VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2017 Michal Polách

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

PREZENTACE DAT Z MYSQL DATABÁZE VE WINDOWS PRESENTATION FOUNDATION
PRESENTATION OF DATA FROM A MYSQL DATABASE IN WINDOWS PRESENTATION FOUNDATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Michal Polách**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Ivo Lattenberg, Ph.D.**
SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Teleinformatika**
Ústav telekomunikací

Student: Michal Polách ***ID:*** 173729
Ročník: 3 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Prezentace dat z MySQL databáze ve Windows Presentation Foundation

POKYNY PRO VYPRACOVÁNÍ:

Naprogramujte vícevrstvou WPF aplikaci pro práci s rozsáhlou knihovní databází, která bude umístěna na MySQL serveru. Pro simulaci rozsáhlé databáze vygenerujte pomocí SQL skriptu větší množství záznamů. Aplikace bude pro zobrazení velkého množství dat používat GridView ve virtuálním módu. Uvažujte synchronizaci mezi více spuštěnými aplikacemi, které jsou napojeny na společnou databázi, stejně tak jako aktualizace zobrazení dat v aplikaci, po změně v databázi pomocí SQL skriptu. Implementujte základní vyhledávání a třídění dle požadavků uživatele.

DOPORUČENÁ LITERATURA:

[1] PETZOLD, C. Mistrovství ve Windows Presentation Foundation, Nakladatelství Computer Press, a.s. 2008, 928 stran, ISBN 9788025121412

[2] VIRIUS, M., C# 2010 Hotová řešení, Computer Press, 2012, 424 s., ISBN 978-80-251-3730-7

Termín zadání: 1.2.2017 ***Termín odevzdání:*** 8.6.2017

Vedoucí práce: doc. Ing. Ivo Lattenberg, Ph.D.
Konzultant:

doc. Ing. Jiří Mišurec, CSc.
předseda oborové rady

UPOZORNĚNÍ:
Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

Abstrakt
V práci je demonstrována a vysvětlena tvorba WPF aplikace podle návrhového vzoru MVVM pro svižné prohlížení, vyhledávání a editaci dat v rozsáhlé knihovní databázi uložené na MySQL serveru. Rychlého načítání dat z databáze je docíleno využitím datové virtualizace pomocí metody stránkování.

Summary
In the thesis there is demonstrated and explained the creation of WPF MVVM design pattern based applications for brisk browsing, searching and editing of data in large library database stored on a MySQL server. Quick loading of data is achieved by using data virtualization using the paging method.

Klíčová slova
WPF, MySQL, databáze, vázání dat, tabulka, virtualizace, Entity Framework, stránkování

Keywords
WPF, MySQL, database, data-binding, table, virtualization, Entity Framework, paging

POLÁCH, M.*Prezentace dat z MySQL databáze ve Windows Presentation Foundation.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2017. 45 s. Vedoucí bakalářské práce doc. Ing. Ivo Lattenberg, Ph.D.

Prohlašuji, že svou bakalářskou práci na téma „Prezentace dat z MySQL databáze ve Windows Presentation Foundation“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení §11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Michal Polách

Obsah

I Teoretická část 7

1 Windows Presentation Foundation 8
1.1 Grafika . . . 8
1.2 Animace . . . 8
1.3 Pixel nezávislý na zařízení . . . 8
1.4 Ovládací prvky, elementy, styly a šablony . . . 9
1.5 Systém vlastností . . . 9
1.6 Databinding . . . 9
1.6.1 Fody/PropertyChanged . . . 10
1.7 Jazyk XAML . . . 11
1.8 Nástroje pro vývoj WPF aplikací . . . 11

2 Návrhový vzor MVVM 12
2.1 View . . . 12
2.2 Model . . . 12
2.3 ViewModel . . . 13
2.3.1 Příkazy . . . 13
2.4 Prostředník (Mediátor) . . . 13
2.5 Balíček nástrojů MVVM Light . . . 14

3 MySQL 15
3.1 Relační model databáze . . . 15
3.2 Jazyk SQL . . . 16
3.2.1 Útok SQL injection . . . 16
3.3 Indexování sloupců . . . 17
3.4 MySQL Connector/Net . . . 17
3.4.1 Connection pooling . . . 17

4 Entity Framework 19
4.1 Verze Entity Freworků . . . 19
4.1.1 Entity Framework 6.X . . . 19
4.1.2 Entity Framework Core . . . 19
4.2 Entity Data Model . . . 19
4.3 Kontextová třída . . . 19
4.4 Entity . . . 20
4.4.1 POCO entity . . . 20
4.4.2 Dynamic Proxy entity . . . 20
4.4.3 Vlastnosti entit . . . 20
4.5 Přístupy k databázi . . . 21
4.5.1 Model First . . . 21
4.5.2 Database First . . . 21
4.5.3 Code First . . . 21

II Praktická část 22

5 Struktura aplikace 23
- 5.1 Třívrstvá architektura aplikace 23
  - 5.1.1 Prezentační vrstva 23
  - 5.1.2 Aplikační vrstva 23
  - 5.1.3 Datová vrstva 23
- 5.2 Rozložení podle MVVM 23
- 5.3 Grafická struktura 23
  - 5.3.1 MainWindow 23
  - 5.3.2 BooksPageView a AuthorsPageView 24
  - 5.3.3 BookDetailView a AuthorDetailView 24
  - 5.3.4 BookFilterView a AuthorFilterView 24
  - 5.3.5 EditBookView a EditAuthorView 25
  - 5.3.6 AuthorSelectWindow a BookSelectWindow 25

6 MySQL databáze 27
- 6.1 Instalace MySQL serveru 27
- 6.2 Tvorba databáze 27
  - 6.2.1 Návrh entit 27
  - 6.2.2 Kontextová třída 29
  - 6.2.3 Inicializace databáze 30
  - 6.2.4 Generování záznamů 30

7 Logika aplikace 31
- 7.1 Modely entit 31
  - 7.1.1 Mapovač 31
- 7.2 Virtualizující kolekce 31
- 7.3 Poskytovatele záznamů 32
  - 7.3.1 Poskytovatel záznamů kolekce 32
  - 7.3.2 Poskytovatel detailů záznamů 32
  - 7.3.3 Poskytovatel MySQL záznamů 32
- 7.4 Řazení 33
- 7.5 Filtrování a vyhledávání záznamů 33
  - 7.5.1 Struktura podmínky 34
  - 7.5.2 Filtr 34
  - 7.5.3 Skládání podmínek 35
- 7.6 Přidávání a editace záznamů 35
  - 7.6.1 Přidávání záznamů 35
  - 7.6.2 Editace záznamů 36
- 7.7 Synchronizace více spuštěných aplikací 36
  - 7.7.1 Uchování označení řádku 37

8 Měření rychlosti a optimalizace 39
- 8.1 Sledování změn v kontextu 39
- 8.2 Doba načítání stránek 39
- 8.3 Doba generování databáze 40

Seznam obrázků

2.1 Schéma návrhového vzoru MVVM 12
2.2 Schéma funkce prostředníka 13
5.1 Hlavní okno aplikace 24
5.2 Zobrazení detailu záznamu knihy 25
5.3 Pohled pro filtrování knih 25
5.4 Pohled pro editaci záznamu knihy 26
5.5 Okno pro přidání autora k záznamu knihy 26
7.1 Kontextové menu pro řazení sloupců 33
7.2 Notifikace změny v databázi 36

Seznam ukázek zdrojových kódů

I Teoretická část 7

1 Windows Presentation Foundation 8
- 1.1 Grafika . . . 8
- 1.2 Animace . . . 8
- 1.3 Pixel nezávislý na zařízení . . . 8
- 1.4 Ovládací prvky, elementy, styly a šablony . . . 9
- 1.5 Systém vlastností . . . 9
- 1.6 Databinding . . . 9
  - 1.6.1 Fody/PropertyChanged . . . 10
- 1.7 Jazyk XAML . . . 11
- 1.8 Nástroje pro vývoj WPF aplikací . . . 11

2 Návrhový vzor MVVM 12
- 2.1 View . . . 12
- 2.2 Model . . . 12
- 2.3 ViewModel . . . 13
  - 2.3.1 Příkazy . . . 13
- 2.4 Prostředník (Mediátor) . . . 13
- 2.5 Balíček nástrojů MVVM Light . . . 14

3 MySQL 15
- 3.1 Relační model databáze . . . 15
- 3.2 Jazyk SQL . . . 16
  - 3.2.1 Útok SQL injection . . . 16
- 3.3 Indexování sloupců . . . 17
- 3.4 MySQL Connector/Net . . . 17
  - 3.4.1 Connection pooling . . . 17

4 Entity Framework 19
- 4.1 Verze Entity Freworků . . . 19
  - 4.1.1 Entity Framework 6.X . . . 19
  - 4.1.2 Entity Framework Core . . . 19
- 4.2 Entity Data Model . . . 19
- 4.3 Kontextová třída . . . 19
- 4.4 Entity . . . 20
  - 4.4.1 POCO entity . . . 20
  - 4.4.2 Dynamic Proxy entity . . . 20
  - 4.4.3 Vlastnosti entit . . . 20
- 4.5 Přístupy k databázi . . . 21
  - 4.5.1 Model First . . . 21
  - 4.5.2 Database First . . . 21
  - 4.5.3 Code First . . . 21

II Praktická část 22

5 Struktura aplikace 23
5.1 Třívrstvá architektura aplikace . . . 23
5.1.1 Prezentační vrstva . . . 23
5.1.2 Aplikační vrstva . . . 23
5.1.3 Datová vrstva . . . 23
5.2 Rozložení podle MVVM . . . 23
5.3 Grafická struktura . . . 23
5.3.1 MainWindow . . . 23
5.3.2 BooksPageView a AuthorsPageView . . . 24
5.3.3 BookDetailView a AuthorDetailView . . . 24
5.3.4 BookFilterView a AuthorFilterView . . . 24
5.3.5 EditBookView a EditAuthorView . . . 25
5.3.6 AuthorSelectWindow a BookSelectWindow . . . 25

6 MySQL databáze 27
6.1 Instalace MySQL serveru . . . 27
6.2 Tvorba databáze . . . 27
6.2.1 Návrh entit . . . 27
6.2.2 Kontextová třída . . . 29
6.2.3 Inicializace databáze . . . 30
6.2.4 Generování záznamů . . . 30

7 Logika aplikace 31
7.1 Modely entit . . . 31
7.1.1 Mapovač . . . 31
7.2 Virtualizující kolekce . . . 31
7.3 Poskytovatele záznamů . . . 32
7.3.1 Poskytovatel záznamů kolekce . . . 32
7.3.2 Poskytovatel detailů záznamů . . . 32
7.3.3 Poskytovatel MySQL záznamů . . . 32
7.4 Řazení . . . 33
7.5 Filtrování a vyhledávání záznamů . . . 33
7.5.1 Struktura podmínky . . . 34
7.5.2 Filtr . . . 34
7.5.3 Skládání podmínek . . . 35
7.6 Přidávání a editace záznamů . . . 35
7.6.1 Přidávání záznamů . . . 35
7.6.2 Editace záznamů . . . 36
7.7 Synchronizace více spuštěných aplikací . . . 36
7.7.1 Uchování označení řádku . . . 37

8 Měření rychlosti a optimalizace 39
8.1 Sledování změn v kontextu . . . 39
8.2 Doba načítání stránek . . . 39
8.3 Doba generování databáze . . . 40

Úvod

Cílem této práce je naprogramovat aplikaci ve Windows Presentation Foundation (dále jen WPF) pro zobrazování dat z knihovní databáze uložené na MySQL serveru.

Aplikace bude schopná virtualizace dat, čímž bude dosažené její rychlé odezvy i při práci s rozsáhlými databázemi. V aplikaci bude možné vyhledávat, filtrovat a řadit záznamy podle požadavků uživatele. Dále bude možné záznamy přidávat, odstraňovat a editovat. Aplikace by měla být schopná automaticky detekovat změny dat v databázi, na kterou je připojena, čímž umožní souběžnou práci více uživatelů na jedné databázi.

Pro zobrazení tabulek bude aplikace používat ovládací prvek ListView se zobrazením GridView, který je nativní součástí .NET frameworku. Pro připojení k MySQL databázi bude použit MySQL Connector/Net a pro objektově relační mapování bude využit Entity Framework 6.1 s přístupem Code First.

Aplikace bude navržena podle návrhového vzoru Model-View-ViewModel, který v současné době patří mezi nejpoužívanější návrhové vzory a poskytuje dvě hlavní výhody: snadné testování výsledné aplikace a oddělení logické části aplikace od grafické.

Aplikace bude využívat vícevrstvou architekturu, díky které bude flexibilní a snadno rozšiřitelná.

Dokument je rozdělen do dvou částí: teoretická část a praktická část.

V teoretické části je čtenář seznámen se základními stavebními bloky aplikace a s principy jejich použití. Je v ní vysvětlena struktura aplikací tvořených ve frameworku WPF, návrhový vzor Model-View-ViewModel, MySQL databáze a Entity Framework.

Praktická část demonstruje postup tvorby aplikace od návrhu až po implementaci jednotlivých logických bloků. Činnosti bloků jsou vysvětleny a je-li třeba, jsou u nich uvedeny ukázky důležitých částí kódu.

Na závěr je provedeno měření výkonnosti aplikace pro různá nastavení, podle kterého je výsledná aplikace následně optimalizována.

Část I

Teoretická část

1. Windows Presentation Foundation

Windows Presentation Foundation, zkráceně WPF, je framework pro tvorbu formulářových aplikací od společnosti Microsoft. Je součástí .NET frameworku od verze 3.0. Původně tento framework nesl označení Avalon. Součástí .NET frameworku je stále i starší framework Windows Forms, který však postrádá řadu funkcionalit, které nabízí WPF [1].

Mezi klíčové vlastnosti WPF patří oddělení grafické a logické části aplikace, grafika a animace, velké možnosti stylizace a šablonování a databinding. WPF poskytuje širokou paletu grafických komponent (ovládacích prvků), které lze snadno poskládat do formulářů, případně si můžeme dotvořit vlastní ovládací prvky. Pro definici formulářů používá jazyk XAML, díky kterému může jednotlivým ovládacím prvkům přiřazovat vlastnosti objektů a může provazovat data s logickou částí aplikace.

1.1. Grafika

Grafika ve WPF využívá knihoven Direct3D, které jsou obsaženy v DirectX. Tyto knihovny umožňují využití hardwarové akcelerace pomocí grafické karty, podporují vektorovou grafiku a vykreslování 3D objektů. Díky hardwarové akceleraci umožňuje aplikaci přemístit část zátěže způsobené vykreslováním grafiky z CPU na GPU a tím docílit vyšší svižnosti aplikace. Výhodou vektorové grafiky je taky to, že vývojáři zůstává kontrola nad nakreslenými objekty a může s nimi dále manipulovat.

1.2. Animace

Obvyklý přístup k animacím je, že se překreslují vždy po několika snímcích, což může být závislé na výkonu počítače, a tak se může stát, že animace běží jinou rychlostí, než by bylo vhodné. Překreslování animace ve WPF je vyvoláváno po nastavených časových intervalech, čímž se tomuto problému vyhne.

Animace se dají ukládat do tzv. storyboardů, které umožňují spouštění, pozastavování a jinou manipulaci s animací. Animace mohou být spouštěny pomocí událostí aplikace, jako třeba stisknutí tlačítka, spuštění aplikace apod.

1.3. Pixel nezávislý na zařízení (Device Independent Pixel)

WPF přepočítává velikosti pixelů podle DPI (dots per inch) zařízení, na kterém je formulář zobrazen, na tzv. Device Independent Pixel (pixel nezávislý na zařízení). Tím udržuje velikost komponent na různých zařízení konstantní a předchází tak nechtěným změnám velikosti zobrazených komponent.

1.4. Ovládací prvky, elementy, styly a šablony

Ovládacím prvkem rozumíme objekt v grafickém rozhraní, jako je např. tlačítko, popisek nebo tabulka. Každý ovládací prvek se skládá z jednoho či více elementů. Všechny tyto elementy jsou ve WPF poskládané do hierarchického vizuálního stromu, který má vždy jeden kořen. Vlastnosti jednotlivých elementů formuláře můžeme definovat buď přímo pomocí jejich atributů, nebo nepřímo pomocí šablon a stylů.

Element je nejmenší stavební jednotka formůlářů ve WPF. Každý element je objekt odvozený z `DependencyObject`, a podporuje tak `DependencyProperty`, což je systém vlastností ve WPF.

Styl je soubor vlastností, který si může vývojář nadefinovat. Jeden styl je možné přiřadit více ovládacím prvkům, a tím umožňuje měnit vlastnosti ovládacích prvků a elementů hromadně.

Šablona definuje alternativní vzhled komponent v částech aplikace. Existuje několik typů šablon: šablona ovládacího prvku `ControlTemplate`, datová šablona `DataTemplate`, `HierarchicalDataTemplate` a `ItemsPanelTemplate`. Z jakých elementů se ovládací prvek skládá, definuje právě jeho šablona. V základu se používá implicitní „šablona“ vytvořená autorem ovládacího prvku.

Použitím šablony ovládacího je možné nahradit jeho implicitní šablonu, a tak modifikovat jeho vzhled. Podobně se používá datová šablona pro změnu reprezentace dat, které by se jinak zobrazovaly jako prostý text.

1.5. Systém vlastností

WPF představuje nový systém vlastností. Tyto vlastnosti se nazývají `DependencyProperty` a můžeme je používat jen v objektech, které dědí z `DependencyObject`. Na rozdíl od CLR vlastností, které jen zapouzdřují proměnnou a vystavují metody na její čtení a změnu, `DependencyProperty` nabízejí řadu výhod. Mimo jiné umožňují dědit hodnoty jiných `DependencyProperty`, vytvářet automatické kontroly hodnot při jejich změně, používat je v šablonách a stylech nebo je používat pro databinding.

1.6. Databinding

WPF poskytuje služby umožňující provazování a manipulaci dat v aplikaci. Těmto službám se říká binding. Pomocí bindingu můžeme provazovat data mezi vlastnostmi elementů (klienty) v XAML dokumentu s objekty (zdroji) ve zdrojovém kódu.
Jako zdroje bindingu mohou sloužit

- CLR objekty,
- Dynamické objekty – implementující rozhraní `IDynamicMetaObjectProvider`,
- Objekty ADO.NET (`DataTable`, `DataView`),
- XML objekty nebo
- `DependencyObject`y.

Existují čtyři módy bindingu:

- one time – klient načte data ze zdroje jednou a změny na zdroji ignoruje,
- one way – klient načte data a aktualizuje se s každou změnou na zdroji,
- two way – zdroj i klient se navzájem udržují aktuální, změny se promítnou u obou,
- one way to source – změny u klienta se projeví na zdroji.

Chceme-li, aby se klient aktualizoval při změně hodnoty zdroje, je nutné, aby vlastnost sloužící jako zdroj bindingu při své změně vyvolávala událost `PropertyChanged`. Toho lze docílit vhodnou implementací rozhraní `INotifyPropertyChanged`.

Ukázka zdrojového kódu 1.1: Příklad implementace `INotifyPropertyChanged`

```
public class ClassWithNotifications : INotifyPropertyChanged
{
    private string _name;
    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
            OnPropertyChanged(nameof(Name));
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new
           PropertyChangedEventArgs(propertyName));
    }
}
```

1.6.1. Fody/PropertyChanged

Fody/PropertyChanged je balíček pro Visual Studio, který umožňuje používání speciálních atributů pro usnadnění bindingu.

Nejdůležitějším atributem je `[ImplementPropertyChanged]`, který při kompilaci u všech tříd označených tímto atributem implementuje rozhraní `INotifyPropertyChanged` a všem vlastnostem přidá kód k vyvolání `PropertyChanged` události.

Dalšími atributy jsou `AlsoNotifyFor`, `DoNotNotify`, `DependsOn`, `DoNotSetChanged` a `DoNotCheckEquality`. Jejich použití je vysvětleno na stránce projektu na GitHubu [2].

1.7. Jazyk XAML

Jazyk XAML (Extensible Application Markup Language, původně Extensible Avalon Markup Language) je značkovací jazyk založený na XML. Byl taktéž vyvinut společností Microsoft a ve WPF je využíván pro popis grafického rozhraní aplikace. Elementy se podobně jako např. v HTML zapisují do lomených závorek, ve kterých je vždy název elementu. Elementy musí být vždy ukončeny a nesmí se navzájem křížit (tzn., že nemůžeme ukončit element, který obsahuje jiný neukončený element).

1.8. Nástroje pro vývoj WPF aplikací

Nástroje od Microsoftu:

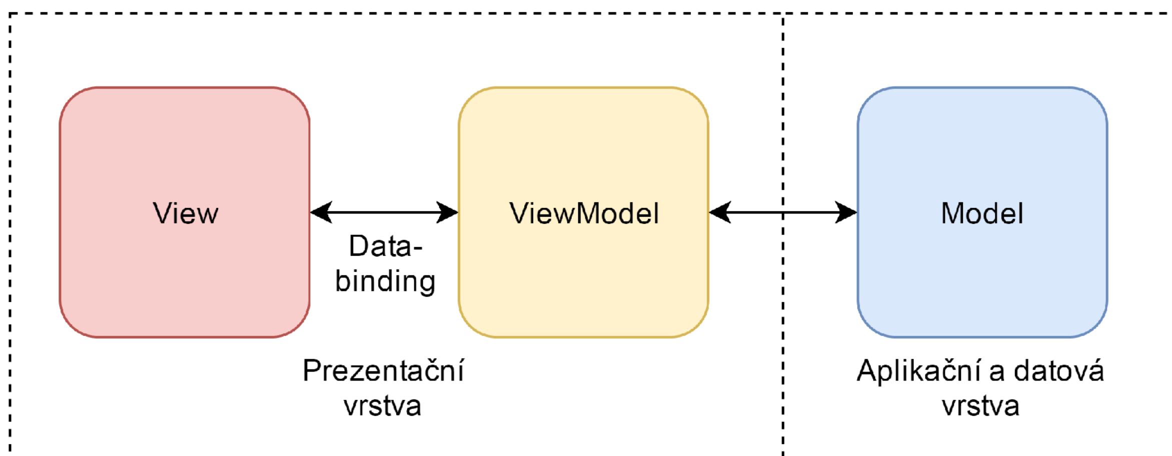
- Microsoft Visual Studio od verze 2008
- Microsoft Visual Studio 2005 s rozšířením Visual Studio 2005 extensions for .NET Framework 3.0 CTP
- Microsoft Blend
- Microsoft Expression Design
- XAMLPad

Nástroje třetí strany:

- SharpDevelop
- PowerBuilder .NET

2. Návrhový vzor MVVM

MVVM (Model-View-ViewModel) je návrhový vzor hojně užívaný při psaní WPF aplikací. Řeší oddělení logiky aplikace od uživatelského rozhraní. Díky tomu je pak kód přehlednější a dá se líp upravovat. Také to umožňuje práci více týmu prakticky nezávisle na sobě. Princip spočívá v rozdělení aplikace na tři bloky: **View**, **Model** a **ViewModel**.



Obrázek 2.1: Schéma návrhového vzoru MVVM

2.1. View

vytváří grafické rozhraní aplikace a jeho úlohou je prezentovat data uživateli ve vzhledné podobě. Samotné View by ideálně nemělo řešit žádnou logiku, někdy by však takové řešení bylo nepřiměřeně složitější, a tak toto pravidlo nemusí být vždy dodrženo.

Ve WPF se View realizuje pomocí vlastního ovládacího prvku (User Control). Ovládací prvek sestává ze dvou souborů – `<název_prvku>.xaml` a `<název_prvku>.xaml.cs`. Oba dva soubory se nacházejí ve stejném jmenném prostoru, a tak navzájem znají všechny své pole, vlastnosti a metody.

Soubor s příponou `.xaml` je napsán v jazyce XAML a popisuje rozmístění prvků na tomto ovládacím prvku a jejich vlastnosti. Hodnoty vlastností může definovat buď přímo, nebo je může vázat na hodnoty jiných vlastností pomocí bindingu.

Druhý soubor (s příponou `.xaml.cs`) se nazývá code-behind ovládacího prvku. V návrhovém vzoru MVVM se nedoporučuje tento soubor využívat pro logické operace, protože tím aplikace ztrácí jednu z hlavních výhod MVVM – oddělitelnost grafické vrstvy od logické.

2.2. Model

Model by měl řešit veškerou logiku aplikace jako například načítání dat z databáze, komunikaci s externími procesy, službami apod.

2.3. ViewModel

ViewModel je oblast, která si uchovává informace o běhu aplikace a ukládá všechna data, která je potřeba zobrazit, do datových struktur, které můžou sloužit jako zdroj bindingu. ViewModel navíc také zpracovává uživatelské akce pomocí příkazů.

2.3.1. Příkazy

Uživatelské akce jsou zpracovávány pomocí příkazů implementujících rozhraní `ICommand`. Toto rozhraní deklaruje metodu `Execute`, funkci `CanExecute` a událost `CanExecuteChanged`.

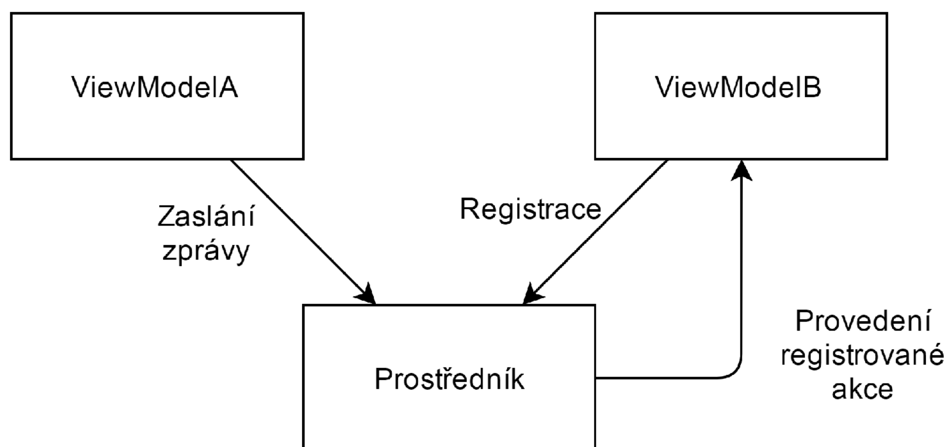
Metoda `Execute` by měla obsahovat funkční tělo, které se má provést při vyvolání příkazu.

Funkce `CanExecute` vracející hodnotu bool udává, zda je možné příkaz spustit.

Událost `CanExecuteChanged` by měla být vyvolána pokaždé, kdy je možné, že se hodnota funkce `CanExecute` změní. Takto implementovaný příkaz se vystaví ve ViewModelu a naváže se na vlastnost `Command` ovládacího prvky, který bude obsluhovat. Hlavní nevýhodou příkazů je, že lze jednoduše nastavit jen na určitých typech ovládacích prvků (např. `Button`). Další nevýhodou je, že přijímají maximálně jeden parametr, což se ale dá obejít vhodným převodníkem.

2.4. Prostředník (Mediátor)

Pro komunikaci mezi ViewModely byl navrhnut návrhový vzor prostředníka. Prostředník je objekt stojící mimo ViewModely schopný mezi nimi předávat zprávy. Díky tomu jsou ViewModely na sobě nezávislé, což vede k flexibilnější aplikaci. Nevýhodou tohoto vzoru je právě to, že ViewModely na sebe navzájem neodkazují, což komplikuje sledování běhu programu při debugování.



Obrázek 2.2: Schéma funkce prostředníka

2.5. Balíček nástrojů MVVM Light

MVVM Light Toolkit je balíček nástrojů pro usnadnění tvorby MVVM aplikací ve frameworcích WPF, Silverlight, Windows Store, Windows Phone a Xamarin. Jedná se o open-source projekt, který je zdarma ke stažení na webové stránce https://mvvmlight.codeplex.com/.

MVVM Light Toolkit implementuje často používané konstrukce, jako jsou například `RelayCommand`, `Messenger` nebo `ViewModelBase`. Také nabízí šablony k vytvoření nových MVVM aplikací.

Třída `RelayCommand` je implementací rozhraní `ICommand`. Vystavuje veřejný konstruktor, jehož parametry jsou delegáty typu `Action` a `Func<bool>`. Prvním parametrem se předává akce, která se má vykonat při spuštění příkazu, druhý parametr je volitelný a předává se jím podmínka spustitelnosti příkazu. Pokud není druhý parametr specifikován, může být příkaz spuštěn kdykoliv.

`Messenger` je vlastní implementace vzoru prostředníka. Poskytuje metody `Send<T>` a `Register<T>`. Všechny třídy se mohou u instance typu `Messenger` metodou `Register<T>` zaregistrovat na příchozí zprávy typu `T`. V případě že jakákoliv jiná třída pomocí této instance `Messenger`u odeslala zprávu metodou `Send<T>`, třídy zaregistrované na tento typ zprávu obdrží a obslouží ji.

Třída `ViewModelBase`, jak už název napovídá, slouží jako bázová třída pro vytvořené ViewModely. Kromě jiných obsahuje také vlastnost `MessengerInstance`, která v základu obsahuje výchozí instanci typu `Messenger`.

3. MySQL

MySQL je systém řízení báze dat využívající relační databáze (zkratka RDBMS – z angličtiny Relation DataBase Management System). Byl vytvořen švédskou firmou MySQL AB v roce 1995 a nyní jej vyvíjí společnost Oracle. MySQL je multiplatformní systém a je vyvíjen jako open-source projekt.

MySQL server se skládá ze tří vrstev. První vrstva obstarává uživatelské rozhraní, aby měl uživatel jak komunikovat s databází. Druhá vrstva obsahuje veškerou logiku systému, jako je třeba analýza a zpracovávání dotazů, optimalizace apod. Třetí vrstva se pak stará o samotné ukládání a čtení dat z databázových úložišť.

Celý systém je odladěn tak, aby byl co nejvýkonnější, mohlo s ním pracovat více uživatelů současně a zároveň aby zůstala všechna data konzistentní. Pro označení požadavků na plnohodnotnou databázi se používá zkratka **ACID** (Atomicity, Consistency, Isolation, Durability).

- **A**tomicity (nedělitelnost) – operace se skládají do **transakcí**, které se provádějí jako jedna nedělitelná operace, takže když by jedna dílčí operace selhala, musí se databáze vrátit do původního stavu.
- **C**onsistency (konzistence) – databáze musí být vždy konzistentní.
- **I**solation (izolace) – operace se navzájem neovlivňují. Pokud by se sešlo víc operací, vykonávají se postupně v pořadí, v jakém přišly.
- **D**urability (trvanlivost) – všechna data jsou zapsaná na trvanlivém úložišti, aby nedocházelo ke ztrátě dat v případě přerušení provozu databáze.

Komunikace mezi uživatelem a databází probíhá pomocí jednoduchého dotazovacího jazyka **SQL**, který je obohacen o některé vlastní funkce.

MySQL nabízí několik databázových úložišť (storage enginů), z nichž nejvýznamnější jsou **InnoDB** a **MyISAM**. Ostatní úložiště nepodporují celou sadu SQL dotazů a bývají využívány pro své specifické účely.

InnoDB je používáno jako výchozí úložiště a ukládají se na něj všechny tabulky, kterým přímo nespecifikujeme úložiště jiné. Tento úložiště podporuje transakce a je oproti MyISAM rychlejší při ukládání a úpravě dat. InnoDB je navíc stále aktivně vyvíjen. Specialitou InnoDB je uzamykání na úrovni řádků [3], čímž umožňuje provádění více operací nad jednou tabulkou současně.

MyISAM je optimalizován pro čtení a vyhledávání v tabulkách, a proto se lépe využije při tvorbě rozsáhlých tabulek, které se tak často nemění. Nevýhodou je, že toto úložiště nepodporuje transakce.

3.1. Relační model databáze

Relační model označuje databázi založenou na tabulkách (relacích), které se skládají z řádků (záznamů), přičemž každý záznam obsahuje určitý počet sloupců (atributů). Každý sloupec má pevně stanovený datový typ (číslo, znak, text, . . .) a může obsahovat jen hodnoty tohoto typu. Každá tabulka by měla obsahovat atribut, pomocí kterého se dá jednoznačně identifikovat záznam, tento atribut se označuje jako primární klíč. Pomocí

cizího klíče je pak možné vyjadřovat vztahy mezi jednotlivými záznamy, a to tak, že do hodnoty atributu označeného se jako cizí klíč se vloží hodnota primárního klíče jiného záznamu.

3.2. Jazyk SQL

Jazyk SQL (Structured Query Language) je strukturovaný dotazovací jazyk pro práci s relačními databázemi. Syntaxe jazyka je účelně podobná syntaxi angličtiny, takže je pro anglicky mluvící lidi celkem snadno čitelný.

Existuje několik standardů jazyka SQL, označené podle roku vydání. Většina databázových systému tyto standardy podporuje, ale ne vždy všechny funkce. Některé systémy naopak přidávají vlastní příkazy, které ve standardech obsaženy nejsou.

Operace se často se dělí do kategorií **DDL**, **DML** a **dotazy**.

DDL (z angličtiny **D**ata **D**efinition **L**anguage) jsou příkazy, které nepostihují jednotlivé řádky tabulek, nýbrž manipulují s celou databází. Patří mezi ně příkazy jako `CREATE`, `ALTER` a `DROP`. Tyto příkazy nelze vrátit zpět.

DML (z angličtiny **D**ata **M**anipulation **L**anguage) jsou příkazy, které provádějí CRUD operace (Create, Read, Update, Delete). Patří mezi ně příkazy `INSERT`, `UPDATE`, `DELETE` a někdy taky `SELECT`[4]. Tyto operace mohou být součástí transakce, a tak může být jejich efekt vrácen.

Dotazy (anglicky queries) jsou příkazy pro čtení dat z jedné či více tabulek. Pokud je dotaz mířený na víc tabulek, označuje se jako **join**. Z historických důvodů se jako dotaz někdy označují i příkazy spadající do předchozích kategorií.

3.2.1. Útok SQL injection

SQL injection (zkráceně SQLI) je technika napadání SQL serverů podstrčením škodlivého kódu přes neošetřený vstup.

SQLI je jedna z nejpoužívanějších technik webového útoku. Většinou se tento útok používá na webových stránkách, má-li uživatel někde zadat například uživatelské jméno nebo ID.

Ukázka zdrojového kódu 3.1: Kód náchylný k SQL injection

```
string userId = Console.ReadLine();
string sqlQuery = "SELECT * FROM Users WHERE UserId = " + userId;
```

Kód v ukázce 3.1 předpokládá, že uživatel zadá uživatelské jméno, podle kterého se vyhledá záznam v databázi. Útočník ale může zadat namísto jména například text
`"1 OR 1=1"`, po jehož dosazení vznikne validní SQL příkaz
`SELECT * FROM Users WHERE UserId = 1 OR 1=1`,
který vypíše všechny záznamy z tabulky `Users`, protože podmínka bude vždy splněna. Pokud by tabulka obsahovala i hesla uživatelů, bezpečnost uživatelů by byla značně narušena.

Kromě vypisování záznamů může útočník provádět i jiné operace, má-li k nim práva. Např. může zadat text `"1; DROP TABLE Users"`, čímž by se po vypsání řádku smazala celá tabulka uživatelů.

Obrana proti SQL injection může být prováděna buď na straně serveru omezením práv uživatelů nebo lépe na straně klienta pomocí kontroly vstupních parametrů.

3.3. Indexování sloupců

Vyhledávání a řazení v rozsáhlých tabulkách jsou časově náročné operace, protože se musí přistupovat k velkému počtu řádků.

Máme-li tabulku o N řádcích, pak pro vyhledání záznamu podle jeho unikátní vlastnosti pomocí lineárního vyhledávání by průměrně bylo potřeba projít $N/2$ řádků tabulky. Pro vyhledání více záznamů by bylo zapotřebí přistoupit ke všem řádkům.

Pro zvýšení výkonnosti vyhledávání a řazení umožňuje MySQL vytváření indexů ke sloupcům v tabulkách. Indexy představují ukazatele na řádky tabulky, které se při vytváření seřadí. Na seřazených seznamech je poté možné aplikovat binární vyhledávání, díky kterému se průměrný počet přístupů k řádkům značně sníží (na $\log_2 N$).

Indexy se vytvářejí jako přidané datové struktury obsahující ukazatel do tabulky o velikosti 2-5 bajtů a hodnotu pole, jehož velikost záleží na datovém typu indexovaného sloupce. Všechny indexy jedné tabulky se ukládají do tabulky indexů používající úložiště MyISAM.

Nevýhodou indexů je, že zabírají další místo na disku a zpomalují přidávání a editaci záznamů, protože se vždy musí upravit i indexy.

Indexy je vhodné používat na sloupce, které nabývají velkého množství rozličných hodnot a podle kterých očekáváme časté vyhledávání nebo řazení. Při velkém množství duplicitních hodnot ve sloupci se snižuje efektivita binárního vyhledávání. Např. pro booleovské proměnné, které nabývají pouze dvou různých hodnot, sníží binární vyhledávání množství dat jen na polovinu, poté se již musí použít vyhledávání lineární.

3.4. MySQL Connector/Net

MySQL Connector/Net je program, který zprostředkovává komunikaci mezi klientskou aplikací a samotnou databází, resp. databázovým systémem. Taktéž jej vyvíjí společnosti Oracle. Umožňuje bezpečné a rychlé připojení MySQL databáze k aplikaci psané v jakémkoliv .NET jazyce. Podporuje Entity Framework, pakety pro posílání a přijímání dat o velikosti až 2GB a connection pooling[5].

3.4.1. Connection pooling

Je pravidlem, že spojení mezi aplikací a databází by mělo být otevřené vždy jen po dobu nezbytnou pro provedení potřebných operací, jinak by mohlo docházet k nežádoucímu blokování dalších spojení na straně serveru. Sestavování spojení je však časově náročné, což by značně zpomalovalo funkci aplikací, které k databázi potřebují přistupovat často.

Connection pooling je metoda sloužící ke zvýšení výkonu a škálovatelnosti takovýchto aplikací. Funguje tak, že spojení, která se v aplikaci uzavřou, se nezruší, ale uloží se do fondu spojení (anglicky connection pool, odsud název connection pooling). V případě otevírání nového spojení se aplikace nejprve podívá do fondu, zda tam takové spojení už není vytvořeno, a případně se použije. Tím se ušetří čas nutný k navazování nového spojení.

V MySQL Connector/Net je connection pooling implicitně aktivní, ale dá se vypnout. Vývojáři říkají, že nejlepší způsob jak se spojeními pracovat, je nechat connection pooling systém, ať se o ně postará sám. Nedoporučují manuální vytváření vlastních spojení, místo toho by se měly používat konstruktory, které v parametru vyžadují připojovací řetězec [6].

4. Entity Framework

Entity Framework (EF) je komplexní řešení pro objektově relační mapování od společnosti Microsoft. Stará se o konverzi dat z relačních databází, kde jsou data vyjádřena jako řádky, do objektů v objektově orientovaném programovacím jazyce. Poskytuje služby jako je sledování změn, opožděné načítání (lazy-loading) nebo překlad dotazů na databázi.

4.1. Verze Entity Freworků

V době psaní tohoto dokumentu existují dvě aktuální verze Entity Frameworku.

4.1.1. Entity Framework 6.X

Entity Framework 6.X (EF6.X) je léty odzkoušená verze, prvně vydaná již v roce 2008. Od té doby se stal nejpoužívanějším Nu-Get balíčkem ve Visual Studiu. Ačkoliv se vývojáři nyní zaměřují spíš na Entity Framework Core, EF6.X jako produkt zůstává stále podporován a budou pro něj dále vydávány opravy chyb a drobná vylepšení [7].

4.1.2. Entity Framework Core

Entity Framework Core (EF Core) je odlehčená, rozšiřitelná a multiplatformní verze Entity Frameworku. EF Core je postavený na zcela novém základu, a tak nenabízí veškeré funkce, které nabízí EF6.X, některé tyto funkce (jako např. lazy-loading) budou postupně implementovány, jiné nikoliv. Na druhou stranu nabízí EF Core několik novinek a vylepšení, jako např. podpora alternativních klíčů nebo generování klíčů na straně klienta.

4.2. Entity Data Model

Pro mapování objektů se používá Entity Data Model (**EDM**), který je napsán v jazyce SDL (Schema Definition Language). Tento model popisuje, jak převést data z databáze do entit a vztahy mezi entitami a jejich vlastnostmi.

Při přidávání nového EDM do Visual Studia jsou nabízeny čtyři možnosti: Empty EF Designer model pro přístup Model First, Empty Code First model a Code First from database pro přístup Code First a EF Designer from database pro přístup Database First.

4.3. Kontextová třída

Pří vytvoření modelu EDM se vygeneruje třída, která dědí ze třídy `DbContext`. Tato třída se označuje jako kontextová třída a slouží jako most mezi třídami entit a databází. Je hlavní třídou zodpovědnou za následující funkce:

- Obsahuje sety entit, které se mapují do tabulek v databázi
- Překládá dotazy a posílá je do databáze
- Sleduje změny provedené na entitách po jejich načtení z databáze

- Je schopna ukládat, upravovat a mazat záznamy v databázi
- Ukládá záznamy, které již byly vytaženy z databáze
- Spravuje reference
- Převádí data z databáze do entit

4.4. Entity

V EF 6.0 existují dva typy entit: POCO entity a dynamické proxy entity.

4.4.1. POCO entity

Zkratka POCO (Plain Old CLR Object) značí instanci obyčejné třídy, která není závislá ne externím frameworku. POCO entity podporují většinu akcí a chování, jako entity vygenerované modelem EDM.

4.4.2. Dynamic Proxy entity

Dynamic Proxy entity (POCO proxy) jsou něco jako schránky POCO entit. POCO proxy navíc podporují lazy-loading a automatické sledování změn, díky čemuž sama ví, zda byla v současném kontextu změněna.

Aby se POCO entity za běhu aplikace staly Dynamic Proxy entitami, musí splňovat tato kritéria:

- Třída entity musí být deklarována s modifikátorem přístupu `public`.
- Ze třídy entity musí být možné dědit, tzn. **nesmí** mít modifikátor přístupu `sealed`.
- Třída entity nesmí být abstraktní.
- Všechny navigační vlastnosti musí být deklarovány jako `public` a `virtual`.
- Všechny kolekce musí být typu `ICollection<T>`.
- Vlastnost `ProxyCreationEnabled` v kontextové třídě musí být nastavena na `true` (v základu tak je).

4.4.3. Vlastnosti entit

Entita může obsahovat vlastnosti buď **skalární** nebo **navigační**.

Skalární vlastnosti jsou vlastnosti, jejichž hodnoty jsou obsaženy přímo v entitě. Mezi tyto vlastnosti se řadí jednoduché typy jako čísla, textové řetězce apod.

Navigační vlastnosti představují ukazatele na jiné entity.

4.5. Přístupy k databázi

Entity Framework podporuje tři různé přístupy pro vývoj aplikací s Entity Frameworkem.

4.5.1. Model First

Přístup Model First spočívá v návrhu databáze pomocí schémata modelu EDM. Podle schémata se vygenerují třídy entit s požadovanými vlastnostmi a samotná databáze. Ve Visual Studiu je pro editaci schémata k dispozici nástroj Entity Designer.

4.5.2. Database First

Přístup Database First se používá, je-li databáze již vytvořena. Database First spočívá v tom, že se schéma EDM nechá vygenerovat podle databáze. Model se může aktualizovat při změně struktury databáze. Tento přístup také podporuje mapování uložených procedur, pohledů a jiných věcí z databáze.

4.5.3. Code First

Při přístupu Code First se vůbec nepracuje se schématem EDM modelu, stačí jen navrhnout třídy pro entity a databázový kontext. Databázi pak lze podle nich vygenerovat.

Např. kdybychom chtěli databázi s jedinou tabulkou Osoby, kde by každý záznam představoval jednu osobu s vlastnostmi Jmeno, Pohlavi a Vek, pak by bylo vhodné vytvořit typ entity Osoba s těmito vlastnostmi. Každá instance této třídy by modelovala právě jednu entitu, tedy osobu.

Část II

Praktická část

5. Struktura aplikace

Aplikace používá třívrstvou architekturu a je psaná podle návrhového vzoru MVVM.

5.1. Třívrstvá architektura aplikace

Aplikace používá třívrstvou architekturu s volným vrstvením, přičemž každé vrstvě odpovídá jeden projekt.

5.1.1. Prezentační vrstva

Vrstva je realizována projektem **DatabaseEditor.App**. Ten zodpovídá za vzhled aplikace, zobrazování dat a komunikaci s uživatelem.

5.1.2. Aplikační vrstva

DatabaseEditor.BLL je projekt pro zpracovávání aplikační logiky (BLL je zkratka pro Bussiness Logic Layer). Obsahuje prostředky pro virtualizaci, filtrování, a vyhledávání dat. Také se stará o mapování entit do modelů, se kterými se pracuje ve vyšších vrstvách.

5.1.3. Datová vrstva

Projekt **DatabaseEditor.DAL** představuje vrstvu pro přístup k databázi (Data Access Layer). Obsahuje třídy entit, kontextovou třídu, třídu pro inicializaci databáze a třídu `Randomizer` pro generování záznamů.

5.2. Rozložení podle MVVM

Model se skládá z projektů DatabaseEditor.BL a DatabaseEditor.DAL. View a ViewModely jsou v projektu DatabaseEditor.App v příslušných složkách.

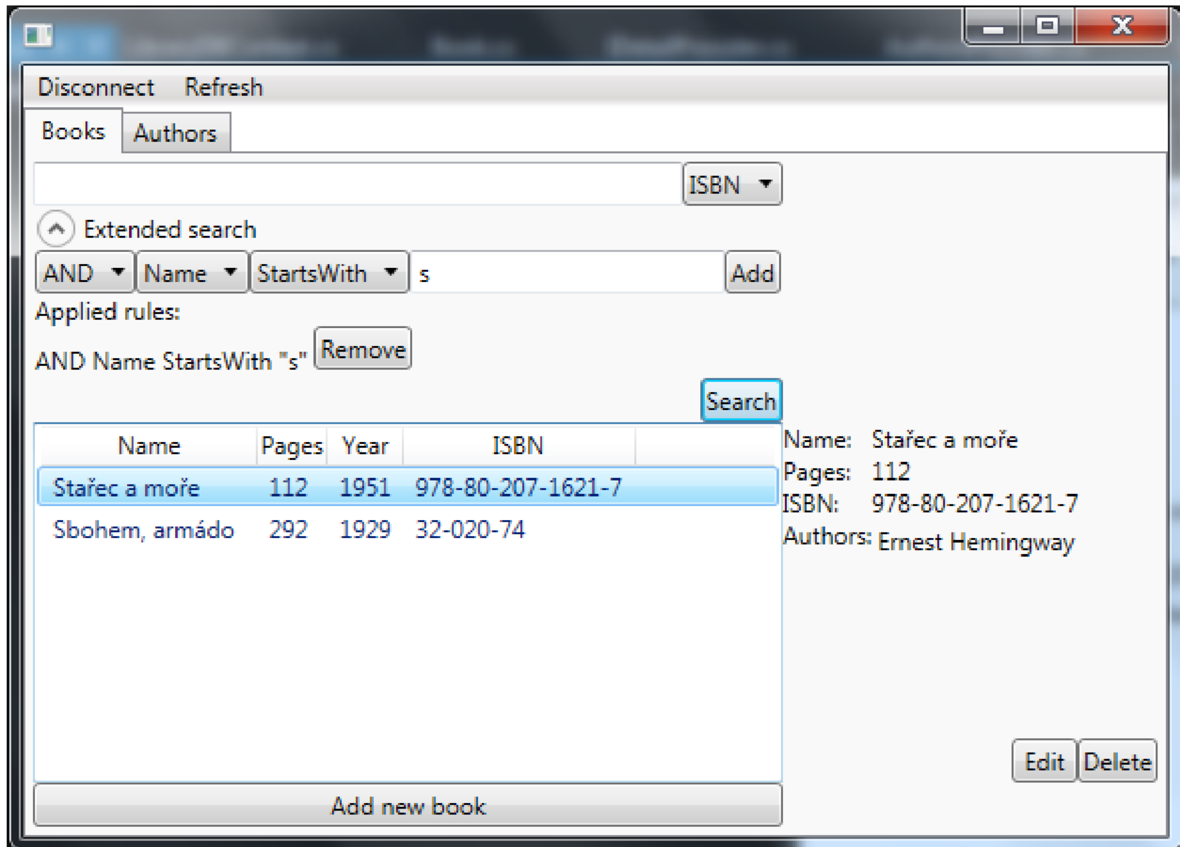
5.3. Grafická struktura

Aplikace se skládá z několika pohledů. Většina pohledů má dvě varianty, jednu pro knihy a druhou pro autory.

5.3.1. MainWindow

Primární pohled, na kterém jsou umístěny hlavní ovládací prvky jako je menu nebo ovládací prvek karty `TabControl`. V menu se nachází tlačítka *Connect* pro připojení k databázi a *Refresh* k aktualizaci tabulky. Na kartách *Books* a *Authors* jsou umístěny prvky `BooksPageView` a `AuthorsPageView`. Datový kontext hlavního okna je navázán na instanci třídy `MainViewModel`.

Přes celý pohled je umístěn ovládací prvek `CreateDatabaseDialog`, který se zobrazuje jen v případě dotazu na vytvoření databáze, když není nalezena. Jeho viditelnost je navázána na vlastnost `CreateDatabaseDialogVisible` ViewModelu.



Obrázek 5.1: Hlavní okno aplikace

5.3.2. BooksPageView a AuthorsPageView

Tyto pohledy slouží k zobrazení dat z databáze. `BooksPageView` je umístěn na kartě *Books* a `AuthorsPageView` na kartě *Authors*. Dominantním prvkem je tabulka `GridView` sloužící k zobrazování seznamu autorů nebo knih. Nad tabulkou se nachází ovládací prvek `BookFilterView`/`AuthorFilterView` pro vyhledávání a filtrování záznamů, vpravo od tabulky se zobrazuje detail právě označeného záznamu pomocí pohledů `BookDetailView` a `AuthorDetailView`, a pod tabulkou je tlačítko pro přidání nového záznamu. ViewModelem těchto pohledů je `PageViewModel<TListModel, TDetail>`.

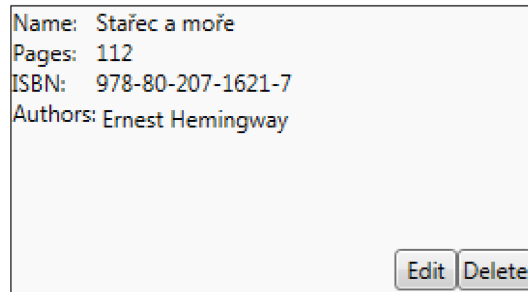
Podobně jako má hlavní okno skrytý dialog na vytváření databáze, tyto pohledy disponují skrytými dialogy pro editaci záznamů `EditBookView`/`EditAuthorView`.

5.3.3. BookDetailView a AuthorDetailView

Slouží k zobrazení detailních informací o záznamu a umožňují jeho editaci nebo odebrání pomocí příslušných tlačítek *Edit* a *Delete*. Oba pohledy používají jako ViewModel třídu `ItemDetailViewModel<TListModel, TDetail>`.

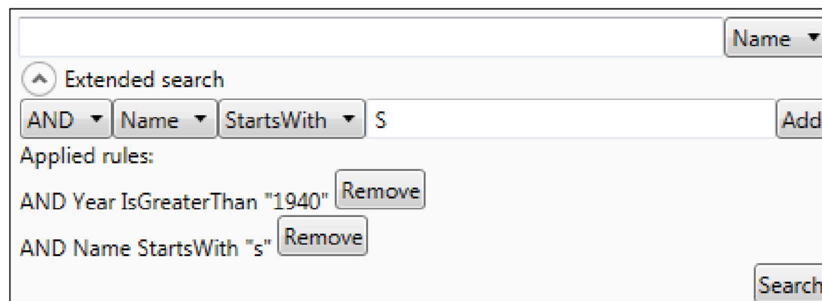
5.3.4. BookFilterView a AuthorFilterView

Pohledy filtrů slouží k vyhledávání a filtrování záznamů. Obsahují textové pole pro rychlé vyhledávání, pod nímž leží rozbalovací část s možností vytváření vlastních pravidel. Jako ViewModel využívají tyto pohledy `FilterViewModel<T>`.



Obrázek 5.2: Zobrazení detailu záznamu knihy

Všechny položky polí se seznamem (ovládací prvek `ComboBox`) v tomto pohledu se plní dynamicky podle typu `T` použitého při vytváření ViewModelu a podle typu zvolené vlastnosti.



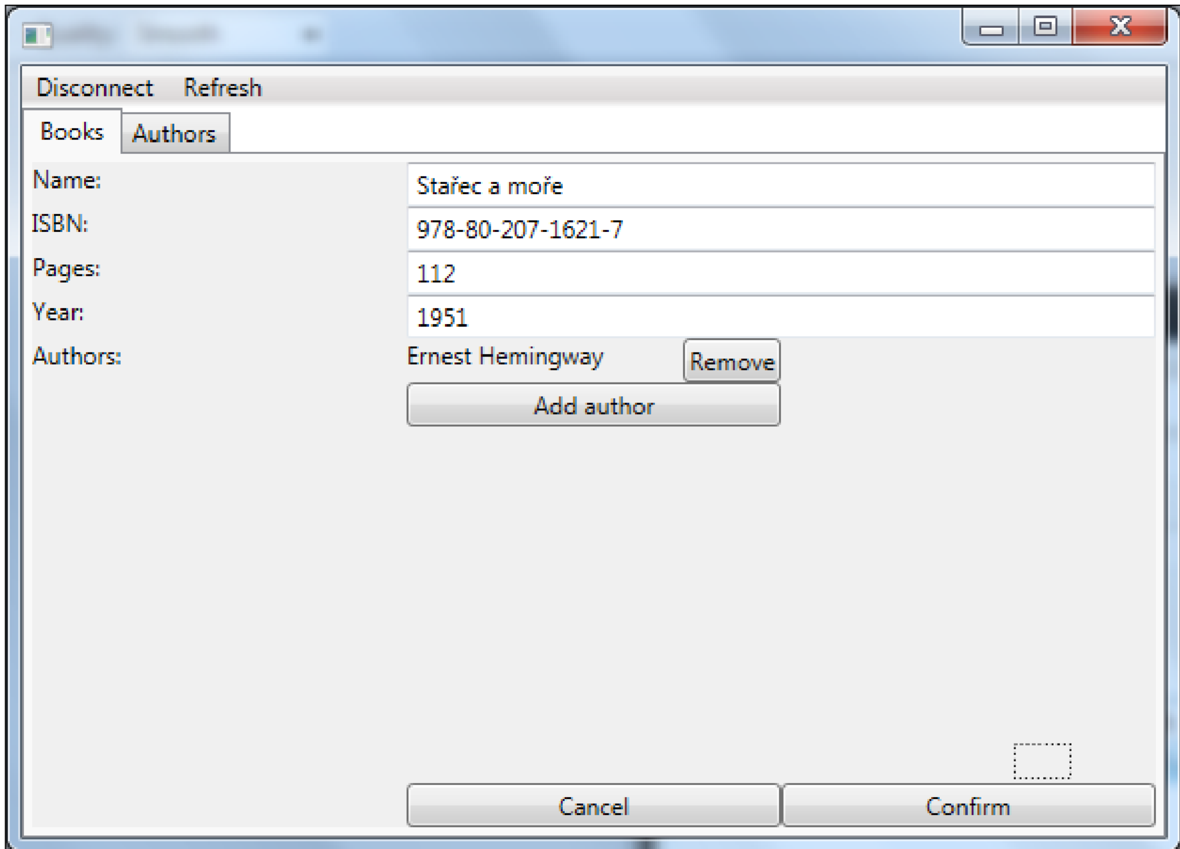
Obrázek 5.3: Pohled pro filtrování knih

5.3.5. EditBookView a EditAuthorView

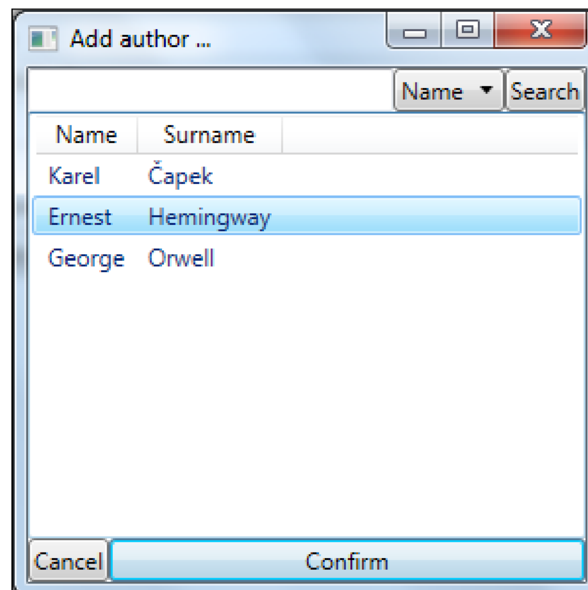
Tyto pohledy umožňují editaci vlastností záznamu. Jak u seznamu autorových knih, tak i u autorů knihy jsou tlačítka na jejich přidání a odebrání. Vespod zobrazení se nachází tlačítka na potvrzení a na zrušení editace. Jako ViewModel je zde použita třída `EditItemViewModel<TDetail, TItem>`.

5.3.6. AuthorSelectWindow a BookSelectWindow

Okno `BookSelectWindow` se zobrazí při žádosti o přidání nové knihy k záznamu autora. Obdobně je tomu u druhého okna, které se pro změnu může zobrazit při editaci knihy. Obsahují vyhledávací pole podobně jako `BooksPageView`, resp. `AuthorsPageView`.



Obrázek 5.4: Pohled pro editaci záznamu knihy



Obrázek 5.5: Okno pro přidání autora k záznamu knihy

6. MySQL databáze

K vytvoření databáze jsem si nainstaloval databázový systém MySQL server, MySQL Connector/Net a pro objektově-relační mapování jsem použil Entity Framework 6.1.

6.1. Instalace MySQL serveru

Na webové stránce `http://dev.mysql.com/downloads/mysql/` je k nalezení nejnovější verze databáze MySQL pro nekomerční použití. Při instalaci jsem postupoval podle pokynů instalátoru. Pomocí instalátor jsem spolu s MySQL serverem nainstaloval mimo jiné i MySQL Connector/Net a Entity Framework.

6.2. Tvorba databáze

Databázi jsem vytvořil pomocí Entity Frameworku 6.1 přístupem Code First. K tomu bylo zapotřebí do projektu nainstalovat Nu-Get balíček `MySql.Data.Entity`, se kterým se automaticky nainstaloval i balíček `EntityFramework`.

6.2.1. Návrh entit

V projektu DatabaseEditor.DAL jsem vytvořil složku Entities, a v ní třídy `Author` a `Book`. Tyto třídy představují třídy entit, se kterými budu později pracovat. Instance třídy `Author` představuje entitu autora a instance třídy `Book` entitu knihy.

Ukázka zdrojového kódu 6.1: Třída entity Author

```
public class Author
{
    public Author()
    {
        Books = new HashSet<Book>();
    }

    public Guid AuthorId { get; set; }

    [Required]
    [Index]
    [MaxLength(20)]
    public string Name { get; set; }

    [Required]
    [Index]
    [MaxLength(15)]
    public string Surname { get; set; }

    public virtual ICollection<Book> Books { get; set; }
}
```

Ukázka zdrojového kódu 6.2: Třída entity Book

```
public class Book
{
    public Book()
    {
        Authors = new HashSet<Author>();
    }

    public Guid BookId { get; set; }

    public string ISBN { get; set; }

    [Required]
    [Index]
    [MaxLength(25)]
    public string Name { get; set; }

    public int Pages { get; set; }

    public int Year { get; set; }

    public virtual ICollection<Author> Authors { get; set; }
}
```

Obě třídy entit splňují všechny podmínky, aby se za běhu programu vytvářely entity typu Dynamic Proxy.

Jak je vidět, každý autor může mít referenci na více knih a každá kniha může mít referenci na více autorů – je mezi nimi vzájemný vztah many-to-many. V relačních databázích se tento vztah řeší tzv. asociační tabulkou, která obsahuje cizí klíče obou záznamů.

Vlastnosti, podle kterých očekávám časté vyhledávání nebo řazení jsem označil atributem `Index`. Ten způsobí, že se při vytváření databáze vytvoří i tabulka s indexem označené vlastnosti.

6.2.2. Kontextová třída

Po dokončení návrhu entit jsem vytvořil kontextovou třídu s názvem `LibraryDbContext`, která dědí ze třídy `DbContext`.

Ukázka zdrojového kódu 6.3: Kontextová třída `LibraryDbContext`

```
[DbConfigurationType(typeof(MySqlEFConfiguration))]
public class LibraryDbContext : DbContext
{
    public LibraryDbContext(string connectionString, int
       authorsCount, int booksCount, int minAuthorsBooks, int
       maxAuthorsBooks)
        : base(connectionString)
    {
        Database.SetInitializer(new LibraryDbInitializer(
           authorsCount, booksCount, minAuthorsBooks,
           maxAuthorsBooks));
        Database.Initialize(false);
    }

    public LibraryDbContext(string connectionString)
        : base(connectionString)
    {
    }

    public LibraryDbContext(DbConnection existingConnection, bool
        contextOwnsConnection)
        : base(existingConnection, contextOwnsConnection)
    {
    }

    public virtual DbSet<Author> Authors { get; set; }
    public virtual DbSet<Book> Books { get; set; }
}
```

Atribut `[DbConfigurationType(typeof(MySqlEFConfiguration))]` slouží pro nastavení konfigurační třídy MySQL. Vlastnosti `Authors` a `Books` typu `DbSet` představují tabulky autorů, resp. knih, v databázi. Ve třídě jsem definoval tři konstruktory, přičemž každý používám ke specifickým účelům.

První konstruktor slouží pro inicializaci databáze a vygenerování testovacích dat. V těle konstruktoru se nastaví inicializátor databáze s parametry, které mu oznamují, kolik záznamů jakých typů má vygenerovat. Poté se jen zavolá metoda pro inicializaci

databáze s parametrem `false`, jenž značí, zda se má databáze vytvořit i kdyby již existovala.

Druhý konstruktor používám pro připojení k existující databázi. K jeho použití není potřeba mít vytvořené spojení s databází – spravuje si je sám. Parametrem je připojovací řetězec pro připojení k databázi.

Třetí konstruktor vyžaduje v parametru již otevřené spojení a používám jej jen pro zjištění, jestli databáze již existuje či nikoliv.

6.2.3. Inicializace databáze

Pro inicializaci databáze je nutné nastavit inicializátor. Lze použít jeden z existujících inicializátorů nebo si můžeme vytvořit vlastní, který z nějakého existujícího dědí. Já jsem si vytvořil třídu `LibraryDbInitializer`, která dědí z již existujícího inicializátoru `DropCreateDatabaseIfModelChanges<LibraryDbContext>`. Bázový inicializátor znovu vytvoří databázi pokaždé, když se změní model nebo cílová databáze není nalezena. Vystavuje virtuální metodu `Seed`, jež slouží k naplnění databáze při její inicializaci. Tuto metodu jsem přetížil vlastní implementací pro vygenerování záznamů a jejich vložení do databáze.

6.2.4. Generování záznamů

Ke generování entit jsem si vytvořil statickou třídu `Randomizer`. Ta poskytuje funkce `RandomString`, `RandomText`, `RandomInt` a hlavně `RandomBook` a `RandomAuthor`.

Funkce `RandomText` vrací řetězec o požadované délce složený pouze z velkých písmen anglické abecedy. Výstup funkce `RandomString` může obsahovat kromě písmen i číslice 0-9. Funkce `RandomInt` vrací náhodné číslo ze zadaného intervalu. Těchto tří funkcí pak využívají funkce `RandomBook` a `RandomAuthor`, které vytváří nové entity s náhodně naplněnými vlastnostmi, kromě autorů u knížek a naopak.

V metodě `Seed` se nejprve vygeneruje kolekce knih, poté se vygeneruje kolekce autorů a každému se vzápětí přiřadí jedna až tři knihy. Všechny entity se postupně po blocích vkládají do kontextu a pomocí metody SaveChanges ukládají do databáze. Vkládání většího počtu záznamů však trvá enormně dlouhou dobu (viz tabulku 8.3).

7. Logika aplikace

7.1. Modely entit

Ve vrchní vrstvě programu nepracuji přímo s entitami, ale pouze s vlastními modely (též nazývanými jako data access object, zkráceně **DAO**), které poskytují pouze omezené množství informací. Pokud bych totiž v prezentační vrstvě aplikace přistoupil k referenčním vlastnostem entity, které se automaticky nenačítají, hrozil by pád aplikace.

Bylo potřeba vytvořit dva typy modelů – jeden plnohodnotný, obsahující všechny vlastnosti entity, a druhý odlehčený pro modelování záznamů v tabulce. Na místo referenčních vlastností detailních modelů jsem taky použil odlehčené modely, protože jinak by se při načítání těchto vlastností mohla spolu s nimi načíst i celá databáze.

Odlehčené modely jsou reprezentovány třídami `AuthorListModel` a `BookListModel` a pro zobrazení detailů jsou použity třídy `AuthorDetailModel` a `BookDetailModel`. Modely detailů dědí ze seznamových modelů, a ty dědí z bázové třídy `ItemModelBase`, která obsahuje pouze vlastnost `Id`.

Některé vlastnosti modelů jsou označeny atributem `SearchablePropertyAttribute`, jež je využit k identifikaci vlastností, podle kterých se dá vyhledávat.

7.1.1. Mapovač

Pro převádění mezi DAO a entitami jsem si vytvořil vlastní třídu `Mapper`. Ta poskytuje metody na vytvoření modelů z entit.

7.2. Virtualizující kolekce

`VirtualizingCollection` je kolekce schopná virtualizovat data. Její řešení bylo inspirováno projektem uživatele Paul McClean [8].

Využívá metodu stránkování s volitelnou velikostí stránek. V aplikaci jsem zvolil velikost stránky odhadem na 100 záznamů. Pro svoji funkčnost využívá zapouzdřeného objektu poskytovatele dat implementujícího rozhraní `IItemsProvider`, který vedle jiných implementuje i funkci `FetchRange` pro načtení rozsahu záznamů. Ta se používá pro plnění stránek. V případě přístupu ke spodnímu okraji stránky se pomocí `FetchRange` z databáze načte následující stránka. Ve výsledku slouží `VirtualizingCollection` jako zdroj dat pro tabulku `GridView`, proto implementuje rozhraní `IList` a `IList<T>`.

Metoda `GetEnumerator` by měla vracet kolekci všech prvků ve zdroji dat, což by u rozsáhle databáze způsobilo značné zpoždění. Řešením by bylo tuto metodu implementovat a vyhnout se jejímu používání. Bohužel při poskytnutí této kolekce jako zdroje dat pro `GridView`, se metoda `GetEnumerator` volala automaticky, záměrně jsem ji tedy implementoval tak, aby vracela prázdnou kolekci. Naštěstí jsem nezpozoroval žádný problém, jež by tato implementace způsobovala. Ovládací prvek obsahující virtualizující kolekci navíc nesmí být umístěna v prvku s automatickou výškou (např. `GridRow` s výškou `"Auto"`), jinak se aplikace pokouší změřit výšku prvku, přičemž prochází všechny prvky v kolekci.

7.3. Poskytovatele záznamů

V projektu jsem vytvořil dva druhy poskytovatelů záznamů – jeden pro poskytování seznamových modelů `IItemsProvider<TListModel>` a druhý pro poskytování detailních modelů `IDetailProvider<TDetail>`.

Obě dvě rozhraní k poskytování záznamů jsou v aplikaci implementována pouze ve třídě `MySQLItemProvider<TListModel, TDetail>`, která slouží jako bázová třída pro třídy `AuthorsProvider` a `BooksProvider`. Původně jsem se snažil tyto třídy sjednotit do jedné generické, ale ukázalo se, že by implementace takové třídy byla příliš složitá.

7.3.1. Poskytovatel záznamů kolekce

Rozhraní `IItemsProvider<TListModel>`, slouží k poskytování záznamů typu `TListModel` virtualizující kolekci ze zdroje dat. Deklaruje funkce `FetchRange`, `FetchCount`, `Contains`, `UseConditions`, `DeleteItem` a vlastnost `SourceChanged`.

Funkce `FetchRange` vrací požadovaný rozsah záznamů ze zdroje dat. V případě řazení nebo filtrování musí být záznamy již seřazeny, resp. vyfiltrovány.

Pomocí funkce `FetchCount` se získá počet záznamů v databázi. Opět tady platí, že při filtrování záznamů musí tato funkce vracet jen počet záznamů splňujících kritéria.

Funkce `DeleteItem` slouží ke smazání záznamu v databázi a může být volána buď s parametrem typu `TListModel` představujícím celý záznam nebo s identifikačním číslem typu `Guid`.

Vlastnost `SourceChanged` slouží ke zjišťování, zda byla data ve zdroji změněna.

7.3.2. Poskytovatel detailů záznamů

Rozhraní `IDetailProvider<TDetail>` poskytuje operace, pro práci s detailními modely entit. Deklaruje pouze funkci `GetItemById` a metodu `UpdateOrInsert`.

Funkce `GetItemById` získá z databáze entitu podle jejího primárního klíče `Id` včetně jejích referenčních vlastností a vytvoří pomocí ní model typu `TDetail`, který předá návratovou hodnotou.

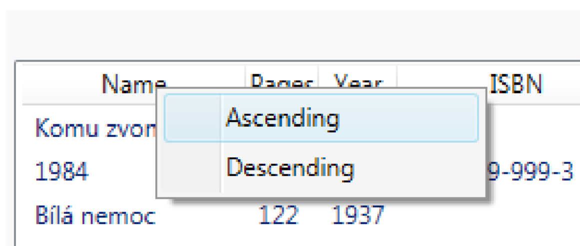
Metoda `UpdateOrInsert` slouží k zapsání záznamů představovaného detailním modelem do zdroje dat.

7.3.3. Poskytovatel MySQL záznamů

Pro získávání a poskytování záznamů z MySQL databáze jsem vytvořil abstraktní třídu `MySQLItemProvider<TListModel, TDetail>`, která implementuje obě rozhraní poskytovatelů záznamů: `IItemsProvider<TListModel>` i `IDetailProvider<TDetail>`. Původně tato třída měla sloužit jako generický zdroj dat, který by poskytoval záznamy autorů nebo knih podle typových parametrů. Nicméně z této představy sešlo, protože by bylo nutné typovým parametrem předat i typ entity, který by v prezentační vrstvě neměl být znám. `MySQLItemProvider` se tak stal šablonou pro dvě implementace – jednu pro poskytování autorů, druhou pro poskytování knih (`AuthorsProvider` a `BooksProvider`).

7.4. Řazení

Seřazení se provede vyvoláním kontextového menu na záhlaví požadovaného sloupce a zvolením směru řazení.



Obrázek 7.1: Kontextové menu pro řazení sloupců

Například kliknutím na směr řazení *Ascending* se vyvolá příkaz `SortTableAscCommand` s parametrem názvu sloupce, na kterém bylo řazení vyvoláno.

Název sloupce, jsem zjistil pomocí bindingu na obsah elementu, na kterém je kontextové menu umístěno (viz ukázku 7.1).

Ukázka zdrojového kódu 7.1: Zjištění názvu sloupce z kontextového menu

```
<MenuItem Header="Ascending"
          Command="{Binding SortTableAscCommand}"
          CommandParameter="{Binding RelativeSource=
                           {RelativeSource FindAncestor,
                              AncestorType={x:Type ContextMenu}},
                           Path=PlacementTarget.Content}" />
```

`SortTableAscCommand` spustí metodu `SortTable` s parametry řazeného sloupce a směru řazení. `SortTable` jen zavolá metodu `SortBy` na poskytovateli záznamů a poskytovatel záznamů si podle požadovaného nastavení upraví způsob dotazování na databázi. Klasicky by se řazení provádělo funkcí `OrderBy` s parametrem vracejícím vlastnost entity. Bohužel, prezentační vrstva nezná entity, a proto ani nemůže předat tento parametr. Proto jsem tady využil knihovnu `System.Linq.Dynamic` umožňující řazení podle názvu sloupce. Samotné řazení řeší až samotný databázový systém.

7.5. Filtrování a vyhledávání záznamů

Filtrování dat je řešeno pomocí podmínek. Každá podmínka obsahuje logický operátor, název vlastnosti, hodnotový operátor a hodnotu.

Uživatel má možnost vytvářet vlastní podmínky, podle kterých jsou následně záznamy filtrovány. Dále je možné filtrování pomocí vyhledávacího pole. Při změně textu ve vyhledávacím poli se automaticky vytvoří podmínka s hodnotovým operátorem `StartsWith` a vlastností vybranou v roletovém menu.

Kliknutím na tlačítko *Search* se všechna vytvořená pravidla vyšlou prostředníkem ve zprávě `FilterChangedMessage`, po jejímž zachycení se podmínky předají příslušnému poskytovateli záznamů, a ten si pomocí nich vytvoří filtr (třída `Filter`). Filtr z podmínek vytvoří textový řetězec, který se použije jako predikát ve funkci `Where`.

7.5.1. Struktura podmínky

Pro modelování podmínek jsem vytvořil abstraktní třídu `PropertyCondition`.

Ukázka zdrojového kódu 7.2: Třída PropertyCondition

```
public abstract class PropertyCondition<T>
{
    public LogicOperator LogicOperator { get; }

    public string PropertyName { get; }

    public abstract string OperatorText { get; }

    public object PropertyValue { get; }

    protected PropertyCondition(LogicOperator logicOperator,
       string propertyName, object propertyValue)
    {
        LogicOperator = logicOperator;
        PropertyName = propertyName;
        PropertyValue = propertyValue;
    }

    public abstract string Predicate { get; }
}
```

`LogicOperator` značí, zda je nutné toto pravidlo dodržet. Logický operátor je stejnojmenného výčtového typu `LogicOperator`, který může nabývat hodnot AND (logický součin) nebo OR (logický součet). Tohoto operátoru se využívá ke slučování více podmínek do jedné funkce.

Vlastnost `PropertyName` udává jméno vlastnosti, na kterou se podmínka má vztahovat. Hodnotové operátory jsou vytvářeny v každé implementaci `PropertyCondition` individuálně, vlastnost `OperatorText` slouží jen k jeho textové reprezentaci. Hodnota vlastnosti `PropertyValue` je typu `object` a udává hodnotu, se kterou by se mělo v podmínce operovat. Textový řetězec `Predicate` slouží jako reprezentace této podmínky pro použití jako predikátu pro filtrování záznamů z databáze.

Vytvořil jsem dvě implementace `PropertyCondition`: `IntCondition` pro celočíselné hodnoty a `StringCondition` pro textové hodnoty. `StringCondition` nabízí pouze operátory `Equals`, `NotEquals`, `Contains` a `StartsWith`. `IntCondition` nabízí jak operátory pro porovnávání hodnot, tak operátor `StartsWith`, který funguje na textové bázi.

Typový parametr `T` slouží pouze pro usnadnění identifikace ViewModelu, kterému mají přicházet zprávy o změně podmínek – každý ViewModel se zaregistruje jen na podmínky typu, které jsou určeny pro něj.

7.5.2. Filtr

Filtr je třída, která se stará o sloučení podmínek do jednoho textového řetězce. Ten je vystaven jako vlastnost `WherePredicate` a je použit v poskytovateli záznamů jako parametr funkce `Where` z knihovny `System.Linq.Dynamic`.

7.5.3. Skládání podmínek

V případě více podmínek se kontrolují podmínky postupně. Mějme podmínky A, B, C a D s následujícími logickými operátory:

1. AND A
2. OR B
3. AND C
4. OR D

Podmínky vracejí hodnotu pravda, když je podmínka splněna, nebo nepravda, když podmínka splněna není. Podmínky se složí do logického výrazu podle jejich logických operátorů. AND značí logický součin a OR značí logický součet. Výsledný výraz by vypadal následovně:

$f(A, B, C, D) = ((A + B) \cdot C) + D = A \cdot B + A \cdot C + D.$

Z toho vyplývá, že výraz by byl splněn, pokud by platila alespoň jedna z kombinací podmínek A a B, A a C nebo D.

7.6. Přidávání a editace záznamů

Přidávání záznamů se provádí stiskem tlačítka *Add new book*, resp. *Add new author*. Jeho stisknutím se zobrazí dialog, umožňující vyplnění vlastností. Potvrzením editace tlačítkem *Confirm* se zašle zpráva `ConfirmEditMessage` obsahující editovaný záznam. Zpráva je zachycena v `PageViewModel`, kde se editovaný záznam vloží do kolekce `EditedItems` a zavolá se metoda `Commit`, která uloží všechny `EditedItems` do databáze pomocí metody `IDetailProvider.UpdateOrInsert`. Editace záznamu funguje na stejném principu a vyvolává se tlačítkem *Edit* na detailu záznamu. Otevře se stejný dialog jako u nového záznamu, akorát předvyplněn vlastnostmi editovaného záznamu. Jediná vlastnost, kterou editovat nelze je `Id`. Její editací by nebylo možné zpětně identifikovat původní záznam v databázi a tak by se vytvořil nový záznam, případně by se editoval jiný.

7.6.1. Přidávání záznamů

Jak už jsem zmínil, přidávání záznamů probíhá v metodě `UpdateOrInsert` poskytovatele záznamů. Tato metoda je implementovaná ve třídách `AuthorsProvider` a `BooksProvider` prakticky stejným způsobem. Pro zkrácení uvedu jen implementaci v `AuthorsProvider`.

Nejprve je nutné vytvořit entitu autora, kterého chceme přidat. Ta se vytvoří pomocí funkce `EntityFromDetailModel`, která mimo jiné i vyhledá odpovídající entity knih autora. Nyní se najde v databázi původní záznam autora podle jeho `Id` a uloží se do proměnné `authorInDb`. Pokud se původní záznam nenajde vytvoří se nový metodou `dbContext.Add`. Jelikož se však knihy nového autora načítaly v jiném kontextu, je třeba je manuálně načíst i do tohoto metodou `Attach`.

7.6.2. Editace záznamů

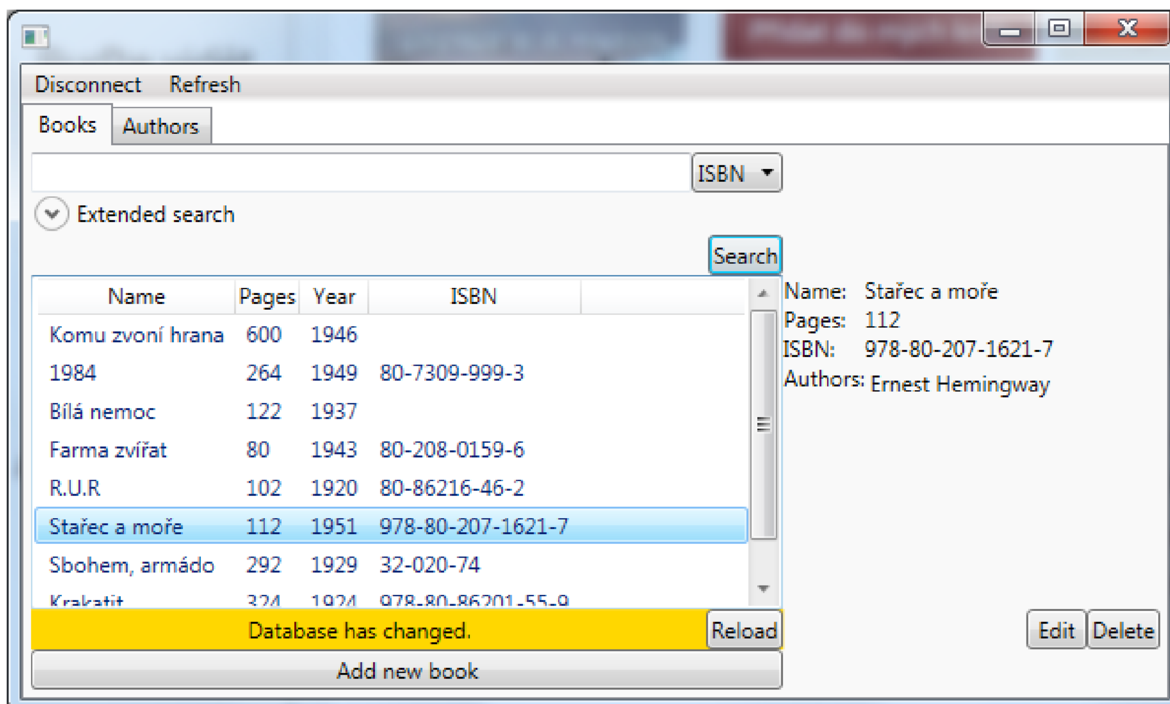
V případě, že se přidávaný autor v databázi najde, editují se jeho vlastnosti. Aktualizace skalárních dat záznamu v databázi se provádí zápisem

`dbContext.Entry(authorInDb).CurrentValues.SetValues(newAuthor);`.

Tímto přístupem se však neaktualizují navigační vlastnosti záznamu. Tento problém jsem vyřešil tak, že se nejprve u autora v databázi odstraní knihy, které v nové entitě nejsou, a poté se přidají knihy, které jsou naopak právě jen v nové entitě. K přidávání knih je opět nutné je načíst do kontextu metodou `Attach`. Nakonec se změny uloží metodou `SaveChanges`.

7.7. Synchronizace více spuštěných aplikací

Aplikace je schopná zjistit, kdy byla provedena poslední změna v databázi. V případě, že má aplikace načtená neaktuální data, pod tabulkou se zobrazí notifikace, pomocí níž je možné tabulku aktualizovat.



Obrázek 7.2: Notifikace změny v databázi

Na změny se aplikace dotazuje každých pět vteřin dotazem na informační tabulku MySQL serveru. Tato informace byla původně poskytována jen úložištěm MyISAM, od

verze MySQL serveru 5.7.2 je však dostupná i pro úložiště InnoDB [9].

Ukázka zdrojového kódu 7.3: Metoda pro získání času poslední aktualizace tabulky

```
private DateTime? GetLastUpdateTime(Type type)
{
    using (var dbContext=new LibraryDbContext(ConnectionString))
    {
        var dbName = GetDatabaseName(dbContext);
        var tableName = GetTableName(type, dbContext);
        return ((IObjectContextAdapter) dbContext)
               .ObjectContext.ExecuteStoreQuery<DateTime?>(
                 "SELECT UPDATE_TIME " +
                 "FROM   information_schema.tables " +
                 "WHERE  TABLE_SCHEMA = @p0 " +
                 "AND TABLE_NAME = @p1", dbName, tableName)
                 .FirstOrDefault();
    }
}
```

Notifikace je zobrazována pomocí řádku `GridRow`, jehož výška je navázána na boolovskou proměnnou `DatabaseChanged`. K překladu z boolovské hodnoty na proměnnou typu `GridLength` jsem si vytvořil konvertor `BoolToGridLengthConverter`.

Ukázka zdrojového kódu 7.4: Převodník z boolovské proměnné na rozměr mřížky

```
[ValueConversion(typeof(bool), typeof(GridLength))]
public class BoolToGridLengthConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object
       parameter, CultureInfo culture)
    {
        var booleanValue = value as bool?;
        return booleanValue == true ? new GridLength(1,
           GridUnitType.Auto) : new GridLength(0);
    }

    public object ConvertBack(object value, Type targetType,
       object parameter, CultureInfo culture)
    {
        return null;
    }
}
```

7.7.1. Uchování označení řádku

U obyčejné kolekce by se implementovalo rozhraní `INotifyCollectionChanged` a při změně dat by se vyvolala událost `CollectionChanged`. To způsobí, že se data v tabulce aktualizují a najde se posledně označený řádek, a ten se znovu označí. Bohužel při vyhledávání tohoto řádků se lineárně prochází všechny záznamy v kolekci, takže je tento postup nepoužitelný pro kolekci, která má data virtualizovat.

Na první pohled vypadá řešení triviálně – před změnou dat se uloží označený řádek, který se po změně vyhledá a opět označí. Při implementaci jsem ale narazil na několik problémů:

Nepodařilo se mi přijít na žádný způsob, jímž bych našel pozici řádku v nově načtené kolekci bez toho, abych načítal celou databázi. Tento problém se mi nepodařilo vyřešit, tak jsem jej obešel tak, že se řádek označí pouze pokud se najde v načtených stránkách.

Druhý problém se ukázal být takový, že stránky se začnou do kolekce načítat, až když je o nich požádáno vláknem uživatelského rozhraní. To však běží s nižší prioritou než hlavní vlákno a tak se stránky načítají opožděně. Musel jsem vymyslet způsob, kterým bych počkal na vykreslení tabulky, abych pak mohl záznam najít a znovu označit.

Problém jsem vyřešil pomocí třídy `Dispatcher`, která umožňuje asynchronní vykonání metody se zadanou prioritou. Metodu pro označení řádku jsem předal příslušnému View pomocí zprávy `CallBackWhenRenderedMessage` a ten ji pomocí `Dispatcher`u vykonal.

8. Měření rychlosti a optimalizace

Pro měření časových úseků jsem použil třídu `Stopwatch`.

Databáze byla uložena na lokálním disku, který byl současně využíván ostatními běžícími programy, takže výsledky můžou být těmito programy ovlivněny. Pro minimalizaci těchto vlivů jsem však všechny testy prováděl za identických podmínek.

8.1. Sledování změn v kontextu

Entity Framework v základním nastavení automaticky sleduje změny entit v kontextu pro případné zpětné ukládání. Jelikož vím, že se načtené entity v kontextu měnit nebudou, můžu při načítání použít funkci `AsNoTracking`. Ta způsobí, že se stav entity sledovat nebude, což urychluje vytváření entit a následnou práci s nimi.

8.2. Doba načítání stránek

Měření jsem prováděl pro databázové tabulky knih o velikostech 10 000 a 100 000 záznamů (viz tabulky 8.2 a 8.2, přičemž se záznamy načítaly po stránkách o velikosti 100 záznamů a načítaly se bez referenčních vlastností (autorů knihy). Měřil jsem doby nutné pro načtení stránek bez řazení, řazení podle indexované vlastnosti (jméno) a řazení podle vlastnosti bez indexu (ISBN).

Měření doby načítání stránek z tabulky se 100 tisíci záznamy jsem prováděl pouze s funkcí `AsNoTracking`. Navíc jsem neprocházel všechny záznamy, ale jen několik desítek stránek po okrajích tabulky, abych zjistil minimální a maximální doby.

K měření hodnot jsem si vytvořil jednoduchou vlastní třídu `Measurer`, do které se automaticky při načtení stránky přidá čas naměřený pomocí `Stopwatch`. Hodnoty se ve třídě ukládají do seznamu, ze kterého se následně díky funkcím `AverageTime`, `MinTime` a `MaxTime` vypočítají průměrný, nejnižší a nejvyšší čas. Tyto hodnoty jsem si nechal vypisovat do logu pomocí funkce `Trace.WriteLine`.

Stránky se načítají tím déle, čím dál jsou od začátku tabulky, protože databázový systém musí nejprve najít všechny předchozí záznamy, aby věděl, u kterého záznamu má začít data číst.

Tabulka 8.1: Doba načítání stránek pro tabulku s 10 000 záznamy

Řazení	Se sledováním změn			Bez sledování změn		
	minimální [ms]	maximální [ms]	průměrná [ms]	minimální [ms]	maximální [ms]	průměrná [ms]
bez řazení	7	138,2	24,7	6,5	73,3	22
podle jména	6,7	315,4	52,2	12,3	164	48,8
podle ISBN	34,5	406,2	98	39,8	192,8	100,6

Tabulka 8.2: Doba načítání stránek pro tabulku se 100 000 záznamy

Řazení	Bez sledování změn	
	minimální [ms]	maximální [ms]
bez řazení	6,1	656,6
podle jména	5,8	764,1
podle ISBN	288,9	1378,8

8.3. Doba generování databáze

V aplikaci je možnost vygenerování databáze, pokud není nalezena. Potýkal jsem se s problémem při ukládání velkého množství dat do databáze, kdy na několik minut aplikace přestane reagovat a uživatel netuší, zda se aplikace ještě rozběhne.

Dobu ukládání jsem se pokusil zmenšit rozdělením dat na menší bloky, které se ukládají postupně, přičemž jsem si nechal do logu vypsat, které záznamy se právě ukládají. Toto řešení navrhl uživatel Slauma na webu StackOverflow [10], kde také uvedl časy pro různé velikosti bloků podle jeho zkušeností. Z jeho pokusů nejlépe vycházely bloky o velikosti 100 záznamů. Také doporučil po každém uložení vytvořit nový kontext, aby se uvolnily záznamy, které již byly uloženy, to si ale v mém případě dovolit nemohu, jelikož uložené knihy dál využívám při vkládání autorů.

Abych určil optimální velikost bloku na své sestavě, provedl jsem vlastní měření. Měření jsem prováděl při ukládání databáze o 100 tisících autorech, 100 tisících knihách, přičemž každý autor napíše 1-3 knihy, tedy celkem se do databáze vkládalo přibližně 400 tisíc záznamů. Zkoušel jsem velikosti bloků 5000, 1000, 100 a 30.

Tabulka 8.3: Doba ukládání přibližně 100 000 záznamů do databáze

Velikost bloků [zaznamů]	Doba ukládání [s]
5000	796
1000	673
100	1687
30	6358

Závěr

Naprogramoval jsem aplikaci, která může sloužit jako nástroj ke správě nebo prohlížení knihovní databáze.

Aplikace poskytuje nástroje pro vyhledávání, filtrování a řazení záznamů v tabulkách autorů a knih, záměrně však neumožňuje mazat či jinak měnit samotné tabulky nebo databáze, o což by se měl starat spíš správce serveru. Aplikace je ošetřena proti útoku pomocí metody SQL injection, čímž zamezuje provádění nežádoucích příkazů na databázi. Přesto by se aplikace neměla dostat do nesprávných rukou, jelikož umožňuje úpravu a mazání dat v zobrazených tabulkách.

Jedinou povolenou funkcí, která spadá do kategorie SQL příkazů DDL, je vytvoření nové databáze s vygenerovanými záznamy. Ta však může být provedena pouze, pokud daná databáze na serveru není nalezena a neumožňuje poškození stávajících dat v databázi.

Díky virtualizaci dat dokáže aplikace pracovat s rozsáhlými sety dat bez výrazného zadrhávání. Dokonce i filtrování, vyhledávání a řazení probíhá na straně serveru, takže je aplikace relativně nenáročná na systémové prostředky. Pomocí výpisu z logů Entity Frameworku jsem si ověřil, že se aplikace vážně dotazuje vždy jen na data, která jsou požadována pro běh aplikace. Tím jsem dokázal, že zpoždění při načítání dat je z převážné části tvořeno zpožděním na straně serveru, takže pokud by databáze byla uložena na výkonnějším úložišti, zpoždění by dále kleslo.

Při měření doby nutné pro vygenerování databáze nejlépe vycházelo ukládání po 1000 záznamech, tak jsem toto nastavení ponechal i ve výsledné aplikaci.

Okamžitá synchronizace mezi více běžícími aplikacemi nebyla možná z důvodu absence jakýchkoliv notifikací klienta ze strany MySQL serveru. Synchronizace je tedy řešena tak, že se aplikace každých pět vteřin dotazuje databáze, zda nebyla provedena změna databázové tabulky a případně tuto skutečnost oznámí uživateli, který má možnost data aktualizovat.

Připojení k serveru je parametrizováno pomocí připojovacího řetězce, takže je možné se připojit k libovolnému MySQL serveru. U verzí MySQL starších než MySQL Server 5.7.2 však nemusí být funkční notifikace o změnách tabulky, kvůli chybě úložiště InnoDB [9].

Aplikace využívá několik převzatých kódů, které jsou však volně dostupné a jejich licence takovéto použití dovoluje.

Literatura

[1] BROWN, Andy. 10 reasons why WPF is better than Windows Forms. In: *WiseOwl Training* [online]. Wise Owl Business Solutions, ©2016 [cit. 2016-12-12]. Dostupné z: http://www.wiseowl.co.uk/blog/s308/wpf-winforms.htm

[2] Fody/PropertyChanged Wiki: Attributes. *GitHub* [online]. GitHub, ©2017 [cit. 2017-05-25]. Dostupné z: https://github.com/Fody/PropertyChanged/wiki/Attributes

[3] Rozdíly - Výhody a nevýhody - MyISAM vs InnoDB v MySQL. In: *Pidi Blog* [online]. PiDi Soft, ©2013 [cit. 2016-12-12]. Dostupné z: http://blog.pidisoft.cz/clanky/380-rozdily---vyhody-a-nevyhody---myisam-vs-innodb-v-mysql/

[4] MySQL 5.7 Reference Manual: MySQL Glossary. *MySQL :: Developer Zone* [online]. Oracle Corporation, ©2016 [cit. 2016-12-12]. Dostupné z: http://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_dml

[5] MySQL Connector/Net Developer Guide: Introduction to MySQL Connector/Net. *MySQL :: Developer Zone* [online]. Oracle Corporation, ©2016 [cit. 2016-12-12]. Dostupné z: http://dev.mysql.com/doc/connector-net/en/connector-net-introduction.html

[6] MySQL Connector/Net Developer Guide: Using Connector/Net with Connection Pooling. *MySQL :: Developer Zone* [online]. Oracle Corporation, ©2017 [cit. 2017-05-25]. Dostupné z: https://dev.mysql.com/doc/connector-net/en/connector-net-programming-connection-pooling.html

[7] Compare EF Core & EF6.x. *Microsoft Docs* [online]. Microsoft, ©2017 [cit. 2017-06-04]. Dostupné z: https://docs.microsoft.com/en-us/ef/efcore-and-ef6/

[8] WPF: Data Virtualization. *CodeProject: For those who code* [online]. CodeProject, ©1999-2017 [cit. 2017-06-03]. Dostupné z: https://www.codeproject.com/kb/wpf/wpfdatavirtualization.aspx

[9] MySQL Bugs: Update_time is NULL for InnoDB tables. *MySQL Bugs* [online]. Oracle, ©2017 [cit. 2017-06-04]. Dostupné z: https://bugs.mysql.com/bug.php?id=14374

[10] Fastest Way of Inserting in Entity Framework. In: *Stack Overflow* [online]. Stack Exchange, ©2017 [cit. 2017-06-03]. Dostupné z: https://stackoverflow.com/questions/5940225/fastest-way-of-inserting-in-entity-framework

Seznam použitých zkratek a symbolů

CLR	Common Language Runtime
CRUD	Create, Read, Update and Delete
EDM	Entity Data Model
EF	Entity Framework
HTTP	Hypertext Transfer Protocol
LINQ	Language Integrated Query
ms	Milisekunda
MVVM	Model-View-ViewModel
ORM	Objektově – relační mapování
POCO	Plain Old CLR Object
RDBMS	Relational database management system
s	Sekunda
SDL	Schema Definition Language
SQL	Structured Query Language
SQLI	SQL Injection
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language

Seznam příloh

A Přiložené CD **45**

A. Přiložené CD

K práci je přiloženo CD s elektronickou verzí dokumentu a se zdrojovými kódy projektů, které byly vytvořeny v rámci této práce. Projekty byly vytvářeny v IDE Microsoft Visual Studio 2015. CD obsahuje i zkompilovaný spustitelný program `DatabaseEditor.exe` a textový soubor `čtimě.txt`, který obsahuje instrukce pro nastavení a spuštění programu.