.

XAML kód pro combobox byl již vložen, takže nezbývá než to celé vyzkoušet.

## Vytvoření tlačítek pro ukládání do databáze nebo vracení změn

V PrehMat.xaml za <TextBox> Komentář vložíme nový Grid s tlačítky:

<Grid Grid.Row="10" Grid.ColumnSpan="2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<Button Name="btnOK" Content="OK" Margin="10,2,10,2" Command="ApplicationCommands.Save" />

<Button Name="btnStrono" Grid.Column="1" Content="Storno" Margin="10,2,10,2"/>

</Grid>

Tlačítko Strono bude řízeno standardně pomocí události Click obsloužené v code-behind (F7). Tlačítko OK obsloužíme pomocí tzv. Commandu. Ten kromě klasikého chování události Click umí kontrolovat, zda lze využívat dané tlačítko (tzv. událost CanExecute) V XAML **pgPrehMat** úplně nahoře za hlavičku (za Title=“pgPrehMat“>) – za nebo před Page.Resources vložíme:

<Page.CommandBindings>

<CommandBinding Command="ApplicationCommands.Save" CanExecute="PotvrditCanExecute" Executed="btnOK\_Click"/>

</Page.CommandBindings>

Poté **CommandBinding** označíme a v seznamu událostí (ikonka „blesku“ v okně pod SolutionExplorer) a poklepeme na událost **CanExecute** (pokud kliknout nelze, projekt uložíme a následně restartujeme VisualStudio). Tím se vygeneruje kód události. Stejným způsobem vygenerujeme kód pro událost **Executed**. Poté upravíme:

private void PotvrditCanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = Globals.IsValid(gbLeva);

}

private void btnOK\_Click(object sender, RoutedEventArgs e)

{

}

## Přiřazení režimů

Do **GroupBoxu gbLeva** byla již přidána vlastnost IsEnabled=“{Binding RezimPridavaniNeboEditace}“, čímž je zajištěno, že do levého panelu bude možné zadávat hodnoty pouze, pokud se budeme nacházet v režimu přidávání nebo editace.

Režim editace (po kliknutí na tlačítko Edit - tužka) jsme již vyřešili.

Dále označíme tlačítko přidávání (+). Otevřeme seznam událostí na plovoucím panelu Properties a poklepeme na událost Click. Kód pak upravíme:

private void btnAddMat\_Click(object sender, RoutedEventArgs e)

{

//nová instance materiálu

Material material = new Material();

//výchozí datum je dnešní

material.Datum = DateTime.Today;

//přidat tuto instaci do kontextu (databáze)

\_context.Materialy.Add(material);

//přidat ji i do kolekce

\_datacontext.MaterialyCol.Add(material);

//nový materiál je vybraným materiálem

\_datacontext.MaterialSelected = material;

//nastavení režimu

\_datacontext.AktualniRezim = pgPrehMatVM.Rezimy.Pridavani;

//fokus komponenty na vybraný materiál

dgMaterialy.ScrollIntoView(\_datacontext.MaterialSelected);

}

Nyní můžeme ve spuštěné aplikaci přidávat do databáze nové materiály kliknutím na ikonu přidávání, zadáním materiálu a uložením pomocí tlačítka OK.

## Ukládání změn

Nyní je zapotřebí nastavit správné ukládání provedených změn. To můžeme udělat kdykoliv pokud nad instancí kontextu zavoláme metodu SaveChanges. Ukládání je potenciálně „nebezpečná“ operace. Při ukládání se může mnohé pokazit (spadne síť, vznikne víceuživatelský konflikt, ukládaná data nejsou validní, atd.) Vytvoříme si tedy vlastní metodu pro ukládání, která všechny tyto problémy bude umět reflektovat – alespoň chybovým hlášením.

Otevřeme **Globals.cs** v SolutionExplorer a vložíme novou statickou (spustitelnou odkudkoliv v rámci využitého namespacu) metodu:

public static void UlozitData ()

{

try

{

\_context.SaveChanges();

// \_parent.PisNaStatusBar("Uloženo. ");

}

catch (DbEntityValidationException e)

{

String text = "";

foreach (var eve in e.EntityValidationErrors)

{

text = text + "Entita typu " + eve.Entry.Entity.GetType().Name + " ve stavu " + eve.Entry.State + " obsahuje tyto chyby ověření:" + Environment.NewLine;

foreach (var ve in eve.ValidationErrors)

{

text = text + "- Vlastnost: " + ve.PropertyName + ", Chyba: " + ve.ErrorMessage + Environment.NewLine;

}

}

MessageBox.Show(text, "CHYBA", MessageBoxButton.OK, MessageBoxImage.Error);

throw;

}

catch (Exception ex)

{

MessageBox.Show("Nastala chyba při ukládání.\n" + ex.ToString(), "CHYBA", MessageBoxButton.OK, MessageBoxImage.Error);

throw;

}

}

Pokud bychom chtěli, aby nás náš software informoval, že bylo uloženo na statový řádek, kde je teď text Připraven, můžeme v předchozím kódu odkomentovat volání metody PisNaStatusBar a do winHlavni.xaml přidat tuto definici metody:

public void PisNaStatusBar(string co)

{

DateTime ted = DateTime.Now;

tbStatusBar.Text = ted.ToLongTimeString() + "> " + co;

}

V XAML samozřejmě musí být TextBlock pojmenován:

<TextBlock Name="tbStatusBar" Grid.Row="2" Text="Připraven ..." Margin="5,0,0,0"/>

Poté otevřeme třídu pgPrehMat a vyhledáme prázdnou metodu private void btnOK\_Click, dopíšeme kód:

private void btnOK\_Click(object sender, RoutedEventArgs e)

{

// \_context.SaveChanges();

Globals.UlozitData();

\_datacontext.AktualniRezim = pgPrehMatVM.Rezimy.Prohlizeni;

}

Pokud bychom místo našeho uložit data použili příkaz \_context.SaveChanges() nejspíše vše bude fungovat. Pokud by však nastal problém s ukládáním, aplikace by nejspíše bez varování spadla. Nová metoda tedy vyzkouší uložit změny, a pokud vše proběhne v pořádku, změny uloží, pokud nastane chyba, vypíše se varovná zpráva s popisem chyby.

## Storno změn

V XAML **pgPrehMat** označíme tlačítko Storno a přejdeme do seznamu událostí a poklepeme na událost Click. Funkci v editoru pak zapíšeme:

private void btnStrono\_Click(object sender, RoutedEventArgs e)

{

if (\_datacontext.AktualniRezim==pgPrehMatVM.Rezimy.Pridavani)

{

\_context.Entry(\_datacontext.MaterialSelected).State = EntityState.Detached;

\_datacontext.MaterialyCol.Remove(\_datacontext.MaterialSelected);

\_datacontext.MaterialSelected = null;

}

if (\_datacontext.AktualniRezim== pgPrehMatVM.Rezimy.Editace)

{

Globals.RollBack();

}

\_datacontext.AktualniRezim = pgPrehMatVM.Rezimy.Prohlizeni;

}

Je potřeba přidat ještě využití namespacu: **using System.Data.Entity**.

Pokud nyní během přidávání nebo editace materiálu zvolíme možnost storno, všechny dosud provedené změny se neuloží a vrátí se původní stav.

*Pozn.: To zajišťuje metoda Rollback z třídy Globals, která ze seznamu změn, které se nacházejí v kolekci ChangeTracker, nalezne ty, které jsou ve stavu Added, Deleted nebo Modified a ty nastaví na původní hodnoty.*

## Ukázka nastavení stylů

Tato podkapitola je pouze ilustrační: pokud byste chtěli změnit globální styly pro danou Page. To můžeme vyzkoušet např. na Page **pgPrehMat**.

V části <Page.Resources> definujeme globální styly pro celou page. V části .Resources lze takto definovat i styly pro dílčí komponentu. Pokud bychom chtěli změnit např. Výšku písma pro textbox u všech textboxů, můžeme to udělat takto:

<Style TargetType="TextBlock" BasedOn="{StaticResource {x:Type TextBlock}}">

<Setter Property="FontSize" Value="20"/>

</Style>

Důležitý je zde zápis BasedOn=“{StaticResource {x:Type TextBlock}}, což znamená, že nový styl bude založen na stylu, který je na komponentě právě použit. Pokud bychom toto neuvedli, zakládali bychom úplně nový styl odvozený od nejzákladnějšího nastavení s ignorování stylu Metro.

Tento zápis okomentujeme.

## Obarvení Celkového množství, pokud bude pod pojistnou zásobou

Otevřeme v **SolutionExplorer Material.cs.** Zde si najdeme property CelkoveMnozstvi, pod ní přidáme nová properties:

public bool PodPojistnymMnozstvim

{

get

{

return CelkoveMnozstvi < MnozPoj;

}

}

Propety PodPojistnymMnozstvim vraci true, pokud jsme s materiálem pod pojistným množstvím.

*Všechny tyto počítané pole mají jen get a žádný set. Automaticky tedy nebudou obsaženy v databázi. Pokud byste někdy chtěli přidat novou property, která má get i set a nechtěli i zahrnout do databáze, je nutné použít datovou anotaci [NotMapped].*

V XAML **pgPrehMat** si najdeme nalezneme počítané množství. Nahradíme stávající **WrapPanel** tímto kódem:

<WrapPanel Grid.Row="1" Margin="0,0,10,0" HorizontalAlignment="Right">

<WrapPanel.Resources>

<Style TargetType="{x:Type TextBlock}" BasedOn="{StaticResource TextBlockStyleProZadavani}">

<Setter Property="FontWeight" Value="Black"/>

<Style.Triggers>

<DataTrigger Binding="{Binding MaterialSelected.PodPojistnymMnozstvim}" Value="true">

<Setter Property="Foreground" Value="Red"/>

<Setter Property="ToolTip" Value="Celkové množství je pod pojistnou zásobou."/>

</DataTrigger>

</Style.Triggers>

</Style>

</WrapPanel.Resources>

<TextBlock Text="Celkové množství: " />

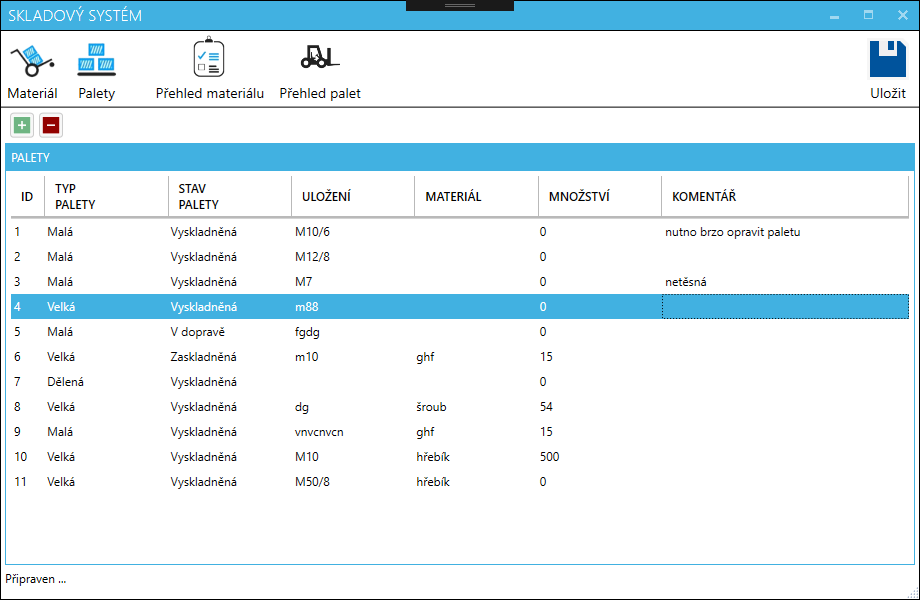
<TextBlock Text="{Binding MaterialSelected.CelkoveMnozstvi}" />

</WrapPanel>

Tak vytvoříme podmíněné formátování celkového množství. Pokud se bude celkové množství nacházet pod pojistným množstvím, tak pole zčervená a bude možno zobrazovat rychlou nápovědu (tooltip) pro najetí kurzorem myši na komponentu, v opačném případě bude pole vyvedeno pomocí tučného písma.

# Formulář Palety

Druhým hlavním formulářem v aplikaci bude formulář přehled palet, pomocí něhož bude možné vkládat nové palety a měnit stávající.



Obr. Hotový formulář (page) palety

## Založení formuláře

Otevřeme si **winHlavni.xaml** a vybereme položku menu Přehled Palet. V XAML kódu jí pak přejmenujeme na <MenuItem Name=“miPalety“>

V SolutionExplorer do adresáře **Presentation** přidáme New Item. Zvolíme typ **Page (WPF)** a pojmenujeme **pgPalety.**

Dále si otevřeme **winHlavni**. Analogicky přidáme novou položku MenuItem do Menu:

<MenuItem Name="miPalety">

<MenuItem.Header>

<StackPanel>

<Image Width="48" Height="48" Source="..\Resources\MainMenu\paleta.png"/>

<TextBlock Text="Palety" HorizontalAlignment="Center"/>

</StackPanel>

</MenuItem.Header>

</MenuItem>

Vybereme tuto položku menu pro palety, najdeme seznam událostí a poklepeme na událost **Click**. Otevřeme událost a upravíme:

private void miPalety\_Click(object sender, RoutedEventArgs e)

{

if (!(frame.Content is pgPalety))

{

frame.Navigate(new pgPalety());

}

}

Tímto zajistíme, že se po kliknutí na ikonku Přehled palet otevře formulář **pgPalety**.

Dále do konstruktoru (za InitializeComponent();) vložíme **frame.Navigate(new pgPrehMat());** čímž zajistíme, že po spuštění aplikace automaticky naběhne okno Přehled materiálu. Pokud bychom vstupní inicializaci „schovali“ do metody VstupniNastaveni, mohl by celý „zaváděcí“ kód vypadat takto (zakomentovaný kód si brzy odkomentujeme):

private SkladContext \_context = Globals.\_context;

// private winHlavniVM \_datacontext;

private void VstupniNastaveni ()

{

Globals.\_context = new SkladContext();

Globals.\_parent = this;

// \_datacontext = new winHlavniVM();

// Globals.\_parentvm = \_datacontext;

// this.DataContext = \_datacontext;

\_context = Globals.\_context;

}

public winHlavni()

{

VstupniNastaveni();

InitializeComponent();

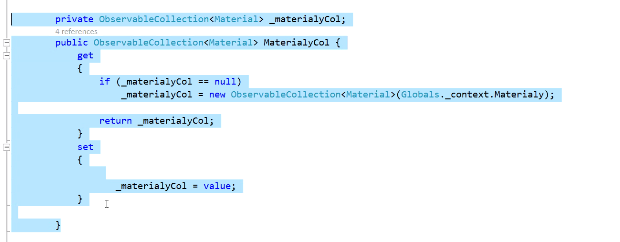
frame.Navigate(new pgPrehMat());

}

## Base ViewModel

Abychom nemuseli do každého nového formuláře psát tytéž properties a metody, můžeme je vložit do sdíleného **BaseVM**, odkud se budou dědit do ostatních viewmodelů (tyto vlastnosti a metody pak mohou využívat všechny formuláře, pokud v příslušném viewmodelu oddědíme BaseVM).

Otevřeme si **pgPrehMatVM** a nalezneme část kódu, kde jsme vytvářeli kolekce viz. Obr. 43



Obr. Vyjmutý kód

Tento kód pomocí Ctrl + X vyjmeme a přesuneme do **BaseVM**. V **BaseVM** tento kód znovu nakopírujeme a přepíšeme jej pro typy palet, tedy:

private ObservableCollection<Material> \_materialyCol;

public ObservableCollection<Material> MaterialyCol

{

get

{

if (\_materialyCol == null)

\_materialyCol = new ObservableCollection<Material>(Globals.\_context.Materialy);

return \_materialyCol;

}

set

{

\_materialyCol = value;

}

}

private ObservableCollection<PaletaTyp> \_paletaTypCol;

public ObservableCollection<PaletaTyp> PaletaTypCol

{

get

{

if (\_paletaTypCol == null)

\_paletaTypCol = new ObservableCollection<PaletaTyp>(Globals.\_context.PaletyTypy);

return \_paletaTypCol;

}

set

{

\_paletaTypCol = value;

}

}

To samé provedeme pro stavy palet, znovu tedy zkopírujeme kód a přepíšeme dle:

private ObservableCollection<PaletaStav> \_paletaStavyCol;

public ObservableCollection<PaletaStav> PaletaStavyCol

{

get

{

if (\_paletaStavyCol == null)

\_paletaStavyCol = new ObservableCollection<PaletaStav>(Globals.\_context.PaletyStavy);

return \_paletaStavyCol;

}

set

{

\_paletaStavyCol = value;

}

}

Tuto úpravu provedeme znovu, tentokrát pro Palety:

private ObservableCollection<Paleta> \_paletyCol;

public ObservableCollection<Paleta> PaletyCol

{

get

{

if (\_paletyCol == null)

\_paletyCol = new ObservableCollection<Paleta>(Globals.\_context.Palety);

return \_paletyCol;

}

set

{

\_paletyCol = value;

}

}

## Vytvoření Viewmodelu pgPaletyVM

Do adresáře **ViewModels** vložíme nový objekt typu **Class** a pojmenujeme **pgPaletyVM**. Tento ViewModel otevřeme a analogicky jako u předchozího viewmodelu vytvoříme dílčí properties:

namespace Sklad.ViewModels

{

[AddINotifyPropertyChangedInterface]

public class pgPaletyVM : BaseVM

{

public Paleta PaletaSelected { get; set; }

public bool JeOznacenaPaleta => PaletaSelected != null;

}

}

Nesmíme zapomenout připojit příslušné namespacy – using.

Dále spojíme ViewModel s prezentační vrstvou. Pomocí F7 otevřeme kód (CodeBehind) pro **pgPalety** a upravíme:

public partial class pgPalety : Page

{

private pgPaletyVM \_datacontext;

public pgPalety()

{

InitializeComponent();

\_datacontext = new pgPaletyVM();

this.DataContext = \_datacontext;

// Globals.\_parentvm.LzeUkladat = true;

}

}

Nezapomeňme na přidání namespacu Sklad.Viewmodels.

## Nastavení formuláře

Otevřeme si XAML **pgPalety.** Jako první nastavíme v hlavičce rozměry okna **Width=“700“ a Height=“300“ a barvu pozadí Background=“White“**

Za hlavičku dále vložíme zdroje stránky:

<Page.Resources>

<CollectionViewSource x:Key="PaletaTypCol" Source="{Binding PaletaTypCol}" />

<CollectionViewSource x:Key="PaletaStavyCol" Source="{Binding PaletaStavyCol}" />

<CollectionViewSource x:Key="MaterialyCol" Source="{Binding MaterialyCol}" />

<CollectionViewSource x:Key="PaletyCol" Source="{Binding PaletyCol}" />

</Page.Resources>

Na tomto místě si předvedeme jeden způsob jak pracovat s kolekcemi: Je možné z nich udělat tzv, statické (nebo dynamické) zdroje, v našem případě tzv. StaticSource. Můžeme si to představit jako tzv. alias – jiné pojmenování naší kolekce. Tato akce je výhodná zejména ve vztahu k tomu, že v rámci našeho formuláře budeme mít jeden DataGrid – v rámci jedné položky DataGridu je datovým kontextem dílčí položka, dotat se tedy na kolekci do viewmodelu může být komplikované (vystvětleno dále).

Do definice page přídáme jméno (Name) a odkaz na namespace validation:

<Page x:Class="Sklad.Presentation.pgPalety" **x:Name="page"**

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:local="clr-namespace:Sklad.Presentation"

**xmlns:validation="clr-namespace:Sklad.Validation"**

mc:Ignorable="d"

d:DesignHeight="300" d:DesignWidth="700"

Title="pgPalety">

Vytvořený **<Grid>** rozdělíme na dvě řádky dle 8.4. Poté vložíme tento XAML:

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<WrapPanel Margin="10,0,0,0">

<Button Padding="3" Grid.Column="1" Name="btnAddPal" ToolTip="Přidat" Width="24" Height="24" Margin="0,5,5,0" >

<Image Source="..\Resources\Icons\Add.png"></Image>

</Button>

<WrapPanel IsEnabled="{Binding JeOznacenaPaleta}">

<Button Padding="3" Grid.Column="1" Name="btnDelPal" ToolTip="Smazat" Width="24" Height="24" Margin="0,5,5,0" >

<Image Source="..\Resources\Icons\Delete.png"></Image>

</Button>

</WrapPanel>

</WrapPanel>

<GroupBox Header="Palety" Grid.Row="1">

<DataGrid Name="dgPalety" ItemsSource="{Binding Source={StaticResource PaletyCol}}" SelectedValue="{Binding PaletaSelected}" CanUserAddRows="False" CanUserDeleteRows="False"

HeadersVisibility="Column" SelectionMode="Single" AutoGenerateColumns="False">

<DataGrid.Columns>

<DataGridTextColumn Binding="{Binding PaletaId}" Header="ID" Width="Auto" IsReadOnly="True"/>

<DataGridComboBoxColumn SelectedValueBinding="{Binding PaletaTypId}" Header="Typ&#x0a;palety" Width="\*"

ItemsSource="{Binding Source={StaticResource PaletaTypCol}}" SelectedValuePath="PaletaTypId" DisplayMemberPath="Popis"/>

<DataGridComboBoxColumn SelectedValueBinding="{Binding PaletaStavId}" Header="Stav&#x0a;palety" Width="\*"

ItemsSource="{Binding Source={StaticResource PaletaStavyCol}}" SelectedValuePath="PaletaStavId" DisplayMemberPath="Popis"/>

<DataGridTextColumn Binding="{Binding AdrUlo, UpdateSourceTrigger=PropertyChanged}" Header="Uložení" Width="\*"/>

<DataGridTemplateColumn Header="Materiál" Width="\*" >

<DataGridTemplateColumn.CellTemplate>

<DataTemplate>

<TextBlock Text="{Binding Material.NazevMat}" />

</DataTemplate>

</DataGridTemplateColumn.CellTemplate>

<DataGridTemplateColumn.CellEditingTemplate>

<DataTemplate>

<ComboBox SelectedValue="{Binding Material, UpdateSourceTrigger=PropertyChanged, NotifyOnValidationError=True, ValidatesOnDataErrors=True, ValidatesOnExceptions=True}"

ItemsSource="{Binding DataContext.MaterialyCol, ElementName=page}" DisplayMemberPath="NazevMat"/>

</DataTemplate>

</DataGridTemplateColumn.CellEditingTemplate>

</DataGridTemplateColumn>

<DataGridTemplateColumn Header="Množství" Width="\*" >

<DataGridTemplateColumn.CellTemplate>

<DataTemplate>

<TextBlock Text="{Binding MnozPa}" />

</DataTemplate>

</DataGridTemplateColumn.CellTemplate>

<DataGridTemplateColumn.CellEditingTemplate>

<DataTemplate>

<TextBox>

<TextBox.Text>

<Binding Path="MnozPa" UpdateSourceTrigger="PropertyChanged" NotifyOnValidationError="True" ValidatesOnDataErrors="True">

<Binding.ValidationRules>

<validation:StringToIntValidationRule />

</Binding.ValidationRules>

</Binding>

</TextBox.Text>

</TextBox>

</DataTemplate>

</DataGridTemplateColumn.CellEditingTemplate>

</DataGridTemplateColumn>

<DataGridTextColumn Binding="{Binding Komentar}" Header="Komentář" Width="2\*"/>

</DataGrid.Columns>

</DataGrid>

</GroupBox>

</Grid>

**Pojďme si nyní dílčí části vysvětlit:**

<DataGrid Name="dgPalety" ItemsSource="{Binding Source={StaticResource PaletyCol}}" SelectedValue="{Binding PaletaSelected}" CanUserAddRows="False" CanUserDeleteRows="False"

HeadersVisibility="Column" SelectionMode="Single" AutoGenerateColumns="False">

Datová mřížka: uživatel nemůže přidávat ani mazat – to zařídíme pomocí tlačítek. ItemSource je směřovný na StaticSource, který jsme vytvořili v Resources – to je zde jen pro ukázku – bylo by možné směřovat přímo na kolekci: ItemSource=“{Binding PaletyCol}” – pak je samozřejmě zbytečná definice StaticSource v Resources Page.

<DataGridComboBoxColumn SelectedValueBinding="{Binding PaletaTypId}" Header="Typ&#x0a;palety" Width="\*"

ItemsSource="{Binding Source={StaticResource PaletaTypCol}}" SelectedValuePath="PaletaTypId" DisplayMemberPath="Popis"/>

V hlavičce je podivný text začínající & a končící ; - jedná se o tzv. HTML character reference – ty jsou možné vkládat kamkoliv do textu (seznam - <https://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references>). V tomto případě se jedná o ENTER.

Typ tohoto datového sloupce je DataGridComboBox. Zde potřebujeme zná, co bude zdrojem pro seznam v comboboxu a která vlastnost bude zobrazena (Popis). Dále je nutné propojit výsledek PaletaTypId – ve vlatsnosti SelectedValuePath na straně kolekce a SelectedValueBinding na straně palety.

*Toto je možno řešit lépe. Bylo by možno propojit přímo objekty – tj. přímo PaletaTyp. Tento přístup bohužel DataGridComboBox nepodporuje a musí se vše zařídit pomocí CellEditingTemplate a standardním Comboboxem – viz.dále.*

**Místo využití StaticSource je možné opět využít přímého přístupu na datacontext – tentokrát však je nutné specifikovat, který datový kontext se má využít, neboť při definování sloupce využíváme datacontext konkrétní Paleta. Takto by vypadalo definice se specifikováním datového kontextu komponenty se jménem page (celá page):**

ItemsSource="{Binding DataContext.PaletaTypCol, ElementName=page}"

Další část kódu:

<DataGridTemplateColumn Header="Materiál" Width="\*" >

<DataGridTemplateColumn.CellTemplate>

<DataTemplate>

<TextBlock Text="{Binding Material.NazevMat}" />

</DataTemplate>

</DataGridTemplateColumn.CellTemplate>

<DataGridTemplateColumn.CellEditingTemplate>

<DataTemplate>

<ComboBox SelectedValue="{Binding Material, UpdateSourceTrigger=PropertyChanged, NotifyOnValidationError=True, ValidatesOnDataErrors=True, ValidatesOnExceptions=True}"

ItemsSource="{Binding DataContext.MaterialyCol, ElementName=page}" DisplayMemberPath="NazevMat"/>

</DataTemplate>

</DataGridTemplateColumn.CellEditingTemplate>

</DataGridTemplateColumn>

Někdy se neubráníte této složitější konstrukci pro vyřešení dílčího sloupce mřížky. Pomocí DataGridTemplateColumn je možno definovat uživatelský vzhled sloupce (buňky v rámci sloupce). Takto je možno třeba vytvořit sloupec, který obsahuje konsttantní měrnou jednotky, nebo informaci a tlačítko pro výběr z číselníku, buňku pro výběr barvy, atp. Obecně je možné definovat jiný vzhled pro prohlížení (CellTemplate) a jiný pro editaci (CellEditingTemplate). U sloupce materiál se této konstrukci nevyhneme, neboť jinak nefungují validace.

U sloupce množství je demonstrováno využití vlastní uživatelské validace – v tomto případě se jedná o validace – byl zadán text. Validační pravidlo je v adreáři Validations (z FTP). V tomto případě by se vše dalo vyřešit jinak:

* Přidat vlastnost propojení ValidatesOnExceptions=True
* Pak se v tomto případě bude objevovat standardní hláška

## Přidávání a odebírání palet

Vybereme tlačítko pro přidávání **(btnAddPal**) a v seznamu událostí poklepeme na událost **Click**. Poté nastavíme událost:

private void btnAddPal\_Click(object sender, RoutedEventArgs e)

{

Paleta pal = new Paleta();

if (\_datacontext.PaletaTypCol.Count>1) pal.PaletaStav = \_datacontext.PaletaStavyCol[1]; //vychozi vyskladnena

if (\_datacontext.PaletaTypCol.Count>0) pal.PaletaTyp = \_datacontext.PaletaTypCol[0];

Globals.\_context.Palety.Add(pal);

\_datacontext.PaletyCol.Add(pal);

\_datacontext.PaletaSelected = pal;

dgPalety.ScrollIntoView(\_datacontext.PaletaSelected);

var column = dgPalety.Columns[3];

var cellToEdit = new DataGridCellInfo (pal, column);

dgPalety.CurrentCell = cellToEdit;

dgPalety.BeginEdit();

}

Dále vybere tlačítko pro odebírání **btnDelPal** a v seznamu událostí poklepeme na událost **Click**.

private void btnDelPal\_Click(object sender, RoutedEventArgs e)

{

Paleta pal = \_datacontext.PaletaSelected;

if (MessageBox.Show("Opravdu chcete vymazat paletu " + pal.PaletaId + "?", "UPOZORNĚNÍ", MessageBoxButton.YesNo, MessageBoxImage.Question) == MessageBoxResult.Yes)

{

\_datacontext.PaletyCol.Remove(pal);

Globals.\_context.Palety.Remove(pal);

}

}

## Ukládání změn a možnost nepoužívat hlavní menu

Ukládání změn provedeme globálně, a sice v rámci hlavního formuláře winHlavni – bude pak možno ukládat kdykoliv.

Nejprve si připravíme datový model. Založíme nový viewmodel do adresáře ViewModels (New Class):

namespace Sklad.ViewModels

{

[AddINotifyPropertyChangedInterface]

public class winHlavniVM

{

public bool LzeUkladat { get; set; } = false;

public bool LzePouzivatMenu { get; set; } = true;

}

}

Přidáme příslušné using PropertyChanged;.

V tomto viewmodelu jsou připdaveny dva příznaky – lze používat menu – to nelze např. když je formulář pgPrehMat v rezimu Přidávání nebo Editace.

Tento viewmodel spojíme v CodeBehind winHlavni.cs (odkomentujeme):

private SkladContext \_context = Globals.\_context;

private winHlavniVM \_datacontext;

private void VstupniNastaveni ()

{

Globals.\_context = new SkladContext();

Globals.\_parent = this;

\_datacontext = new winHlavniVM();

Globals.\_parentvm = \_datacontext;

this.DataContext = \_datacontext;

\_context = Globals.\_context;

}

V Globals.cs nyní chybí globální odkaz na viewmodel – což je užitečné:

public static winHlavniVM \_parentvm { get; set; }

Dále do hlavního gridu ve XAML winHlavni vložíme ikonu diskety pro uložení (za stávající menu, tag </Menu>).

<Menu HorizontalAlignment="Right" IsEnabled="{Binding LzePouzivatMenu}" >

<MenuItem Margin="0,0,0,0" Name="miUloz" IsEnabled="{Binding LzeUkladat}" Click="miUloz\_Click" >

<MenuItem.Header>

<StackPanel>

<Image Width="48" Height="48" Source="..\Resources\MainMenu\save.png" />

<TextBlock Text="Uložit" HorizontalAlignment="Center"/>

</StackPanel>

</MenuItem.Header>

</MenuItem>

</Menu>

Do původního menu je také přidána vazba na LzePouzivatMenu:

<Menu IsEnabled="{Binding LzePouzivatMenu}">

Do CodeBehind winHlavni.cs přidáme novou metodu, která bude testovat, zda jsou data na formuláři validní (to je třeba udělat před ukládáním nebo změnou page) :

private bool TestNaValiditu ()

{

if (frame.Content is pgPalety)

{

Page pg = (Page)frame.Content;

if (!Globals.IsValid(pg))

{

MessageBox.Show("Data nejsou validní. Před pokračováním je opravte.", "UPOZORNĚNÍ", MessageBoxButton.OK, MessageBoxImage.Warning);

return false;

}

if (Globals.HasUnsavedChanges())

{

if (MessageBox.Show("Chcete uložit provedené změny?", "OTÁZKA", MessageBoxButton.YesNo, MessageBoxImage.Question) == MessageBoxResult.Yes)

{

Globals.UlozitData();

}

else

{

Globals.RollBack();

}

}

}

return true;

}

Nyní je možno upravit i spouštění dílčích pages proti validaci:

private void miPrehMat\_Click(object sender, RoutedEventArgs e)

{

if (!TestNaValiditu()) return;

if (!(frame.Content is pgPrehMat))

{

frame.Navigate(new pgPrehMat());

}

}

private void miPalety\_Click(object sender, RoutedEventArgs e)

{

if (!TestNaValiditu()) return;

if (!(frame.Content is pgPalety))

{

frame.Navigate(new pgPalety());

}

}

Testování na validatu u miPalety\_Click sice teď nedává smysl, ale pokud do metody TestNaValiditu do budoucna zahneme i další pages, smysl to dává.

A konečně vygenerujeme novou událost pro položku menu Uložení:

private void miUloz\_Click(object sender, RoutedEventArgs e)

{

if (isAllValid()) Globals.UlozitData();

}

A k tomu příslušnou metodu:

public bool isAllValid(bool silent = false)

{

if (!Globals.IsValid(this))

{

if (!silent) MessageBox.Show

("Data obsahují chyby, před tím než bude možno pokračovat, je potřeba je opravit.", "CHYBA OVĚŘENÍ", MessageBoxButton.OK, MessageBoxImage.Error);

return false;

}

return true;

}

Parametr silent je nepovinný parametr, který určuje, zda se bude vypsisovat varovná hláška. Pokud zavoláme tuto funkci např.bool x=isAllValid(true); tak se do x jen uloží, zda jsou data validní bez propagace chybové hlášky.

Ve viewmodelu pgPrehMat dokonce můžeme už odkomentovat nastovení využitelnosti menu při změně režimu:

private Rezimy \_aktualniRezim = Rezimy.Prohlizeni;

public Rezimy AktualniRezim {

get

{

return \_aktualniRezim;

}

set

{

\_aktualniRezim = value;

Globals.\_parentvm.LzePouzivatMenu = \_aktualniRezim == Rezimy.Prohlizeni;

}

}

## Alternativa: Ukládání pomocí command

**ALTERNATIVA:** Pro případ, že bychom potřebovali zařídit ukládání rychleji (bez validace, globálního tlačítka a řízení použitelnosti menu), je možno alternativně přidat tlačítko přímo na formulář pgPalety:

V **pgPalety** XAML vložíme do kódu za **</WrapPanel>,** kam jsme vkládali tlačítka pro přidávání a mazání nové tlačítko:

<Button Name="btnUloz" Content="Uložit" Margin="5" Command="ApplicationCommands.Save"/>

Dále za hlavičku kódu za **</Page.Resources>** nadefinujeme:

<Page.CommandBindings>

<CommandBinding Command="ApplicationCommands.Save" CanExecute="UlozitCanExecute" Executed="Ulozit"/>

</Page.CommandBindings>

Pravým tlačítkem pak klikneme na **UlozitCanExecute** a **Go to Definition** (nebo poklepeme v seznamu událostí na událost **CanExecute** jako jsme dělali u předchozího formuláře). To samé uděláme s událostí **Executed** a definujeme je:

private void UlozitCanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = Globals.HasUnsavedChanges();

}

A přidáme ještě kód pro uložení:

private void Ulozit(object sender, ExecutedRoutedEventArgs e)

{

Globals.UlozitData();

}

Tímto jsme vytvořili cvičnou verzi aplikace pro skladový systém. V Aplikaci se nachází dva formuláře, jeden pro správu materiálu a druhý pro správu palet. Po spuštění aplikace by nyní mělo být možné aktualizovat databázi (přidávat, upravovat, či odebírat položky).

## Optimalizace paměti při navigaci na page

Problémem současného řešení je, že všechny otevřené page zůstávají v paměti. Je třeba je průběžně mazat. To provedeme voláním metody RemoveBackEntry po zobrazení libovolné page:

Na **winHlavni** si označíme frame a v seznamu událostí nalezneme událost Navigate, pro ní vložíme tento kód:

private void frame\_Navigated(object sender, NavigationEventArgs e)

{

frame.NavigationService.RemoveBackEntry();

}

# Aplikace Skladový systém

Na Courseware v záložce cvičení v pracovních podkladech FTP je možné stáhnout finální verzi aplikace SkladFINAL.zip. Zde si můžeme stáhnout kompletní verzi aplikace, kterou si nyní projdeme.

## Provedené změny

Oproti námi vytvořené aplikaci byly zde provedeny některé změny. Ve třídě Paleta.cs, kde jsme definovali sloupce, jsme provedli změnu v zápisu:

public int MaterialId {get; set;} na public int? MaterialId {get; set;}

Symbol otazníku znamená, že připustíme u MaterialuId int nulovou hodnotu. To znamená, že paleta v tomto případě nemusí obsahovat žádný materiál.

Dále byla to třídy Configuration v projektu DataEntity byl přidán ukázkový kód pro doplnění tabulek Typy palet, Stavy palet a Měrné jednotky. Dále jsou plněna testovací data (dva druhy materiálu a tři palety). Příkaz:

Context.Materialy.AddOrUpdate(x=> x.NazevMat, mathrebik, matsroub)

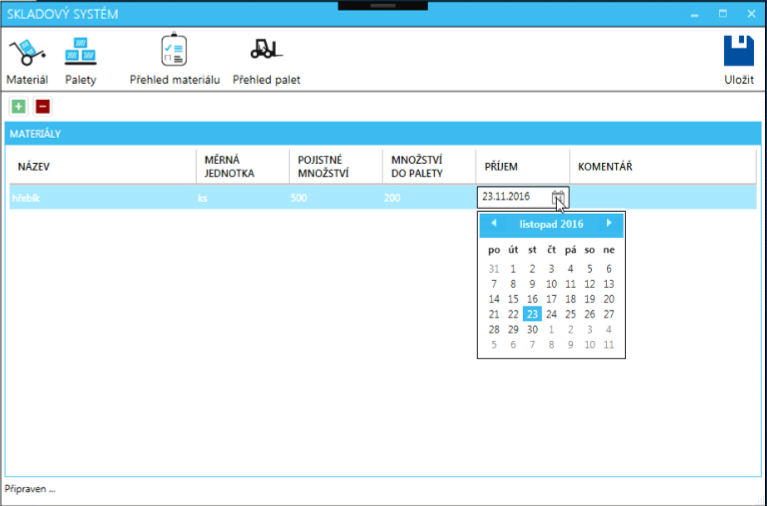
zajistí, že pokud přidáme určité množství materiálu, který pojmenujeme „hřebík“, tak mohou nastat dvě situace. Pokud se materiál „hřebík“ v databázi nenachází, založí se nový materiál. Pokud se již materiál hřebík v databázi nachází, dojde k aktulizaci této rádky, např. množství.

Zápis context.Database.ExecuteSqlCommand(dotaz); umožní nad úrovní databáze zavolat jakýkoliv SQL dotaz/příkaz.

## Správa materiálu a palet

Pro spuštění aplikace na lokálním počítači je opět nutné v souboru App.config změnit connectionString viz. 1.7.

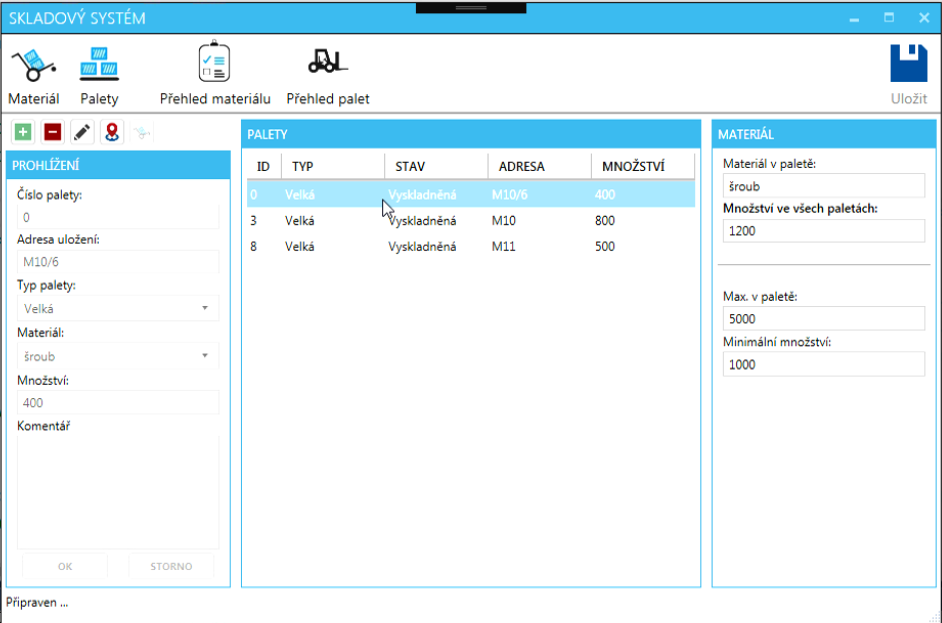
V aplikaci najdeme čtyři formuláře: Materiál, Palety, Přehled materiálu, Přehled palet. První formulář Materiál slouží k přidávání, či odebírání materiálů z databáze. Při zadávání materiálu vložíme název materiálu, vybereme měrnou jednotku, zadáme pojistné množství a množství do palety, příjem (příjem je vytvořen jako DatePicker, kde můžeme vybírat konkrétní datum viz. Obr. 44) a komentář. Jakmile zadáme všechny potřebné údaje, můžeme menu Uložit vpravo na panelu ikon provedené změny uložit. Rovněž tak můžeme již vytvořené materiály odebírat. Stejným způsobem funguje formulář pro správu palet.



Obr. Formulář pro správu materiálu – DatePicker

## Přehled palet

Formulář přehled palet funguje podobně jako námi vytvořený formulář Přehled materiálu. Formulář se může nacházet v několika stavech – prohlížení, přidávání, editace, přemístění a rezervace. Režim prohlížení viz. Obr. 45 umožňuje pouze prohlížet údaje o vybraných paletách. Těmito údaji jsou Typ, Stav, Adresa uložení a množství. Dále umožňuje zobrazení informací o materiálu v paletě v pravém sloupci formuláře.

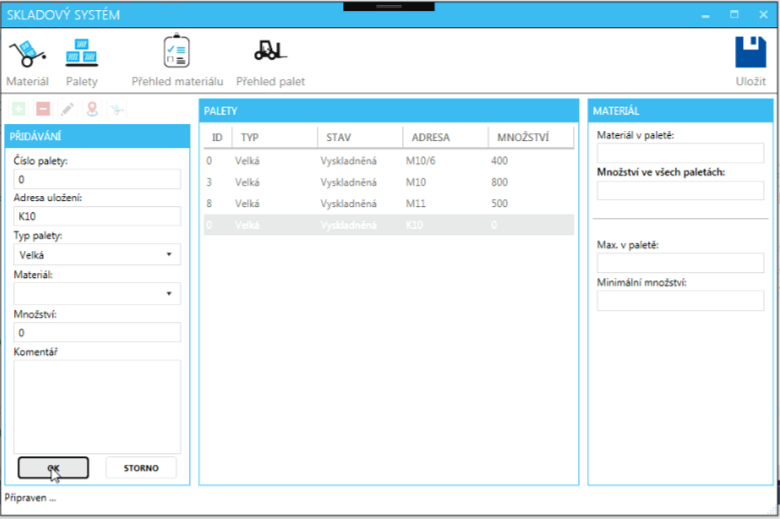


Obr. Přehled palet - režim prohlížení

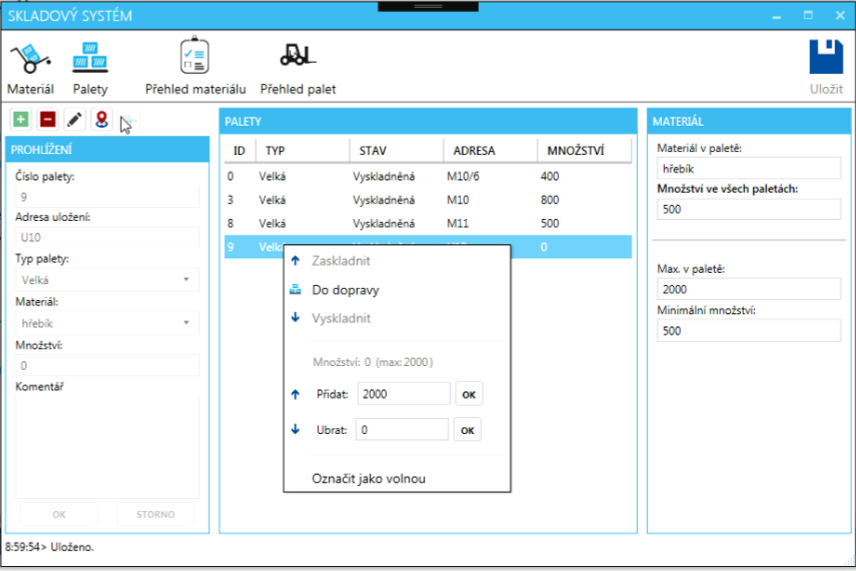
Režimy se přepínají pomocí tlačítek v levém sloupci. Režim přidávání viz. Obr. 46 můžeme aktivovat pomocí tlačítka pro přidávání. Tím se nám zpřístupní levý panel pro zadávání údajů o nově zakládané paletě (adresa uložení, typ palety, materiál, množství v paletě, komentář). Pokud nevybereme žádný materiál, zadávané množství musí být 0.

Režim editace a režim přemístění mohou být aktivní pouze tehdy, pokud je vybrána určitá paleta z prostředního sloupce formuláře.

Pravým kliknutím na paletu z nabídky se nám zobrazí menu viz. Obr. 47. Zde můžeme pomocí možnosti Označit jako volnou odebrat veškerý materiál ze zvolené palety. Pokud vybere volnou paletu, můžeme se dostat do režimu Rezervace (pátá ikonka), kdy se nám v levém panelu zpřístupní pouze možnost pro zadávání materiálu. Dále je možné měnit stavy palet podle stavu, ve kterém se právě nachází (zaskladněná, vyskladněná, v dopravě). Stejně tak množství můžeme měnit pouze u vyskladněných palet. Pokud odebereme množství materiálu tak, že se nachází pod pojistným množstvím, v pravém panelu nám zčervená Množství ve všech paletách, které říká, že jsme pod pojistným množstvím.

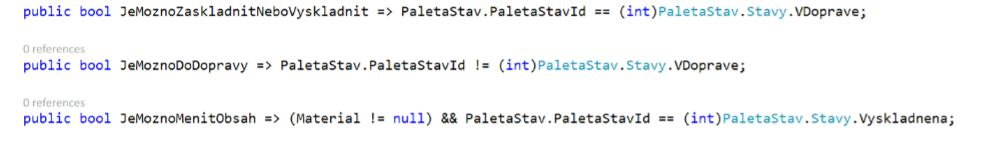


Obr. Přehled palet - režim přidávání



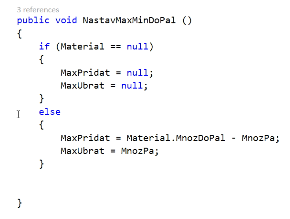
Obr. Kontextové menu

Kód na Obr. 48 ukazuje nastavení dříve zmíněné možnosti přepínání stavů palet. Paletu je možno zaskladnit nebo vyskladnit pouze pokud se nachází ve stavu v dopravě. Stejně tak je paletu možné dát do dopravy pouze, pokud se nenachází ve stavu v dopravě. Poslední zápis říká, že je možné měnit množství v paletě, pouze pokud se v ní nachází nějaký materiál a je ve stavu Vyskladněná.



Obr. Možnosti nastavení palet

Zápis na Obr. 49 nastavuje, že pokud není v paletě žádný materiál, tak není možné přidávat ani ubírat množství materiálu. Pokud se v paletě nějaký materiál nachází, možnost přidávání materiálu se uvolní.

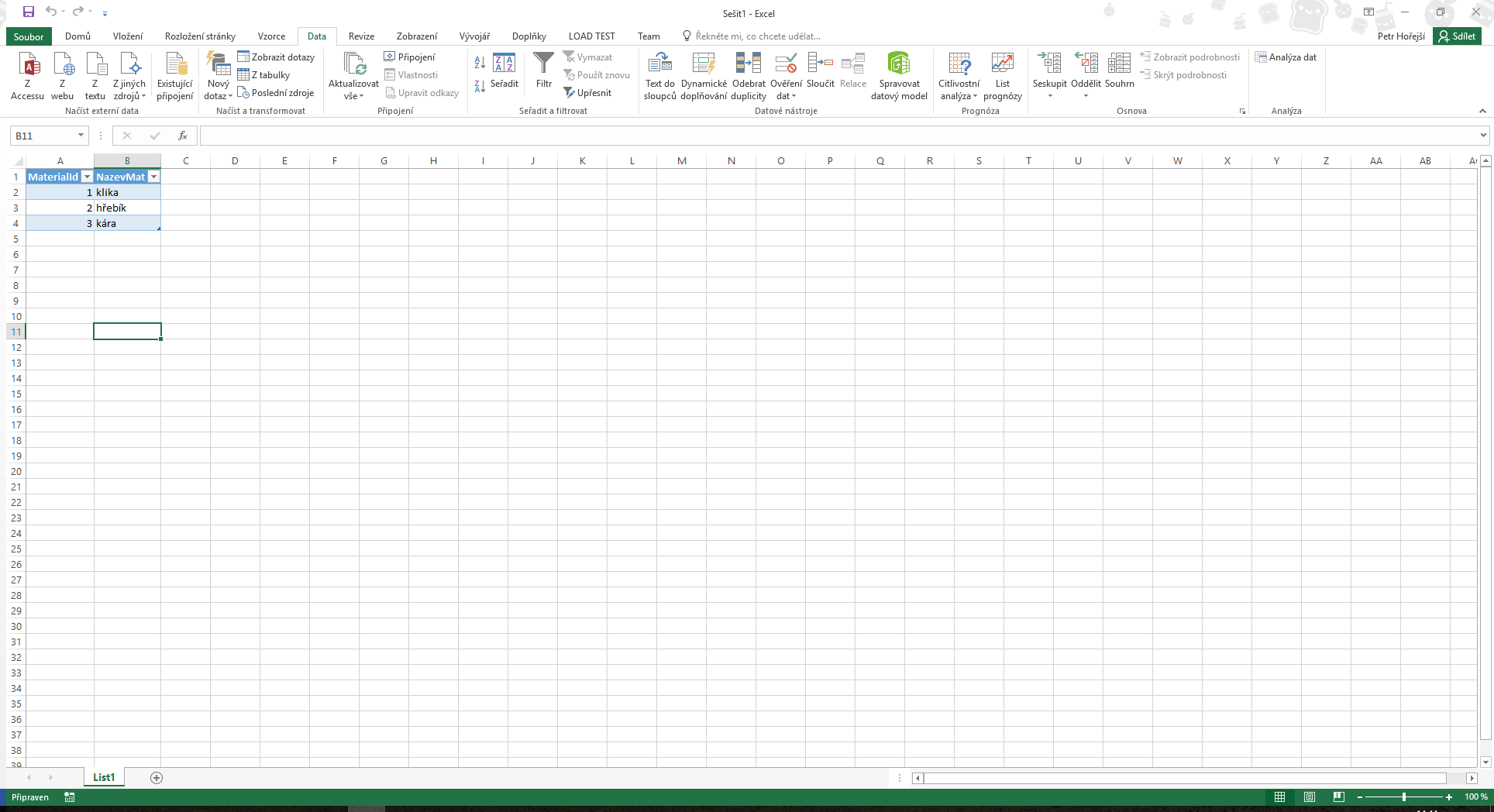


Obr. Nastavení přidávání materiálu do palety

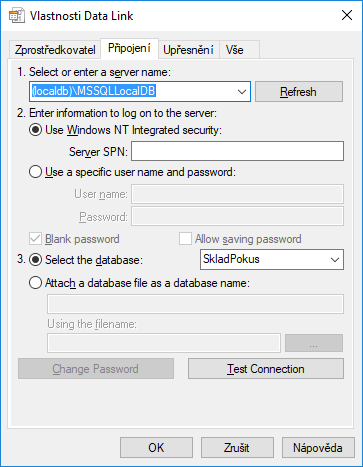
Dodatek A – Připojení k LocalDB v Excelu

Této dodatek odkazujte na kapitolu 2.3, kde je zmínka o možnosti připojení na LocalDB v Excelu. V případě SQL serveru nebo cloudu je to snadné, na LocalDB se musí jít rafinovaněji:

1. V Excelu zvolte možnost Data, načíst externí data, Z jiných zdrojů



1. Vyberte průvodce datovým připojením a zvolte upřesnit jiný, v nové okně zvolte možnost SQL Server Native Client 11.0
2. V následujícím okně vyberte:
   1. lokální SQL server - (localdb)\MSSQLLocalDB
   2. Přihlašování pomocí Windows – Use Windows integrated security
   3. Vyberte databázi



1. Nyní je možno buď otestovat připojení nebo vkládat tabulky přímo do Excelu

Dodatek B – Generický list (seznam)

Zde budeme pracovat s původní třídou List, vše předvedené samozřejmě funguje i s třídou ObservableCollection.

List je seznam čehokoliv. Může to být třeba seznam čísel (int) nebo textů (string), ale nebo také může obsahovat vlastní třídy. Tam, kde v relační databázi pracujeme s tabulkou a jejími řádky, taky pracujeme se seznamem a jeho položkami (ty reprezentuje vlastní třída).

Generický list založíme pomocí konstruktoru:

var list = new List<string>();

ve špičatých závorkách je uveden datový typ, který reprezentuje jednu položku seznamu (listu).

Ukládat data do listu, můžete třeba takto:

list.Add("Pepa");

Někdy je výhodné ukládat prvky do seznamu přímo při volání konstruktoru:

var list = new List<string>() { "Pepa", "Karel", "Mirek", "Kryštof" };

Položky ze seznamu je možno přidávat na libovolné místo, mazat, řadit atd. (viz dále)

Zajímavější bude, pokud bychom chtěli pracovat se seznamem vlastních tříd. Pro jednoduchost vezměme třeba třídu Student, která umí držet informaci o jménu studenta a jeho získané body. Pro zjednodušení práce obsahuje třída i konstruktor. Dále obsahuje i počítané pole Popis, které vrací textovou informaci jméno studenta - body:

public class Student

{

public Student (string jmeno, int body)

{

Jmeno = jmeno;

Body = body;

}

public string Jmeno { get; set; }

public int Body { get; set; }

public string Popis => Jmeno + " - " + Body;

}

Takto založíme seznam studentů:

var seznamstudentu = new List<Student>()

{ new Student ("Pepa", 5),

new Student ("Karel", 12),

new Student ("Mirek", 8),

new Student ("Kryštof", 4)

};

Nyní můžeme třeba zjistit počet záznamů:

Console.WriteLine(seznamstudentu.Count);

Console.WriteLine je výpis na konzoli.

Pojďme zkusit sečíst všechny body. Seznam tedy musíme říci informaci, co chceme sčítat. Pak je potřeba zavolat metodu Sum. Tato metoda projde všechny záznamy, v rámci zkoumání každého záznamu zavolá metodu, která je parametrem metody Sum a výsledek přičte k mezivýsledku. Problémem je tato metoda, která se posílá jako parametr – ta musí mít vracet číslo a vstupem pak musí být něco co je datového typu shodného s type každého prvku seznamu – v našem případě student, čili:

public int VyberBody (Student stud)

{

return stud.Body;

}

To můžeme při sčítání zavolat takto:

Console.WriteLine(seznamstudentu.Sum (VyberBody));

Protože toto dá dost práce, začalo se používat tzv. anonymních metod (ty nemají název – jméno VyberBody neexistuje a lze je použít jen na jednom místě). Zjednodušení:

Console.WriteLine(seznamstudentu.Sum(x=>x.Body));

Tento zápis funguje úplně stejně a nepotřebuje deklaraci metody. => je tzv. Predikát a používá se právě v kontextu anonymních metod. x je v tomto případě parametr anonymní třídy (dříve stud) a lze použít jakýkoliv přípustný název.

Zkusme to ještě zkombinovat s kritériem. Sečteme jen ty body, jejichž hodnota je větší než 6:

Console.WriteLine(seznamstudentu.Where(x=>x.Body>6).Sum(x => x.Body));

Je dokonce možné tyto studenty vypsat:

var vybranistudenti = seznamstudentu.Where(x => x.Body > 6).ToList();

vybranistudenti.ForEach(x => Console.WriteLine(x.Popis));

ForEach je extenzní metoda, která pro každou položku seznamu zavolá funkci, která je v parametru (vstupem do této funkce x je dílčí student).

Celý kód konzolové aplikace, ve které je vyzoušena jak práce s generickým listem řetězců (stringů) a generickým listem, který obsahuje třídu Student:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Seznamy

{

class Program

{

static void Main(string[] args)

{

//ukázka seznamu stringů

var list = new List<string>() { "Pepa", "Karel", "Mirek", "Kryštof" };

//výpis seznamu

Console.WriteLine(String.Join(",", list));

//přidej nový záznam

list.Add("Ivan");

Console.WriteLine(String.Join(",", list));

//odeber záznam

list.Remove("Pepa");

Console.WriteLine(String.Join(",", list));

Console.WriteLine();

//nový student - naplnění vlastností - na tři řádky

var s1 = new Student(); s1.Jmeno = "Dušan"; s1.Body = 14;

//alternativa - lepší, naplnění rovnou

var s2 = new Student() { Jmeno = "Pepa", Body = 15 };

//alternativa - pomocí konstruktoru s parametry

var s3 = new Student("Karel", 75);

//vytvoření a naplnění seznamu

var liststud = new List<Student>();

liststud.Add(s1); liststud.Add(s2); liststud.Add(s3);

//výpis jmen vedle sebe

foreach (var s in liststud)

{

Console.Write(s.Jmeno + " ");

}

Console.WriteLine();

Console.WriteLine();

Console.WriteLine();

//naplnění seznamu - jinak a rychleji

var seznamstudentu = new List<Student>()

{ new Student ("Pepa", 5),

new Student ("Karel", 12),

new Student ("Mirek", 8),

new Student ("Kryštof", 4)

};

//výpis jmen vedle sebe

foreach (var s in seznamstudentu)

{

Console.Write(s.Jmeno + "("+s.Body+") ");

//možno i takto

//Console.Write($"{s.Jmeno}({s.Body}) ");

}

Console.WriteLine();

Console.WriteLine();

Console.WriteLine("Počet: " + seznamstudentu.Count);

//volání neanonymní funkce

Console.WriteLine("Suma:" + seznamstudentu.Sum(VyberBody));

//to samé lépe

Console.WriteLine("Suma2:" + seznamstudentu.Sum(x => x.Body));

//možno kombinovat - to samé, ale 2 funkce

Console.WriteLine("Suma>6:" + seznamstudentu.Where(x => x.Body > 6).Sum(x => x.Body));

//nový seznam - vybráni jen někteří, jak SQL dotaz

var vybranistudenti = seznamstudentu.Where(x => x.Body > 6).ToList();

//+elegantní výpis, vypiš každého ze seznamu

vybranistudenti.ForEach(x => Console.WriteLine(x.Popis));

//následující okomentovaný text je alternativou k předchozímu:

//foreach (var s in vybranistudenti)

//{

// Console.WriteLine(s.Popis);

//}

Console.ReadKey();

}

public static int VyberBody(Student stud)

{

return stud.Body;

}

}

public class Student

{

public Student()

{

}

public Student(string jmeno, int body)

{

Jmeno = jmeno;

Body = body;

}

public string Jmeno { get; set; }

public int Body { get; set; }

public string Popis => Jmeno + " - " + Body;

}

}

A výpis na konzoli:

Pepa,Karel,Mirek,Kryštof  
Pepa,Karel,Mirek,Kryštof,Ivan  
Karel,Mirek,Kryštof,Ivan

Dušan Pepa Karel

Pepa(5) Karel(12) Mirek(8) Kryštof(4)

Počet: 4  
Suma:29  
Suma2:29  
Suma>6:20  
Karel – 12  
Mirek - 8

Vlastnosti na listu

* Count - Funguje jako Length na poli, vrací počet prvků v kolekci.

Metody na listu

* Add(položka) - Metodu Add() jsme si již vyzkoušeli, jako parametr bere položku, kterou vloží na konec listu.
* AddRange(kolekce) - Přidá do listu více položek, např. z pole.
* Clear() - Vymaže všechny položky v listu.
* Contains(položka) - Vrací true/false podle toho, zda List obsahuje předanou položku.
* CopyTo(pole) - Zkopíruje položky do předaného pole. Můžeme přidat parametr startovní index a počet prvků.
* IndexOf(položka) - Vrátí index prvního výskytu položky (jako u pole). Vrací -1 při neúspěchu.
* Insert(index, položka) - Vloží položku na daný index (pozici) v Listu.
* InsertRange(index, kolekce) - Vloží prvky dané kolekce na daný index v Listu.
* LastIndexOf(po­ložka) - Vrací index posledního výskytu položky v Listu. Vrací -1 při neúspěchu.
* Remove(položka) - Vymaže první nalezenou položku.
* RemoveAt(index) - Vymaže položku na daném indexu.
* RemoveRange(index, počet) - Vymaže daný počet prvků od zadaného indexu.
* Reverse() - Funguje stejně jako u pole, obrátí List tak, že je první položka poslední a naopak. Metoda nic nevrací, změny se provedou přímo v Listu.
* Sort() - Sort již také známe, setřídí položky v listu. Metoda opět nic nevrací.
* ToArray() - Zkopíruje položky z Listu do pole a to vrátí.

Další metody

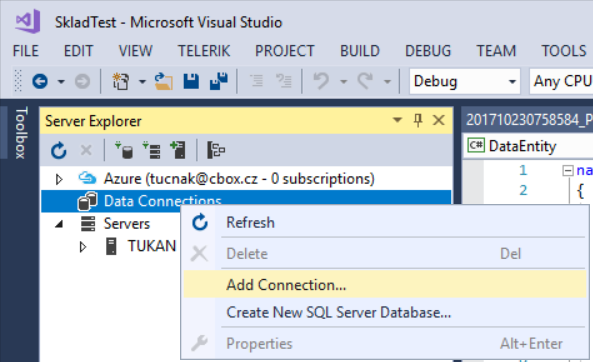
List poskytuje i další zajímavé metody, které užijeme při databázovém programování:

* Average() - Vrátí průměr z položek v Listu jako double.
* Distinct() - Vrátí unikátní elementy z Listu.
* First() - Vrátí první element.
* Last() - Vrátí poslední element.
* Intersect(kolek­ce) - Vrátí průnik Listu se zadanou kolekcí.
* Union() - Vrátí sjednocení Listu se zadanou kolekcí.
* Min() - Vrátí nejmenší prvek.
* Max() - Vrátí největší prvek.
* Sum() - Vrátí součet prvků.
* Take(počet) - Vrátí prvky od začátku Listu.

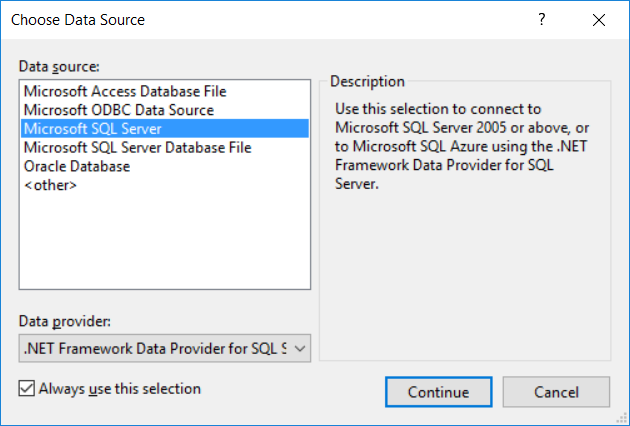
Dodatek C – Když chybí lokální databáze

Pokud není v plovoucím okně SQL Server Object Explorer viditelná lokální databáze, pokusíme se ji „občerstvit“.

To provedeme, tak, že si zobrazíme plovoucí okno Server Explorer (View, Server Explorer). Zde by mělo stačit nastavit Data Connection – datové připojení a dojde i k načtení lokální databáze. Klepněte pravým tlačítkem na Data Connections a zvolte Add Connection.



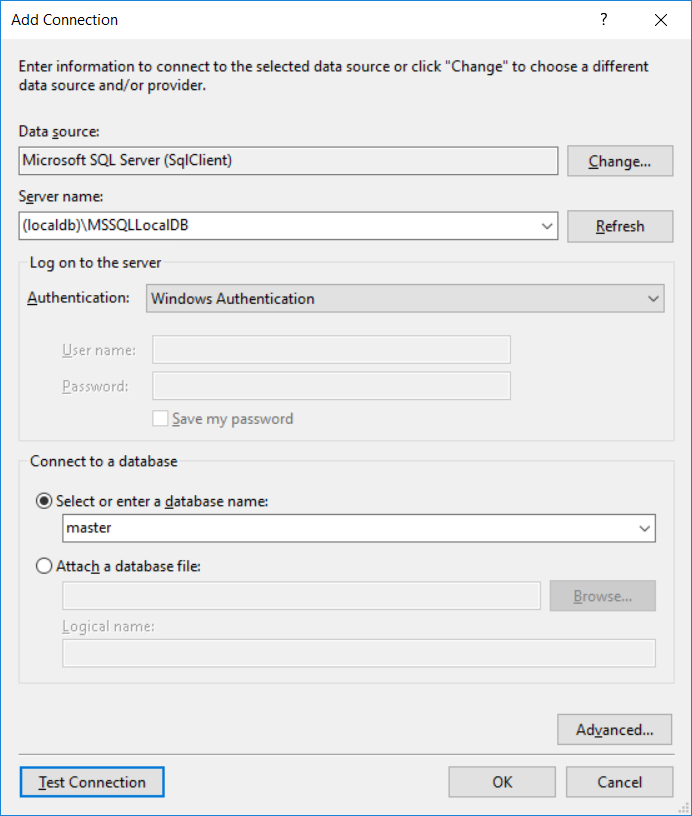
V dalším okně zvolte MS SQL Server a řekněte Continue:



V následujícím okně do Data connection vypiště ručně: **(localdb)\MSSQLLocalDB**

Vyberte databázi (Select or enter database name) – standardně by všude měla být databáze **master**

Můžete vyzkoušet připojení – **Test connection** a pak potvrdit.



Nyní by se mělo přidat jak připojení do Data connections. Hlavně byste však měli vidět databázi v SQL Server Object Explorer.

**Pokud by v následujícím postupu něco selhalo (typicky Test Connection), nainstalujte SQL Server 2016 Express LocalDB – možno snadno vygooglit. Pak zkuste vše znovu.**

Dodatek D – základy XAML

Tato kapitola je inspirována stránkou <http://www.wpf-tutorial.com/>

**Tuto stránku můžete navštívit pokaždé, když si nebudete vědět rady s dílčí komponentou.**

V XAML pro každou komponentu (kontrolku) požíváme značku (tag) – tak např. pro tlačítko je to:

<Button>

To samo o sobě „nefunguje“. Každá značka (tag) musí být uzavřena:

<Button></Button>

Nebo:

<Button/>

Každý tag má standardní (defaultní) vlastnost, který lze napsat mezi začínající a ukončovací tag. U tlačítka je to Content. Takže tento zápis:

<Button>Ahoj</Button>

Je ekvivalentní tomuto zápisu:

<Button Content="Ahoj"></Button>

Nebo tomuto:

<Button Content="Ahoj"/>

Jak je vidět vlastnosti se zapistují do tagu ve formátu název vlastnosti=“něco“. Takto můžeme změnit písmo na tučné:

<Button FontWeight="Bold" Content="Ahoj"/>

Vlastnosti je možno nastavovat také pomocí tečkové notace. Ekvivalentní zápis:

<Button>

<Button.Content>Ahoj</Button.Content>

<Button.FontWeight>Bold</Button.FontWeight>

</Button>

V tomto případě nemá význam vše zapisovat „takto složitě“. Účelnost tohoto typu zápisu dobře uvidíme na příkladu, kdy můžeme takto „dovnitř“ komponenty Button vložit jiné komponenty:

<Button FontWeight="Bold">

<Button.Content>

<WrapPanel>

<TextBlock Foreground="Blue">Více</TextBlock>

<TextBlock Foreground="Red">barevné</TextBlock>

<TextBlock Text=" tlačítko"/>

</WrapPanel>

</Button.Content>

</Button>

V rámci tohoto příkladu jsou střídány různé přístupy. Je vidět, že u komponenty TextBlock je výchozí vlastností Text. Do vlastnosti Content je možno vložit pouze jednu komponentu. Zde je to tzv. Wrap Panel, který slučuje podkomponenty horizontálně (vedle sebe) – tzv. StackPanel je umí řadit pod sebe.

Celý XAML kód jednoho window, page, usercontrol, …. je de facto jen rozšířením (partial class) třídy, která je v CodeBehind (stisk F7). Kód, který je v našem případě generován vypadá nějak takto:

Button btn = new Button();

btn.FontWeight = FontWeights.Bold;

WrapPanel pnl = new WrapPanel();

TextBlock txt = new TextBlock();

txt.Text = "Více";

txt.Foreground = Brushes.Blue;

pnl.Children.Add(txt);

txt = new TextBlock();

txt.Text = "barevné";

txt.Foreground = Brushes.Red;

pnl.Children.Add(txt);

txt = new TextBlock();

txt.Text = " tlačítko";

pnl.Children.Add(txt);

btn.Content = pnl;

grMain.Children.Add(btn);

Pokud byste si Grid, ve kterém je tlačítko pojmenovali grMain a tento kód vložili do konstruktoru v CodeBehind, výsledek bude totožný.

**Důležité: Abyste mohli přistupovat ke komponentě v CodeBehind, musí být veřejná – to lze provést jejím pojmenováním – tj. naplněním vlastnosti Name.**

<Button Name="btnPokus" FontWeight="Bold">

Rozhodně zajímavé jsou formátovací vlastnosti, které jsou víceméně pro všechny komponenty stejné. Příklad:

<Button Name="btnPokus" FontWeight="Bold" Width="150" Height="25" HorizontalAlignment="Left" VerticalAlignment="Top" Margin="10">

Width, Height – šířka, výska

HorizontalAlignment – odstazení vodorovné (Left, Right, Center, Stretch)

VerticalAlignment – odstazení svislé (Top, Bottom, Center, Stretch)

Margin – odsazení, Je možné napsat i Margin=”10,0,5,15” a tím definovat různá odsazení z různých stran

Dodatek E – nové okno s předaným datakontextem

Občas je potřeba otevřít nové okno, např. s detaily nějakého objektu. V našem případě si naznačíme, jak je možné otevřít nové okno s detaily k materiálu.

Vytvoříme si nové okno. Pod Presentation založíme nový objekt okno – Add, New Item, Window (XAML). Toto okno nazveme winMatDetails. Toto okno bude mít následující kód:

<Window x:Class="Sklad.Presentation.winMatDetails"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Sklad.Presentation"

xmlns:vm="clr-namespace:Sklad.ViewModels"

xmlns:de="clr-namespace:DataEntity;assembly=DataEntity"

mc:Ignorable="d"

d:DataContext="{d:DesignInstance Type=de:Material, IsDesignTimeCreatable=False}"

Title="winMatDetails" Height="300" Width="300">

<Grid>

<StackPanel>

<TextBlock Text="{Binding NazevMat}"/>

<TextBlock Text="{Binding CelkoveMnozstvi}"/>

</StackPanel>

</Grid>

</Window>

V rámci tohoto kódu je ukázáno jak v XAML můžeme využívat namespace Presentation.Viewmodels:

xmlns:vm="clr-namespace:Sklad.ViewModels"

a hlavně, a to tady využijeme, jak využít namespace z jiného projektu (zde DataEntity):

xmlns:de="clr-namespace:DataEntity;assembly=DataEntity"

zde to děláme proto, abychom si ukázali, jak je možné přímo v XAML definovat datacontext a sice jen pro účely tvorby Designu:

d:DataContext="{d:DesignInstance Type=de:Material, IsDesignTimeCreatable=False}"

Pak funguje „napovídání“ přímo v XAML Designeru.

Nyní si toto okno spustíme:

Na page PrehMat přidáme tlačítko Detaily např. do horního WrapPanelu:

<Button Name="Detail" Content="Detail" Click="Detail\_Click" />

A CodeBehind:

private void Detail\_Click(object sender, RoutedEventArgs e)

{

var okno = new winMatDetails();

okno.DataContext = \_datacontext.MaterialSelected;

okno.ShowDialog();

}

Tím jsme do tohoto okna předali datový context a pak pomocí ShowDialog spouští tzv. Modální okno – je nutné jej zavřít než je možno vrátit se k PrehMat.