Gymnázium Christiana Dopplera, Zborovská 45, Praha 5

ROČNÍKOVÁ PRÁCE

Název práce

Vypracoval: Filip Mach
Třída: 8.M
Školní rok: 2025/2026
Seminář: Seminář z programování

Prohlašuji, že jsem svou ročníkovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím s využíváním práce na Gymnáziu Christiana Dopplera pro studijní účely.

V Praze dne 16.1.2026 Filip Mach

Obsah

1. Úvod

V této ročníkové práci se budu zabývat tvorbou aplikace pro správu členské databáze v C#. Motivace za zpracováním tohoto tématu je možnost vytvořit funkční aplikaci, která by mohla být v budoucnu využívána existujícími organizacemi pro správu jejich členů.

Práce je rozdělena na dvě části, a to teoretickou a praktickou. V teoretické části se budeme zabývat architekturou MVVM, kterou jsme využili při tvorbě této aplikace. V praktické části se poté podíváme konkrétně na naši aplikaci a její implementaci.

2. Teoretická část

2.1 MVVM architektura

MVVM architektura (Model-View-ViewModel) je způsob architektury, který se soustředí na vzájemnou nezávislost uživatelského rozhraní a logiky aplikace. To zaručuje pomocí rozdělení aplikace do tří vrstev: Model, View a ViewModel. Tyto vrstvy spolu navzájem komunikují takovým způsobem, aby na sobě byly co nejméně závislé.

2.1.1 Motivace pro použití

Jaká je motivace pro rozdělení aplikace do několika téměř nezávislých celků? Výhody přicházejí jak při vývoji, tak při údržbě programu. V prvotní fázi tvorby je ideální mít možnost programovat nezávisle na sobě grafické rozhraní a logiku aplikace. To umožní, aby na každé části pracoval samostatný tým, který nebude zpomalován pokrokem týmu druhého. S tímto nezávislým vývojem souvisí i to, že v případě potřeby úpravy jedné z částí se nemusí vůbec upravovat část druhá. Například pokud by bylo třeba přidat nové textové pole do grafického rozhraní, tato úprava by se nijak nedotkla samotné logiky. Tato vlastnost také umožňuje testování každé části zvlášť, což je užitečné především při vývoji nových částí aplikace. Tento způsob architektury je tedy ideální pro větší projekty, na kterých pracuje několik týmů nebo pro projekty, u kterých se očekává častá potřeba úprav. Naopak nevhodný je pro malé projekty, jelikož je příliš komplikovaný.

2.1.2 Model

Model slouží jako vrstva, která reprezentuje doménu aplikace a její data. Jeho účelem je tedy vymezit, jaká data aplikace obsahuje a jaká pravidla pro ně platí. Je nezávislý na uživatelském rozhraní, neslouží tedy k omezení toho, jak se data zobrazují nebo jak s nimi uživatel interaguje.

Model typicky obsahuje entity (například v naší aplikaci Person, Team atd.), jejich vlastnosti a vztahy mezi nimi. Může také zahrnovat business logiku, což jsou pravidla určující, co je při nakládání s daty povoleno a co ne (například že není možné smazat tým, který má nějaké členy). Jak jsme psali výše, zásadní pro MVVM je, že Model je naprosto nezávislý na uživatelském rozhraní, které lze tedy vždy libovolně upravovat. Oddělení datového modelu a business logiky od prezentační vrstvy je základním principem moderních aplikačních architektur. [2]

2.1.3 View

View je vrstva, která se stará o uživatelské rozhraní a jeho vzhled. Laicky řečeno jde o to, co uživatel první uvidí, když zapne aplikaci. V rámci aplikace je jeho hlavním úkolem zobrazit data, která pocházejí z Modelu, a umožnit uživateli s nimi interagovat. Tyto interakce zajišťuje takzvaný data binding.

Jednou z důležitých vlastností View je, že by mělo být co nejvíce „hloupé“. To znamená, že by se nemělo vůbec rozhodovat, co zobrazí, pouze by mělo správně zobrazit cokoli, co mu přijde. Stejně tak by se nemělo rozhodovat, co se stane po kliknutí na tlačítko, místo toho by mělo pouze volat příkazy, které jsou definovány ve ViewModelu. Tento přístup odpovídá doporučením architektury MVVM v prostředí WPF, kde je View určeno pouze k prezentaci dat a veškerá logika je přesunuta do ViewModelu [1].

Existuje několik způsobů, jak View implementovat, ovšem v okenních aplikacích tvořených pro Windows a Linux ho většinou tvoří několik XAML souborů. V naší práci pracujeme s Windows Presentation Foundation, která volí právě tento přístup k View.

2.1.4 ViewModel

Hlavním účelem ViewModelu je získávat data z Modelu. Zároveň zpracovává vstupy z View a reaguje na ně tak, aby součástí View nebyla žádná logika. Slouží tedy jako jakýsi prostředník mezi Modelem a View.

Jak se ale data mezi vrstvami přesouvají? Samotný ViewModel umí načíst data z Modelu, ovšem o View vůbec „neví“. Závislosti v MVVM jdou totiž pouze jedním směrem, a to sice View → ViewModel → Model. To znamená, že View umí nahlížet do ViewModelu, ViewModel do Modelu, ale ne do View, a Model nikam. Přesun informací z Modelu do ViewModelu je tedy jednoduchý. ViewModel si pomocí funkce načte data z Modelu, aby je mohl získat View.

Zobrazení dat ve View je trochu složitější a probíhá přes dříve již zmíněný data binding. Každý soubor ve View (typicky jeden soubor pro jedno okno aplikace) má nastavený DataContext. Ten odkazuje na ViewModel příslušející danému oknu. Samotné View žádná data neobsahuje (aby bylo co nejjednodušší), pouze se odkazuje na data ve ViewModelu. DataContext určuje, kde má View potřebná data hledat, ovšem nedefinuje, jaká konkrétní data chce View získat. Zde přichází na řadu data binding, který propojí entitu z View s vlastností ViewModelu. Například pokud chceme načíst seznam lidí, který je ve ViewModelu uložen pod názvem People, odkážeme se na tento název ve View. View sice žádnou vlastnost People nemá, ale data binding se „podívá“ do ViewModelu, tuto vlastnost najde a předá ji View. Pokud tedy View zná názvy vlastností ViewModelu, může data z něj kdykoli získat, přestože ViewModel o View „neví“.

3. Praktická část

V této části se budu zabývat aplikací, kterou jsme vytvořili v rámci této ročníkové práce.

3.1 Cíl

Cílem naší aplikace je poskytnout jednoduché řešení pro správu členů určitého spolku. Tato správa by měla zahrnovat možnost ukládat obecné informace (jméno, datum narození, bydliště apod.), informace potřebné pro spolek (třída, družstvo, role apod.) i základní prostředky komunikace se samotnými členy. Mezi ty by se dalo zařadit posílání zpráv členům, případně jejich rodičům, omlouvání absence nebo přihlašování na různé soutěže.

Jelikož se nám nepodaří naprogramovat všechny funkcionality v rámci této práce, je pro aplikaci naprosto zásadní její rozšiřitelnost. Je velmi pravděpodobné, že se aplikace bude nadále vyvíjet a bude potřeba doplňovat nové funkce. K jednoduchým úpravám nám poslouží jak dříve popsaný architektonický vzor MVVM, tak správné navržení databáze a funkcí.

3.2 Databáze

Samotná data o uživatelích je třeba nějakým způsobem ukládat. My jsme se rozhodli, že využijeme databázi. Nechceme, aby byla aplikace omezena maximálním počtem dat, která lze uložit, jelikož by to výrazně limitovalo velikost spolku schopného naši aplikaci využívat. To tedy vyřazuje jednoduché metody ukládání dat, jako je například ukládání do textového souboru. Zároveň potřebujeme, aby data byla bezpečně uložena, zejména ta citlivá, a abychom k nim měli rychlý přístup. Jako nejlepší možnost se tedy jeví využití databáze.

3.2.1 Architektura

Našim potřebám nejvíce vyhovuje takzvaný relační model databáze. To znamená, že databáze se bude skládat z několika tabulek, které spolu budou propojeny určitými vztahy. Nejprve je ale třeba rozhodnout, jaká data o členech budeme shromažďovat.

Zcela jistě budeme chtít ukládat základní osobní údaje o každém členovi. Mezi ty se řadí: jméno, příjmení, datum narození, rodné číslo, údaje o zdravotním pojištění a údaje o trvalém bydlišti. Dále můžeme potřebovat rozdělit členy na různé skupiny v rámci našeho spolku. Budeme předpokládat, že toto rozdělení proběhne podle dvou typů: role každého člena (trenér, cvičenec, učitel apod.) a tým každého člena (družstvo, třída apod.). Tyto role a týmy vytvoří správce každého klubu podle svých potřeb. Dále budeme ukládat informaci o tom, zda je konkrétní člen stále aktivní. To nám zajistí možnost uchovávání členů, kteří již nejsou v našem spolku, aniž by se pletli s aktivními členy. Dále každému členovi přiřadíme

takzvaný primární klíč v podobě ID. Všechny tyto údaje budeme zapisovat do tabulky s názvem People.

Nejprve ale budeme potřebovat vytvořit seznam rolí, které bude moci jednotlivý člen mít, a seznam týmů, do kterých bude moci patřit. Pro každou z těchto vlastností tedy vytvoříme vlastní tabulku, do které samotný správce spolku doplní konkrétní hodnoty (například název týmu). V tabulce People se na ně pak budeme odkazovat pomocí cizích klíčů. Použití primárních a cizích klíčů pro vyjádření vztahů mezi tabulkami je základním principem relačního databázového modelu [3].

Jedním z možných rozšíření naší aplikace je například omlouvání absence nebo její zapisování. Pro tyto potřeby by se vytvořilo několik dalších tabulek pro zaznamenávání tréninků, zapisování absence na nich a také pro ukládání přijatých omluvenek od členů, případně jejich rodičů.

3.2.2 Implementace

Databázi je možné implementovat několika různými způsoby. Pro nás je důležité, abychom s ní mohli jednoduše komunikovat pomocí našeho kódu psaného v C#. V kompletní aplikaci připravené na použití je samozřejmě potřeba, aby databáze byla na vzdáleném serveru, ke kterému se budeme připojovat přes internet. To proto, aby mohlo více uživatelů z různých zařízení upravovat a sledovat údaje v ní. Vzhledem k rozsahu této ročníkové práce jsme se ale rozhodli databázi vytvořit pouze lokální. Pokud se tedy aplikace spustí na novém zařízení, bude vytvořena zcela prázdná databáze.

Naši databázi jsme se rozhodli vytvořit pomocí Entity Framework Core (dále jen EF Core), a to takzvaným přístupem „code first“ [4]. To znamená, že komunikaci s databází bude zajišťovat rozšíření EF Core pro Visual Studio a databáze bude na míru vytvořena pro tento přístup. Jak již název napovídá, „code first“ znamená, že databázi budeme generovat pomocí kódu. Opakem tohoto přístupu je takzvaný „database first“, ve kterém již databáze existuje a my kolem ní chceme postavit logiku aplikace. Tento přístup ovšem není ideální pro nás, jelikož si můžeme databázi navrhnout přesně tak, aby se nám s ní dobře pracovalo.

Samotný přístup „code first“ funguje tak, že každá tabulka v databázi je reprezentována určitou třídou. Tato třída má stejné vlastnosti, jaké má jeden záznam v dané tabulce. Zjednodušeně řečeno lze říci, že každý sloupec v tabulce odpovídá jedné vlastnosti této třídy. Jak bylo uvedeno výše, pro nás jsou důležité tabulky People, Roles a Teams, vytvoříme jim tedy odpovídající třídy. Poté vytvoříme databázový kontext, ve kterém zadáme EF Core, aby tabulky na základě zadaných tříd vytvořil. Výsledná databáze bude odpovídat schématu uvedenému v příloze.

3.3 Architektura

V rámci naší aplikace jsme se rozhodli využít architekturu MVVM. Jelikož ta je již teoreticky popsaná dříve v této práci, v této kapitole se budu věnovat spíše její implementaci.

3.3.1 Model

Do vrstvy Modelu patří v naší aplikaci doménové entity, DbContext a datové služby sloužící k získávání dat z databáze. Doménové entity a DbContext jsme již podrobněji popsali v kapitole výše, budeme se tedy věnovat datovým službám.

Datové služby (data services) jsou funkce a metody, jejichž úkolem je pracovat s daty uloženými v databázi. To zahrnuje jak základní CRUD operace, tak složitější úkony. CRUD operace, neboli Create, Read, Update a Delete, umožňují nakládání s daty přesně takovým způsobem, který je zřejmý z jejich pojmenování. To znamená, že pro vytváření nových dat a jejich uložení do databáze slouží funkce Create, pro načtení dat funkce Read, pro úpravu již uložených dat funkce Update a pro úplné odstranění existujícího záznamu funkce Delete. Tyto operace jsme v naší aplikaci implementovali v rámci několika různých datových služeb.

GenericDataService je obecná datová služba umožňující práci z daty jakéhokoli platného typu. To znamená, že lze použít pro nahrávání dat jak typu Person, Team nebo třeba Role. Dále jsme vytvořili služby *PersonDataService, RoleDataService* a *TeamDataService*, které slouží ke specifické práci s těmito konkrétními entitami.

GenericDataService je obecná datová služba umožňující práci s daty jakéhokoli platného typu. To znamená, že ji lze použít pro práci s daty typu Person, Team nebo například Role. Dále jsme vytvořili služby *PersonDataService*, *RoleDataService* a *TeamDataService*, které slouží ke specifické práci s těmito konkrétními entitami.

PersonDataService slouží k načítání záznamů o jedné osobě z několika tabulek v rámci databáze. Jak jsme psali výše, zápisy o jedné osobě jsou v rámci databáze propojeny pomocí primárních a cizích klíčů. V praxi tedy máme v jedné tabulce uloženo několik týmů, z nichž každému přísluší konkrétní ID. V druhé tabulce máme uložena všechna potřebná data o jednom členovi a ve sloupci, který jej přiřazuje k danému týmu, je odkaz na ID v první tabulce. *PersonDataService* načte nejen informace z tabulky osob, ale prostřednictvím cizího klíče také informace příslušející danému členovi z jiné tabulky. Lze tak například načíst název týmu, do kterého daný člen patří, přestože není uložen v původní tabulce členů. *PersonDataService* také implementuje několik různých funkcí pro načítání osob podle určité vlastnosti, jako například *GetByLastName*.

3.3.2 View

Vrstva View se v naší aplikaci skládá z několika souborů. Nejdůležitější jsou *Views*, které reprezentují jednotlivé obrazovky či stránky aplikace. K těm jsou navázány takzvané Controls, což jsou menší nezávislé objekty, které jsou implementovány ve vlastním souboru, díky čemuž je lze používat v několika různých *Views*. Tato vrstva také obsahuje *Styles*, které upravují, jak se jednotlivé objekty zobrazují.

Hlavní okno naší aplikace je rozděleno do několika částí. V levé části se nachází logo a název projektu a pod nimi je navigační menu. Vpravo se pak zobrazují jednotlivé *Views* v závislosti na tom, jakou položku z navigačního menu máme zrovna vybranou.

Navigační menu je jedním z příkladů využití *Controls*. Jelikož je v celé aplikaci stejné, je ideální jej implementovat ve vlastním souboru a pro zobrazení pak pouze využívat odkaz na tento soubor. Další výhodou je, že pro přidání například dalšího tlačítka pro navigaci stačí změnit zdrojový soubor tohoto navigačního menu a změna se projeví všude.

Navigační menu zároveň obsahuje několik tlačítek, u kterých bylo třeba změnit jejich vzhled. K tomu slouží styly, kterými lze upravovat podobu konkrétních objektů napříč celou aplikací. V tomto případě bylo potřeba změnit několik vlastností v závislosti na tom, zda máme na tlačítku v menu myš, nebo zda je toto tlačítko momentálně stisknuté. Všechny tyto podmínky a chování, která nastávají v případě jejich splnění, jsme popsali v souboru se stylem menu a tento soubor jsme poté nastavili jako zdroj stylování tohoto menu.

Různé styly a *Controls* jsme v naší aplikaci použili několikrát, ovšem všechny tyto případy fungují na podobném principu, nebudeme se jimi tedy dále zabývat.

3.3.3 ViewModel

Jak jsme psali výše, hlavním úkolem ViewModelu je získávat data z Modelu a upravit je tak, aby bylo možné je zobrazit ve View. Každému View přísluší tedy právě jeden ViewModel. Níže se budeme zabývat funkcí tří z nich.

3.3.4 PeopleViewModel

PeopleViewModel slouží k tomu, aby načetl z databáze informace o všech členech a uložil je do kolekce. *PeopleView* pak tyto informace zobrazí v tabulce. Samotný ViewModel je tedy velice jednoduchý. Má jednu veřejnou vlastnost *People*, ve které je uložena kolekce lidí. V konstruktoru voláme funkci, která pomocí *PeopleDataService* načte data z databáze a uloží je do této kolekce. Toto je tedy krok, ve kterém probíhá komunikace mezi ViewModelem a Modelem. V samotném View se poté pomocí data bindingu odkazujeme přímo na data v kolekci *People*. Nakonec se tato data zobrazí v tabulce ve výstupu samotného View. Tímto způsobem se tedy k uživateli dostanou všechny potřebné informace o všech členech.

3.3.5 PeopleSearchViewModel

Velice podobný, ovšem mírně složitější, je *PeopleSearchViewModel*. Ten odpovídá View, které zobrazuje informace o členech, umožňuje s nimi manipulovat a umožňuje vyhledávat konkrétní členy podle určitého kritéria. Zobrazení dat tedy probíhá stejným způsobem jako v předchozím případě.

Úprava informací o členovi probíhá tak, že uživatel klikne na příslušné pole v tabulce a hodnotu v něm přepíše. Ve chvíli, kdy úpravu dokončí, se spustí metoda definovaná v code-behind View, která předá ViewModelu upravená data a ten je uloží do databáze. To probíhá pomocí funkce *Update*, která najde příslušného člena a jeho informace upraví. Nakonec opět načteme upravenou osobu z databáze, aby se změny projevily v uživatelském rozhraní.

Vyhledávání členů podle určitého kritéria je třetí funkcionalitou tohoto View. Uživatel si v menu vybere, podle jaké vlastnosti chce členy vyhledávat, a poté zadá hledanou hodnotu. Například pokud si vybere hledání podle názvu týmu, zadá do vyhledávacího TextBoxu název tohoto týmu. Poté klikne na tlačítko Hledat, což spustí příkaz definovaný ve ViewModelu.

ViewModel má vlastnost *SearchText*, která se pomocí data bindingu automaticky mění při každé úpravě textu ve View, a vlastnost *SelectedSearchField*. Obě tyto vlastnosti jsou viditelné pro funkci *Search*, která se spustí po kliknutí na tlačítko Hledat. Funkce *Search* se podívá na *SelectedSearchField*, čímž zjistí, podle jaké vlastnosti má hledat, a následně vyhledá členy s hodnotou uloženou ve vlastnosti *SearchText*. Nakonec všechny nalezené členy vrátí View, které je zobrazí.

3.3.6 TeamRoleCRUDViewModel

Posledním zajímavým ViewModelem je *TeamRoleCRUDViewModel*. Ten umožňuje provádět CRUD operace pro role a týmy. Jelikož jsou všechny operace pro týmy a role analogické, budeme popisovat pouze to, jak fungují pro týmy.

Přidání nového týmu je jednoduché. Uživatel napíše do okénka název nového týmu a klikne na tlačítko Uložit. Spustí se funkce, která provede jednoduchou operaci Create, již jsme popsali dříve. Zobrazení všech týmů je také stejné jako v ostatních případech, pouze se tentokrát týmy zobrazují pomocí ComboBoxu. To znamená, že když uživatel klikne na pole, rozbalí se názvy všech týmů, ze kterých si může vybrat. To slouží k tomu, aby mohl uživatel s vybraným týmem dále pracovat.

Odstranění týmu je trochu složitější. Nechceme totiž dovolit, aby byl smazán tým, do kterého někdo patří. Došlo by pak ke kaskádovému mazání, jak jsme popisovali v **kapitole 2.1.3**. Pro mazání tedy využíváme *TeamDataService*, který při pokusu o odstranění nevyhovujícího týmu vyvolá výjimku. Díky bloku *try-catch* v kódu dokážeme tuto situaci zachytit a mazání týmu přerušit. Uživatel je na tuto skutečnost upozorněn prostřednictvím vyskakovacího okna.

4. Závěr

Cílem této ročníkové práce bylo vytvořit aplikaci, která by mohla být v budoucnu využívána existujícími organizacemi pro správu jejich členů. Tohoto cíle jsme dosáhli částečně.

Naše aplikace sice nemá všechny původně zamýšlené funkcionality, ovšem je postavena tak, aby byla dále rozšiřitelná a modifikovatelná. Díky těmto vlastnostem je tedy ideálně připravena na to, aby byla dále vyvíjena a v budoucnu skutečně využitelná.

Závěrem lze tedy říci, že práce byla úspěšná. Vytvořili jsme funkční aplikaci, která je připravena na další vývoj, s cílem vytvořit reálně použitelnou aplikaci.

Literatura

[1] MICROSOFT. *Model-View-ViewModel (MVVM)*. Online. Microsoft learn. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm. [cit. 2025-12-21].

[2] Fowler, Martin. *Patterns of enterprise application architecture*. Addison-Wesley, 2012.

[3] Silberschatz, Abraham, Henry F. Korth, and Shashank Sudarshan. *Database system concepts*. Vol. 5. New York: Mcgraw-hill, 2002.

[4] Microsoft. *Entity Framework Core*. Online. Microsoft learn. Dostupné z: https://learn.microsoft.com/en-us/ef/core/modeling/. [cit. 2025-12-21].

Přílohy