Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Vývoj aplikace pomocí WPF

Jan Prokop

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Jan Prokop

Systémové inženýrství a informatika
Informatika

Název práce

Vývoj aplikace pomocí WPF

Název anglicky

Application development using WPF

Cíle práce

Tato bakalářská práce je zaměřena na problematiku vývoje desktopových aplikaci s použitím technologie WPF (Windows Presentation Foundation) a moderních návrhových vzorů (MVVM architektura). Hlavním cílem je zhotovení aplikace, která bude zobrazovat uživateli data získaná ze zařízeních pro zabezpečení perimetru. Dílčím cílem je pak popis postupů implementace a použitých technologií.

Metodika

Metodika práce je založená na analyticko-syntetickém přístupu. Na základě analýzy odborných informačních zdrojů souvisejících s tématem práce a syntézy takto získaných poznatků budou popsána obecná pravidla vývoje aplikací pomocí technologie WPF v jazyce C# za využití moderních návrhových vzorů (MVVM architektura). Teoretické poznatky budou následně aplikovány při vývoji prototypu aplikace pro prezentaci dat ze zařízení pro sledování perimentru. Při návrhu a vývoji bude využito standardních metod a postupů softwarového inženýrství. Poznatky z vývoje a testování budou popsány a zhodnoceny.

Doporučený rozsah práce

40-50 stránek

Klíčová slova

WPF, C#, MVVM, framework, technologie, desktop

Doporučené zdroje informací

ALBAHARI, Joseph and Ben ALBAHARI, 2020. C# 8.0 in a Nutshell: The Definitive Reference. O'Reilly Media. ISBN 9781492051138.

MICROSOFT, 2020. .NET Documentation | Microsoft Docs [online]. 13.8.2020. Dostupné z: https://docs.microsoft.com/en-us/dotnet/

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

Ing. Petr Hanzlík, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 23. 2. 2021

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 23. 2. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 14. 03. 2021

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Vývoj aplikace pomocí WPF" jsem vypracoval(a) samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor(ka) uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.3.2022 _____

Poděkování

Rád(a) bych touto cestou poděkoval(a) Ing. Petru Hanzlíkovi, Ph.D., za vedení mé bakalářské práce a poskytnuté rady.

Vývoj aplikace pomocí WPF

Abstrakt

Tato bakalářská práce se zabývá problematikou vývoje okenní aplikace psané v C# .NET za pomocí technologie WPF a tvorbou vlastní REST API. V úvodní teoretické části jsou představeny technologie a nástroje, které jsou spjaty s vývojem na platformě C# a serverové části API. Jedná se o objektové programování za pomocí frameworku WPF, PRISM a Unity kontejnerem pro Dependency Injection, dále jsou využity speciální UI komponenty, a to Progress Telerik pro lepší a modernější grafické rozhraní aplikace a jeho kontrolky. Simulaci REST API zajišťuje json server. Vývoj samotné aplikace se odehrává v Microsoft Visual Studiu. Tyto nástroje jsou následně využity pro praktickou část vlastního vývoje

Klíčová slova: WPF, technologie, framework, C#, REST API, Model, View, ViewModel, třída, desktop, vývoj

Application development using WPF

Abstract

This bachelor thesis deals with a problematic of the development of desktop application using C# .NET with a technology WPF and creating my own REST API. There are introduced technologies and tools in the introduction of theoretical part that relate to a development on the C# platform and the server part. It is object programming using framework WPF, PRISM and Unity container for Dependency Injection. Json server simulates the REST API. Development environment is the Microsoft Visual Studio. These tools are subsequently used for the practical part of own development.

Keywords: development, technology, WPF, REST API, C#, Framework, MVVM, Model, View, ViewModel, class, desktop

Obsah

1 Úvod

Na úvod bych chtěl popsat strukturu této bakalářské práce, z jakých částí je složena a s čím se v průběhu každé části seznámíte.

Bakalářská práce je rozdělena do dvou hlavních kapitol – teoretická část a praktická část. V teoretické části nastíním použité technologie, jejich využití a moderní pojetí vývoje aplikace s těmito nástroji. V praktické části si rozebereme samotný vývoj aplikace, od analýzy aplikace až po její samotnou implementaci.

V první kapitole teoretické části se budu věnovat jazyku C# obecně, jeho případné srovnání s ostatními jazyky a jeho využití nejen v komerční sféře. Dozvíte se zde, z čeho jazyk vychází, jeho stručnou historii ale i nynější pojetí jazyka.

Ve druhé kapitole teoretické části se zaměříme na moderní pojetí objektového programování a vůbec celkovému návrhu aplikace za pomocí WPF, a to architektonický návrhový vzor MVVM (Model-View-ViewModel), který nejen že umožňuje snadnou rozšiřitelnost aplikace, ale také její testování a celkový přehled v kódu.

Ve třetí kapitole teoretické části se dozvíte něco o frameworku PRISM, který je skvělým nástrojem pro snazší a lepší orientaci v architektonickém návrhu.

Čtvrtá kapitola teoretické části pojednává o architektuře REST, jež je cesta, jak jednoduše dotazovat server o data pomocí jednoduchých HTTP dotazů.

V předposlední kapitole teoretické části popíšu vývojové prostředí, které jsem použil k naprogramování aplikace, a to Microsoft Visual Studio, jež je mocným nástrojem pro tvorbu desktopové aplikace v jazyce C#.

Poslední kapitola teoretické části pojednává o verzovacích systémech používaných ke správě a udržitelnosti kódu samotného.

Naopak v praktické části se budu věnovat prvotní analýze aplikace, jejímu návrhu, a to z pohledu rozvržení tříd nebo použitých technologií a architektur. V poslední řadě zde bude popsána implementace jednotlivých řešení, ukázky kódu, popis tříd a návrhové vzory v praxi.

Na závěr této bakalářské práce proběhne zhodnocené odvedené práce a její případné vylepšení.

Cíl práce a metodika

1.1 Cíl práce

Práce je zaměřena na seznámení čtenáře s průběhem vývoje aplikace v programovacím jazyku C# za použití technologie Windows Presentation Foundation (WPF). Hlavním cílem je analýza a implementace desktopové aplikace pro přehlednou prezentaci dat z chytrých zařízeních zabezpečující perimetr (kamery, senzory, …). Dalším cílem je popsat vývoj na této platformě.

1.2 Metodika

Teoretická část, která slouží jako podklad pro zpracování praktické části, představuje první část práce. Tato část bude vypracována na základě poznatků a studií z odborné literatury a článků.

Praktická část práce spočívá na základě analýzy, návrhu a implementaci desktopové aplikace sloužící jako přehledné uživatelské rozhraní pro chytré zařízení. Během analýzy a návrhu budou použity standardní metody softwarového inženýrství, zatímco při implementaci bude využit programovací jazyk C# s použitou technologií WPF.

Nakonec bude aplikace otestována a zhodnocena pro její další možné změny, vylepšení či následný budoucí rozvoj.

2 Teoretická východiska

V této části bakalářské práce si představíme technologie a nástroje, které použijeme pro zhotovení desktopové aplikace.

2.1 C#

C# (C sharp) je programovací jazyk vyvíjený společností Microsoft, která ho spolu s celým vývojovým prostředím .NET poprvé představila v roce 2002. Jedná se tedy o poměrně nový objektově orientovaný jazyk. Už podle společnosti, která tento jazyk vyvíjí, tak podle názvu lze soudit, že C# vychází z programovacího jazyka C/C++, ale na první pohled nejen syntaxí se podobá jazyku Java. [1]

Zde se může objevit otázka – Jaký je tedy rozdíl mezi C# a .NET? Je to vcelku jednoduché, C# je programovací jazyk, zatímco .NET je aplikační rámec (framework), na kterém je tento jazyk postaven. .NET může podporovat více jazyků než jen samotný C#, ale jelikož C# a .NET jsou od stejné firmy, jejich integrace je o to jednodušší. [2]

C# je objektově orientovaný programovací jazyk, což znamená, že při programování můžeme používat moderní filozofii objektově orientovaného přístupu. Je to vlastně o jiném způsobu přemýšlení při vývoji, kde klademe důraz na znovupoužitelnost. Programování je tedy lépe srozumitelnější pro člověka, jelikož se odpoutáváme od toho, jak program vidí stroj a program píšeme z pohledu člověka. U objektově orientovaného přístupu je základní entitou objekt, který nám může reprezentovat cokoliv z reálného světa (např. máme objekt auto, které má své vlastnosti – značka, model, barva, SPZ atd.). S tímto objektem a jeho vlastnosti – atributy, dále pracujeme dle dalších specifikací aplikace. [3]

2.1.1 Vlastnosti jazyka C#

V C# neexistuje vícenásobná dědičnost. Jedna třída může dědit z pouze jedné třídy, ale může implementovat kolik chce rozhraní.

Neexistují žádné globální proměnné a metody. Všechny proměnné jsou deklarovány uvnitř třídy nebo metody. Pokud jsou deklarovány na úrovni třídy, jsou dostupné skrz celou třídu, ve všech metodách. Pokud jsou deklarovány v nějaké metodě, jsou dostupné pouze pro danou metodu. Jako náhrada za globální proměnné jsou statické metody a proměnné.

C# ulehčuje přístup k datovým atributům. Toho docílíme tak, že si můžeme definovat tzv. propertu, která se zvenku jeví jako datový atribut, ale skutečně obsahuje prostor pro definici metod get a set, čímž nám usnadňuje práci s datovým atributem a zároveň zachová princip zapouzdření dat. Naopak v jiných objektově orientovaných jazycích je zvykem, že se používá vzor, kde se k datovým atributům přistupuje nepřímo, a to samotnými metodami get (accessor) a set (mutator).

C# je typově bezpečný. Všechny předdefinované implicitní konverze jsou bezpečné. Jako příklad může být rozšiřování celočíselných typů – z 32bit na 64bit nebo konverze z odvozeného typu na rodičovský. Není tu implicitní konverze z celočíselných typů na boolean a ani implicitní konverze mezi celočíselným a výčtovým typem.

C# nepotřebuje a ani neobsahuje dopřednou deklaraci. Takže je na nás, v jakém pořadí deklarujeme jednotlivé metody, důležité je pak až jejich samotné volání.

Jazyk C# je case sensitive. Rozlišuje mezi velkými a malými písmeny. Identifikátory „promenna“ a „Prommena“ tedy nejsou ekvivalentní. [1]

2.1.2 .NET

.NET je vývojářská platforma s plně otevřeným, přístupným zdrojovým kódem, která nám umožňuje vyvíjet rozličné typy aplikací. Vývojářská platforma v tomto případě zahrnuje použitý programovací jazyk a použité knihovny. .NET je z hlavní části vyvíjen společností Microsoft, ale jelikož se jedná o platformu s přístupným zdrojovým kódem, k jeho rozvoji může přispět každý. Konkrétně k rozvoji přispělo přes šedesát tisíc vývojářů a tři tisíce sedm set společností.

Jako příklad programovacích jazyků, které můžeme použít jsou:

- C#
- Visual Basic
- F#

Použité knihovny se liší na druhu námi zvolené platformy, nad kterou vyvíjíme aplikaci. Máme několik typů platforem:

- .NET Core – Vývoj pro Windows, Linux a macOS aplikací s příslušnými knihovnami
- .NET Framework – Vývoj webových stránek, služeb a aplikací pro Windows s příslušnými knihovnami
- Xamarin/Mono – Vývoj mobilních .NET aplikací
- .NET Standard – Standardní .NET knihovny, které obsahují všechny zmíněné .NET knihovny seshora.

S .NETem lze tvořit různá škála aplikací:

- Webové aplikace
- Desktopové aplikace
- Mobilní aplikace
- Mikroslužby
- 2D a 3D hry pro počítač, mobily a konzole
- Strojové učení
- Cloudové služby
- IoT aplikace – Raspberry Pi atd.

I přes to, kolik toho .NET umožňuje, všude používáme ten stejný zaběhlý standard, stejné knihovny a stejné zkušenosti, na které je .NET vývojář zvyklý. [4]

2.1.3 **WPF**

WPF neboli Windows Presentation Foundation je nejaktuálnější (k r. 2022) přístup, jakým lze tvořit grafické uživatelské rozhraní v .NET platformě. V zásadě nám umožňuje specifikovat vzhled naší aplikace, ať už okenní, webové nebo jiné.

Již v základu obsahuje mnoho použitelných grafických komponent, jako jsou textová pole, tlačítka, dialogové boxy a spoustu dalších známých elementů. Dále obsahuje elementy pro definici rozložení stránky jako takové, např. mřížky, do níž můžeme zakreslovat elementy a mít kontrolu nad jejich umístěním a vzhledem.

Zvládá také jakékoliv uživatelské interakce s těmito elementy, ať už se jedná o přejetí myši nad elementem nebo stisknutím tlačítka. [5]

Pro návrh uživatelského prostředí používáme deklarativní jazyk **XAML** (Extensible Application Markup Language), který je primárně určený pro návrh grafického uživatelského prostředí ve WPF, to ale neznamená, že je určený jenom na toto. Dále se dá použít jako jazyk pro napsání definice konfigurace aplikace.

```
<UserControl

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:DekstopTelerikApp.Views"
xmlns:conv="clr-namespace:DekstopTelerikApp.Converters"
xmlns:telerik="http://schemas.telerik.com/2008/xaml/presentation"
xmlns:cc="clr-namespace:DekstopTelerikApp.CustomControls"
xmlns:cam="clr-namespace:MjpegProcessor;assembly=MjpegProcessor"
x:Name="userControl"
xmlns:b="http://schemas.microsoft.com/xaml/behaviors"
x:Class="DekstopTelerikApp.Views.ElementsTreeListView"
mc:Ignorable="d" >

    <UserControl.Resources>
        <conv:BooleanToVisibilityConverter x:Key="VisConverter"/>
    </UserControl.Resources>

    <Grid x:Name="menuBarGrid"
                  VerticalAlignment="Stretch"
                  HorizontalAlignment="Stretch"
                  Grid.Column="1">

                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>

                <Border Grid.ColumnSpan="3"
                        Background="{StaticResource GrayAppColor}"
                        BorderBrush="{StaticResource DefaultBorderBrush}"
                        BorderThickness="0 0 0 2"/>

                <Button x:Name="ExportAllCSVButton"
                        Command="{Binding ExportElementToCsvCommand}"
                        CommandParameter="{x:Null}"
                        Content="Export All"
                        Grid.Column="0"
                        Style="{DynamicResource MenuButtonStyle}"/>
            </Grid>
</UserControl>
```

Code snippet 1 - Příklad jazyka XAML

2.2 MVVM Architektura

MVVM neboli Model-View-ViewModel, je jeden z návrhových vzorů pro aplikace vyvíjené s WPF technologií. Umožňuje úplné oddělení logické implementace od uživatelského rozhraní, čímž nám dále nevzniká závislost uživatelského rozhraní na konkrétním typu dat. Kód se stává přehlednějším, dále rozšiřovatelnějším, je ho méně a následné implementace jsou jednodušší.

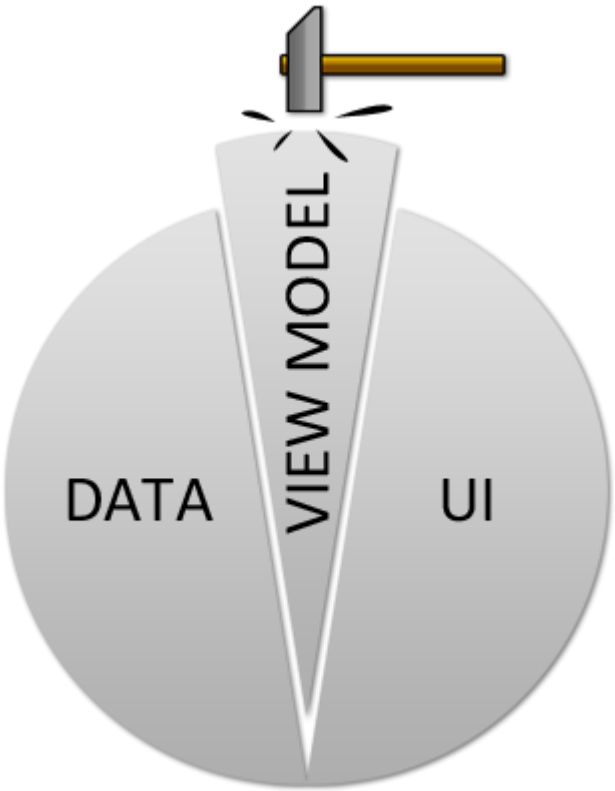
2.2.1 Jednotlivé komponenty

MVVM architektura se skládá ze tří hlavních komponent:

1. **Model –** Model je nosičem dat a logiky práce s daty. Pro představu můžeme mít model *Auto*, který jakožto objekt bude mít určité vlastnosti a atributy např. *Model, Značka, Barva, Rok výroby* atd. Dále může na sobě metody typu *ZiskejStariVozu, JeTechnickyZpusobile* atd.

2. **View –** View nám představuje grafické okno neboli to, co vidí uživatel. Je prezentováno značkovacím jazykem XAML a toto okno by nemělo mít vůbec ponětí o tom, jaké typy dat budou vstupovat do výstupu. Naopak musí být připraveno na jakýkoliv vstup dat. Jinými slovy, musí být kompletně odděleno od business části aplikace. Pro představu máme uživatelské rozhraní, které obsahuje ComboBox. Tento ComboBox by neměl vědět, jaké data budou přicházet, jestli to bude kolekce Aut, Nemovitostí, Lidí, … Na všechny typy musí přistupovat stejně bez lpění na jejich vlastnostech.

3. **ViewModel –** ViewModel nám spojuje oba tyto celky a zajišťuje efektivní komunikaci a implementaci. Předává tedy data z Modelu do View a naopak musí reagovat na změny v uživatelském prostředí, které přepisuje zpět do Modelu. Pro představu – v uživatelském rozhraní nám uživatel smaže řádek v Gridu s daty, ViewModel na to musí zareagovat a tyto data korektně smazat i z Modelu. Naopak, když nám např. serverová část pozmění nějaké data, tak tyto data musí ViewModel aktualizovat i v uživatelském rozhraní.



Obrázek 1 - Princip MVVM [6]

2.2.2 **Proč je důležité používat MVVM architekturu?**

- Pokud budeme chtít dělat nějaké významné změny v business logice aplikace a implementace modelu tuto logiku zapouzdřuje, tak tyto změny jsou nanejvýš nepříjemné, obtížné a rizikové. Proto máme *ViewModel*, který funguje jako adaptér mezi jednotlivými *Modely*, a tudíž se vyhneme provádění významných změn v *Modelu*.

- Jednodušší vytváření unit testů a celkové testování aplikace, kdy vývojáři provádí testy pro *ViewModely,* ale nijak nezasáhnou do *View,* grafické komponenty aplikace. Testy pro jednotlivé *ViewModely* testují stejné funkce, které používá *View*.

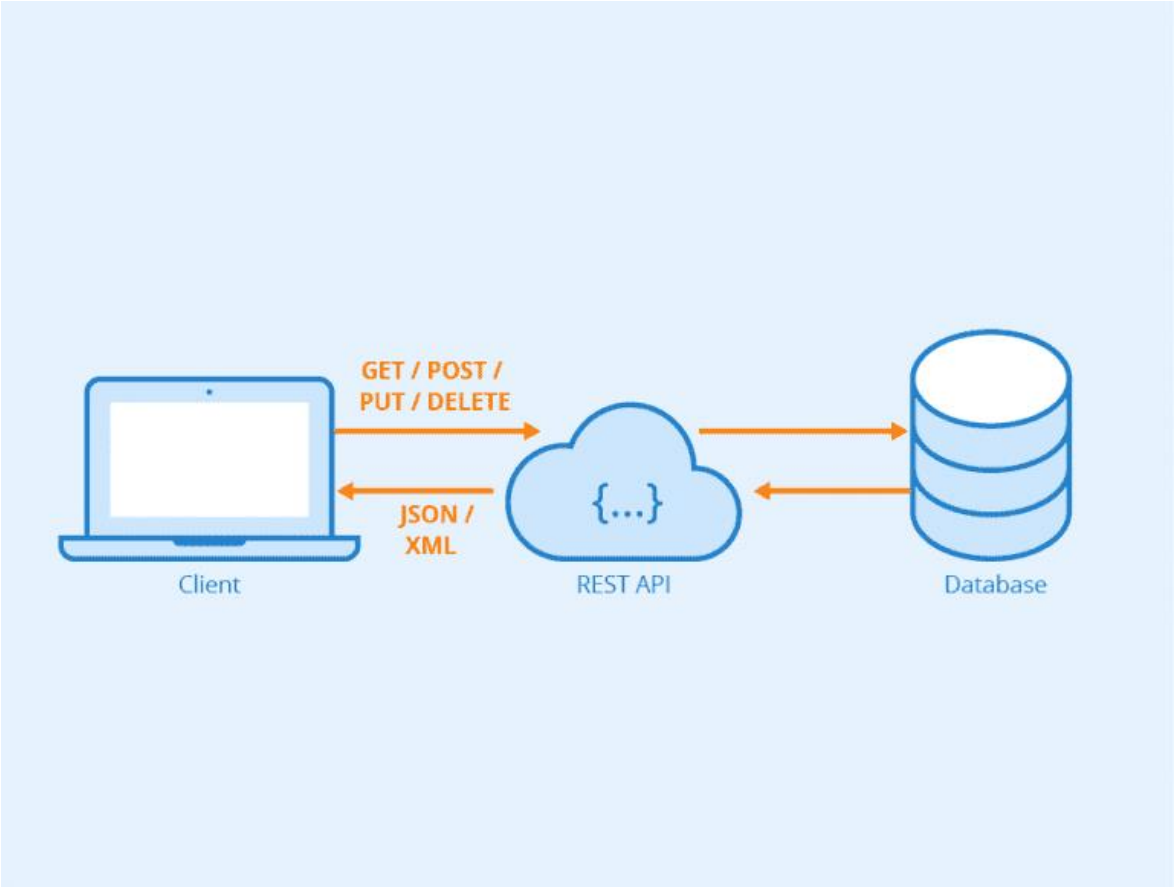
- Uživatelské rozhraní neboli *View*, je možné přepracovat bez zásahy do kódu za předpokladu, že je *View* zcela implementováno jazykem XAML. Proto by neměl být problém, že pokud se něco změní ve *View, ViewModel* to přijme a nevyhodí žádnou chybu.

- Návrháři a vývojáři mohou pracovat nezávisle a souběžně na svých komponentách během procesu vývoje. Návrháři se mohou soustředit na *View*, zatímco vývojáři mohou pracovat na komponentách *ViewModelu* a *Modelu*. [7]

- Snazší rozšiřitelnost aplikace, kdy při přidávání nových modulů, komponent, prostě čehokoliv, nemusíme procházet starý kód a bát se nějakých chyb.

2.3 **REST API**

REST je architektura pro webové API. Její zkratka znamená *Representational State Transfer* a jejím autorem je spoluautor webového protokolu HTTP, Roy Fielding. REST je datově orientovaný, nespouští tedy žádné vzdálené procedury, ale pracuje čistě s daty. Na každé data přistupujeme přes unikátní URI, pro které jsou definovány čtyři základní metody nazývané **CRUD**. [8]



Obrázek 2 - REST API [11]

2.3.1 **CRUD Metody**

Tyto metody jsou základními metodami dotazujícími se na server. Pomocí těchto metod lze měnit nebo získávat data ze serveru. Každá metoda vrací návratový stav. Při implementaci těchto metod se musí brát v potaz autentifikace, která nejčastěji probíhá pomocí jména a hesla a posléze vygenerovaného autentifikačního tokenu, který se posílá s dotazy a tím validuje oprávnění na manipulaci s daty.

1. **Create –** Slouží k vytvoření objektu. Používá se metoda **POST**, která s určitými parametry vytvoří daný objekt.

2. **Read –** Slouží k získání objektu. Používá se metoda **GET**, jež je klasickou základní metodou, která se volá při zadání jakékoliv adresy do prohlížeče.

3. **Update –** Slouží k aktualizaci objektu. Používá se metoda **PUT** (někdy můžeme narazit, že je implementováno opět přes **POST**).

4. **Delete –** Slouží ke smazání objektu. Používá se metoda **DELETE** (opět se můžeme setkat s častou implementací přes metodu **POST**). [8]

2.3.2 Stavové kódy

Ať už při úspěšném či neúspěšném provedení jednotlivých metod, nám metody vrátí stavový kód. Neboli informaci o provedení dané operace. Vypíšu zde ty nejdůležitější a nejznámější:

- **200 –** OK, požadavek proběhl v pořádku
- **204 –** No Content, požadavek proběhl v pořádku, ale nic nevrátil
- **400 –** Bad Request, požadavek na server je nečitelný
- **401 –** Unauthorized, klient není ověřen
- **404 –** Not Found, zdroj není nalezen
- **500 –** Server error, interní chyba serveru [9]

2.3.3 Struktura dat

Při dotazování se serveru na data, očekáváme nějaký návratový formát. Tím je nejčastěji objekt typu JSON (JavaScript Object Notation), který se lehce čte a snadno převádí do kódu.

Příklad struktury JSON dat:

```
{
        “jmeno“: “Jan“,
        “prijmeni“: “Prokop“,
        “univerzita“: “CZU“
        “rocnik“: 3
}
```

2.4 Dependency Injection

Dependency injection (**DI**), neboli vkládání závislostí, je technika, kterou lze vkládat závislosti mezi jednotlivými komponenty bez toho, aniž by se v každé komponentě musel daný závislý objekt znovu vytvářet a inicializovat. Dalším cílem je, aby vkládané komponenty neměli mezi sebou referenci při sestavování programu. Tento způsob lze použít při objektově orientovaném přístupu psaní aplikace.

Vkládání závislostí lze docílit třemi způsoby:

1. **Vkládání rozhraním** – jako v případě implementace našeho řešení, kde nám DI zajišťuje framework, kontejner Unity. Implementuje rozhraní, jež se poté očekává při sestavení programu.

2. **Vkládání konstruktorem** – předávání závislostí mezi komponenty v jednotlivých konstruktorech jako parametry.

3. **Vkládání setter metodou** – objekt má metodu *Setter* ve které lze závislost předávat. [10]

Příklad užití Unity kontejneru se zabývám při implementaci aplikace v praktické části bakalářské práce.

2.5 Vývojové prostředí Microsoft Visual Studio

Microsoft Visual Studio je vývojové prostředí od společnosti Microsoft. Je vyvíjeno v jazyce C++ a C#. Jelikož i C# a .NET, který používám pro vývoj této aplikace, je od společnosti Microsoft, následná integrace a implementace je o to snazší a intuitivní.

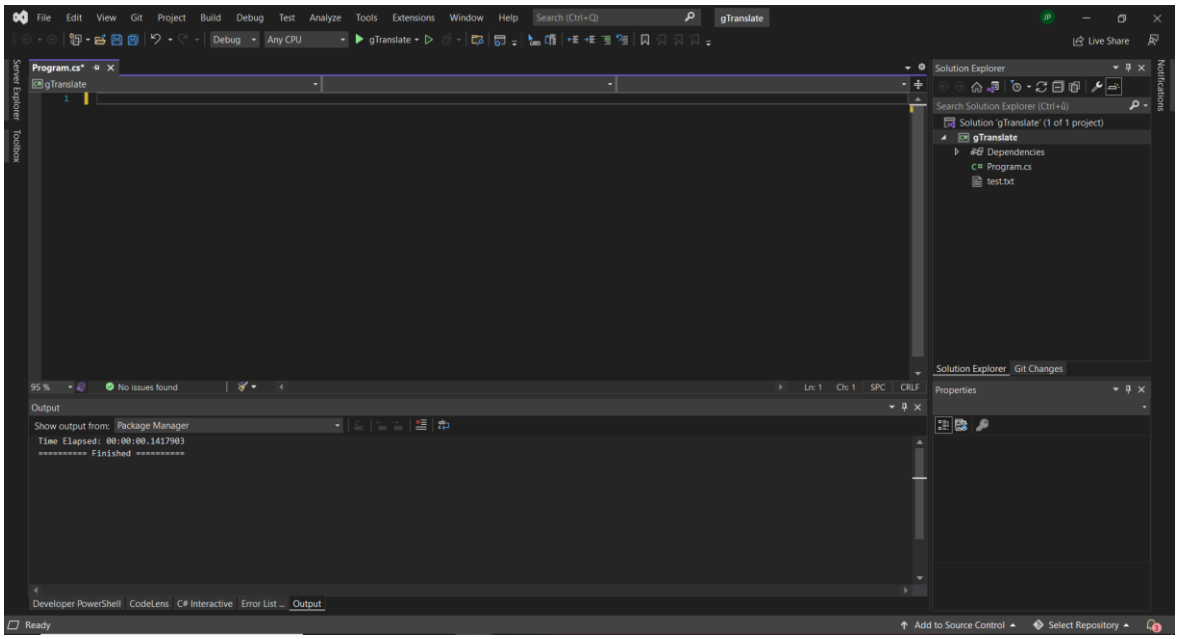
Vývojové prostředí je rozděleno do několika panelů. Po levé straně většinou máme integrované kontrolky pro XAML a následně strukturu XAML souboru.

V hlavní prostřední části se nachází kód daného souboru.

Po pravé straně je průzkumník našeho řešení a soborů. Při debugování aplikace tu běží monitoring zátěže počítače.

V dolní části vývojového prostředí se nachází okno s Outputem aplikace, případně s errory.

Toto rozložení je defaultní a krásné na tom je to, že si to každý programátor může pozměnit, jak uzná za vhodné.



Obrázek 3 - Vývojové prostředí Microsoft Visual Studio 2019

3 Vlastní práce

V předešlé části byly přestaveny technologie a nástroje, které byly použity pro vývoj desktopové aplikace. V této části práce budou představené technologie použity v praxi a bude zde popsán postup implementace jednotlivých technologií desktopové aplikace pro zobrazení dat z chytrých zařízení sloužící k zabezpečení perimetru.

Na aplikace pro zobrazení dat bude představena technologie Windows Presentation Foundation (WPF) za použití architektonických návrhových vzorů a to Model-View-ViewModel (MVVM), Singleton a Dependency Injection.

Data bude aplikace přebírat z nasimulovaného serverového prostředí, na kterém běží REST API a na data se bude dotazovat základními http dotazy.

3.1 Analýza

Nejprve je třeba si definovat, proč vlastně takovou aplikaci vyvíjet a k čemu bude sloužit. Dále jaké jsou možné požadavky na aplikaci a s jakými daty bude manipulováno.

3.1.1 Business analýza

Hlavním cílem této aplikace bude vytvořit jednotné a přehledné grafické rozhraní pro uživatele, kteří mají několik zařízení pro zabezpečení perimetru, jako jsou kamery, parkovací senzory, poplašné hlásiče, kabel pro zabezpečení perimetru a další.

Tato zařízení mohou být každá od jiného výrobce, což pro uživatele znamená, že každé zařízení může mít svojí aplikaci pro zobrazování dat. V dnešní době, kdy uživatel hledá co nejpohodlnější způsob, jak mít data přehledně pohromadě, toto řešení není ideální.

Aplikace bude sloužit pro integraci všech těchto zařízení a pohodlné zobrazení dat uživateli na jednom místě.

3.1.2 Technologická analýza

Pro vývoj okenní aplikace na operačním systému Windows bylo nutné zvolit vhodnou technologii, která je lehce dostupná a kompatibilní napříč prostředími. Na výběr je široká škála programovacích jazyků a technologií, se kterými by šlo pracovat. Máme

zde například programovací jazyk Java a s ním spojené frameworky JavaFX, Swing, AWT, kde by se také daly aplikovat architektonické návrhy Model-View-Controller, ale pro vývoj této aplikace byl zvolen jazyk C# s frameworkem .NET za použití technologie WPF.

.NET za použití WPF se jeví jako moderní multiplatformní pojetí objektového přístupu k vývoji desktopové aplikaci. Dá se přenášet na ostatní platformy – např. Linux. Je zde možnost implementovat mnoho pomocných knihoven a balíčků usnadňující vývoj aplikace.

3.1.3 **Data**

Data se ukládají a získávají ze serveru, na němž běží RESTová API. Na data se dotazuje běžnými HTTP dotazy – GET pro získání dat, POST pro vytvoření dat, DELETE pro smazání dat a PUT pro update dat.

3.1.3.1 Zabezpečení

Celá komunikace mezi desktopovou aplikací (klientem) a serverem je šifrována.

Pro úspěšné přihlášení na server je třeba znát uživatelské jméno a heslo, kterým se vygeneruje autorizační token potřebný pro další dotazy. Pro komunikaci mezi tímto klientem a serverem je potřeba autorizační token typu Baerer.

Bez tohoto tokenu není možné data získat a při dotazování na data bez tokenu server vyhodnotí dotaz jako neplatný a následně vyhodí chybu.

3.1.3.2 Formát

Data jsou nahrávána a získávána ve formátu JSON, který umožňuje ukládat data jako objekty. To usnadňuje následnou deserializaci dat ze serveru do objektových tříd v aplikaci.

Každý prvek (kamera, senzor, …) je uložen jako objekt typu „*element*“, který má své definované vlastnosti, získávající ze zařízení. Každý prvek má své unikátní ID, kterým se lze následně párovat na další prvky, a tudíž tvořit hierarchii prvků.

3.2 Návrh aplikace

Při prvotním návrhu aplikace je nutné myslet na co nejlepší způsob, jak aplikaci vůbec napsat. Kód musí být udržitelný, lehce rozšiřitelný a také musí být umožněno testování.

Tato práce pojednává o moderním objektově orientovaném programování, a právě s tím přicházejí i architektonické vzory. Zde bude demonstrovaný hlavní návrhový vzor Model-View-ViewModel (MVVM), který umožnuje rozdělení aplikace na samo soběstačné moduly, které se dají znovupoužít kdekoliv v rámci aplikace.

Každý modul musí fungovat i jako samostatný prvek, když je tohoto docíleno, moduly se poté propojí a vytvoří jednotný funkční celek – aplikaci. Na tomto odděleném principu dále také stojí testování, kde pomocí Unit testů lze snadno testovat samotný kód.

Pro lepší pochopení určitých scénářů v aplikaci bude navržen diagram případu užití.

V poslední řadě je také kladen důraz na návrh přehledného uživatelského prostředí, které bude intuitivní a uživatelsky přívětivé.

3.2.1 Návrhové vzory

Důležitým aspektem při návrhu aplikace je struktura kódu, jak kód bude členěn a jak bude vypadat. K tomuto nám pomáhají návrhové architektonické vzory, které v kódu udržují jakýsi pořádek a správnou strukturu.

3.2.1.1 Model-View-ViewModel

Hlavním demonstrovaným vzorem v této aplikaci je MVVM vzor. Ten nám zajišťuje oddělené uchování dat mezi oknem, tedy tím, co vidí uživatel, třídou se souborem funkcí metod a vlastností, která definuje chování daného okna a datový model, který slouží jako objekt se souhrnem vlastností.

Ve třídě ViewModelu můžeme pracovat s business objekty, které poté používáme v samotném okně a zobrazujeme uživateli.

Okno neboli „view“ prezentuje samotná data uživateli. View definujeme pomocí XAML jazyku. Není žádoucí, aby se v „code behind“ pracovalo s business objekty.

```
<DataTemplate DataType="{x:Type vm:ElementsTreeListViewModel}">
       <views:ElementsTreeListView />
</DataTemplate>
<DataTemplate DataType="{x:Type vm:LoginViewModel}">
       <views:LoginView />
</DataTemplate>
</telerik:RadWindow.Resources>

<telerik:RadWindow.DataContext>
       <vm:MainWindowViewModel/>
</telerik:RadWindow.DataContext>
```

Code snippet 2 - Příklad propojení View a jeho ViewModelu

Také je důležité dbát na pojmenovávání svých tříd, aby v tom nebyl zmatek. Jedna z používaných syntaxí je *nazevView* pro okna a *nazevViewModel* pro ViewModel daného okna.

3.2.2 Diagram případu užití

Je třeba definovat seznam kroků, který probíhá mezi dvěma aktéry – nejčastěji uživatele a systému. Uživatel něco očekává a systém to uživateli zobrazí.

3.2.2.1 Případ užití – Přihlášení do aplikace

Tabulka 1 - Případ užití přihlášení do aplikace

Use case	*Scénář*

• Uživatel očekává možnost přihlášení se na server pomocí IP adresy, uživatelského jména a hesla
• Po zadání validních údajů očekává správné přihlášení a načtení hlavního okna aplikace
• Po zadání špatných údajů očekává informaci o nesprávnosti údajů
	• Systém zobrazí okno s formulářem pro přihlášení:
	  • Textové pole pro zadání IP adresy
	  • Textové pole pro zadání uživatelského jména
	  • Textové pole pro zadání hesla
	• Systém zobrazí tlačítka:
	  • Tlačítko pro přihlášení

	• Tlačítko pro ukončení aplikace
	
	• Po stisku tlačítka pro přihlášení nebo stiskem klávesy „Enter“ systém zhodnotí předaná data:
	  • Pokud jsou data validní, systém se spojí se serverem a zobrazí hlavní okno aplikace
	  • Pokud data validní nejsou, systém zobrazí chybovou hlášku uživateli

3.2.2.2 Případ užití – Hlavní stránka s prvky

Tabulka 2 - Případ užití hlavní stránka s prvky

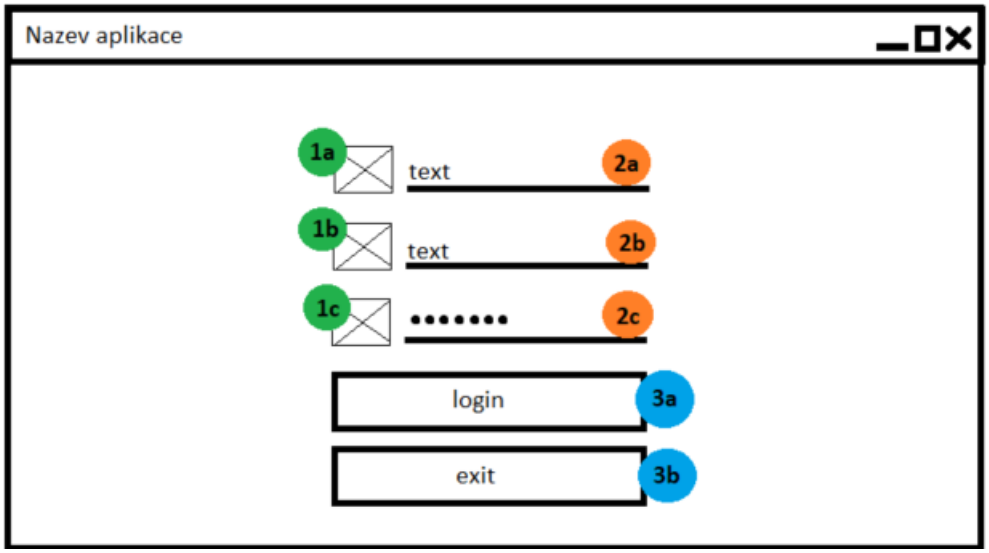
Use case	*Scénář*

• Uživatel očekává že uvidí svá zařízení
 	
• Uživatel očekává možnost vyhledávání prvků
 	
• Uživatel očekává možnost zobrazení detailů u jednotlivých zařízení
 	
• Uživatel očekává možnost exportu dat
 	
• Uživatel očekává možnost odhlášení se ze serveru
	• Systém zobrazí hlavní okno rozdělené na dva sektory:
	  • Na levé straně systém zobrazí strom prvků s hierarchickou strukturou
	  • Na pravé straně systém zobrazí informace o daném prvku
	
	• Systém zobrazí pomocnou lištu nad stromem prvků, kde se nachází:
	  • Textové pole pro vyhledávání
	  • Tlačítko pro hierarchické rozšíření prvků
	  • Tlačítko pro složení prvků ve stromu a smazání obsahu textového pole pro vyhledávání

	• Systém zobrazí v pravé části informace o daném prvku:
	  • Textové pole s názvem prvku
	  • Další textová pole s informacemi o prvku
	
	• Systém zobrazí pomocnou horní lištu, kde se nachází:
	  • Tlačítko pro export dat všech prvků
	  • Tlačítko pro export dat vybraného prvku
	  • Tlačítko pro odhlášení se ze serveru

3.2.3 **Návrh uživatelského rozhraní**

3.2.3.1 Logický model – Přihlášení do aplikace



Obrázek 4 - Logický model přihlášení

1) **Ikony jednotlivých textových polí**
    a. Ikona serveru

b. Ikona uživatele

c. Ikona hesla

2) Textové pole

a. IP adresa serveru

b. Uživatelské jméno

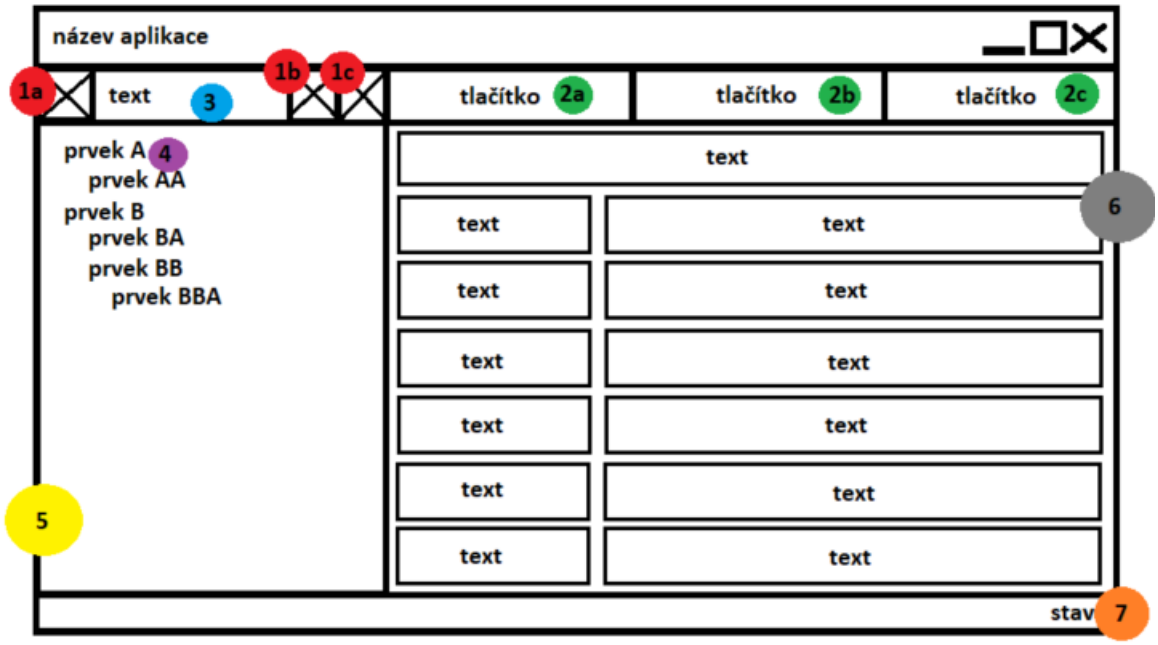
c. Heslo

3) Tlačítka

a. Přihlášení

b. Ukončení aplikace

3.2.3.2 Logický model – Hlavní stránka s prvky



Obrázek 5 - Logický model hlavní stránky

1) Ikony

a. Ikona lupy

b. Klikací ikona rozšíření

c. Klikací ikona křížku

2) Tlačítka

a. Export všech prvků

b. Export vybraného prvku

c. Odhlášení se ze serveru

3) Textové vyhledávací pole

4) **Jednotlivé prvky ve stromu prvků**
5) **Strom prvků**
6) **Informace o vybraném prvku**
7) **Aktuální stav aplikace – Selected, Export, Ready**

3.3 Implementace

V této části budou popsány kroky při vytváření aplikace. Nebudou popsány všechny kroky, jelikož to není možné při takovém rozsahu práce, ale budou představeny nejdůležitější části aplikace.

3.3.1 Sestavení projektu

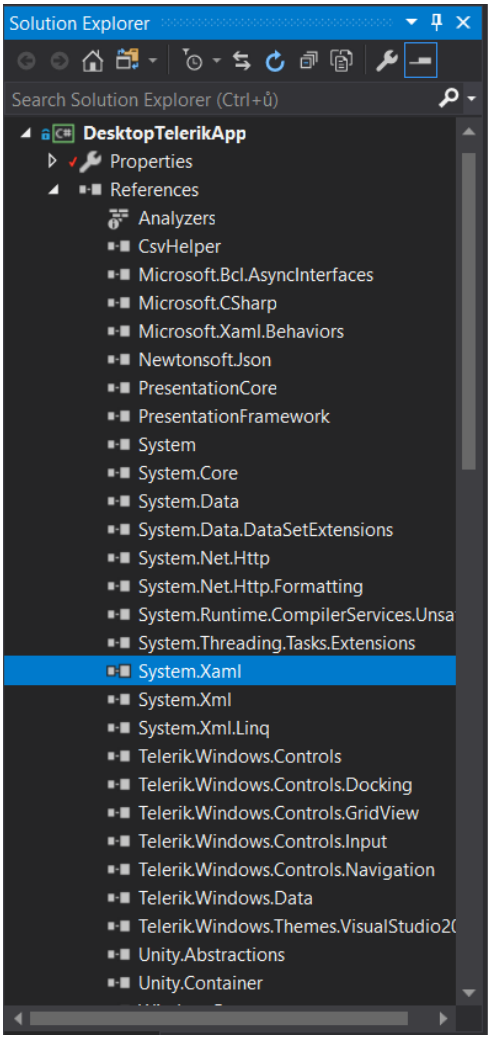
Aplikace tedy bude psána v jazyce C# za pomocí frameworku .NET, za použití technologie WPF. Programování, debug a sestavení bude probíhat ve vývojářském prostředí Microsoft Visual Studio 2019.

Pro začátek je tedy nutné si nainstalovat Microsoft Visual Studio 2019 a při instalaci vybrat balíček modulů určených pro vývoj desktopové aplikace, ve kterém se nacházejí knihovny WPF a jiné. Po dokončení instalace jednoduše zapneme Visual Studio a vytvoříme nový projekt, kde si vybereme přednastavený template – WPF App.

Dále si nainstalujeme PRISM knihovnu, kterou si lehce nainstalujeme pomocí integrovaného NuGet prostředí. Pod naším projektem klikneme na „References“ pravým tlačítkem, vybereme „Manage NuGet packages…“ a vyhledáme PRISM a nainstalujeme Prism.Wpf, Prism.Core a Prism.Unity – každý jeden tento balíček budeme potřebovat pro snazší vývoj naší aplikace.

Dalším balíčkem z NuGet repositáře, který budeme potřebovat je CsvHelper, pro snazší export dat.

Všechny tyto externí knihovny – balíčky, najdeme v „References“ našeho projektu.



Obrázek 6 - Reference projektu

3.3.2 **Vytvoření oken pro zobrazení dat – View**

3.3.2.1 MainWindow

S vytvořeným projektem se nám také vytvoří první okno aplikace – *MainWindow.xaml* a jeho třída pro volání logiky okna tzv. Code Behind – *MainWindow.xaml.cs*. Toto okno, neřekneme-li jinak, se nám bude defaultně pouštět jako výchozí okno naší aplikace.

Toto výchozí okno nám poslouží jako vstupní bod aplikace. Zde si budeme přepínat další příslušné okna a také jim nastavovat jejich ViewModel třídy.

Jediné, co se bude v tomto okně nacházet je tedy grid, který bude obsahovat vnořený control s právě používaným oknem a jeho ViewModelem. Pro představu, máme tedy hlavní okno aplikace, v něm vytvořený grid a jeho obsah se nám bude měnit v závislosti na tom, který obsah právě chceme uživateli zobrazit.

```
<Grid>
       <telerik:RadTransitionControl Content="{Binding CurrentViewModel}"
                                     Duration="00:00:02">
            <telerik:RadTransitionControl.Easing>
                <PowerEase EasingMode="EaseInOut" Power=".4"/>
            </telerik:RadTransitionControl.Easing>

            <telerik:RadTransitionControl.Transition>
                <telerikTransitions:SlideAndZoomTransition/>
            </telerik:RadTransitionControl.Transition>
       </telerik:RadTransitionControl>
</Grid>
```

Code snippet 3 - MainWindow.xaml content pro ViewModel

3.3.2.2 ElementsTreeListView

Toto View bude již trochu složitější, jelikož zde bude probíhat hlavní část prezentací dat uživateli. Logický model je zmíněn v kapitole 4.2.3, takže ho již nemusím představovat.

Toto okno se nám načte po úspěšném přihlášení. Po načtení okna se spustí Trigger s událostí „*Loaded*“, jejíž chování je popsáno ve ViewModelu.

Hlavním nosičem prvků je zde grid – WrapperGrid, který obaluje celé okno a do něj vnořujeme další prvky.

K zobrazení všech prvků, tím myšleno uživatelovo chytrá zařízení, nám slouží *TreeView* element, kterému nastavíme DataContext – obsah, ze kterého bude čerpat data. Nastavíme ho pomocí bindingu. Okno jako takové by nemělo mít vůbec žádné ponětí o tom, jaké data bude zobrazovat, nesmí být konkretizována, aby při dalším použitím okna, s jiným ViewModelem, mohlo stejně efektivně data zobrazovat.

```
<telerik:RadTreeView Grid.Row="1"
                     x:Name="ElementsTreeView"
                     SelectedItem="{Binding DataContext.SelectedElement,
                     RelativeSource={RelativeSource AncestorType={x:Type
UserControl}}, Mode=TwoWay}"
                     ItemsSource="{Binding RootElements}"
                     IsVirtualizing="True"
                     ItemTemplate="{StaticResource RootElement}"
                     BorderThickness="0"
                     Background="{StaticResource GrayAppColor}">

</telerik:RadTreeView>
```

Code snippet 4 – TreeView

TreeView v této podobě ale žádná data nezobrazí, sice už má přehled o tom, že nějaká data má zobrazovat, jelikož má nastavený *„ItemsSource“* ale ještě neví, jak je má zobrazit. K tomu nám slouží DataTemplaty, které definují vzhled a hierarchii zobrazení prvků.

```
<DataTemplate x:Key="ElementTypeDetail">
      <TextBlock Text="{Binding Name}" />
</DataTemplate>

<HierarchicalDataTemplate x:Key="Child_level3"
                          ItemTemplate="{StaticResource ElementTypeDetail}"
                          ItemsSource="{Binding Children}">
      <TextBlock Text="{Binding Name}"/>

</HierarchicalDataTemplate>

<HierarchicalDataTemplate x:Key="Child_level2"
                          ItemTemplate="{StaticResource Child_level3}"
                          ItemsSource="{Binding Children}">
      <TextBlock Text="{Binding Name}"/>

</HierarchicalDataTemplate>

<HierarchicalDataTemplate x:Key="RootElement"
                          ItemTemplate="{StaticResource Child_level2}"
                          ItemsSource="{Binding Children}">
      <TextBlock Text="{Binding Name}"/>

</HierarchicalDataTemplate>
```

Code snippet 5 - DataTemplate k TreeView

Dále okno zobrazuje tlačítka, vyhledávací pole a podrobné informace ke každému prvku, ty jsou představeny v logickém modelu v kapitole 4.2.3 a pro rozsah této práce je podrobněji představovat nebudu.

3.3.3 Vytvoření tříd pro zacházení s daty – ViewModel

Každý ViewModel dědí základní přepis z třídy *BindableBase,* kterou nám poskytne PRISM framework. Tato třída nám umožňuje zajišťuje správné vystřelení eventu vždy a následné aktualizaci vlastnosti, která je předaná jako parametr. Dále také vyšle informaci do View daného ViewModelu o tom, že vlastnost se změnila a vynutí si její aktualizaci.

ViewModel by se neměl odkazovat na žádnou vlastnost, která se nastavuje ve View, neměl by vědět ani o žádném elementu, který obsahuje View.

3.3.3.1 MainWindowViewModel

Jedná se o ViewModel k výchozímu oknu MainWindow, který není až tak obsáhlý, za to ale velice důležitý. Bez něj by nebylo možné přepínat mezi ostatními ViewModely a tudíž by se ani uživatelovi nezobrazovaly žádné jiné okna, než okno výchozí.

Máme zde důležitou vlastnost třídy, *CurrentViewModel,* který si udržuje informace o momentálně používaném ViewModelu. Po načtení třídy, tedy v konstruktoru, nastavíme tuto *CurrentViewModel* na ViewModel stránky s přihlášením.

```
#region Public Properties
        public BindableBase CurrentViewModel
        {
            get { return _currentViewModel; }
            set { SetProperty(ref _currentViewModel, value); }
        }
#endregion

#region Constructor
        public MainWindowViewModel()
        {
            _elementsTreeListViewModel =
ContainerHelper.Container.Resolve<ElementsTreeListViewModel>();

            _loginViewModel = ContainerHelper.Container.Resolve<LoginViewModel>();

            CurrentViewModel = _loginViewModel;

            _loginViewModel.SwitchViewModel += NavToElementsTreeList;
            _elementsTreeListViewModel.SwitchViewModel += NavToElementsTreeList;
        }
#endregion
```

Code snippet 6 - MainWindowViewModel nastavení ViewModelu

Je zde také řešena logika přepínání ViewModelů, která funguje na bázi Eventů. Z jednotlivých ViewModelů se vystřelí event, který zachytí rodičovský ViewModel – MainWindowViewModel a poté provede příslušnou metodu.

```
private void NavToElementsTreeList(string navParam)
{
	switch (navParam)
	{
		case "Login":
			CurrentViewModel = _elementsTreeListViewModel;
			break;
		case "Logout":
			CurrentViewModel = _loginViewModel;
			break;
	}
}
```

Code snippet 7 - MainWindowViewModel navigace mezi ViewModely

Jako parametr si metoda vyžádá *string*, který určí, na jaký ViewModel se má *CurrentViewModel* nastavit, a tudíž i zobrazit uživateli.

3.3.3.2 ElementsTreeListViewModel

V této třídě probíhá veškerá logika k oknu ElementsTreeListView, probíhá zde načtení prvků z repositáře, následně je uloží do Listu *_allElements*. S tímto listem dále pracuje tak, že si vytřídí prvky, sestaví hierarchii a již hotový seznam prvků s hierarchií uloží do *ObservableCollection*, kterou předává View.

```
	private void EstablishRelationships()
	{
		_elementsRepo.AddElementTypeToElements(_elementTypes, _allElements);

		_allElements.ForEach(e => e.Children = _allElements.Where(child =>
	child.ParentId == e.Id).OrderBy(x => x.Name).ToList());
		RootElements = new ObservableCollection<Element>(_allElements.Where(e =>
	e.ParentId == null).OrderBy(x => x.Name));

		StatusText = States.Loaded.ToString();
	}
```

Code snippet 8 - Složení hierarchie

Je zde také řešena logika zacházení s tlačítky. Každé tlačítko je typu *RelayCommand*, které bere jako parametr příslušnou metodu, jež se vykoná vždy po stisku tlačítka.

```
#region Commands
        public RelayCommand ClearFilterInputCommand { get; set; }
        public RelayCommand<Element> ExportElementToCsvCommand { get; set; }
        public RelayCommand LogoutCommand { get; set; }
#endregion


private void OnFilterInputClear()
{
       FilterInput = null;
       IsElementSelected = false;
       EstablishRelationships();
}

private void OnExportToCsv(Element selectedElement)
{
       StatusText = States.Export.ToString();
       string defaultPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

       if (selectedElement != null)
       {
              ExportToCsv(defaultPath, selectedElement.Name, selectedElement.ToList());
       }
       else
       {
              ExportToCsv(defaultPath, $"Export_{ DateTime.UtcNow:dd/MM/yyyy}",
       _allElements);
       }

       StatusText = States.Ready.ToString();
}

private void OnLogout()
{
       SwitchViewModel("Logout");
}
```

Code snippet 9 - Logika pro tlačítka

3.3.4 Vytvoření tříd modelů – Model

Třída modelu obsahuje předpis atributů, které naplníme z repositáře. Model neví nic o stavu ovládacích prvků, je pouze nosičem dat, ke kterým pak přistupujeme instanci jeho objektu.

Vytvoříme si tedy novou public třídu, kam nasypeme public atributy, ve kterých budou jednotlivá data.

Pro demonstraci jsem si vybral nejobsáhlejší modelovou třídu *Element.cs.*

```
public string Id { get; set; }
public string ParentId { get; set; }
public string TechParentId { get; set; }
public int? AliasId { get; set; }
public string Name { get; set; }
public string TypeId { get; set; }
public string ExternalId { get; set; }
public string Data { get; set; }
public string UserData { get; set; }
public string StateId { get; set; }
public DateTime CreatedOn { get; set; }
public DateTime ModifiedOn { get; set; }
public DateTime? StateSetOn { get; set; }
public int? Icon { get; set; }
public object Bypass { get; set; }
public List<Spatial> Spatial { get; set; }
public List<Element> Children { get; set; }
public ElementType ElementType { get; set; }
```

Code snippet 10 - Atributy modelu Element

3.3.5 **Vytvoření repozitářů**

Všechny repozitáře si implementují interface, v závislosti na typu prvků, se kterými budou jednat. Implementují je proto, aby bylo snadné zjistit, jaké metody, funkce nebo vlastnosti mají dané třídy mít.

Repozitáře nám získají data ze serveru, následně je zpracují a uloží do datových modelů.

3.3.6 **Připojení se k serveru**

Připojení k serveru funguje na bázi HttpClienta (System.Net.Http). Vytvoříme si tedy třídu *ClientHelper.cs,* která poslouží jako zjednodušení celého procesu připojení. V této třídě máme dvě metody:

a. GetClient za užití *string username* a *string password:*

```
        public static HttpClient GetClient(string username, string password)
        {
            var authValue = new AuthenticationHeaderValue("Basic",
Convert.ToBase64String(Encoding.UTF8.GetBytes($"{username}:{password}")));

            var client = new HttpClient()
            {
                DefaultRequestHeaders = { Authorization = authValue },
                BaseAddress = new Uri(Properties.Settings.Default.BaseUrlAddress)

            };
            return client;
        }
```

Code snippet 11 - GetClient(username, passsword)

b. GetClient za použití *string token:*

```
        public static HttpClient GetClient(string token)
        {
            var authValue = new AuthenticationHeaderValue("Bearer", token);

            var client = new HttpClient()
            {
                DefaultRequestHeaders = { Authorization = authValue },
                BaseAddress = new Uri(Properties.Settings.Default.BaseUrlAddress)
            };
            return client;
        }
```

Code snippet 12 - GetClient(token)

Tyto metody nám na základě buď poskytnutého tokenu nebo uživatelského jména a hesla vrátí vytvořenou instanci objektu HttpClient, se kterým budeme dále pracovat. Demonstraci příkladu provedu na repozitáři *ElementsRepository.cs,* díky kterému se můžou načíst jednotlivé prvky ze serveru.

V repozitáři si otevřeme tedy nového *HttpClienta* a vytvoříme *HttpResponseMessage* (System.Net.Http), do které uložíme výsledek našeho dotazu na server. V tomto případě se jedná o GET dotaz. Pokud je vlanost objektu *HttpResponseMessage.IsSuccessStatusCode* nastavená na *true,* můžeme uložit výsledek do námi avizovaného modelu.

```
        public async Task<List<Element>> GetElementsAsync()
        {
            using (var httpClient =
ClientHelper.GetClient(AccessToken.GetAccessToken().GetAccessTokenString().Replace("\n
", "")))
            {
                using (HttpResponseMessage response = await httpClient.GetAsync(URL))
                {
                    if (response.IsSuccessStatusCode)
                    {
                        var elements = await
response.Content.ReadAsAsync<List<Element>>();

                        return elements;
                    }
                    else
                    {
                        throw new Exception(response.ReasonPhrase);
                    }
                }
            }
        }
```

Code snippet 13 - Načítání elementů

3.3.7 **Dependency injection kontejner**

Kontejner pro dependency injection nám usnadňuje práci při inicializaci závislostí, a to tak, že to celé udělá za nás. Tedy ještě před spuštěním aplikace nám inicializuje celý strom závislostí, obsahující v kontejneru.

Na výběr je z několika typů kontejnerů, pro moji aplikaci jsem zvolil Unity kontejner, který lze získat z úložiště balíčků NuGet – Unity.Container.

V kontejneru si zaregistrujeme typy, kde asociujeme Interface třídy se samotnou třídou. Poté kdykoliv v aplikaci si řekneme v konstruktoru o danou instanci třídy, tak jí dostaneme a nemusíme si inicializovat novou instanci třídy pomocí syntaxe *new*. Toto je žádoucí v případě dalšího testování a také je to dosti pohodlné.

```
static ContainerHelper()
{
    _container = new UnityContainer();
    _container.RegisterType<IElementsRepository, ElementsRepository>(
        new ContainerControlledLifetimeManager());
    _container.RegisterType<ITokenRepository, TokenRepository>(
        new ContainerControlledLifetimeManager());
    _container.RegisterType<IElementTypeRepository, ElementTypeRepository>(
        new ContainerControlledLifetimeManager());
    _container.RegisterType<IFileExportRepository, FileExportRepository>(
        new ContainerControlledLifetimeManager());
    _container.RegisterType<ICovidRepository, CovidRepository>(
        new ContainerControlledLifetimeManager());
}
```

Code snippet 14 - Registrace typů v kontejneru

3.3.8 **Zabezpečení**

Připojení k serveru je chráněno přístupovými údaji. Při prvotním připojení je nutné zadat jméno a heslo, při validních údajích nám server vrátí odpověď s autorizačním tokenem, který následně používáme při každém dotazu na data.

Tento token je ukládán ve třídě *AccessToken.cs*, která funguje jako návrhový vzor Singleton – Jedináček. Jeho instance je sdílena na několika místech a není žádoucí vytvářet stále novou instance, ale přebírat data z již vytvořených instancí.

Třídě je nutné zabránit, aby existovala vícekrát, proto vytvoříme prázdný privátní konstruktor. A vytvoříme třídní proměnnou, do které budeme ukládat aktuální instanci a tu na dotaz vracet. Nyní si tedy instance vytvoří sama sebe a tu si uloží do statické proměnné.

```
public sealed class AccessToken
{
    private static readonly AccessToken _accessTokenInstance = new AccessToken();
    private ITokenRepository _repo = new TokenRepository();
    private string _tokenString;


    private AccessToken()
    {

    }

    public static AccessToken GetAccessToken()
    {
        return _accessTokenInstance;
    }

    public async Task InicializeTokenString(string username, string password)
    {
        try
        {
            _tokenString = await _repo.GetTokenAsync(username, password);
        }
        catch (System.UnauthorizedAccessException)
        {
            throw;
        }

    }

    public void SetAccessTokenString(string tokenString)
    {
        _tokenString = tokenString;
    }

    public string GetAccessTokenString()
    {
        return _tokenString;
    }

}
```

Code snippet 15 - Ukázka Singletonu

Singleton je jednoduché a elegantní řešení pro tento typ situací, když máme jednu instanci objektu a potřebujeme si jí předávat na různých místech v aplikaci.

Když si tedy inicializujeme token, následně ho používáme v metodě GetClient viz. kapitola 4.3.6 s parametrem *string token* a přidáváme ho do hlavičky každého requestu. Hlavička je typu Bearer a hodnotou tokenu. Bez tokenu není možné provést žádný úspěšný dotaz na server.

4 Výsledky a diskuse

4.1 Zhodnocení vývoje

Vývoj aplikace probíhal v pořádku a bez nějakých větších komplikací. Pro vývoj byly použity veškeré technologie popsané v teoretické i praktické části. Velké usnadnění programování WPF aplikace s orientací na MVVM architekturu umožnil PRISM framework, který je bezpochyby skvělým nástrojem každého vývojáře aplikací WPF.

4.2 Možná vylepšení

Po celkovém dopsání aplikace a jejím zhodnocení a otestování mě napadly další možná vylepšení, a to:

- Přidat více možností uživateli pro práci s prvky.
- Možnost prvky přidávat i v rámci aplikace, nejen získávání ze serveru.

5 Závěr

Hlavním cílem aplikace byla demonstrace objektově orientovaného programování v .NETu za pomocí technologie Windows Presentation Foundation s ukázkou architektonických návrhů, zejména návrhu Model-View-ViewModel. Tyto cíle byly splněny.

V teoretické části byly představeny technologie použité k vývoji aplikace. Programovací jazyk C# a jeho framework .NET, který slouží nejen k pohodlnému vývoji okenních aplikací. Technologie Windows Presentation Foundation, která je moderním pojetí při psaní okenních aplikací. Následně byla představena architektura použita při návrhu aplikace MVVM – Model-View-ViewModel, její principy a vrstvy. Dále byl byl čtenář uveden do základů knihovny PRISM a jejím obrovským přínosem do vývoje aplikace zaměřené na MVVM architekturu. V neposlední řadě bylo řečeno něco o RESTové API, která zajišťovala data potřebná k aplikaci. Posledními body bylo představení vývojářského prostředí Microst Visual Studio a systém správy verzí, jež je dnes už nezbytný při vývoji aplikací.

Ve vlastní práci proběhl analýza, proč vůbec je dobré, aby takováto aplikace vznikala, jaké technologie jsou vhodné pro vývoj takovéto aplikace a jak by mělo vypadat členění a struktura dat, které do aplikace budou proudit. V návrhu aplikace jsme se vrhli na strukturu návrhových vzorů, vytvořili jsme si diagramy užití s podrobnými scénáři a navrhnuli jsme si základní logický model, který byl předlohou pro vytvoření přehledného a přívětivého grafického uživatelského rozhraní. V poslední řadě jsem popsal implementaci klíčových a zajímavých částí aplikace, od prvotního sestavení projektu, po vytváření oken, jejich tříd s chováním a logikou, datové modely, získávání dat ze serveru, samotné připojení k serveru, návrhový vzor dependency injection a v poslední řadě zabezpečení přístupu k datům ze serveru.

Silná stránka aplikace je určitě v přehlednosti a objektovém přístupu kódu, jeho možné testování pomocí unit testů a také další možná rozšiřitelnost. Díky použití architektonických návrhů je kód krásně rozdělen do malých samosprávních funkčních celků.

Slabá stránka aplikace bude v dosavadní omezené funkcionalitě, kde uživatel nemůže přes aplikaci přidávat žádné nové prvky, momentálně by je musel přidat ručně zásahem do serveru.

Výsledkem celé této práce je funkční okenní aplikace na platformu Windows.

Řádná studie, seznámení se a také již nějaká zkušenost s použitými technologiemi autorovi usnadnila práci při vývoji aplikace. Pro další vývoj okenních aplikací pro platformu Windows autor rád použije stejné technologie.

6 Seznam použitých zdrojů

[1] C Sharp. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-2022 [cit. 2022-03-11]. Dostupné z: https://cs.wikipedia.org/wiki/C_Sharp

[2] POPLE, Aiden. What's the difference between C# and .NET?. In: *CAPITA: IT Blogs* [online]. CAPITA, 2018 [cit. 2022-03-11]. Dostupné z: https://www.capitaitresourcing.co.uk/blogs/whats-the-difference-between-c-and-net-81122210121

[3] ČÁPKA, David. Lekce 1 - Úvod do objektově orientovaného programování v C#. *Itnetwork.cz* [online]. Česká republika: itnetwork.cz [cit. 2022-03-11]. Dostupné z: https://www.itnetwork.cz/csharp/oop/c-sharp-tutorial-uvod-do-objektove-orientovaneho-programovani

[4] HANSELMAN, Scott a Kendra HAVENS. What is .NET? | .NET Core 101 [1 of 8]. In: *Youtube* [online]. dotNET, 2019 [cit. 2022-03-11]. Dostupné z: https://www.youtube.com/watch?v=eIHKZfgddLM&list=PLdo4fOcmZ0oWoazjhXQzBKMrFuArxpW80

[5] What is WPF?. *WPF Tutorial* [online]. wpf-tutorial.com, c2007-2022 [cit. 2022-03-11]. Dostupné z: https://wpf-tutorial.com/about-wpf/what-is-wpf/

[6] Idea. In: *Dotnetportal.cz* [online]. dotnetportal.cz [cit. 2022-03-11]. Dostupné z: https://dotnetportal.blob.core.windows.net/files/Windows-Live-Writer/MVVM-Model-View-ViewModel_DB59/idea_2.png

[7] Model Model-View-ViewModel. *Microsoft Docs* [online]. Microsoft, 2019 [cit. 2022-03-11]. Dostupné z: https://docs.microsoft.com/cs-cz/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

[8] Co je to REST API. In: *Parse error.cz* [online]. Parse-error.cz [cit. 2022-03-11]. Dostupné z: https://www.parse-error.cz/nodejs-tutorial/4/co-je-to-rest-api

[9] HANÁK, Drahomír. Stopařův průvodce REST API. In: *Itnetwork.cz* [online]. itnetwork.cz [cit. 2022-03-11]. Dostupné z: https://www.itnetwork.cz/programovani/nezarazene/stoparuv-pruvodce-rest-api

[10] Vkládání závislostí. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-2022 [cit. 2022-03-11]. Dostupné z: https://cs.wikipedia.org/wiki/Vkládání_závislostí#cite_note-1

[11] Rest. In: *NitroPack* [online]. [cit. 2022-03-11]. Dostupné z: https://cdn-ajfbi.nitrocdn.com/GuYcnotRkcKfJXshTEEKnCZTOtUwxDnm/assets/static/optimized/rev-1e26f48/wp-content/uploads/2020/01/rest-768x580.png

7 Seznam obrázků, code snippetů a tabulek

8 Přílohy

Seznam příloh na flash disku:

1. Zdrojové kódy aplikace
2. Soubory lokálního serveru
3. Spustitelnou aplikaci
4. Bakalářskou práci ve formátu PDF