Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Vývoj desktopové aplikace pro půjčovnu zvedacích plošin

Bc. Ladislav Marks

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Ladislav Marks

Informatika

Název práce

Vývoj desktopové aplikace pro půjčovnu zvedacích plošin

Název anglicky

Development of desktop application for lift platform rental company

---

Cíle práce

Hlavním cílem diplomové práce je implementace interní desktopové aplikace na platformě .NET pro společnost vypůjčující pracovní plošiny. Aplikace bude umožňovat vedení evidence vlastněných plošin a jejich výpůjček. V aplikaci budou dále automatizovány některé opakovaně vykonávané činnosti, přičemž dojde k jejich zrychlení a zamezení lidských chyb.

Metodika

Diplomová práce se skládá ze dvou částí. V teoretické části bude metodikou analýza informačních zdrojů. Na základě zjištěných poznatků budou formulována teoretická východiska práce.

V praktické části bude navržena a implementována konkrétní aplikace na zvolené platformě s využitím nabytých teoretických východisek. Následně bude aplikace nasazena, otestována a bude zhodnoceno její budoucí možné rozšíření. Dále budou shrnuty zkušenosti a poznatky z její implementace.

Doporučený rozsah práce

60-80 stran

Klíčová slova

Desktopová aplikace, .NET, WPF, Windows, C#, Entity Framework, Visual studio

---

Doporučené zdroje informací

LITVINAVICIUS, Taurius. Exploring Windows Presentation Foundation: with practical applications in .NET 5 [online]. New York, NY: Apress, 2021. ISBN 978-1-4842-6637-3.

PRICE, Mark J. C# 10 and .NET 6 – modern cross-platform development: build apps, websites, and services with ASP.NET Core 6, Blazor, and EF Core 6 using Visual Studio 2022 and Visual Studio Code. Sixth edition. Birmingham: Packt, 2021. ISBN 978-1-80107-736-1.

VIRIUS, Miroslav. Programování v C#: od základů k profesionálnímu použití. Praha: Grada Publishing, 2021. Myslíme v.. ISBN 978-80-271-1216-6.

Vývojářské nástroje, technická dokumentace a příklady kódování | Microsoft Docs. [online]. Microsoft, 2022. Dostupné z: https://docs.microsoft.com/cs-cz/

Předběžný termín obhajoby

2022/23 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 4. 11. 2022

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 28. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 28. 03. 2023

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Vývoj desktopové aplikace pro půjčovnu zvedacích plošin“ jsem vypracoval samostatně pod vedením vedoucího diplomové 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 uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31. března 2023 _____

Poděkování

Rád bych touto cestou poděkoval panu Ing. Jiřímu Brožkovi, Ph.D. za vstřícnost, ochotu a poskytnutí cenných rad při zpracování mé diplomové práce.

Vývoj desktopové aplikace pro půjčovnu zvedacích plošin

Abstrakt

Tato práce se je zaměřena na problematiku vývoje desktopových aplikací v rámci platformy .NET.

V teoretické části je nejprve popsán objektově orientovaný přístup k vývoji aplikací, historie jazyka C# a platformy .NET. Dále jsou popsány možnosti vývoje desktopových aplikací na platformě .NET s využitím různých frameworků, technika objektově relačního mapování s využitím Entity Framework Core, architektura Model-View-ViewModel a jazyk UML pro návrh informačních systémů.

Předmětem praktické části aplikace je vývoj informačního systému pro společnost půjčující pracovní plošiny. Nejprve jsou analyzovány požadavky na aplikaci a stanoveny případy užití. Dále je navržen datový model aplikace, její architektura, prezentační vrstva a je vybrána vhodná technologie pro implementaci. Následně je zdokumentována implementace aplikace zahrnující implementaci základní infrastruktury, vrstvy přístupu k datům, prezentační vrstvy a dalších služeb. Nakonec je provedeno uživatelské testování a jsou navrženy směry budoucího rozšíření.

Klíčová slova: Desktopová aplikace, .NET, WPF, Windows, C#, Entity Framework, Visual studio

Development of desktop application for lift platform rental company

Abstract

This thesis is focused on the development of desktop applications within the .NET platform.

The theoretical part first describes the object-oriented approach to application development, the history of the C# language and the .NET platform. It also describes the features of development of desktop applications on the .NET platform using various frameworks, the technique of object-relational mapping using the Entity Framework Core, the Model-View-ViewModel architecture and the UML language for the design of information systems.

The practical part describes the process of an information system development for work platforms lending company. First, the requirements for the application are analyzed and use cases are determined. Next, the data model of the application, its architecture, presentation layer and the technology for implementation is selected. Then, the implementation of the application is documented, including the implementation of the application infrastructure, data access layer, presentation layer and other services. Finally, user testing is carried out and directions for future expansion are proposed.

Keywords: Application for desktop, .NET, WPF, Windows, C#, Entity Framework, Visual studio

Obsah

Seznam obrázků

Seznam tabulek

Seznam kódů

Seznam použitých zkratek

OOP – Objektově orientované programování

LINQ – Language Integrated Query

WPF – Windows Presentation Foundation

MVVM – Model-View-ViewModel

EF – Entity Framework

UML – Unified Modeling Language

1 Úvod

Malé podniky poskytující služby, jako např. půjčování pracovních nástrojů, často nepoužívají vhodný informační systém pro evidenci záznamů o předmětu jejich podnikání. Evidencí těchto informací jsou často pověřeny osoby, které nedisponují znalostmi o fungování informačních systémů a pro evidenci různých informací používají některý tabulkový procesor. Evidence pomocí tabulkového procesoru umožňuje vykonávat tuto činnost osobami, které nemusí mít velkou znalost v oblasti IT a jedná se také o zdánlivě finančně výhodné řešení. Vedení společností také často nevidí potenciál ve zvýšení efektivity při vykonávání pracovních procesů, které by bylo výsledkem zavedení vhodného informačního systému, a tak jsou spokojeni se zavedenými podnikovými procesy, které mohou být značně neefektivní a které mohou zaměstnance vytěžovat více, než je nutné.

Zavedení informačního systému, který by automatizoval opakovaně vykonávané podnikové procesy, by přineslo snížení pracovní zátěže zaměstnanců, kterým by mohla připadnout další agenda. Mohl by být tak snížen počet zaměstnanců, což by snížilo náklady na provoz společnosti a zvýšilo zisk. Zavedení takového systému sice vyžaduje počáteční investici, ale ta se může během času vrátit úsporou na provozních nákladech. Zároveň je možné zabezpečit evidované záznamy tak, aby byly vždy v konzistentní formě a nebylo možné s nimi nesprávně manipulovat. Bylo by tak sníženo riziko pochybení v důsledku lidského faktoru.

Nový informační systém by mohl také nabídnout funkce pro podporu rozhodování, které by v rámci řešení evidence pomocí tabulkového procesoru nebylo možné implementovat, nebo by to bylo značně obtížné.

Tato práce je zaměřena na vývoj systému, který optimalizuje vybrané podnikové procesy ve společnosti. Při implementaci je vycházeno z analýzy stávajících procesů a návrhu architektury aplikace, která umožní její další rozvoj v budoucnu.

2 Cíl práce a metodika

2.1 Cíl

Cílem této práce je vytvoření informačního systému pro společnost vypůjčující pracovní plošiny, který nahradí stávající řešení tvořené nezávislými tabulkami v tabulkovém procesu. Na základě provedené analýzy procesu a návrhu softwarového řešení bude implementována desktopová aplikace, ve které bude možné spravovat záznamy o plošinách, jejich výpůjčkách a údržbách. Pro evidenci nových záznamů budou sloužit přehledné formuláře, které budou validovány, aby byla zajištěna konzistence uložených záznamů. V aplikaci bude možné provádět pouze operace, které neohrozí konzistenci datové struktury. Data budou ukládána mimo samotnou aplikaci v databázovém systému, čímž bude zajištěna možnost přístupu k nim z více stanic bez přístupu k internetu. Aplikace bude nabízet přívětivé uživatelské rozhraní, což zajistí snadnou orientaci uživatelů v interakci s ní. Dále budou implementovány přehledy dat, které budou fungovat jako podpora při rozhodování vedení společnosti a export pohledů na data do tabulkového procesoru pro zajištění možnosti provádět operace s daty, které nejsou v aplikaci implementovány.

2.2 Metodika

Diplomová práce se skládá ze dvou částí, a to teoretické části a praktické časti.

V teoretické části bude metodikou analýza informačních zdrojů. Na základě syntézy budou formulována teoretická východiska k problematice vývoje aplikací v jazyce C# na platformě .NET, technologií pro vývoj desktopových aplikací v rámci platformy .NET, objektově relačního mapování a nástrojů pro analýzu a návrh softwarového řešení.

V praktické části bude analyzován současný stav podnikových procesů ve společnosti. Dále budou pomocí rozhovorů s budoucími uživateli formulovány požadavky na funkčnost aplikace. Následně bude za pomocí jazyka UML vytvořen model případů užití a budou specifikovány scénáře případů užití.

Na základě takto provedené analýzy bude navržen pomocí jazyka UML datový model aplikace. Bude navržena architektura aplikace a uživatelské prostředí reprezentované drátovými modely.

Následně bude aplikace implementována ve zvolené technologii pro vývoj desktopových aplikací. Po implementaci bude zhodnocen její vývoj, bude provedeno testování budoucími uživateli budou navrženy směry dalšího vývoje.

3 Teoretická východiska

3.1 Objektově orientované programování

Objektově orientovanému přístupu předcházely funkcionální a imperativní přístupy používané od 30. let 20. století ve spojení s logikou Lambda kalkulu a Turingovým strojem, což jsou univerzální formulace pro obecné výpočty. Nízkoúrovňové jazyky užívající tyto přístupy se objevily ve 40. letech 20. století a mezi ně patřily strojový kód nebo assembler. Na konci 50. let se již objevily jazyky FORTRAN nebo COBOL, které se již řadily mezi jazyky vysokoúrovňové. [1]

Pojem objektově orientované programování (OOP) uvedl Alan Kay mezi lety 1967–1968. Hlavní ideou bylo používání zapouzdřených výpočetních entit, které spolu komunikují předáváním zpráv, a nikoliv přímým sdílením dat a výpočetních postupů. Základními principy OOP dle Alana Kaye jsou:

- Předávání zpráv
- Zapouzdření
- Dynamická vazba [1]

3.1.1 Vlastnosti OOP

V této kapitole byly využity zdroje: [2]

Jednou z hlavních vlastností OOP je **existence tříd a objektů**. Třída je tzn. předpis pro vytvoření vlastního datového typu, která seskupuje související informace a chování. Objekt je poté samotný výskyt (instance) třídy.

Další vlastností je **abstrakce**. Pod tímto pojmem je možné chápat proces filtrování dostupných informací, které jsou potřebné pro konkrétní použití.

Procesem abstrakce je tedy možné definovat, jaká data jsou pro aplikaci potřeba. Proces **zapouzdření** se týká předepsání způsobu jejich zápisu či čtení. Spočívá ve svázání dat a operací do tříd a definici způsobu přístupu k nim. K tomu slouží vlastnosti a metody třídy.

Běžné aplikace obsahují množství tříd. Některé třídy mohou sdílet nějaké informace či operace. Pro vyhnutí se duplicitě atributů v třídách je využívána **dědičnost**. Dědičnost umožňuje izolovat sdílené vlastnosti a metody do obecné třídy a poté vytvořit konkrétní třídu založenou na ní. Obecné třídy se nazývají bázové a specializované třídy se označují jako odvozené. Mezi těmito třídami platí vztah *odvozená třída je bázová*. Odvozená třída může

pak přidávat své atributy nebo měnit chování definované bázovou třídou. Přínosem dědičnosti je znovupoužitelnost a organizace kódu do hierarchické struktury.

Poslední důležitou vlastností OOP je **polymorfismus**. Pod tímto pojmem si lze představit mnoho variant něčeho. V aplikaci mohou existovat dvě třídy, které provádí navenek stejné operace, ale vnitřně mohou tyto operace vykonávat jiným způsobem. Toto je možné implementovat přes již dříve popsanou dědičnost, kdy je vytvořena abstraktní bázová třída s implementací, která ale nemůže být instanciována, nebo pomocí rozhraní, které pouze definuje signaturu třídy, tj. definuje vlastnosti a metody bez jejich implementace.

3.1.2 SOLID principy

Díky OOP je možné vyvíjet aplikace relativně jednoduše v porovnání s předchozími přístupy. Dříve popsané vlastnosti ale nezaručují správnost implementované logiky. SOLID představuje standardy a pokyny, které je vhodné dodržovat pro docílení flexibility a jednoduché rozšiřitelnosti kódu. [2]

Prvním principem je **Single Responsibility princip**. Tento princip definuje, že „třída by měla dělat jednu věc, a proto by měla mít pouze jediný důvod ke změně“. [3] Příkladem může být třída sloužící jako úložiště dat. Tato třída by se měla změnit pouze v případě, kdy je změněn datový model a neměla by být odpovědná za více aspektů aplikace. [3]

Druhým principem je **Open-Closed princip**. Tento princip definuje, že „třídy by měly být otevřeny pro rozšíření a uzavřeny pro úpravu“. [3] Zatímco úpravu lze chápat jako změnu kódu již existující třídy, rozšíření znamená přidávání nové funcionality. Rozšíření funkcionality by se mělo obejít bez zásahů do existujícího kódu, který může být považován za otestovaný a spolehlivý. [3]

Třetí princip se nazývá **Liskov Substitution princip**. Podstatou tohoto principu je nahraditelnost odvozených tříd svými bázovými třídami. V případě, že „třída B je podtřída třídy A, mělo by být možné předat objekt třídy B jakékoli metodě, která očekává objekt třídy A a tato metoda by neměla v takovém případě vracet neočekáváný výstup“. [3]

Čtvrtým principem je **Interface Segregation princip**. Segregace v tomto případě znamená oddělení rozhraní. Rozhraní by mělo být specifické pro konkrétní účely a nemělo by být natolik obecné, aby nutilo implementovat metody, které daná třída nepotřebuje implementovat. [3]

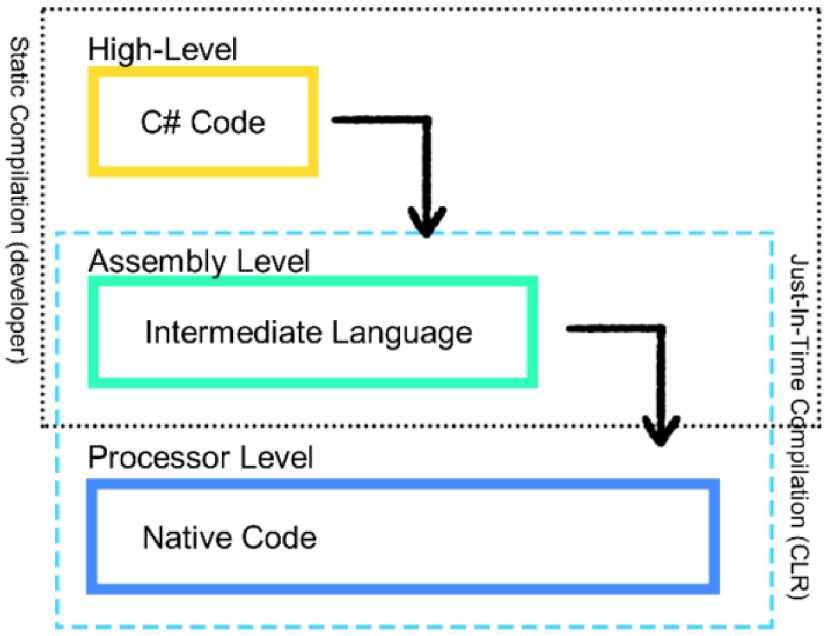
Posledním principem je **Dependency Inversion princip**. Ten říká, že třídy v aplikaci by měly být závislé na rozhraních či abstraktních třídách, a nikoliv na třídách s konkrétní implementací. [3]

3.2 Jazyk C#

Programovací jazyk C# byl vytvořen společností Microsoft. Uveden byl roku 2002 jako součást vývojového prostředí Visual Studio .NET 2002. Jedná se o čistě objektový programovací jazyk. [4]

C# se řadí do kategorie vyšších programovacích jazyků, které se vyznačují tím, že jejich syntaxe je pro vývojáře snadněji čitelná než strojové instrukce pro procesor, do kterých je popis řešené úlohy překládán a který umí procesor zpracovat. Pro přeložení do strojového kódu je používán další program, tzn. kompilátor. S kompilátorem společně pracuje také sestavovací program, který spojí nezávislé části kódu a použité knihovny, což jsou předem naprogramované soubory kódu určené k opakovanému použití. [4]

Zdrojový kód v jazyce C# je nejprve přeložen do pomocného jazyka CIL (Common intermediate language). Pomocí dalšího překladače je poté přeložen do strojového kódu. Přeložení do strojového kódu probíhá až v momentě, kdy je program spuštěn. Překladač do strojového kódu se proto nazývá JIT (Just in time). [4]



Obrázek 1 - Kompletní proces kompilace C# kódu. Zdroj: [51]

3.2.1 Historie verzí

První verze C# 1.0 vyšla v roce 2002 a její součástí byly hlavní vlastnosti staticky typovaného objektově orientovaného jazyka. Verze 2.0 přinesla mimo jiné nullovatelné typy. S verzí 3.0 přišla technologie LINQ umožňující deklarativní dotazy a s tím spojené vlastnosti, jako anonymní datové typy a lambda výrazy. Verze 4.0 přinesla mimo jiné dynamické datové typy, nepovinné parametry metod a pojmenované argumenty. Hlavní novinkou ve verzi 5.0 bylo zjednodušení podpory asynchronních operací. S verzí 6.0 přišly malé vylepšení jazyka, jako např. interpolované textové řetězce. Verze 7.0, 7.1, 7.2 a 7.3 přinesly např. typ *Tuple* a další menší vylepšení jazyka. C# 8.0 přinesla již větší změny, a to např. nullovatelné referenční typy, zjednodušené switch konstrukce atd. Verze 9.0 přinesla možnost vytvářet konzolové aplikace s minimálním množstvím kódu, typ *record* atd. Verze 10.0 byla zaměřena na minimalizaci kódu u běžných scénářů, globální definice jmenných prostorů, jmenné prostory na úrovni souboru, kontrola nullovatelných parametrů a další změny. [5]

3.3 Platforma .NET a její vývoj

Historie prostředí .NET začíná v roce 2002, kdy společnost Microsoft uvedla první verzi .NET Framework a během let se pojmenování platformy několikrát měnilo. Nejdříve se platforma jmenovala .NET Framework, poté .NET Core a nakonec .NET. Mimo názvy se také měnila samotná implementace prostředí. [6]

3.3.1 .NET Framework

.NET Framework je technologie pro vývoj, běh aplikací a webových služeb v operačním systému Windows. Cílem je poskytnutí konzistentního objektově orientovaného prostředí pro kód uložený a vykonávaný buď lokálně, nebo na webu, dále poskytnutí prostředí, které usnadní publikování softwaru a eliminuje výkonnostní problémy skriptovaných nebo interpretovaných prostředí. [7] Ačkoli byl původně .NET Framework navržen tak, aby v něm byla možnost vyvíjet multiplatformní aplikace, společnost Microsoft jej vyvíjela tak, aby nejlépe pracoval na platformě Windows. [5]

Vývojová platforma .NET Framework obsahuje běhové prostředí Common Language Runtime (CLR), které řídí vykonání instrukcí v kódu, a knihovnu Base Class Library (BCL) obsahující třídy, díky kterým je možné vyvíjet vlastní aplikace. [5] První verze prostředí byla vydána v roce 2002 a poslední verze nese označení 4.8. [6]

3.3.2 .NET Core

V roce 2016 byl .NET Framework nahrazen novou multiplatformní implementací platformy nazvanou .NET Core. [6] Během implementace byly refaktorovány a odstraněny části platformy, které nebyly považovány za jádro frameworku. .NET Core tedy zahrnuje novou multiplatformní implementaci CLR pojmenovanou CoreCLR a BCL pojmenovanou CoreFX. Díky odlehčení jádra platformy je možné ji nasazovat společně s aplikací, takže se může často měnit, aniž by ovlivnila ostatní aplikace běžící v .NET Core na stejném počítači. Většina změn, které Microsoft provádí na .NET Core, by nemohla být tak jednoduše provedena na .NET Framework. [5]

3.3.3 .NET

Poslední verze .NET Core byla 3.1. Další verze byla vydána v listopadu roku 2020 a nesla již zkrácené označení .NET 5. [6] Nová hlavní (major) aktualizace by měla vycházet každý rok v listopadu a počínaje .NET 6.0 bude každá druhá verze bude LTS (Long Term Support), což znamená podporu těchto verzí buď 3 roky po jejich vydání, nebo 1 rok po vydání další LTS verze. [5]

3.3.4 Modely aplikací

Na platformě .NET je možné vyvíjet aplikace do různých běhových prostředí. Těmito základními prostředími jsou web, desktop a mobilní zařízení. .NET pro tyto prostředí poskytuje následující technologie:

- Web:
    - ASP
    - ASP.NET
    - ASP.NET WebForms
    - Silverlight
    - ASP.NET MVC
    - ASP.NET Core
- Desktop:
    - Windows Forms (WinForms)
    - Windows Presentation Foundation (WPF)
    - UWP
- Mobilní zařízení:

- Xamarin a Mono
- Multi-Platform App UI (MAUI) [6]

3.4 Desktopové aplikace v .NET

Na platformě .NET lze vyvíjet desktopové aplikace v několika technologiích. Některé technologie jsou určeny pouze pro aplikace na OS Windows, jako Windows Forms (WinForms), Windows Presentation Foundation (WPF) a Universal Windows Platform (UWP). Mezi technologie pro vývoj multiplatformních aplikací se řadí .NET Multi-platform App User Interfaces (MAUI) a další technologie třetích stran, jako např. Uno Platform či Avalonia. [5]

3.4.1 Windows Forms

V této kapitole byly využity zdroje: [8]

Framework Windows Forms od společnosti Microsoft je technologie pro vytváření desktopových aplikací. Tato technologie nabízí sadu nástrojů pro vytváření uživatelského rozhraní, která zahrnuje ovládací prvky, grafické elementy, napojení dat a uživatelské vstupy. Při použití s vývojářskou aplikací Visual Studio nabízí grafický nástroj pro návrh UI s funkcí drag-and-drop.

Existují dvě implementace této technologie, a to open-source varianta běžící na nejnovější platformě .NET a starší varianta běžící na .NET Framework 4.

Základní komponentu představuje formulář, který slouží k zobrazení informací a ovládacích prvků určených pro uživatele. V momentě, kdy je uživatelem provedena operace s některým ovládacím prvkem, tento prvek vytvoří událost, kterou je poté možné zachytit a provést nějakou operaci. Mezi ovládací prvky patří textová pole, tlačítka, rozevírací pole, přepínače atd. Zobrazení dat v tabulkové formě umožňuje prvek DataGridView, ke kterému je možné napojit různé datové zdroje. Také je možné si vytvořit vlastní ovládací prvek, tzn. UserControl. Grafické prvky je možné vytvářet pomocí tříd ve jmenném prostoru System.Drawing.

3.4.2 Universal Windows Platform (UWP)

V této kapitole byly využity zdroje: [9]

Stejně jako Windows Forms, UWP je technologie pro vytváření klientských aplikací pro OS Windows. Pro poskytnutí výkonného uživatelského rozhraní a pokročilých

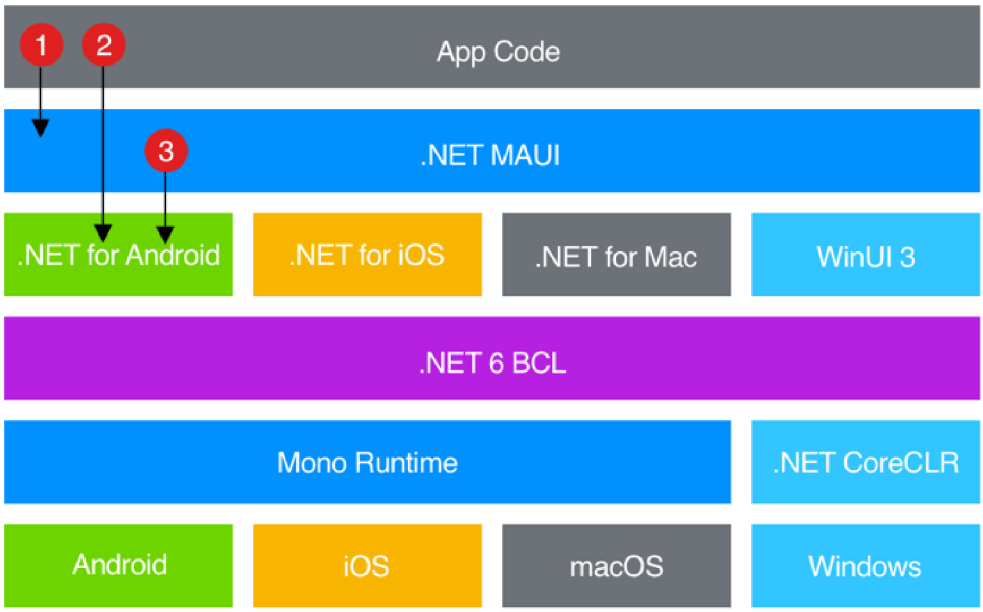
asynchronních funkcí využívá sadu aplikačních rozhraní WinRT. Technologie umožňuje použití společného aplikačního rozhraní pro všechna zařízení s OS Windows 10 nebo Windows 11. Prezentační část aplikace je možné vyvíjet s použitím WinUI, XAML, HTML nebo DirectX, a část vnitřní infrastruktury v jazycích C#, C++, Visual basic nebo JavaScript.

Vytvořená aplikace může běžet na desktopu, tabletu, zařízení Xbox, HoloLens, SurfaceHub a dalších. Distribuována může být prostřednictvím Microsoft Store jako MSIX balíček, díky čemuž může být během času aktualizována.

3.4.3 .NET Multi-platform App User Interfaces (MAUI)

V této kapitole byly využity zdroje: [10]

.NET MAUI je open-source technologie od společnosti Microsoft určená pro vývoj multiplatformních aplikací pro desktop a mobilní zařízení. Cílovými operačními systémy mohou být Windows, MacOS, iOS nebo Android. Tato technologie vznikla evolucí frameworku Xamarin.Forms. Hlavním cílem je možnost implementace co největší části aplikační logiky a uživatelského rozhraní do jedné báze kódu.



Obrázek 2 - Architektura .NET MAUI aplikace. Zdroj: [10]

Na Obrázek 2 lze vidět architekturu .NET MAUI aplikace. Aplikační rozhraní .NET MAUI poskytuje pro každou platformu vlastní framework pro tvorbu aplikací (pro desktopové aplikace je určen framework WinUI 3). Každý tento framework sdílí s ostatními knihovnu BCL. Pro mobilní zařízení a MacOS je určeno běhové prostředí Mono a pro OS Windows je určen .NET CoreCLR. Zatímco v aplikaci je standardně vyvíjena logika

spolupracující s aplikačním rozhraním .NET MAUI (bod 1), které dále komunikuje s nativním rozhraním zařízení (bod 3), aplikační logika může být také vykonáváno přímo na nativním rozhraní (bod 2).

3.4.4 Uno Platform

Uno Platform je multiplatformní aplikační framework třetí strany. Aplikace je možné vyvíjet s pomocí jazyků XAML a C# [11] a nasazovat na mobilní zařízení se systémem Android, iOS, dále jako web pomocí technologie WebAssembly a na desktopová zařízení se systémy macOS, Linux nebo Windows. [12]

Pro prezentační část aplikace je využívána knihovna Uno.UI, díky které lze nasazovat aplikace pro všechny dříve zmíněné platformy s výjimkou OS Windows, u které je použito rozhraní WinUI nebo UWP. Pomocí Uno.UI se pro ostatní platformy převádí XAML prvky do vhodných nativních prvků. U webové aplikace to jsou HTML elementy, u iOS a Mac Catalyst prvky typu UIView, u Android prvky typu ViewGroup a u MacOS prvky typu NSView. V prostředí OS Linux se XAML prvky vykreslují na plátno Skia. [13]

3.4.5 Avalonia

Avalonia je obdobně jako Uno Platform framework třetí strany pro vývoj multiplatformních aplikací. Aplikační logika může být nasazena na desktopová zařízení s OS Windows, Linux a MacOS. K tomu je vyvíjena podpora pro mobilní zařízení s Android a iOS a webovou technologii WebAssembly. [14] Pro vykreslení uživatelského rozhraní používá Avalonia vlastní vyskreslovací nástroj založený na Skia nebo Direct2D. Díky tomu je možné nasazovat zařízení také na vestavěná a energeticky nenáročná zařízení. [15]

3.4.6 WPF

V této části byly využity zdroje: [16]

Windows Presentation Foundation je framework určený pro vývoj desktopových aplikací na OS Windows. Díky vektorově založenému vykreslovacímu nástroji umožňuje vytvářet uživatelská rozhraní nezávislá na rozlišení obrazovky. Pro vývoj uživatelského rozhraní aplikace je určen jazyk XAML. Aplikační logika je vyvíjena v jiném programovacím jazyce (např. C#). Jednou z výhod oddělení vzhledu a aplikační logiky je to, že na aplikaci mohou nezávisle pracovat návrháři designu aplikace a vývojáři aplikační logiky.

Existují dvě implementace WPF, ta starší funguje na platformě .NET Framework 4, a druhá běžící na platformě .NET, která je open-source. Hlavní funkcí WPF je implementace obchodní logiky, která odpovídá na uživatelovy interakce.

```
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="SDKSample.AWindow"
Title="Window with Button"
Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button" Click="button_Click">Click Me!</Button>

</Window>
```

Kód 1 - Prezentační část WPF aplikace. Zdroj: [16]

Kód 1 představuje ukázkovou implementaci prezentační části WPF aplikace. Základním elementem je zde *Window*, které reprezentuje samotné okno aplikace. V tomto elementu jsou definovány některé atributy, mezi které patří *xmlns:x*, pomocí kterého je namapováno schéma pro podporu typů aplikační logiky. Odvozený atribut *x:Class* přiřadí konkrétní třídu aplikační logiky k tomuto oknu. Dále je zde element tlačítko, kdy po kliknutí na něj je vyvolána událost, která může být zachycena ve vrstvě aplikační logiky.

```
using System.Windows;

namespace SDKSample
{
    public partial class AWindow : Window     {
        public AWindow()
        {
            // InitializeComponent call is required to merge the UI
            // that is defined in markup with this class, including
            // setting properties and registering event handlers
            InitializeComponent();
        }

        void button_Click(object sender, RoutedEventArgs e)
        {
            // Show message box when button is clicked.
            MessageBox.Show("Hello, Windows Presentation Foundation!");
        }
    }
}
```

Kód 2 – Aplikační logika WPF aplikace. Zdroj: [16]

Kód 2 je ukázkou aplikační logiky, konkrétně třídou spojenou s prezentační částí (Kód 1). Jelikož je tato třída spojena s elementem *Window*, musí dědit z třídy *Window*. V konstruktoru je volána vygenerovaná metoda pro sloučení uživatelského rozhraní definovaného v prezentační vrstvě aplikace. Metoda *button_Click* zachycuje událost kliknutí na definované tlačítko.

3.4.6.1 XAML

V této kapitole byly využity zdroje: [17]

XAML je deklarativní značkovací jazyk založený na jazyku XML pro tvorbu uživatelského rozhraní WPF aplikace. Jak již bylo dříve řečeno, díky tomuto jazyku je možné vytvářet uživatelské rozhraní odděleně od aplikační logiky. Objekty vytvořené pomocí XAML přímo reprezentují typy definované v sestaveních aplikace (assemblies). Tím se odlišuje od ostatních značkovacích jazyků, které jsou typicky interpretované bez vazby na definovanou sadu typů.

Základním elementem je objekt, pomocí kterého je deklarována instance konkrétního typu (třídy). Počáteční značka je zapisována ve formátu <*jméno*> a ukončovací značka ve formátu </*jméno*>. V případě, že není potřeba v elementu deklarovat další elementy objektu, je možné deklarovat element pouze jednou značkou ve formátu <*jméno*/>. Pomocí atributu elementu je možné nastavovat různé vlastnosti typu či definovat metody zachycující různé události.

3.4.6.2 Životní cyklus aplikace

V této kapitole byly využity zdroje: [18]

„Všechny aplikace mají tendenci sdílet společný soubor funkcí, který se týká implementace a správy aplikací.“ [18] V případě WPF je aplikační funkcionalita zapouzdřena ve třídě *Application*. Jedná se o následující funkcionality:

- Řízení životního cyklu aplikace
- Zpracování parametrů příkazového řádku
- Detekování neošetřených výjimek
- Sdílení vlastností a zdrojů aplikace
- Správa oken v samostatných aplikacích
- Správa navigace

Třída *Application* je schopna vyvolat události týkající se životního cyklu aplikace, jako např. spuštění, vypnutí, aktivace a deaktivace. Přiřazení metod, které zpracovávají jednotlivé události, je definováno v jazyce XAML (viz. Kód 3) a metody jsou implementovány v jazyce C# (viz. Kód 4).

```
<Application   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"   x:Class="SDKSample.App"
Startup="App_Startup" />
```

Kód 3 - Přiřazení metody zpracovávající událost spuštění aplikace. Zdroj: [18]

```
using System.Windows;

namespace SDKSample
{
    public partial class App : Application     {
        void App_Startup(object sender, StartupEventArgs e)
        {
            // Open a window
            MainWindow window = new MainWindow();
            window.Show();
        }
    }
}
```

Kód 4 - Implementace metody zpracovávající událost spuštění aplikace. Zdroj: [18]

3.5 Generický hostitel aplikace

Generický hostitel aplikace umožňuje, díky oddělení spuštění a inicializace, jednoduše konfigurovat a spouštět aplikaci s jejími službami. Jedná se o výchozího hostitele webových aplikací ASP.NET Core. Hostitel zodpovídá za životní cyklus aplikace [19] a je využíván k zapouzdření zdrojů, mezi které se řadí:

- Vkládání závislostí (Dependency injection)
- Logování
- Konfigurace
- Různé implementace rozhraní *IHostedService* [20]

3.5.1 Použití s WPF

V této části byly využity zdroje: [21]

Idea vytvoření generického hostitele, který by mohl být použit pro různé typy aplikací, vznikla z vývoje webového hostitele v ASP.NET Core. S použitím ve WPF aplikaci je možné využít všech dříve popsaných zdrojů hostitele, jako konfigurace aplikace, logování a DI kontejner.

```
public partial class App : Application
{
    private IHost _host;

    public App()
    {
        _host = new HostBuilder().Build();
    }

    private async void Application_Startup(object sender, StartupEventArgs e)
    {
        await _host.StartAsync();
    }

    private async void Application_Exit(object sender, ExitEventArgs e)
    {
        using (_host)
        {
```

```
            await _host.StopAsync(TimeSpan.FromSeconds(5));
        }
    }
}
```

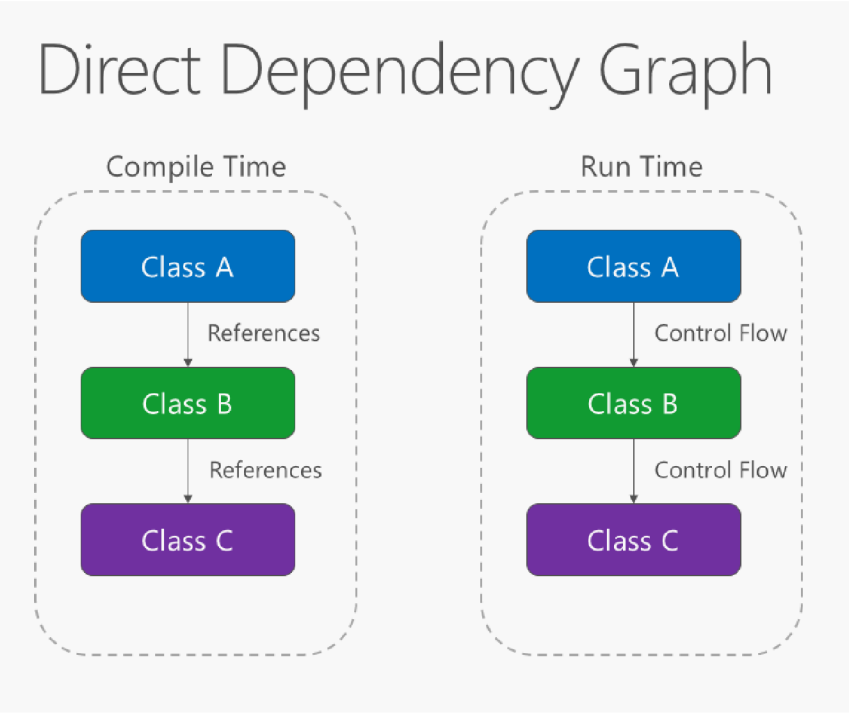
Kód 5 - Použití generického hostitele ve WPF aplikaci. Zdroj: [21]

Kód 5 představuje příklad použití generického hostitele ve WPF aplikaci. Generický hostitel je inicializován a sestaven v konstruktoru třídy, dále jsou zde metody zachycující události spuštění a vypnutí aplikace, které spouštějí a ukončují běh hostitele.

3.5.2 **Dependency injection**

V této části byly využity zdroje: [22] [23] [24]

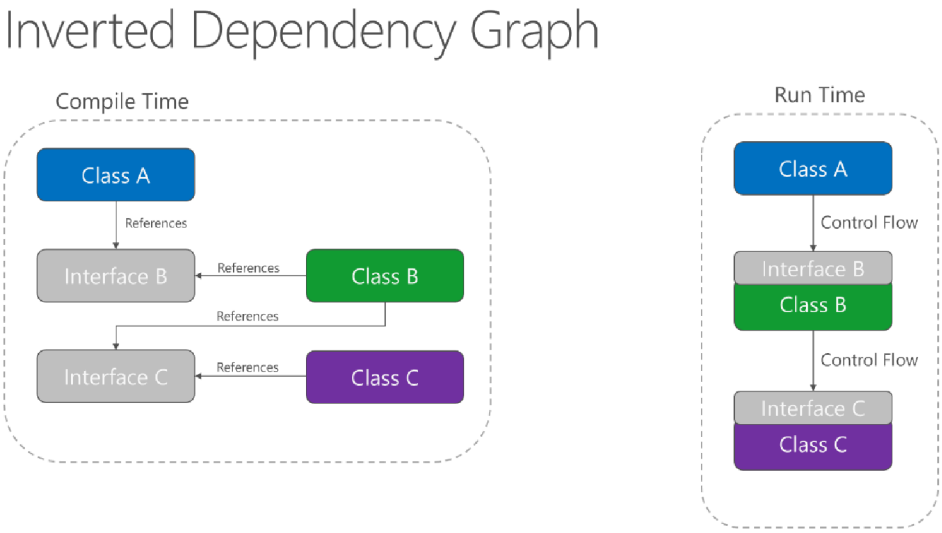
Inverze směru závislostí mezi třídami byla již vysvětlena v kapitole 3.1.2. Tento princip říká, že směr závislosti by neměl být ve směru implementačních detailů, ale ve směru abstrakce. Bez uplatnění tohoto principu vzniká mezi třídami přímý graf závislostí (viz. Obrázek 3).



Obrázek 3 - Graf přímé závislosti. Zdroj: [22]

Na předchozím příkladu je třída A v čase kompilace i v čase běhu aplikace přímo závislá na metodách třídy B a třída B je přímo závislá na metodách třídy C.

Použitím principu inverze závislostí je invertována závislost mezi třídami v době kompilace, ale závislost v čase běhu zůstává nezměněna. Toto je docíleno tím, že třída není závislá na implementaci druhé třídy, ale na její abstrakci, tzn. třída je závislá pouze na rozhraní, které druhá třída implementuje (viz. Obrázek 4).

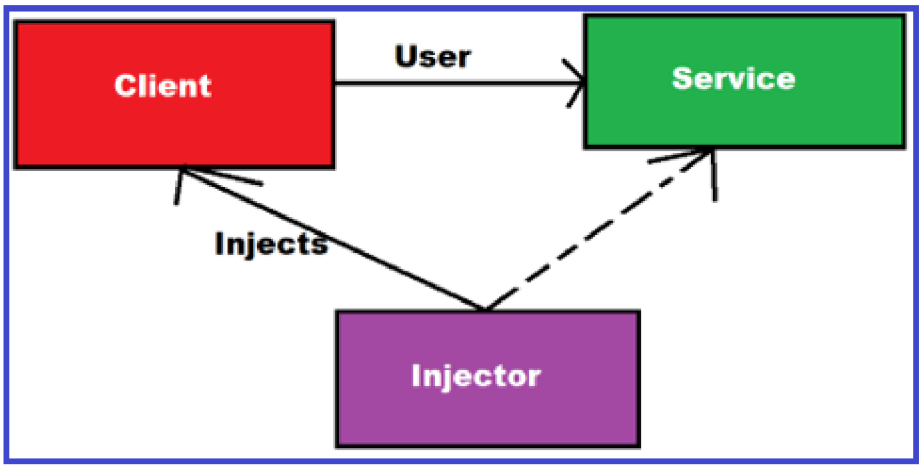


Obrázek 4 - Graf invertované závislosti. Zdroj: [22]

Mezi výhody tohoto principu se řadí lepší testovatelnost, modularita (jelikož zde vzniká závislost na abstrakci, tak může být implementace snadno změněna) a udržitelnost implementovaného řešení.

Dependency injection je návrhový vzor pro invertování závislostí mezi třídami. Zahrnuje 3 typy tříd, a to:

- Klientská třída – třída závislá na třídě služby
- Třída služby – třída poskytující službu klientské třídě
- Třída Injector – třída vkládající instanci třídy služby do klientské třídy (viz. Obrázek 5)



Obrázek 5 - Vztahy mezi třídami návrhového vzoru Dependency injection. Zdroj: [23]

Závislost lze vkládat prostřednictvím konstruktoru, vlastnosti nebo metody třídy. Při registrování služby do DI kontejneru je nastavována její životnost. Prvním typem životnosti je **Transient**. Transientní služba je vytvořena pokaždé, kdy je požadována z DI kontejneru a její prostředky jsou uvolněny na konci procesu, který tuto službu využívá. Služba s touto životností je registrována metodou *AddTransient*. Druhým typem životnosti je **Scoped**. Tato životnost služby se týká webových aplikací, kdy je vytvořen objekt třídy při každém požadavku od klienta (připojení). Po vyřízení požadavku jsou prostředky služby uvolněny. Posledním typem životnosti je **Singleton**. Objekty služby s touto životností jsou vytvořeny pouze jednou, a to buď v čase, kdy je služba poprvé požadována z DI kontejneru, nebo v případě, kdy vývojář zaregistruje do DI kontejneru přímo konkrétní instanci třídy. Každý požadavek poté využívá tu samou instanci. Prostředky této služby jsou pak uvolněny až s uvolněním prostředků poskytovatele služeb při ukončení běhu aplikace. Singleton služby jsou registrovány metodou *AddSingleton*.

3.5.3 Logování

V této kapitole byly využity zdroje: [25] [26]

Platforma .NET obsahuje aplikační rozhraní pro zaznamenávání zpráv při běhu aplikace. Díky tomuto rozhraní lze upravovat obsah a formát logovaných zpráv, což usnadňuje ladění chyb aplikace. Dále je také možné určit umístění zpráv. Aplikační rozhraní pro logování je tvořeno těmito komponentami:

- *ILogger* – Poskytuje metodu *Log()*, která slouží k zápisu zprávy logu.

- *ILoggerProvider* – Implementované různými logovacími poskytovateli, kteří určují, kam budou logované zprávy zapisovány (např. konzole). Může také sloužit k implementaci vlastního poskytovatele.
- *ILoggerFactory* – Zaregistruje jeden nebo více poskytovatelů logování a poskytuje metodu *CreateLogger()* pro vytvoření instance *ILogger*.

Toto aplikační rozhraní lze využít buď se zabudovanými poskytovateli, nebo s poskytovateli třetích stran. Mezi poskytovatele zabudované v .NET se řadí:

- Console
- Debug
- EventSource
- EventLog

Poskytovatelé logování třetích stran nabízí další možnosti umístění zpráv a jejich formátování. Mezi tyto poskytovatele se řadí:

- io
- JSNLog
- Log4Net
- NLog
- Serilog

Kód 6 ilustruje zaregistrování konzolového poskytovatele logování a Kód 7 použití objektu ILogger v konkrétní třídě pro zapsání zprávy.

```
var builder = WebApplication.CreateBuilder(args);
builder.Host.ConfigureLogging(logging =>     {
        logging.ClearProviders();
        logging.AddConsole();
    }
);
```

Kód 6 - Zaregistrování poskytovatele logování. Zdroj: [26]

```
public class IndexModel : PageModel
{
    private readonly ILogger<IndexModel> _logger;

    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        _logger.LogInformation( "This is a custom message from the Loggly Guide");
    }
}
```

Kód 7 – Využití ILogger pro zapsání informace. Zdroj: [26]

Při logování zprávy lze nastavit její závažnost. Tabulka obsahuje různé úrovně logů společně s metodami pro jejich zápis a doporučeným použitím:

Tabulka 1 - Úrovně logů

Úroveň	Hodnota	Metoda	Popis
Trace	0	LogTrace	Pro detailní a citlivé informace o běhu aplikace. Tyto zprávy by neměly být povoleny v aplikaci, která je již nasazena do produkce.
Debug	1	LogDebug	Pro ladění chyb a vývoj aplikace. Její povolení v produkční apliakci by se mělo zvážit vzheledem k vysokému objemu zpráv
Information	2	LogInformation	Pro zaznamenání hlavních událostí běhu aplikace.
Warning	3	LogWarning	Pro zaznamenání neočekávaných událostí, které ale nezpůsobí pád aplikace.
Error	4	LogError	Pro chyby a výjimky, které nejsou, nebo nemohou být zpracovány. Zprávy často indikují chybu v konkrétních operacích či požadavcích.
Critical	5	LogCritical	Pro chyby, které si vyžadují bezprostřední pozornost a musí být přednostně vyřešeny. Příkladem může být ztráta dat či nedostatek místa na disku.
None	6		Určuje, že by se neměly psát žádné zprávy.

Zdroj: [25]

Konfigurace logování je uložena ve formátu JSON v souboru *appsettings.json*. V názvu souboru může být specifikováno prostředí, ve kterém má být konfigurační soubor použitý (např. pro vývoj - *appsettings.Development.json*). V tomto souboru lze specifikovat, jaká nejnižší úroveň zpráv má být zpracována poskytovatelem u jakých kategorií (viz Kód 8).

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"     }
  }
}
```

Kód 8 - Konfigurace nejnižších povolených úrovní pro různé kategorie zpráv. Zdroj: [25]

3.5.4 Konfigurace

Generický hostitel dále poskytuje rozhraní *IConfiguration*, pomocí kterého je možné získat konfigurační data z různých zdrojů. Přístup ke konfiguračním datům zajišťují poskytovatelé a zdroji konfigurace mohou být:

- Soubory nastavení, jako např. *appsettings.json*
- Proměnné prostředí
- Azure Key Vault
- Azure App Configuration
- Argumenty příkazového řádku
- Vlastní zprostředkovatelé, nainstalovaní nebo vytvoření
- Soubory adresářů
- Objekty .NET v paměti
- Poskytovatelé třetích stran [27]

Klíčovou vlastností konfigurace .NET je možnost svázání hodnot ve zdroji konfigurace do objektu třídy. K tomu se používá vzor Options, který poskytuje silně typovaný přístup k nastavení [27] a následnou validaci konfiguračních dat. Pro svázání konfiguračních dat do Options vzoru je nutné vytvoření třídy s vlastnostmi odpovídajícími ve zdroji dat (Kód 9, Kód 10) a zavolání metody *Configure* poskytovatele služeb (Kód 11). Poté je možné již vložit třídu Options s konfiguračními hodnotami do konkrétní služby pomocí Dependency injection (Kód 12). [28]

```
"Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
}
```

Kód 9 - Konfigurační data ve formátu JSON. Zdroj: [28]

```
public class PositionOptions
{
    public const string Position = "Position";

    public string Title { get; set; } = String.Empty;
    public string Name { get; set; } = String.Empty;
}
```

Kód 10 - Třída podle vzoru Options. Zdroj: [28]

```
builder.Services.Configure<PositionOptions>(
 builder.Configuration.GetSection(PositionOptions.Position));
```

Kód 11 - Svázání konfiguračních hodnot k třídě Options. Zdroj: [28]

```
private readonly PositionOptions _options;

public Test2Model(IOptions<PositionOptions> options)
{
```

```
    _options = options.Value;
}
```

Kód 12 - Vložení konfigurace do konkrétní služby. Zdroj: [28]

K přístupu ke konfiguračním datům svázaným do vzoru Options slouží tato rozhraní:

- *IOptions<TOptions>* - konfigurace ve stavu, jaká byla před spuštěním aplikace. Životnost tohoto rozhraní je *Singleton*.
- *IOptionsSnapshot<TOptions>* - použití v případě, že je očekávána změna konfiguraci při každém požadavku. Životnost tohoto rozhraní je *Scoped*.
- *IOptionsMonitor<TOptions>* - podporuje upozornění na změnu konfigurace a její opakované čtení. Životnost tohoto rozhraní je *Singleton*. [28]

3.6 Objektově relační mapování

Objektově relační mapování (ORM) je technika pro vytvoření spojení mezi objektově orientovanou aplikací a relační databází. Spočívá v namapování relační datové struktury do objektové (viz. Tabulka 2), ve které je poté možné provádět operace, jako vytváření, čtení, úprava a odstranění záznamů již v objektovém prostředí. [29]

Tabulka 2 - Mapování relační datové struktury do objektové v Entity Framework Core.

Relační databáze	**.NET platforma**
Tabulka	.NET třída
Sloupce tabulky	Vlastnosti/pole třídy
Řádky tabulky	Elementy v kolekcích .NET - např. *List*
Primární klíč – jedinečný řádek	Objekt třídy
Cizí klíč – definice vztahu	Odkaz na jinou třídu
SQL - např. *WHERE*	.NET LINQ - např. *Where(p => ...*

Zdroj: [30]

Pro vytvoření vrstvy ORM jsou používány různé nástroje a mezi tyto nástroje pro platformu .NET patří:

- Entity Framework
- NHibernate
- Dapper
- Base One Foundation Component [29]

Použití ORM nástrojů má své výhody i nevýhody. Mezi výhody patří snížení nákladů na vývoj, jelikož není nutné ovládat dotazovací jazyk SQL a dále zvýšení ochrany proti

útokům SQL injection. Nevýhodami mohou být nižší výkonnost dotazů a možná časová náročnost osvojení si ORM nástrojů. [29]

3.6.1 Entity Framework Core

V této části byly využity zdroje: [30] [31] [32] [33] [24]

Entity Framework Core (EF Core) je open-source nástroj pro objektově relační mapování od společnosti Microsoft. První verze byla vydána v roce 2016. Tento nástroj podporuje mapování do mnoha databázových systémů, jako například:

- Microsoft SQL Server
- SQLite
- PostgreSQL
- MySQL

Pro implementaci ORM v rámci EF Core lze využít přístupy jako Code First, Database First a Model First. Přístup **Code First** spočívá v prvotním vytvoření objektového modelu a následné migrace do relačního databázového systému pomocí prostředků EF Core. U tohoto přístupu vývojář vůbec nepřistupuje do relačního databázového systému. Pro definici cizích klíčů, indexů atd. slouží nástroje Fluent API nebo Data Annotations. Případné změny v modelu lze přenést do relačního databázového systému pomocí dalších migrací. Přístup **Database First** spočívá ve vygenerování tříd podle již existujícího relačního modelu v databázovém systému. Je používán v případě, kdy je relační databáze již implementována nebo i nasazena v provozu. Posledním přístupem je **Model First**. V rámci tohoto přístupu je model vytvořen v návrháři EF Core. Model je pak reprezentován souborem typu EDMX. Pomocí tohoto souboru je možné vygenerovat relační databázové schéma a také třídy určené k interakci s relační databází. Pro přístup k datům je určen model tvořený třídami reprezentujícími entity a objektem typu kontext reprezentujícím relaci s databází (viz. Kód 13). Tento kontext umožňuje pomocí dotazovacího jazyka LINQ provádět operace, jako čtení, ukládání, modifikace a odstranění dat (viz. Kód 14). Databázový kontext lze zaregistrovat do DI kontejneru pomocí metody *AddDbContext*. Zaregistrovaný kontext má poté životnost *Scoped*.

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
```

```
            @"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True");
    }
}
```

Kód 13 - Příklad implementace kontextové třídy. Zdroj: [33]

```
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
          .Where(b => b.Rating > 3)
          .OrderBy(b => b.Url)
          .ToList();
}
```

Kód 14 - Čtení z databáze s použitím jazyka LINQ. Zdroj: [33]

3.6.1.1 Metody k dotazování souvisejících entit

V této kapitole byly využity zdroje: [30]

Jelikož EF Core minimalizuje přístup do databáze, tak není možné jednoduchým dotazem na jednu entitu získávat také související entity. Odkazy na související entity v získané třídě by byly rovny *null*. Pro docílení přístupu k více entitám propojeným vztahy v rámci jednoho dotazu je nutné použít přístupy, mezi které patří:

- Eager loading
- Explicit loading
- Select loading
- Lazy loading

Eager loading funguje na principu načtení souvisejících entit v rámci jednoho dotazu na primární entitu. Toto je realizováno pomocí metod *Include* pro načtení entit v primární úrovni vztahu a *ThenInclude* pro načtení entit v sekundární úrovni vztahu. V rámci metod *Include* a *ThenInclude* je možné také řadit či filtrovat související entity. Výhodou tohoto přístupu je efektivní načtení dat s minimálním množstvím přístupů do databáze (tzn. *database round-trips*). Jelikož tato metoda načte celé požadované entity, neměla by se používat v případě potřeby přístupu pouze k části souvisejících entit z důvodu snížení výkonnosti dotazu s narůstajícím objemem dat.

Dalším přístupem je **Explicit Loading**. Podstatou je oddělené načtení primární entity a souvisejících entit. Související entity jsou explicitně načítány zavoláním metody *Load* na *Entry* primární entity. Výhodou tohoto přístupu je skutečnost, že související entity se načítají později. Mohou být požadovány jen za nějakých pozdějších okolností, či jen určitou částí logiky a není nutné je načítat najednou. Nevýhodou je větší množství přístupů do databáze, což může být neefektivní. Tento přístup by neměl být používán v případě, kdy je již od začátku jasné, jaké všechny entity budou požadovány.

Select Loading spočívá ve výběru specifických vlastností entity. K tomu je využita metoda *Select* jazyka LINQ. Výhodou tohoto přístupu je efektivnost dotazu, jelikož nejsou načítány všechny vlastnosti entity. Je také minimalizován počet a složitost přístupů k databázi. Nevýhodou je nutnost definice všech požadovaných vlastností či výpočtů v metodě *Select*.

Posledním přístupem je **Lazy Loading**. Tento přístup spočívá v jednoduchosti dotazování na související entity na úkor výkonnosti dotazů. Pro jeho využití u dotazů je nutné provést změny buď v databázovém kontextu, nebo v třídách entit:

- Využití knihovny *Microsoft.EntityFrameworkCore.Proxies* při konfiguraci databázového kontextu
- Vložení Lazy Loading metod do tříd entit přes jejich konstruktor

Po provedení jedné z těchto změn jsou při dotazování související entity automaticky načteny při přístupu k nim přes primární entitu. Tento přístup se vyznačuje neefektivností dotazů, jelikož dotazy mohou způsobovat velké množství přístupů do databáze.

3.6.1.2 Fluent API a Data Annotations

V rámci vytvoření databázového modelu metodou Code First je možné definovat způsob namapování entit do relační podoby prostřednictvím tzn. modelu metadat. Tento model může být upraven pomocí mapovacích atributů (Data Annotations) anebo metod třídy ModelBuilder (Fluent API) v metodě OnModelCreating databázového kontextu. Při použití obou těchto metod má vyšší prioritu Fluent API a konfigurace touto metodou přepíše konfiguraci Data Annotations. [32]

V rámci Fluent API je možné konfigurovat každou vlastnost entity pomocí metody *Property*, která vrací objekt konfigurace dané vlastnosti. Možnosti konfigurace se liší podle datového typu vlastnosti. Entitám je možné nastavovat primární a cizí klíče, maximální délku obsahu atributů, povinné atributy, indexy atd. [34]

```
modelBuilder.Entity<Department>().Property(t => t.Name).IsRequired();
```

Kód 15 - Nastavení povinného atributu pomocí Fluent API. Zdroj: [34]

Konfigurace metodou Data Annotations je zapisována přímo ve třídě definující entitu k řádek před jednotlivými vlastnostmi jako atributy uzavřené hranatými závorkami. Je možné definovat obdobné atributy vlastností jako pomocí metody Fluent API. Mezi tyto atributy patří například:

- *[Key]* – primární a složené klíče

- *[Required]* – povinné vlastnosti
- *[ForeignKey("columnName")]* – definice vztahu [35]

```
[Required]
public string Title { get; set; }
```

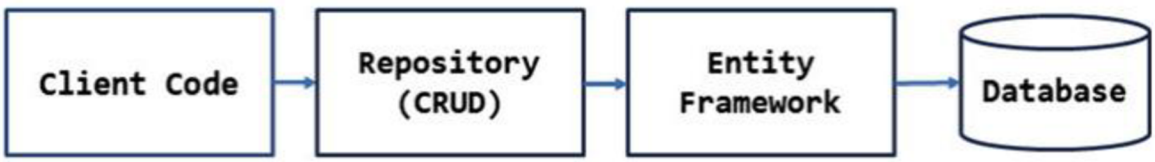
Kód 16 - Nastavení povinné vlastnosti pomocí Data Annotations. Zdroj: [35]

3.6.1.3 Návrhové vzory Repository a Unit Of Work

V této části byly využity zdroje: [2]

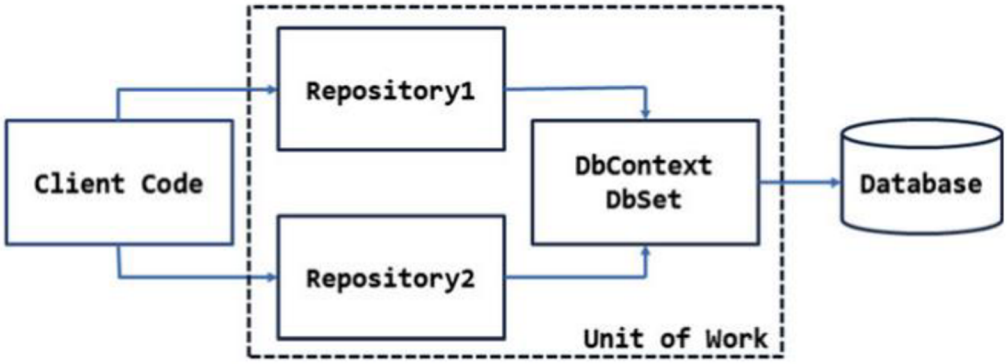
Aplikace běžně používá datovou vrstvu a provádí operace jako čtení, zápis, úprava a odstranění záznamů (CRUD). Tato vrstva bývá využívána z více míst a může se stát, že jsou tyto operace implementovány na více místech, což není žádoucí. Pro jejich zapouzdření je využíván návrhový vzor **Repository**. Tento vzor poté poskytuje jednotný přístup k datové vrstvě jakékoli komponentě aplikace. Výhodami tohoto vzoru jsou:

- Izolace CRUD operací od prezentační a logické vrstvy
- Jednodušší testování aplikace, kdy může být třída Repository pro testování nahrazena simulovanou (mockovanou) třídou Repository



Obrázek 6 - Přístup k datové vrstvě s použitím Repository. Zdroj: [2]

Některé logické operace se mohou sestávat z více operací na datové vrstvě. Tento sled operací se nazývá transakce. V případě, že by jedna z operací selhala, je nutné, aby se ostatní operace také neprovedly, i když by mohly být provedeny úspěšně. K tomuto procesu je určen návrhový vzor **Unit of Work**. Ideou tohoto vzoru je zaznamenání všech potřebných operací a jejich vykonání v rámci jedné transakce. Vykonáním transakce je poté úspěšně převeden stav datové vrstvy do očekávaného a konzistentního stavu. V případě, že jedna nebo více operací v rámci transakce selže, zůstane datová vrstva v původním (konzistentním) stavu.



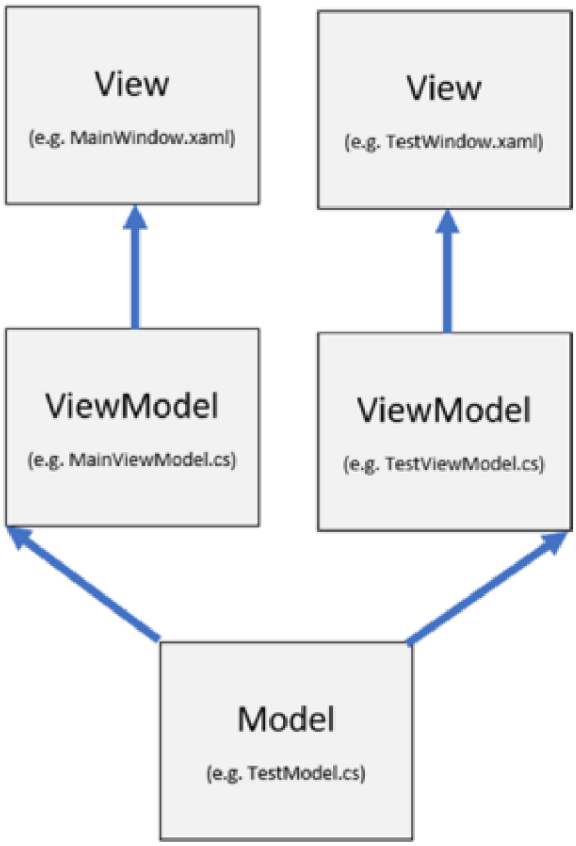
Obrázek 7 - Implementace Unit of Work společně s Repository. Zdroj: [2]

Obrázek 7 ilustruje implementaci návrhového vzoru Unit of Work. Sled operací využívajících více tříd Repository musí sdílet stejný databázový kontext, aby tyto operace mohly být vykonány v rámci jedné databázové transakce.

3.7 Model-View-ViewModel

Se zvyšující komplexností aplikace je čím dál více žádoucí udržovat kód v rámci pevně dané architektury pro jeho pochopitelnost, udržitelnost a rozšiřitelnost. Jedním s návrhových vzorů architektury aplikace je Model-View-ViewModel (MVVM). [36]

Principem této architektury je oddělení prezentační vrstvy a aplikační logiky. Část View reprezentuje okno aplikace určené pouze pro prezentaci dat. Zpracování událostí a svázání prvků prezentační vrstvy s daty je realizováno částí ViewModel. Část Model pak reprezentuje business logiku, se kterou pracuje ViewModel. Jedna třída ViewModel nemusí být svázána jen s jednou třídou View a jedna třída Model může být vložena do více tříd ViewModel (viz. Obrázek 8). [37]



Obrázek 8 - Architektura MVVM. Zdroj: [37]

3.7.1 **Použití s WPF**

V této části byly využity zdroje: [38]

Vzor Model-View-ViewModel lze využít při implementaci WPF aplikace. Pro svázání dat mezi View a ViewModel je využívána komponenta *Binding*, pomocí které je nastavena hodnota ovládacího prvku View na hodnotu vlastnosti ve ViewModel (viz. Kód 17). S použitím komponenty *Binding* musí být také nastavena ve View vlastnost *DataContext* na příslušný ViewModel. Aby změna svázané hodnoty na jedné straně vztahu byla reflektována na druhou stranu, je nutné, aby třída ViewModel implementovala rozhraní *INotifyPropertyChanged*. Toto rozhraní definuje událost typu *PropertyChangedEventHandler*, která upozorní jednu stranu o změně hodnoty svázané vlastnosti provedené na straně druhé (viz. Kód 18).

```
<TextBlock Text="{Binding Path=FirstName}" VerticalAlignment="Center"
HorizontalAlignment="Center"/>
```

Kód 17 - Svázání hodnoty textu na straně View pomocí Binding. Zdroj: [38]

```
public class ViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public string FirstName { get; set; }
    public void OnPropertyChanged(string propertyName) => PropertyChanged?.Invoke(this, new
          PropertyChangedEventArgs(propertyName));
```

Kód 18 - Implementace části ViewModel s událostí PropertyChangedEventHandler. Zdroj: [38]

V rámci architektury MVVM je možné také zpracovávat interakce s uživatelským rozhraním na straně ViewModel. UI prvky implementující rozhraní *ICommandSource* podporují vlastnost *Command*, která je vyvolána při interakci s daným prvkem (např. kliknutí na tlačítko). Při svázání objektu *ICommand* na vlastnost *Command* UI prvku je odeslána zpráva na stranu ViewModel, který poté tuto událost může zpracovat.

3.8 UML

Modelovací jazyk Unified Markup Language (UML) je soubor grafických notací a postupů pro objektově orientovanou analýzu a návrh softwarových řešení. Umožňuje modelovat jednoduché i složité aplikace s využitím stejné formální syntaxe. Právě skutečnost, že jsou při modelování dodržována přesně definovaná pravidla, je jednou z hlavních výhod jazyka UML, jelikož je možné modelovanou problematiku sdílet i s ostatními aktéry tohoto procesu (ostatní návrháři, zadavatelé) a tito aktéři jsou schopni dané problematice snadno porozumět. První verze jazyka byla vydána roku 1997 a stále probíhá její vývoj. [39]

3.8.1 Typy diagramů

V této části byly využity zdroje: [39], [40]

Pomocí jazyka UML lze dekomponovat navrhovaný systém do různých částí z různých úhlů pohledu. Tyto části jsou reprezentovány diagramy a dělí se do skupin:

- **Strukturální diagramy** – diagram tříd, diagram komponent, diagram nasazení, diagram složených struktur, objektový diagram a diagram balíčků
- **Diagramy chování** – diagram časování, diagram přehledu interakcí, diagram komunikace, stavový diagram, diagram případů užití, sekvenční diagram a diagram aktivit

Diagram tříd vyjadřuje statickou strukturu systému. Je tvořen třídami, které obsahují různé atributy, vyznačují se různým chováním a mají mezi sebou různé vztahy. U atributů je definován jejich název, formát (datový typ) a viditelnost určující úroveň přístupu k nim. Chování třídy je definováno signaturami operací (název, seznam parametrů a návratová hodnota). Mezi typy vztahů se řadí agregace (třída je součástí jiné třídy), kompozice

(speciální typ agregace, kdy podřízený objekt nemůže existovat bez nadřízeného objektu), asociace (abstrakce množiny spojení mezi instancemi tříd, u které je definována násobnost vztahu) a generalizace (dědičnost tříd).

Diagram komponent lze chápat jako specializovanější druh diagramu tříd. V rámci tohoto diagramu je komplexní systém rozložen do samostatných komponent a jsou definovány vztahy mezi nimi.

Diagram nasazení vyjadřuje způsob nasazení aplikace do produkčního prostředí s ohledem na softwarové a hardwarové komponenty v systému.

Diagramy složených struktur „jsou v podstatě plány vnitřní struktury klasifikátoru. Mohou být také použity k zobrazení chování spolupráce nebo interakce klasifikátoru s jejich prostředím prostřednictvím portů. Mohou snadno zobrazit vnitřní komponenty libovolného hardwaru, aby lépe porozuměly vnitřnímu fungování.“ [40]

Objektový diagram je používán k zobrazení příkladů stavu datových struktur v určitém časovém bodě. Může být využitý jako doplněk k diagramu tříd v podobě testovacích scénářů pro ověření jeho správnosti.

Pro zobrazení závislostí mezi balíčky slouží diagram balíčků. Balíček může představovat elementy modelu, jako případy užití nebo třídy.

Mezi diagramy chování se řadí diagram časování. Tento diagram je využíván pro popis interakcí mezi objekty v daném časovém úseku a pro nalezení procesů, které je vhodné optimalizovat.

Diagram přehledu interakcí poskytuje přehled o toku řízení mezi uzly, které spolu interagují. Je tvořen počátečními uzly, koncovými uzly, rozhodovacími uzly atd.

Diagram komunikace je využíván k zobrazení vztahu objektů z hlediska přenosu informací mezi nimi ve formě zpráv.

Stavový diagram je technika pro znázornění stavů, které může konkrétní objekt v systému nabývat. Zároveň popisuje změny stavů v závislosti na událostech, které se tohoto objektu týkají.

Diagram případů užití popisuje funkčnost informačního systému z pohledu uživatelů. Je tvořen případy užití, z nichž každý reprezentuje jedinečný způsob využití systému. Každý případ užití je tvořen stručným popisem, základním scénářem, alternativními scénáři a vstupními/výstupními podmínkami. Scénáře představují sekvenci kroků v interakci mezi aktéry (např. uživatel a systém). Případy užití mohou mít mezi sebou vztah *include* (případ

užití obsahuje stejnou část scénáře jiného případu užití), *extend* (případ užití rozšiřuje jiný případ užití) nebo zobecnění (generalizace).

Sekvenční diagram je využíván pro popis interakcí mezi objekty během času. Zobrazují životní cyklus objektů a předávání zpráv mezi objekty v rámci toku sekvence.

Diagram aktivit slouží pro vizualizaci kroků v rámci případu užití. Tyto kroky (aktivity) mohou být sekvenční či souběžné a mohou se větvit na základě různých podmínek.

3.9 Požadavky na aplikaci

V této části byly využity zdroje: [39]

V rámci zahájení tvorby softwarového řešení je nutné formulovat sadu požadavků na funkcionalitu aplikace. Tyto požadavky by měly definovat, co má být funkcionalitou aplikace, a nikoliv jak to má být aplikací řešeno. Jsou rozlišovány 2 typy požadavků, a to:

- Funkční požadavky – specifikace funkcionality systému
- Nefunkční požadavky – specifikace vlastností a podmínky správného fungování systému

Zdroji požadavků mohou být legislativní úpravy, budoucí uživatelé, již existující systémy používané uživateli, jejich pracovní procesy atd. Prostřednictvím jazyka UML mohou být specifikovány funkční požadavky v rámci případů užití, ale nikoliv ty nefunkční. V rámci nefunkčních požadavků může být specifikováno:

- Dodržení různých standardů, protokolů
- Rychlost vykonávání operací
- Architektura aplikace
- Zabezpečení systému

3.10 Vývoj řízený chováním

„Vývoj řízený chováním (Behavior Driven Development – BDD) je vývojová metodika, která klade důraz na uspokojování obchodních potřeb softwaru. Vyvinula se z vývoje řízeného testy neboli TDD. BDD používá pro vývoj testů doménově specifický jazyk a pevnou syntaxi.“ [41] Jazykem používaným pro BDD je Gherkin. Každá testovaná funkcionalita je označována klíčovým slovem **Feature**. V rámci funkcionality jsou pak definovány scénáře (označované slovem **Scenario**). Scénář začíná vstupní podmínkou spuštění (klíčové slovo **Given**) a pokračuje vnitřní podmínkou označující (**When**), po nichž

následuje reakce na podmínku (**Then**). Jednotlivé součásti scénáře mohou být složeny z více částí pomocí slova **And**. [41]

4 Vlastní práce

V této části práce je zdokumentován proces vývoje softwarového řešení pro společnost vypůjčující pracovní plošiny. V rámci této části je cílem vytvoření aplikace, která optimalizuje a do určité míry automatizuje vybrané pracovní procesy, které jsou opakovaně vykonávány. Přínosem pro uživatele by mělo být nižší časové zatížení, zjednodušení vykonávaných procesů a do určité míry zamezení lidských chyb. Benefitem pro vlastníka společnosti by poté mělo být snížení zatížení zaměstnanců, což může vést snížení nákladů a tím pádem zvýšení zisku společnosti.

Při vývoji byly využity formulovaná teoretická východiska. Proces vývoje aplikace je rozdělen do těchto částí:

- Analýza požadavků
- Návrh softwarového řešení
- Implementace aplikace
- Testování

4.1 Popis stávajícího procesu

Předmětem podnikání společnosti, pro kterou je aplikace vyvíjena, je půjčování pracovních plošin. Velikostí se společnost řadí mezi malé podniky. Společnost plošiny nakupuje a uchovává je na skladě. V případě poruchy si společnost sama plošiny opravuje. Plošiny se dělí podle druhu pohonu (elektrický, dieselový) a podle maximálního zdvihu a jsou pravidelně revidovány.

Stěžejním pracovním procesem ve společnosti je vypůjčení plošiny. Tento proces začíná kontaktováním společnosti zákazníkem. Obchodní zástupce se v závislosti na požadovaném časovém rozpětí výpůjčky dohodne se zákazníkem na vypůjčení plošiny požadovaného typu a poté zaeviduje novou výpůjčku. V den počátku výpůjčky je poté vlastními prostředky plošina dopravena k zákazníkovi a ke konci výpůjčky opět dopravena zpět do skladu. Ve skladu je poté plošina revidována pro zjištění možných poškození, ať zaviněných zákazníkem či nikoliv. V případě zjištění závady zaviněné zákazníkem je zákazník sankciován dodatečnými poplatky. Plošina je poté dočasně vyřazena a je provedena oprava. Po skončení výpůjčky je zákazníkovi vystavena faktura.

Všechna data využívaná v tomto procesu jsou uchovávána v souborech tabulkového procesoru, které jsou uloženy na veřejné cloudové službě. Jedná se o soubory:

- Výpůjčky – Záznamy s informacemi o vypůjčení plošiny
- Plošiny – Seznam plošin
- Údržby – Záznamy o provedení údržeb plošin
- Faktury – Záznamy s informacemi o vystavených fakturách zákazníkovi a jejich zaplacení

Záznamy v těchto souborech nejsou nijak chráněny před nesprávnými operacemi a není nijak zajištěna jejich integrita. Je zde tedy velké riziko ztráty či zneplatnění dat v důsledku provedení nesprávných operací, které se zvyšuje s jejich častějším vykonáváním.

4.2 Analýza požadavků

Tato část se věnuje specifikaci požadavků pro nové softwarové řešení. Na základě sérií rozhovorů se zaměstnanci, kteří jsou součástí vybraných pracovních procesů, byly formulovány požadavky na aplikaci. Formulované požadavky jsou rozděleny na funkční a nefunkční a jsou stěžejní pro následný návrh aplikace, jelikož výsledná aplikace by měla všechny tyto požadavky splňovat. Na základě požadavků na aplikaci byl poté vytvořen diagram případů užití a jejich specifikace.

4.2.1 Funkční požadavky

- **Prohlížení dat spojených s procesem vypůjčení plošiny –** V aplikaci by mělo být možné prohlížet zaevidované záznamy spojené s procesem vypůjčování plošin.
- **Správa záznamů o výpůjčkách –** Aplikace by měla umožňovat přidání nového záznamu mezi záznamy, se kterými pracuje. Dále by měla umožňovat úpravu a odstranění záznamů, u kterých to integritní omezení povoluje.
- **Evidence plošin –** Aplikace by měla umožňovat zaevidování nových plošin a zaevidování jejich případného vyřazení.
- **Zobrazení volných plošin z daném časovém rozpětí –** V aplikaci by mělo být možné zobrazovat seznam volných plošin k vypůjčení v zadaném časovém období.
- **Evidence informací o údržbách plošin –** Aplikace by měla umožňovat přidávání záznamů o údržbách plošin, které omezují její dostupnost v daném časovém období.
- **Evidence faktur –** Aplikace by měla umožňovat evidování vystavených faktur a uchovávat informaci o jejich zaplacení. Dále by měla umožňovat export faktur do souboru PDF, který je poté možné odeslat zákazníkovi.

- **Poskytnutí statistických přehledů o půjčování plošin** – Aplikace by měla poskytovat vybranou sadu grafických přehledů k prohlížení. Tato sad je tvořena přehledy:
    - Souhrnné tržby podle dní
    - Tržby ve dnech podle zákazníků
    - Počet vypůjčených plošin ve dnech podle typu pohonu
    - Srovnání tržeb v měsících vybraných let
- **Export záznamů do tabulkového procesoru** – Aby bylo možné se záznamy provádět operace, které nejsou v aplikaci implementované, měla by být zajištěna možnost exportu záznamů do tabulkového procesoru Microsoft Excel. Aplikace by měla umožňovat exportovat záznamy výpůjček, plošin, zákazníků, údržeb a faktur do oddělených listů, přičemž u množin záznamů výpůjček, údržeb a faktur by mělo být možné omezit počet exportovaných záznamů podle data počátku vypůjčení, resp. data počátku údržby nebo data vystavení faktury.

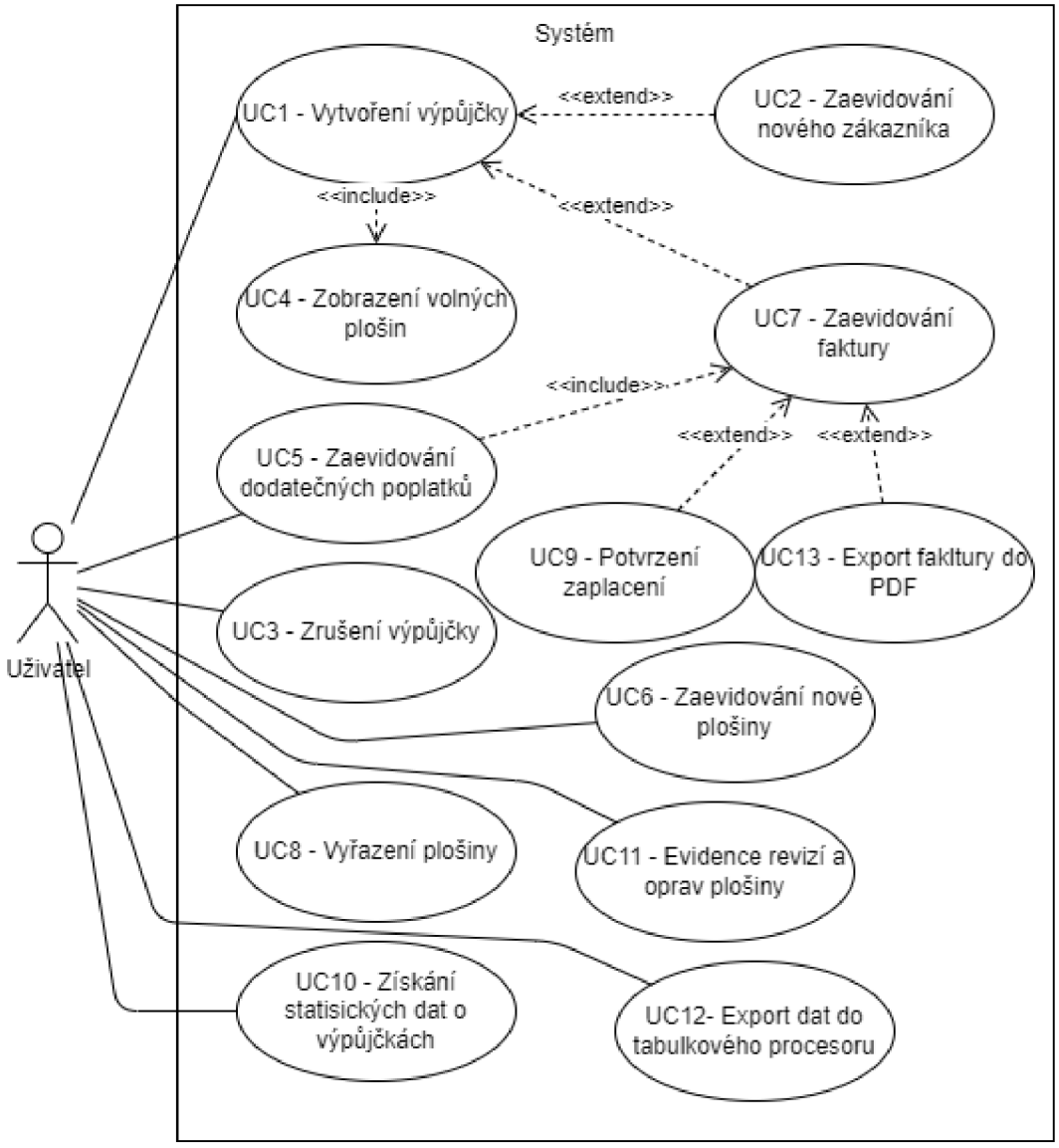
4.2.2 Nefunkční požadavky

- **Platforma** – Požadovanou cílovou platformou pro nasazení aplikace je OS Windows.
- **Persistence a sdílení dat** – Uložené záznamy by měly být stále a měly by být uchovávány mimo klientskou aplikaci, aby mohly být sdíleny s více klienty.
- **Konzistence a integrita datového modelu** – Navržený datový model, se kterým aplikace pracuje, by měl být neměnný, ale zároveň rozšiřitelný. Dále by měla být zajištěna integrita dat pro zamezení ukládání nových neplatných záznamů či zneplatnění stávajících.
- **Přívětivost uživatelského rozhraní** – Aplikace by pro uživatele měla být přehledná a vzhled aplikace konzistentní pro dosažení uživatelovy snadné orientace.

4.2.3 Diagram případů užití

Diagram případů užití znázorňuje interakce mezi uživatelem a systémem. Mezi aktéry interagující se systémem se řadí pouze neurčitý uživatel, jelikož systém nijak uživatele nerozlišuje. Uživatel má v diagramu ikonu člověka. Jednotlivé případy užití jsou v diagramu reprezentovány oválem, který obsahuje jejich název doplněný unikátním identifikátorem. Plné čáry vyjadřují asociaci mezi aktéry a případem užití. Některé případy užití jsou

asociovány pouze s jiným případem užití, tzn. že jsou součástí jiného případu užití (*include*) nebo určitý případ užití rozšiřují (*extend*). Pro vytvoření diagramu případů užití byl využit nástroj **Draw.io**. [42]



Obrázek 9 - Diagram případů užití. Zdroj: vlastní

4.2.4 Specifikace případů užití

Tabulka 3 představuje dokumentaci stanovených případů užití v předchozí části. Každý případ užití má svůj identifikátor (formát *Identifikátor – Název případu užití*), aktéry, kteří spolu interagují, podmínky pro zahájení a dokončení scénáře, základní scénář a alternativní scénář. Kroky v základním scénáři jsou číslovány jednoúrovňově a v alternativním scénáři

dvouúrovňově, přičemž číslo první úrovně kroku vyznačuje, na jaký krok v základním scénáři navazuje.

Tabulka 3 - Specifikace případů užití

Název případu užití		**UC1 – Vytvoření výpůjčky**
Aktéři		Uživatel, Systém
Podmínky pro zahájení		
Základní scénář	1.	Uživatel přejde do sekce Výpůjčky
	2.	Uživatel klikne na tlačítko Nová výpůjčka
	3.	Uživatel zadá časové rozpětí výpůjčky
	4.	Systém vyplní rozevírací seznam dostupnými plošinami
	5.	Uživatel vybere ze seznamu plošinu k vypůjčení
	6.	Uživatel vybere zákazníka ze seznamu již evidovaných zákazníků
	7.	Uživatel potvrdí vytvoření nové výpůjčky
	8.	Systém zkontroluje všechny potřebné údaje, zapíše zadané údaje do databáze, informuje uživatele o úspěchu a zavře formulář
	9.	Systém zobrazí dialog, zda chce uživatel vystavit fakturu – UC7 **krok 3.**
Alternativní scénář	4.1	V případě, že není dostupná žádná plošina, systém nevyplní seznam žádnými plošinami
	4.2.	Dále tok pokračuje krokem 3., nebo uživatel ukončí operaci vytvoření výpůjčky
	6.1.	Uživatel zaeviduje nového zákazníka – UC2 **krok 2.**
	6.2.	Dále tok pokračuje krokem 7.
	8.1.	Systém zobrazí hlášku o neplatných vstupech
Podmínky pro dokončení		V zadaném časovém rozpětí výpůjčky musí být dostupná alespoň jedna plošina; Přístup k databázi
Název případu užití		**UC2 – Zaevidování nového zákazníka**
Aktéři		Uživatel, Systém
Podmínky pro zahájení		
Základní scénář	1.	Uživatel klikne na tlačítko Nový Zákazník
	2.	Uživatel zvolí typ nového zákazníka (Nepodnikatel, OSVČ, nebo Právnická osoba)
	3.	Uživatel vyplní kontaktní údaje
	4.	Uživatel potvrdí vytvoření nového zákazníka
	5.	Systém ověří, že se v databázi nevyskytuje záznam s daným identifikačním číslem

	6.	Systém zkontroluje potřebné vstupy, uloží zadané údaje do databáze, informuje uživatele o úspěchu a zavře formulář
Alternativní scénář	6.1.	Systém zobrazí hlášku o neplatných vstupech
Podmínky pro dokončení		Nový zákazník ještě není evidován v databázi; Přístup k databázi
Název případu užití		**UC3 – Zrušení výpůjčky**
Aktéři		Uživatel, Systém
Podmínky pro zahájení		Přístup k databázi
Základní scénář	1.	Uživatel přejde do sekce Výpůjčky
	2.	Uživatel vybere jednu nebo více výpůjček ze seznamu a klikne na tlačítko Odstranit
	3.	Systém zobrazí dialog pro potvrzení odstranění záznamů
	4.	Uživatel klikne na tlačítko potvrdit
	5.	Systém odstraní výpůjčku z databáze a informuje uživatele
Alternativní scénář	5.1.	Systém zobrazí hlášku, že výpůjčka již proběhla nebo její faktura je již zaplacena není ji možné odstranit
Podmínky pro dokončení		Přístup k databázi
Název případu užití		**UC6 – Zaevidování nové plošiny**
Aktéři		Uživatel, Systém
Podmínky pro zahájení		
Základní scénář	1.	Uživatel přejde do sekce Plošiny a klikne na tlačítko Přidat plošinu
	2.	Uživatel zadá do formuláře všechny potřebné údaje
	3.	Systém zkontroluje vstupní pole
	4.	Systém zapíše nový záznam do databáze a informuje uživatele o úspěchu
Alternativní scénář	4.1.	Systém zobrazí hlášku, že zadané údaje nejsou platné
Podmínky pro dokončení		Přístup k databázi
Název případu užití		**UC7 – Zaevidování faktury**
Aktéři		Uživatel, Systém
Podmínky pro zahájení		
Základní scénář	1.	Uživatel přejde do sekce Výpůjčky
	2.	Uživatel vybere výpůjčku a klikne na tlačítko Nová faktura

	3.	Systém zobrazí formulář pro zaevidování faktury
	4.	Uživatel vyplní potřebné údaje
	6.	Systém zkontroluje vstupní údaje, zapíše nový záznam do databáze a informuje uživatele
Alternativní scénář	6.1.	Systém zobrazí hlášku, že zadané údaje nejsou platné
Podmínky pro dokončení		Přístup k databázi
Název případu užití		**UC8 – Vyřazení plošiny**
Aktéři		Uživatel, Systém
Podmínky pro zahájení		Přístup k databázi
Základní scénář	1.	Uživatel přejde do sekce Plošiny
	2.	Uživatel vybere ze seznamu plošinu a klikne na tlačítko Vyřadit
	3.	Systém zapíše údaj o vyřazení do databáze a informuje uživatele
Alternativní scénář	3.	Systém zobrazí hlášku, že plošina je přiřazena budoucí výpůjčce
Podmínky pro dokončení		Přístup k databázi
Název případu užití		**UC9 – Potvrzení zaplacení**
Aktéři		Uživatel, Systém
Podmínky pro zahájení		Přístup k databázi
Základní scénář	1.	Uživatel přejde do sekce Faktury
	2.	Uživatel vybere fakturu a klikne na tlačítko Potvrdit zaplacení
	3.	Systém zapíše informaci o zaplacení faktury do databáze
Alternativní scénář		
Podmínky pro dokončení		Přístup k databázi
Název případu užití		**UC10 – Získání statistických dat o výpůjčkách**
Aktéři		Uživatel, Systém
Podmínky pro zahájení		
Základní scénář	1.	Uživatel přejde do sekce Statistiky
	2.	Uživatel vybere přehled z předem definované nabídky
	3.	Systém zobrazí vybraný přehled
Alternativní scénář		Přístup k databázi
Podmínky pro dokončení		
Název případu užití		**UC11 – Evidence revizí a oprav plošiny**

Aktéři		Uživatel, Systém
Podmínky pro zahájení		Přístup k databázi
Základní scénář	1.	Uživatel přejde do sekce Plošiny
	2.	Uživatel vybere plošinu ze seznamu a klikne na tlačítko Přidat opravu/revizi
	3.	Uživatel vyplní údaje o opravě/revizi a potvrdí formulář
	4.	Systém zkontroluje údaje, odešle je do databáze a informuje uživatele
Alternativní scénář	4.1.	Systém informuje uživatele, že jsou zadané údaje neplatné nebo že v daném termínu je plošina přiřazena výpůjčce
Podmínky pro dokončení		V termínu opravy nesmí být plošina přiřazena výpůjčce; Přístup k databázi
Název případu užití		**UC12 – Export dat do tabulkového procesoru**
Aktéři		Uživatel, Systém
Podmínky pro zahájení		Přístup k databázi
Základní scénář	1.	Uživatel přejde do sekce Export
	2.	Systém zobrazí výběr možných přehledů k exportu
	3.	Uživatel vybere požadované přehledy k exportu a u vybraných může vybrat časové omezení záznamů
	4.	Systém zobrazí dialog pro výběr umístění nového souboru
	5.	Uživatel vybere složku pro uložení souboru
	6.	Systém vygeneruje soubor Excelu s požadovanými přehledy dat a uloží jej do zvoleného umístění
Alternativní scénář		
Podmínky pro dokončení		
Název případu užití		**UC13 – Export faktury do PDF**
Aktéři		Uživatel, Systém
Podmínky pro zahájení		Přístup k databázi
Základní scénář	1.	Uživatel přejde do sekce Faktury
	2.	Uživatel vybere fakturu ze seznamu a klikne na tlačítko Export do PDF
	3.	Uživatel vybere požadované přehledy k exportu a u vybraných může vybrat časové omezení záznamů
	4.	Systém zobrazí dialog pro výběr umístění nového souboru
	5.	Uživatel vybere složku pro uložení souboru

	6.	Systém vygeneruje soubor PDF a uloží jej do zvoleného umístění
Alternativní scénář		
Podmínky pro dokončení		

Zdroj: vlastní

V předchozí tabulce nejsou specifikovány dva případy užití, a to UC4 a UC5. Případ užití *UC4 – Zobrazení volných plošin* je součástí *UC1* a reprezentuje část scénáře týkající se výběru plošiny pro výpůjčku. Případ užití *UC5 – Zaevidování dodatečných poplatků* reprezentuje proces evidence faktury pro zákazníka týkající se např. vzniklých škod během výpůjčky a má téměř totožný scénář jako *UC7*, který se od liší pouze ve vyplnění informace, že jde o mimořádnou fakturu. Na základě této informace je poté zaevidována faktura, která má nulovou sazbu DPH.

4.3 Návrh softwarového řešení

Na základě analýzy uživatelských požadavků a formulovaných případů užití je následně možné navrhnout softwarové řešení, které bude tyto požadavky splňovat a bude v ní možné provádět specifikované operace. V rámci této kapitoly je nejprve popsán výběr technologie, pomocí které bude aplikace implementována, dále je navržena architektura aplikace, datový model a komponenty v prezentační vrstvě.

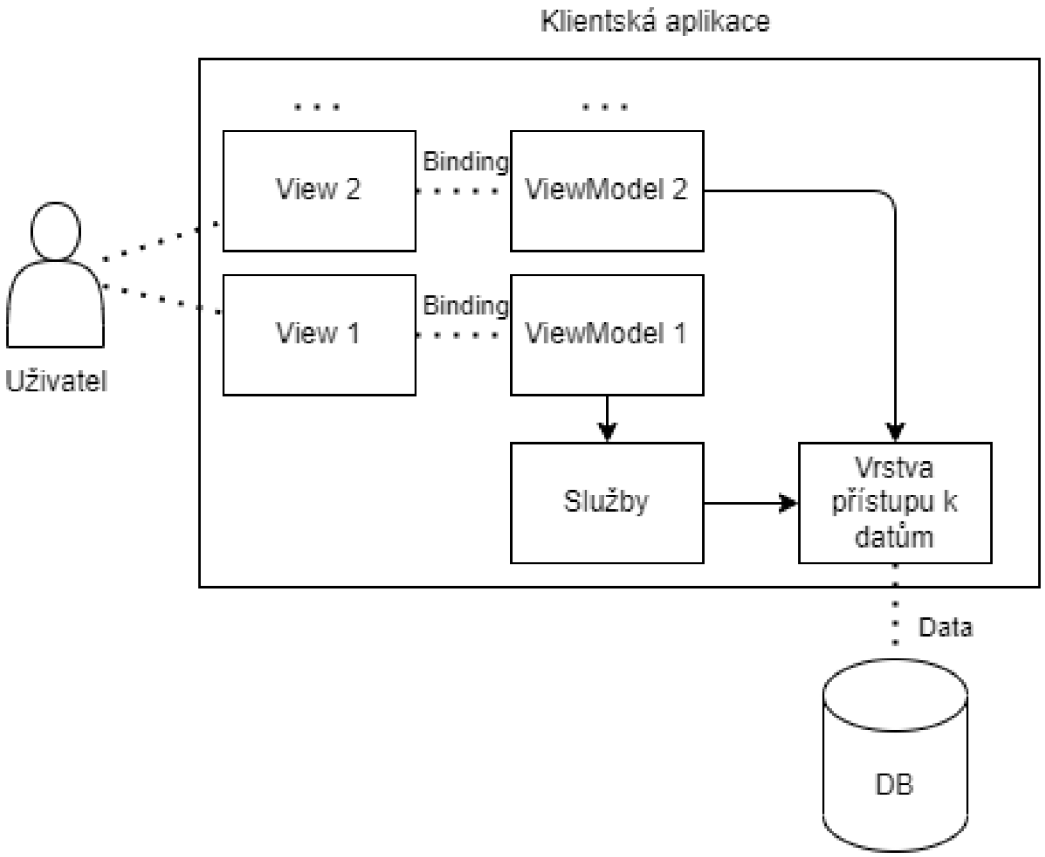
4.3.1 Technologie pro implementaci

Prvním krokem při návrhu aplikace je výběr technologie pro implementaci aplikace. V rámci platformy .NET je možné vyvíjet aplikace pro různé běhové prostředí. Mezi nefunkčními požadavky je specifikováno, že aplikace bude používána výhradně na pracovních stanicích s operačním systémem Windows. Na základě této informace bylo rozhodnuto, že aplikace bude implementována pomocí frameworku WPF. Framework WPF je používán pro implementaci nejrůznějších druhů desktopových aplikací pro OS Windows a při výběru byl mezi ostatními multiplatformními frameworky upřednostněn právě z důvodu požadované platformy OS Windows.

4.3.2 Architektura aplikace

Po výběru frameworku, ve kterém bude aplikace implementována, následuje navržení architektury aplikace. Navržení architektury je klíčové pro následnou implementaci, jelikož ovlivňuje složitost, přehlednost a rozšiřitelnost kódu aplikace. Následující schéma ilustruje

navrženou architekturu aplikace a komponenty, které tuto architekturu tvoří. K vytvoření schématu byl využit nástroj **Draw.io**. [42]



Obrázek 10 - Schéma architektury aplikace. Zdroj: vlastní

Pro propojení služeb s prezentační vrstvou je využita třívrstvá architektura Model-View-ViewModel, jejíž výhodou je oddělení business logiky aplikace a prezentační vrstvy. Část Model reprezentují služby s různou business logikou a služby ve vrstvě přístupu k datům. Data jsou ukládána mimo klientskou aplikaci v databázovém systému a je k nim přistupováno právě přes vrstvu přístupu k datům. Se službami komunikují třídy typu ViewModel, které fungují jako propojovací služby business logiky a prezentační vrstvy. Prezentační vrstva je reprezentována sadou pohledů, která slouží pro prezentaci dat uživateli a obsahuje ovládací prvky, které umožňují komunikaci se službami a manipulaci s daty. Tato vrstva neobsahuje žádnou aplikační logiku a ovládací prvky jsou svázány (nabindovány) na vlastnosti třídy ViewModel, ve které jsou těmto prvkům nastavovány hodnoty a spouštěny metody reagující na uživatelovy interakce.

Vrstva přístupu k datům je tvořena sadou zapouzdřených tříd. Služby nebo ViewModely přistupují k datům pomocí služby, která je navržena podle návrhového vzoru Unit of Work. Tato třída umožňuje složit databázovou transakci a provedení změn záznamů

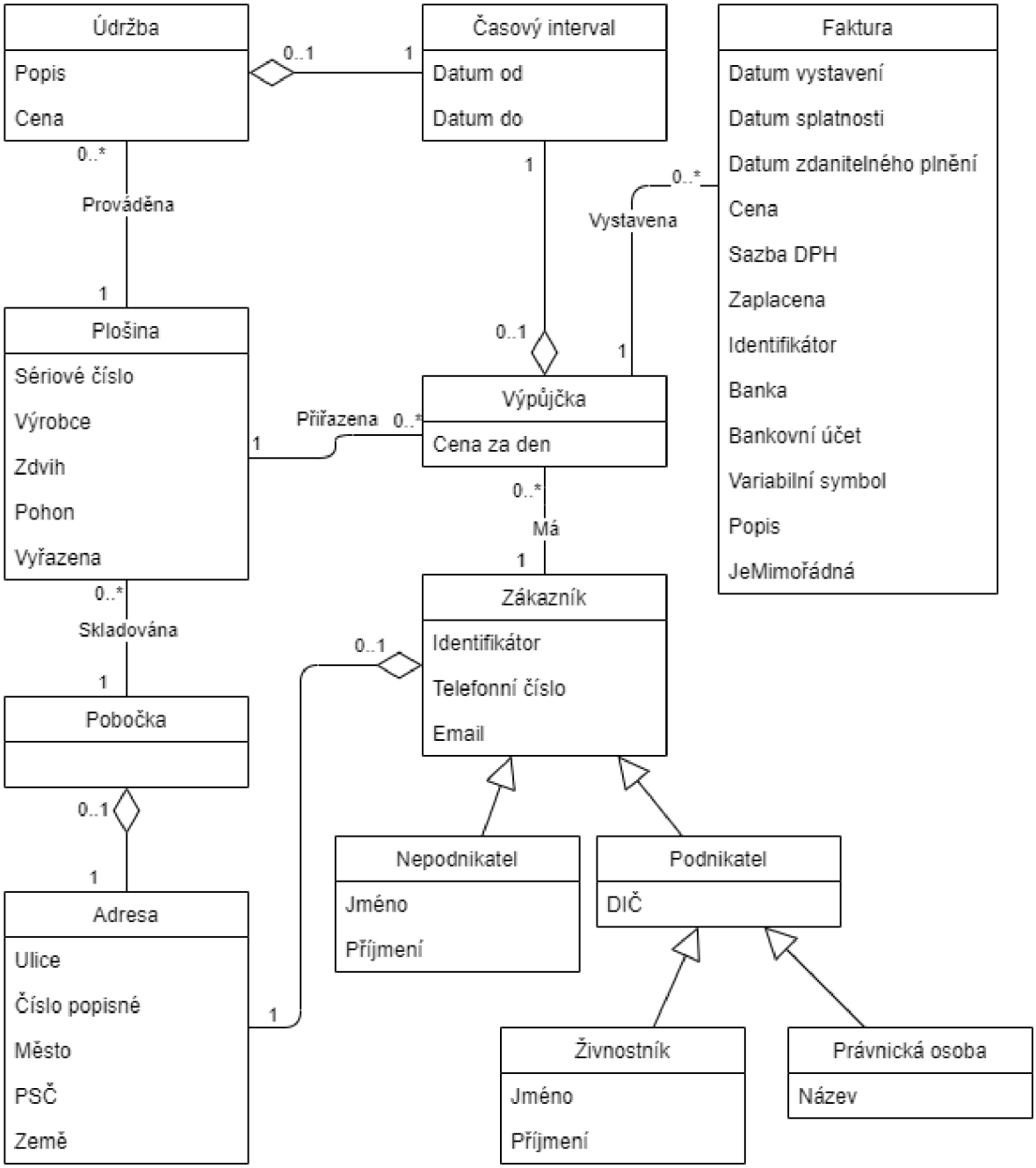
v databázi. Díky vykonání sledu operací v rámci jedné databázové transakce je zajištěn přechod z jednoho konzistentního stavu do druhého. K jednotlivým databázovým tabulkám je přistupováno přes vlastnosti třídy vytvořené podle vozru Unit of Work, které jsou navrženy podle vzoru Repository. Tento vzor umožňuje provedení operací na daty, jako čtení, vytvoření, úprava a odstranění záznamů (CRUD). Pro každou databázovou tabulku sloužící k uchování záznamů existuje vlastnost typu Repository ve třídě Unit of Work.

4.3.3 Datový model

V rámci návrhu datového modelu aplikace jsou definovány data, se kterými bude aplikace pracovat. Nejdříve je vytvořen diagram tříd, jednotlivé entity, ze kterých se model skládá, jsou poté popsány a poté je navržen způsob zajištění persistence dat.

4.3.3.1 Diagram tříd

Pro zachycení datových entit a vztahů mezi nimi, se kterými aplikace pracuje, byl vytvořen diagram tříd, který je součástí grafického jazyka UML (Obrázek 11). Jednotlivé třídy jsou v diagramu vyznačeny obdélníkovým boxem, jejich název je umístěn v oddělené horní části boxu a vlastnosti jsou obsahem dolní části boxu. Mezi třídami jsou vyznačeny vztahy typu agregace (čára s kosočtvercem), asociace (prostá čára) a generalizace (čára s šipkou). U vztahů typu agregace a asociace jsou vyznačeny kvantifikátory vztahu a u vztahu typu asociace pojmenování vztahu.



Obrázek 11 - Diagram tříd. Zdroj: vlastní

4.3.3.2 **Zajištění persistence dat**

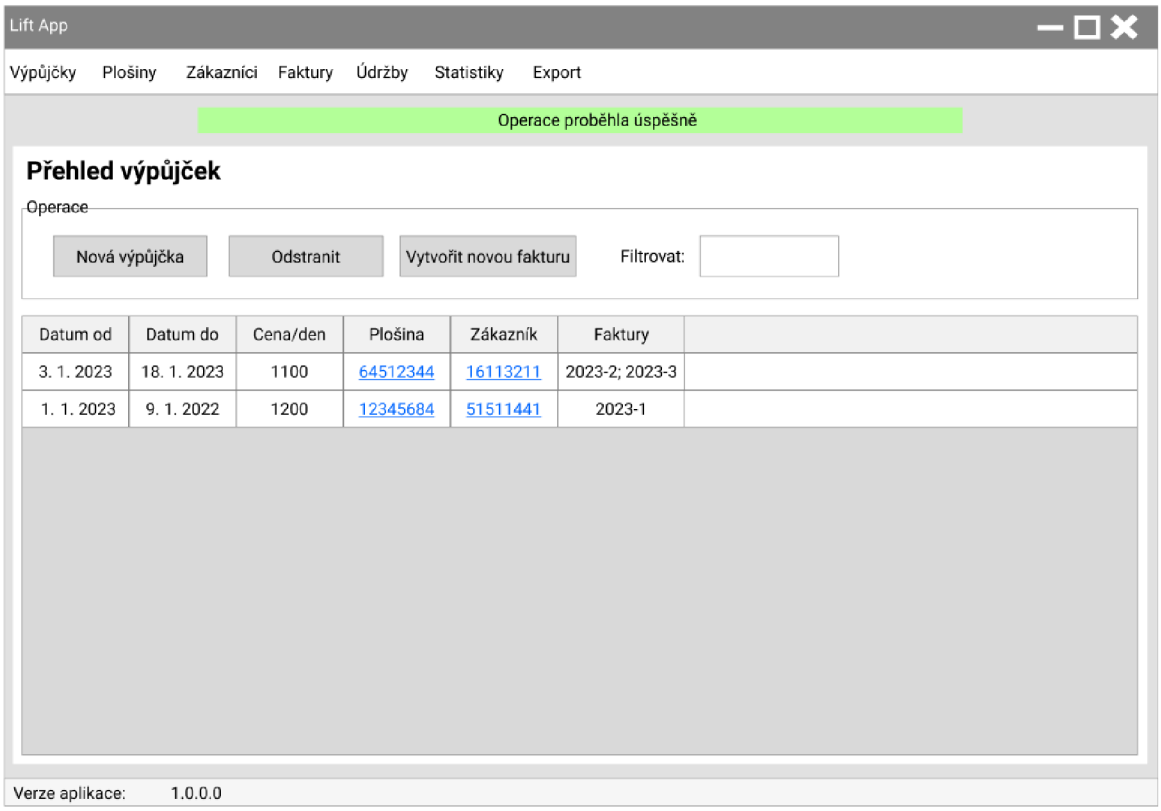
Navrhovaná aplikace má sloužit k provádění operací se záznamů, které je nutné uchovávat v dlouhodobém měřítku a je žádoucí, aby ke stejné množině záznamů mohlo být přistupováno z více instancí aplikace. Z tohoto důvodu je nutné zajistit persistenci dat, se kterými aplikace pracuje.

Už při návrhu aplikační architektury bylo stanoveno, že záznamy budou uchovávány mimo samotnou aplikaci v databázovém systému. Bylo rozhodnuto, že bude použit relační databázový systém, a to konkrétně **PostgreSQL** [43]. „PostgreSQL je výkonný objektově-relační databázový systém s otevřeným zdrojovým kódem s více než 35 lety aktivního vývoje, který mu získal silnou pověst spolehlivosti, robustnosti funkcí a výkonu.“ [43]

Vytvoření datového modelu v relační podobě a komunikace s databázovým systémem budou zajištěny pomocí technologie **Entity Framework Core**. Bude využita technika Code-First pro migraci datové struktury z objektové podoby do relační a jako zprostředkovatel komunikace s databázovým systémem PostgreSQL bude využita knihovna **Npgsql** [44]. „Npgsql je open source ADO.NET Data Provider pro PostgreSQL, umožňuje programům napsaným v C#, Visual Basic a F# přístup k databázovému serveru PostgreSQL. Je ze 100 % implementován ve C# kódu, je zdarma a je open source.“ [44]

4.3.4 Prezentační vrstva

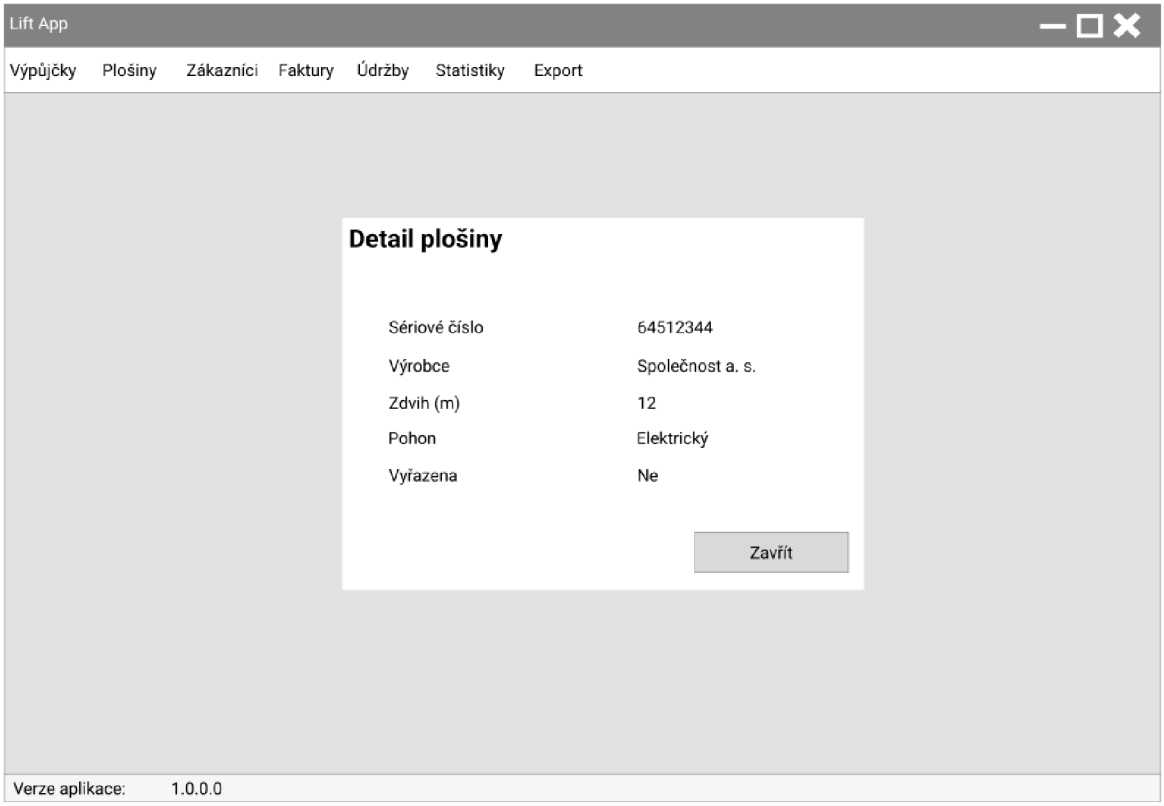
Při tvorbě návrhu uživatelského rozhraní bylo vycházeno z požadavku na přehlednost a univerzálnost komponent pro prezentaci dat a provádění operací. Návrh UI byl proveden pomocí vytvoření drátových modelů (Wireframe), které byly konzultovány s budoucími uživateli. V rámci implementace budou poté vytvořeny pohledy, které respektují rozložení prvků v drátových modelech. Pro tvorbu drátových modelů byl využit software **Figma** [45].



Obrázek 12 - Drátový model sekce Výpůjčky. Zdroj: vlastní

Obrázek 12 představuje drátový model sekce Výpůjčky. Tento model také specifikuje základní rozložení uživatelského rozhraní (layout). Každý pohled v aplikaci obsahuje horní menu, které funguje jako navigace napříč celou aplikací. Tato navigace obsahuje tlačítka Výpůjčky, Plošiny, Zákazníci, Faktury, Statistiky a Export, které po kliknutí přepnou obsah hlavního kontejneru na zvolenou sekci. Pod horním menu je mizející box, který obeznámí uživatele o úspěšnosti provedené operace se záznamy. Pod tímto boxem je již hlavní kontejner sekce, který v horní části obsahuje popis zobrazené sekce, pod ním skupinu ovládacích prvků, které je možné v dané sekci provádět a tabulku pro zobrazení konkrétních záznamů uložených v databázovém systému. Pod hlavním kontejnerem je poté už jen box pro zobrazení používané verze aplikace. Výše popsané základní rozvržení aplikace je používáno napříč všemi pohledy a je měněn pouze obsah hlavního kontejneru. Rozložení hlavního kontejneru podle předchozího modelu je totožné v sekcích Plošiny, Zákazníci, Faktury a Údržby.

V sekci Výpůjčky je v tabulce u každého záznamu link pro zobrazení detailu plošiny a link pro detail zákazníka, kdy po kliknutí na jeden z těchto linků je zobrazen detail záznamu plošiny/zákazníka (viz. Obrázek 13). Tento detail dále obsahuje tlačítko pro navigaci zpět do seznamu výpůjček.



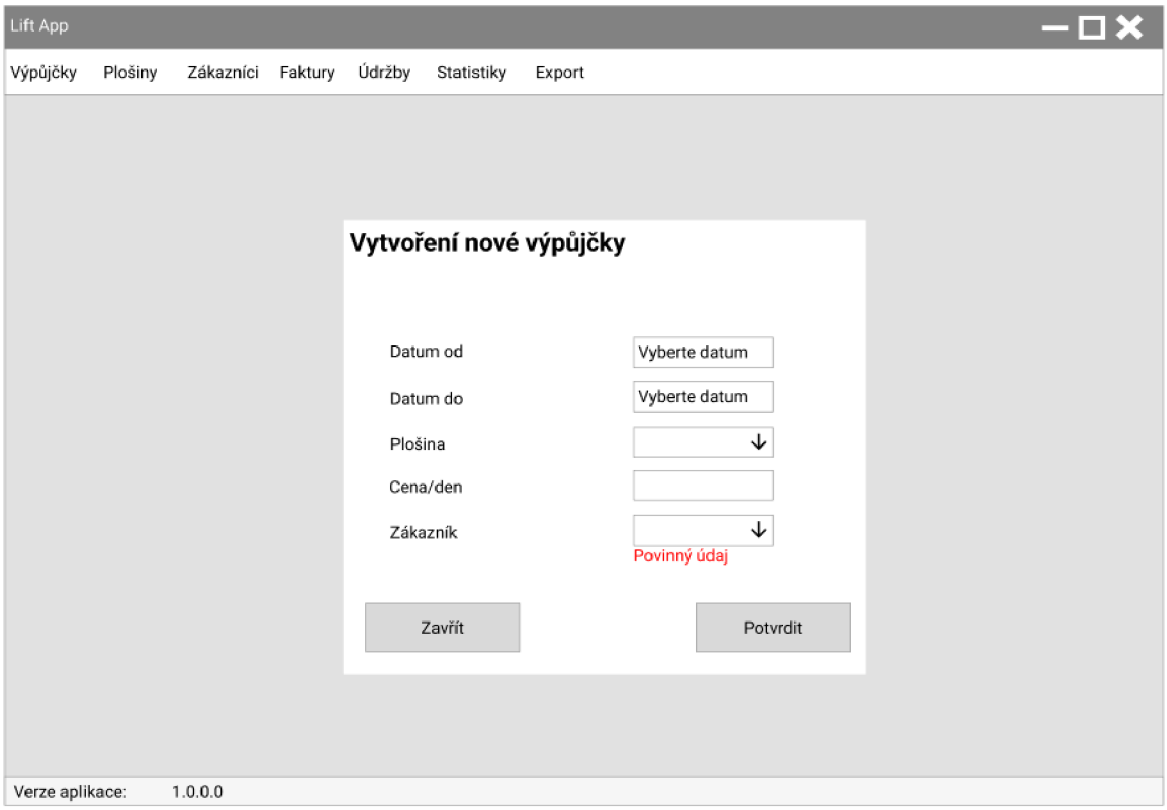
Obrázek 13 - Drátový model detailu plošiny. Zdroj: vlastní

V sekci Výpůjčky je v boxu Operace umístěno tlačítko Nová výpůjčka a po kliknutí na toto tlačítko je obsah hlavního kontejneru změněn na formulář pro vytvoření nového záznamu výpůjčky (viz. Obrázek 15). Tento formulář obsahuje vstupní pole pro Datum od a Datum do, dále rozevírací seznam pro výběr plošiny k vypůjčení, vstupní pole pro zadání ceny vypůjčení za den, a rozevírací seznam pro výběr zákazníka. Všechny údaje jsou po potvrzení formuláře validovány a případné neplatné vstupy jsou označeny hláškou pod vstupním polem. Pokud ještě nikdy nebyla zákazníkovi vypůjčena plošina, je možné vytvořit nový záznam zákazníka po kliknutí na tlačítko Nový u rozevírací seznamu se zákazníky. Po kliknutí na toto tlačítko je zobrazen formulář se vstupními poli pro zaevidování nového zákazníka (viz. Obrázek 14). Tento formulář je rozdělen na dvě skupiny prvků, a to skupina se vstupními poli pro kontaktní údaje zákazníka a skupina se vstupními poli pro adresu zákazníka. Zobrazení vstupních polí v první skupině je závislé na výběru druhu zákazníka:

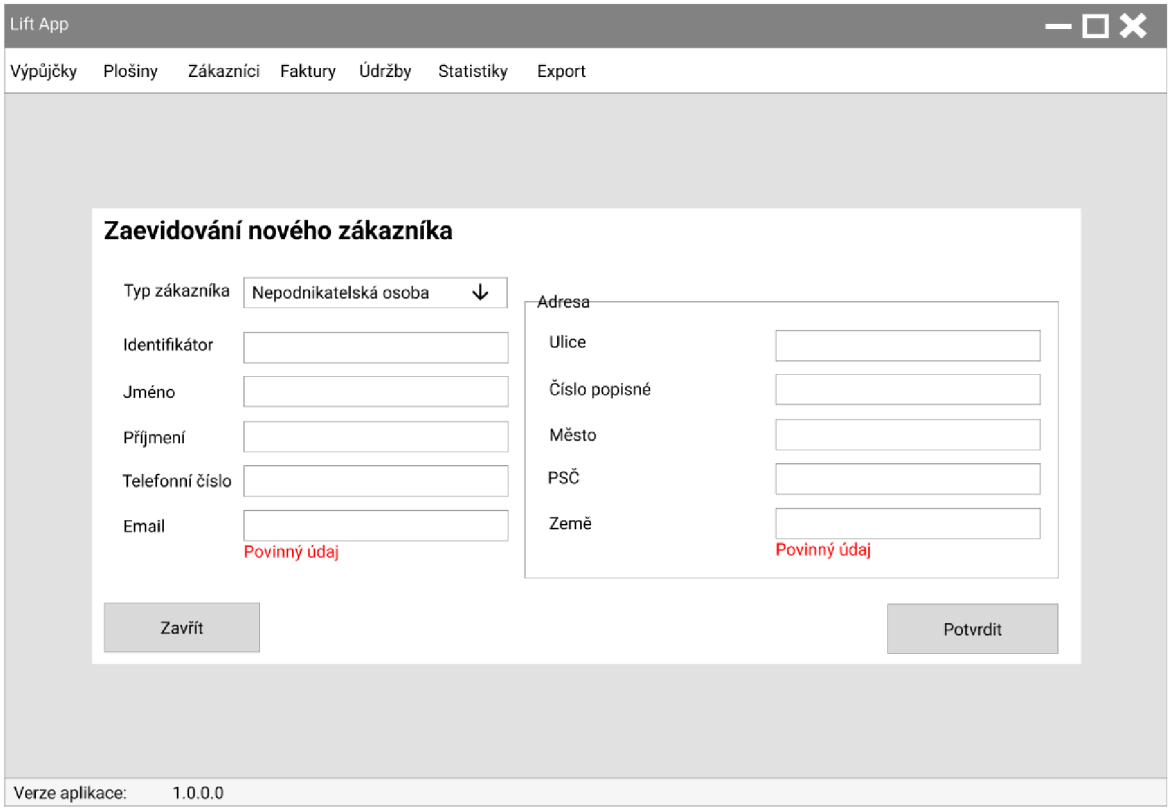
- Nepodnikatel – Identifikátor, Jméno, Příjmení, Telefonní číslo, Email

- OSVČ – Identifikátor, DIČ (nepovinné), Jméno, Příjmení, Telefonní číslo, Email
- Společnost – Identifikátor, DIČ (nepovinné), Název, Telefonní číslo, Email

V dolní části formuláře jsou poté tlačítka pro navigaci zpět a potvrzení formuláře. Obdobě jako u formuláře pro novou výpůjčku jsou validovány hodnoty ve vstupních polích po potvrzení formuláře.



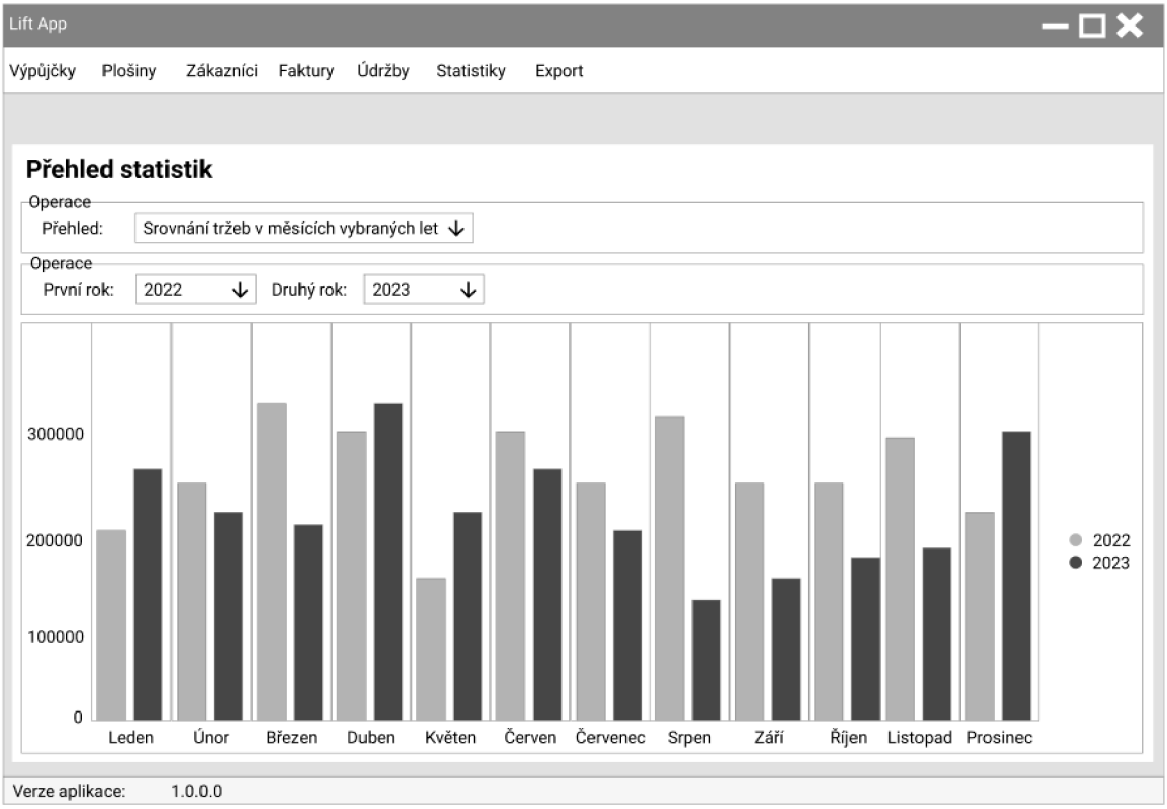
Obrázek 15 - Drátový model Vytvoření nové výpůjčky. Zdroj: vlastní



Obrázek 14 - Drátový model Zaevidování nového zákazníka. Zdroj: vlastní

Ostatní formuláře pro vytvoření nového záznamu nebo jeho úpravu mají obdobné rozložení a mění se u nich pouze seznam vstupních polí.

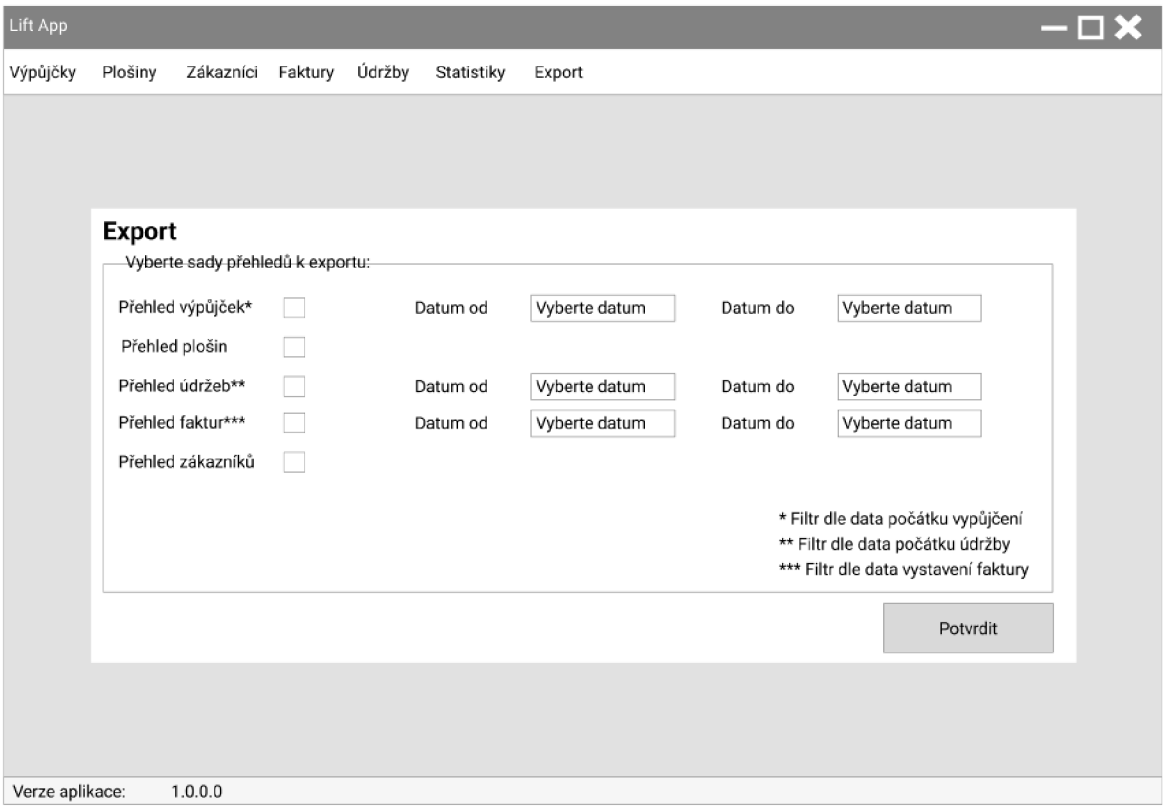
U sekcí Statistiky a Export je navržena odlišná podoba hlavního kontejneru. Hlavní kontejner u sekce Statistiky obsahuje obdobě jako u sekcí Výpůjčky, Plošiny, Zákazníci, Faktury a Údržby box s výběrem operací. Tento box obsahuje pouze jeden prvek, a to rozevírací seznam s vybranými přehledy dat. Po výběru je pod tímto boxem načten graf reprezentující vybraný přehled (viz. Obrázek 16). Toto se netýká přehledu Srovnání tržeb v měsících vybraných let, kdy je nad prostorem pro načtený graf navíc umístěn box se rozevíracími seznamy pro výběr let, které mají být použity pro srovnání. Tyto vstupy jsou validovány, aby bylo zajištěno, že vybraný rok v prvním poli je dříve než vybraný rok v druhém poli.



Obrázek 16 - Drátový model sekce Statistiky. Zdroj: vlastní

Rozložení prvků hlavního kontejneru v sekci Export čerpá z dříve navrženého rozvržení formulářů pro vytvoření nového záznamu (viz. Obrázek 17). Obsahem tohoto kontejneru je seznam dostupných přehledů záznamů k exportu a u každého přehledu je zaškrtávací pole pro výběr dané přehledu, který má být exportován. U některých jsou navíc vstupní pole omezení množiny záznamů podle časového rozpětí, přičemž u každého reportu se jedná o jiný časový údaj, k čemuž jsou umístěny vysvětlivky v dolní části kontejneru. Pod

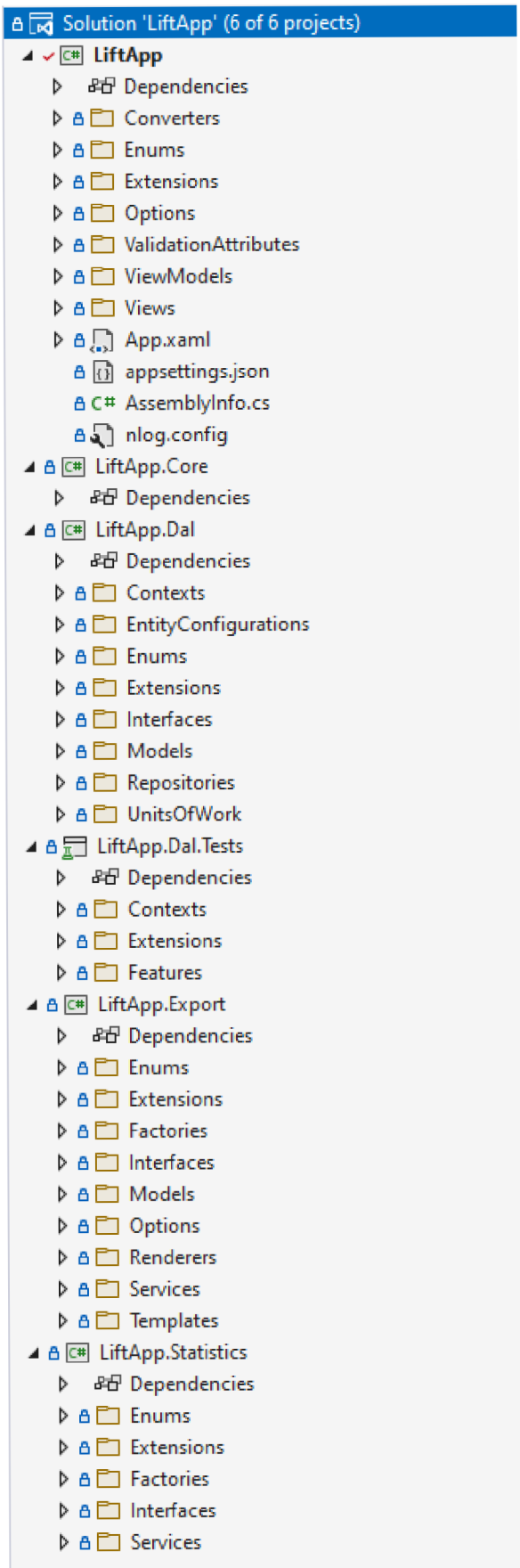
těmito vysvětlivkami je pak už jen tlačítko pro potvrzení výběru přehledů. Po kliknutí na toto tlačítko je zobrazen dialog pro výběr umístění souboru.



Obrázek 17 - Drátový model sekce Export. Zdroj: vlastní

4.3.5 Struktura projektu

Před samotnou implementací aplikace je vhodné navrhnout strukturu softwarového řešení. Vhodně navržené rozdělení komponent do projektů a složek zvyšuje přehlednost pro vývojáře, lepší testovatelnost a snadnější rozšiřitelnost. Zdrojový kód by měl být rozdělen do co nejmenších jednotek jmenných prostorů a tyto jmenné prostory by měly vhodně popisovat, jaké komponenty jsou její součástí.



Obrázek 18 - Adresářová struktura aplikace. Zdroj: vlastní

Obrázek 18 představuje navrženou strukturu aplikace. Řešení je rozděleno do 6 projektů pojmenovaných *LiftApp*, *LiftApp.Core*, *LiftApp.Dal*, *LiftApp.Dal.Tests*, *LiftApp.Export* a *LiftApp.Statistics*.

LiftApp představuje projekt pro prezentační vrstvu. Nejobsáhlejšími složkami v tomto projektu jsou *ViewModels a Views*, kde jsou umístěny jednotlivé pohledy aplikace a třídy typu ViewModel umožňující ovládání těchto pohledů. Složka *Converters* obsahuje třídy pro převod hodnot jednoho typu do druhého využívané v pohledech, složka *Enums* obsahuje výčtové typy používané v tomto projektu, napříč projekty se vyskytuje složka *Extensions* (třídy obsahující metody pro zaregistrování služeb do DI kontejneru), dále projekt obsahuje složky *Options* (třídy pro přístup ke konfiguraci aplikace) a *ValidationAttributes* (třídy definující validační atributy pro validaci formulářů). Významnými soubory, které nejsou umístěny v žádné složce, jsou *App.xaml*, *App.xaml.cs*, *appsettings.json* a *nlog.config*. První soubor představuje základní pohled WPF aplikace, druhý obsahuje třídu *App*, kde je inicializováno jádro aplikace, třetí obsahuje konfiguraci aplikace a soubor *nlog.config* konfiguruje logování.

LiftApp.Core je určen pro služby a reference na knihovny, které jsou používány napříč více projekty.

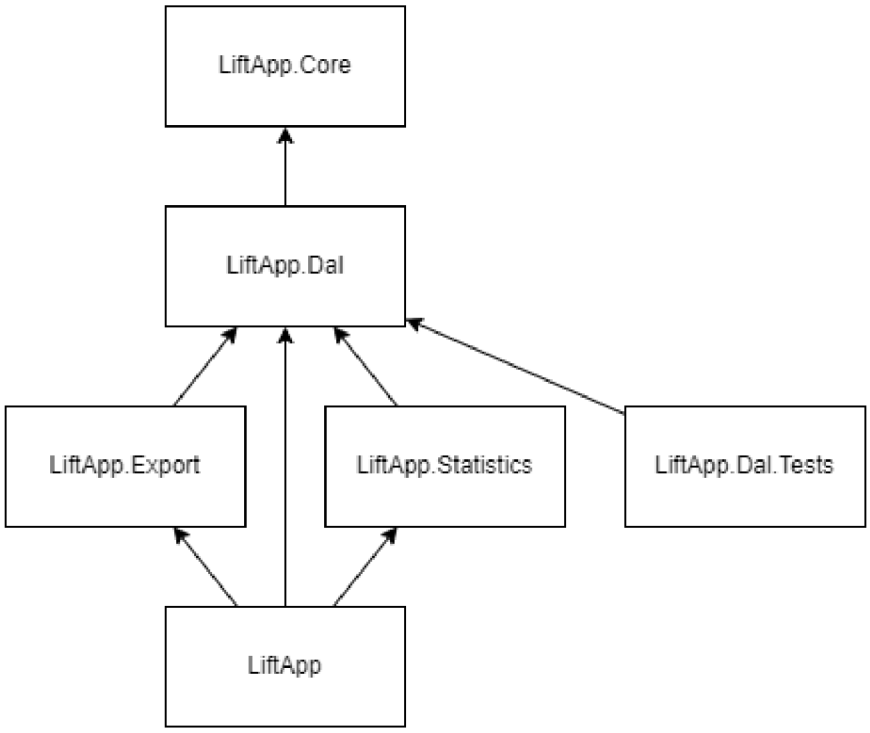
LiftApp.Dal je projekt pro vrstvu přístupu k datům. Složky *Contexts*, *EntityConfigurations*, *Interfaces*, *Repositories* a *UnitsOfWork* obsahují služby, které realizující vrstvu přístupu k datům. Složky *Models* a *Enums* obsahují třídy definující entity dříve navrženého datového modelu.

LiftApp.Dal.Tests je projekt určen pro testování služeb tvořících vrstvu přístupu k datům.

LiftApp.Export reprezentuje projekt se službami pro export přehledů dat z aplikace. Obsahuje složky *Renderers* a *Services* s třídami s business logikou realizující proces exportu, dále *Models* a *Options* (třídy pro přístup ke konfiguraci aplikace), *Enums*, *Factories* a *Interfaces* (modely, služby a rozhraní pro získání konkrétní služby pro export přehledu) a *Templates* (šablony přehledů).

Posledním projektem je *LiftApp.Statistics*. Tento projekt realizuje načtení přehledů dat v sekci Statistiky. Obsahuje složky *Services* (business logika realizující vytvoření grafů), *Enums*, *Factories* a *Interfaces* (modely, služby a rozhraní pro získání konkrétní služby pro vytvoření grafu).

Spolu s návrhem struktury aplikace jsou navrženy také závislosti mezi jednotlivými projekty. Navržené závislosti ilustruje Obrázek 19.



Obrázek 19 - Schéma závislostí mezi projekty. Zdroj: vlastní

4.4 Implementace

Po analýze problému a návrhu softwarového řešení následuje samotná implementace. Jedná se o časově nejnáročnější krok a kvalita implementované aplikace je závislá na kvalitě provedené analýzy a návrhu.

V předchozí části bylo navrženo, že implementace aplikace bude provedena pomocí frameworku WPF určeného pro vývoj desktopových aplikací na OS Windows. Nejprve je implementována základní infrastruktura aplikace realizující její běh, dále je implementována vrstva pro přístup k datům zajišťující jejich persistenci a následně jsou implementovány jednotlivé sekce aplikace zahrnující jejich pohledy a používané služby. Během vývoje aplikace jsou také implementovány jednotkové testy pro vybrané komponenty.

4.4.1 Základní infrastruktura aplikace

Pro běh aplikace je využit generický hostitel popsaný v kapitole 3.5. Základní pohled aplikace (*App.xaml*) obsahuje definuje události zapnutí a vypnutí aplikace, které jsou

zachytávané v třídě *App*. Dále jsou zde definovány některé styly používané napříč více pohledy (viz. Kód 19).

```
<Application x:Class="LiftApp.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:LiftApp"
             Startup="Application_Startup"
             Exit="Application_Exit">
    <Application.Resources>
        <Style x:Key="MenuItem" TargetType="MenuItem">
            <Setter Property="Padding" Value="5" />
        </Style>
            …
    </Application.Resources>
</Application>
```

Kód 19 - Základní pohled aplikace. Zdroj: vlastní

V třídě *App* (soubor *App.xaml.cs*) je při startu aplikace inicializován hostitel aplikace, dále jsou zde nastaveny služby pro logování událostí a přístup ke konfiguraci aplikace, a v samostatné metodě jsou zaregistrovány služby ze všech ostatních projektů (viz. Kód 20). Jednotlivé metody registrující služby volané z této metody jsou umístěny v příslušných projektech, což umožňuje testování jednotlivých projektů mimo hlavní projekt *LiftApp* (metody mohou být přepoužity pro zaregistrování služeb v samostatných projektech pro testy).

```
private static void ConfigureServices(IHostBuilder builder)
{
    builder.ConfigureServices((context, services) =>
    {
        services.RegisterDataAccessLayerServices(context.Configuration);
        services.RegisterExportServices(context.Configuration);
        services.RegisterReportServices(context.Configuration);
        services.RegisterViews(context.Configuration);
        services.RegisterViewModels(context.Configuration);

        services.Configure<NewInvoiceOptions>(
            context.Configuration.GetSection(nameof(NewInvoiceOptions)));
    });
}
```

Kód 20 - Metoda ConfigureServices pro zaregistrování služeb do DI kontejneru. Zdroj: vlastní

Jako architektura aplikace byla navržena architektura *Model-View-ViewModel*. Pro zjednodušení vývoje aplikace s touto architekturou je využívána knihovna **CommunityToolkit.Mvvm** [46] verze 8.1.0, která nabízí pomocné nástroje a služby pro vývoj MVVM aplikací a je možné ji využít i při vývoji v jiných frameworcích.

Prezentační vrstva aplikace je implementována ve stylu *Single Page Application*. Je implementováno hlavní okno s navrženým rozložením prvků, na který je navázána třída *ViewModel* pojmenovaná *MainWindowViewModel*. V hlavním okně je umístěn ovládací prvek *ContentControl*, ke kterému je svázána vlastnost *CurrentViewModel* obsažená ve *MainWindowViewModel*. Přepnutí obrazovky je poté prováděno nastavením vlastnosti *CurrentViewModel* na příslušnou instanci *ViewModelu*. Pohled, který má být zobrazen

v závislosti na konkrétním *ViewModelu*, je definován ve zdrojích hlavního okna elementy *DataTemplate*. Dále pohled obsahuje element spouštějící příkaz, který zavolá metodu při načtení obrazovky a zobrazí úvodní pohled (viz. Kód 21).

```
<Window x:Class="LiftApp.Views.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:b="http://schemas.microsoft.com/xaml/behaviors"
        xmlns:local="clr-namespace:LiftApp"
        xmlns:views="clr-namespace:LiftApp.Views"
        xmlns:viewModels="clr-namespace:LiftApp.ViewModels"
        d:DataContext="{d:DesignInstance Type=viewModels:MainWindowViewModel}"
        xmlns:converters="clr-namespace:LiftApp.Converters"
        mc:Ignorable="d"
        Title="Lift App" Height="720" Width="1280"
        Style="{StaticResource ControlBackgoundColor}">

    <b:Interaction.Triggers>
        <b:EventTrigger EventName="Loaded">
            <b:InvokeCommandAction
                Command="{Binding WindowLoadedCommand}"/>
        </b:EventTrigger>
    </b:Interaction.Triggers>

    <Window.Resources>
        <DataTemplate DataType="{x:Type viewModels:BorrowalsViewModel}">
            <views:BorrowalsUserControl />
        </DataTemplate>
        ...
    </Window.Resources>

    <Grid>
        Navigační lišta hlavního okna...

        <!-- Box pro informování o úspěchu operace -->
        <ContentControl Grid.Row="1"
                        Content="{Binding InfoBarViewModel}"
                        Visibility="{Binding InfoBarVisible, Converter={StaticResource
                                 BoolToVisibilityHiddenConverter}}"/>
        <!-- Ovládací prvek pro zobrazení pohledů aplikace -->
        <ContentControl Grid.Row="2" Content="{Binding CurrentViewModel}" Margin="10"/>
        <!-- Stavová lišta -->
        <StatusBar Grid.Row="3">
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="Verze aplikace:" Margin="0 0 10 0"/>
                <TextBlock Text="{Binding AppVersion}"/>
            </StackPanel>
        </StatusBar>
    </Grid>
</Window>
```

Kód 21 - Hlavní pohled MainWindow. Zdroj: vlastní

Třída *ViewModel* hlavní okna obsahuje poté instance *ViewModelů* všech zobrazovaných pohledů a služeb, které jsou při přepínání pohledů používány. Dále obsahuje metody, které obsahují logiku přepínání pohledů a zpracování událostí v jednotlivých pohledech (viz. Kód 22). *ViewModely* ostatních pohledů zobrazovaných v *ContentControl* obsahují pouze vlastnosti svázané na uživatelské vstupy, dále *Commandy*, které jsou přiřazeny na jednotlivé metody v hlavním *ViewModelu* a některé metody validující uživatelské formuláře.

```
public partial class MainWindowViewModel : ObservableObject
{
    private IServiceProvider _serviceProvider;
    // Ostatní služby...

    [ObservableProperty]
    private ObservableObject? _currentViewModel;
    [ObservableProperty]
    private BorrowalsViewModel _borrowalsViewModel;
    // Ostatní ViewModely...

    [ObservableProperty]
    private IAsyncRelayCommand _windowLoadedCommand;
    // Ostatní Commandy...

    [ObservableProperty]
    private bool _infoBarVisible = false;
    [ObservableProperty]
    private string? _appVersion;

    public MainWindowViewModel(IServiceProvider serviceProvider)
    {
        AppVersion = Assembly.GetExecutingAssembly()
            .GetName().Version!.ToString();

        _serviceProvider = serviceProvider;
        // Inicializace ostatních služeb...

        _borrowalsViewModel = _serviceProvider.GetRequiredService<BorrowalsViewModel>();
        // Inicializace ostatních ViewModelů...

        _windowLoadedCommand = new AsyncRelayCommand(MenuNavigateToBorrowalsAsync);
        // Inicializace ostatních Commandů...
    }

    // Metody spouštěné Commandy...
}
```

Kód 22 - Náhled třídy MainWindowViewModel. Zdroj: vlastní

Kód 22 představuje náhled *ViewModelu* hlavního okna *MainWindow.xaml*. Jsou zde používány komponenty knihovny *CommunityToolkit.Mvvm*:

- Třída *ObservableObject* – třída, která zpřístupňuje metody rozhraní *INotifyPropertyChanged*, umožňující promítnutí změn svázaných vlastností *ViewModelu* do pohledu
- Atribut *ObservableProperty* – Generuje pro privátní pole veřejnou vlastnost a tím pádem snižuje množství kódu
- Rozhraní *IAsyncRelayCommand* – Umožňuje svázání *Commandu* s asynchronní operací, která nezablokuje vlákno, na kterém běží uživatelské rozhraní

V konstruktoru třídy jsou vloženy některé závislosti z DI kontejneru, jako vybrané služby a *ViewModely*. Dále jsou zde inicializovány *Commandy* spouštějící metody reagující na uživatelovy interakce.

4.4.2 Vrstva přístupu k datům

Pro přístup k datům v databázi je v rámci projektu *LiftApp.Dal* implementována vrstva přístupu k datům tvořená několika službami. Nejprve byly podle diagramu tříd vytvořeny

modely entit (viz. Kód 23). Modely kromě atributů entit obsahují navigační vlastnosti, pomocí kterých lze přistupovat k souvisejícím entitám. Modely entit, které v diagramu tříd neobsahují přirozený primární klíč, obsahují umělý primární klíč, aby byla zajištěna entitní integrita.

```
public class Borrowal
{
    public int Id { get; set; } = default!;

    public int PriceADay { get; set; } = default!;

    public int TimeIntervalId { get; set; } = default!;

    public TimeInterval TimeInterval { get; set; } = default!;

    public string LiftSerialNumber { get; set; } = default!;

    public Lift Lift { get; set; } = default!;

    public List<Invoice>? Invoices { get; set; };

    public string CustomerIdentifier { get; set; } = default!;

    public Customer Customer { get; set; } = default!;
}
```

Kód 23 - Model entity Výpůjčka. Zdroj: vlastní

Ke každé entitě je nutné dále nastavit pravidla pro jednotlivé atributy a definovat vztahy se souvisejícími entitami. Tato pravidla jsou definována ve třídách ve složce *EntityConfigurations*. Pro jednotlivé entity jsou nastavovány povinné atributy, způsob generování hodnot pro umělé primární klíče a kvantifikátory vztahů se souvisejícími entitami pomocí FluentAPI (viz. Kód 24).

```
internal class BorrowalEntityConfiguration : IEntityTypeConfiguration<Borrowal>
{
    public void Configure(EntityTypeBuilder<Borrowal> builder)
    {
        builder.HasKey(b => b.Id);
        builder.Property(b => b.Id).ValueGeneratedOnAdd();

        builder.Property(b => b.Id).IsRequired();
        builder.Property(b => b.PriceADay).IsRequired();

        builder.HasOne(b => b.TimeInterval)
            .WithOne(ti => ti.Borrowal)
            .HasForeignKey<Borrowal>(b => b.TimeIntervalId)
            .IsRequired();

        builder.HasOne<Lift>(b => b.Lift)
            .WithMany(l => l.Borrowals)
            .HasForeignKey(b => b.LiftSerialNumber)
            .IsRequired();

        builder.HasOne<Customer>(b => b.Customer)
            .WithMany(c => c.Borrowals)
            .HasForeignKey(b => b.CustomerIdentifier)
            .IsRequired();
    }
}
```

Kód 24 - Nastavení pravidel pro atributy modelu Výpůjčka. Zdroj: vlastní

Mapování entit odvozených od abstraktních entit Zákazník a Podnikatel do relační databáze bylo nutné provést jedním ze způsobů, které Entity Framework Core nabízí. Byl

zvolen způsob *Table per concrete type (TPC)*. *TPC* spočívá v uložení entit do samostatných tabulek podle jejich typu, zatímco pro abstraktní typy tabulka vytvořena není. Mapování tímto způsobem je provedeno zavoláním metody *UseTpcMappingStrategy()* v rámci konfigurace bázové entity. [47] Při mapování objektů do relační podoby jsou tedy vytvořeny tabulky pro entity Nepodnikatel, Živnostník a Právnická osoba. Metoda *UseTpcMappingStrategy()* je volána v rámci konfigurace mapování abstraktní entity Zákazník.

Dále je implementována třída *AppDbContext*, která poskytuje přístup k namapovaným tabulkám prostřednictvím vlastností *DbSet* a obsahuje přetěžuje metodu *OnModelCreating*, která volá metody pro konfiguraci entit při mapování (migraci) objektového datového modelu do relačního a vkládá vzorová data určená pro vývoj aplikace (viz. Kód 25). Při registraci třídy *AppDbContext* do DI kontejneru je poté určena knihovna Npgsql jako zprostředkovatel komunikace s databázovým systémem PostgreSQL (viz. Kód 26).

```
public class AppDbContext : DbContext
{
    public DbSet<Borrowal> Borrowals { get; set; } = default!;
    // Ostatní kolekce pro přístup k entitám...

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.ApplyConfiguration(new BorrowalEntityConfiguration());
        // Aplikace konfiguračních pravidel pro ostatní entity...
        // ...

        // Vložení vzorových dat použitých při vývoji aplikace...
        // ...
    }
}
```

Kód 25 - Implementace třídy typu kontext. Zdroj: vlastní

```
services.AddDbContext<AppDbContext>(options =>
options.UseNpgsql(configuration.GetConnectionString("DefaultConnection")));
```

Kód 26 - Použití Npgsql při zaregistrování databázového kontextu. Zdroj: vlastní

Po implementaci služby *AppDbContext*, která poskytuje abstrakci databázových tabulek ve formě kolekcí objektů, byla implementována další vrstva zapouzdřující přístup k těmto kolekcím. Tato vrstva je implementována podle návrhového vzoru Repository popsaném v kapitole 3.6.1.3. Byla implementována generická třída s metodami provádějící CRUD operace, která může být použita pro přístup ke všem typům *DbSetů* obsažených ve třídě *AppDbContext*.

Přístup k instancím třídy *Repository* je zajištěn prostřednictvím služby *MainUnitOfWork*, která je implementována podle návrhového vzoru Unit of Work

zmíněného také v kapitole 3.6.1.3. V konstruktoru této třídy je vložena instance *AppDbContext* a jsou inicializovány instance třídy *Repository*. Při inicializaci těchto instancí je přes konstruktor vložena získaná instance *AppDbContext*, čímž je zajištěno, že operace napříč více instancemi *Repository* je možné složit do jedné databázové transakce. Vykonání transakce je provedeno metodou *SaveChangesAsync* ve třídě *MainUnitOfWork*.

4.4.3 Sekce Výpůjčky

Sekce Výpůjčky je úvodní sekcí zobrazovanou po spuštění aplikace. Přepnutí na pohled sekce Výpůjčky provádí metoda *MenuNavigateToBorrowalsAsync*, která je spuštěna buď příkazem vyvolaným spuštěním aplikace, kliknutím na tlačítko v navigační liště nebo je dále volána po provedení operace odstranění záznamu výpůjčky, po potvrzení formuláře nové výpůjčky nebo po zavření formuláře pro vytvoření nové faktury.

```
private async Task MenuNavigateToBorrowalsAsync()
{
    try
    {
        ShowWaitingView();

        var borrowals = await _mainUnitOfWork.BorrowalRepository.GetAsync(include:
            borrowals => borrowals
            .Include(b => b.TimeInterval)
            .Include(b => b.Customer).ThenInclude(c => c.Address)
            .Include(b => b.Lift)
            .Include(b => b.Invoices),
            orderBy: borrowals => borrowals.OrderByDescending(borrowal =>
                        borrowal.TimeInterval.DateFrom));
        BorrowalsViewModel.Borrowals = new ObservableCollection<Dal.Models.Borrowal>(borrowals);
        CurrentViewModel = BorrowalsViewModel;

        CloseWaitingView();
        _logger.LogInformation("Navigated to Borrowals view");
    }
    catch (Exception ex)
    {
        CloseWaitingView();
        ShowErrorWindow(ex.Message);
        _logger.LogError(ex, "");
    }
}
```

Kód 27 - Metoda pro přepnutí pohledu na sekci Výpůjčky. Zdroj: vlastní

Kód 27 představuje implementaci metody pro přepnutí pohledu na sekci Výpůjčky. Celá logika je vykonávána v rámci bloku *try/catch*, který zajišťuje zpracování výjimky. Pokud by došlo k jakékoli chybě v bloku, informace o chybě je zalogována a je zobrazena chybová hláška uživateli. Nejdříve je prostřednictvím služby *MainUnitOfWork* získán seznam výpůjček s potřebnými souvisejícími entitami, které jsou poté předány vlastnosti třídy *BorrowalsViewModel*, která je svázaná s pohledem sekce Výpůjčky. Nakonec je nastavena vlastnost *CurrentViewModel* na naplněný *BorrowalsViewModel*, čímž je pohled s daty zobrazen uživateli.

Každý záznam výpůjčky v tabulce obsahuje položky s linkem plošiny a zákazníka. Po kliknutí na tyto linky jsou spuštěny příkazy, které zavolají metodu pro zobrazení detailu plošiny/zákazníka, který respektuje rozložení navrženého drátového modelu (Obrázek 13).

Tento pohled obsahuje několik prvků v boxu Operace, a to tlačítko pro vytvoření nové výpůjčky, odstranění výpůjčky, vytvoření nové faktury pro výpůjčku a vstupní pole pro filtrování záznamů. Tlačítko pro vytvoření výpůjčky přepne pohled na odpovídající formulář. Tento formulář v horní části obsahuje vstupní pole pro datum od a datum do, po jejichž vyplnění či změně jednoho z nich je naplněn rozevírací seznam dostupnými plošinami. Po výběru plošiny je možné zadat cenu za den, vybrat zákazníka ze seznamu a v případě, že zákazník ještě není zaevidován, je možné kliknout na tlačítko Nový, po čemž je přepnut pohled na formulář nového zákazníka. Po vyplnění tohoto formuláře je vyplněn seznam pouze novým zákazníkem a je možné kliknout na tlačítko Potvrdit, po čemž jsou všechny vstupy zvalidovány a je spuštěn příkaz volající metodu pro zapsání záznamu do databáze. Validace je řešena na straně *ViewModelů* jednotlivých formulářů, které dědí z třídy *ObservableValidator*. Pravidla jsou specifikována technikou *DataAnnotations* a upozornění zobrazovány pomocí atributu *NotifyDataErrorInfo*, který je společně s třídou *ObservableValidator* součástí *CommunityToolkit.Mvvm*. U vstupů pro datum od a datum do je implementován vlastní validační atribut, který zajišťuje, že datum od je dříve nebo ve stejný den než datum do.

Druhou operací je odstranění výpůjčky. Kliknutí na tlačítko pro spuštění této operace je možné pouze v případě, že je vybrán nějaký záznam výpůjčky v tabulce a daná výpůjčka není již proběhlá. Po kliknutí na toto tlačítko se nejdříve zobrazí dialog pro potvrzení operace a v případě potvrzení dialogu je zavolána metoda pro odstranění záznamu z databáze.

Třetí operací je vytvoření nové faktury pro vybranou výpůjčku z tabulky. Po kliknutí na příslušné tlačítko je přepnut pohled na formulář pro zaevidování nové faktury. Tento formulář může být také zobrazen po úspěšném vložení nové výpůjčky a kliknutím na tlačítko pro vytvoření nové faktury v sekci Faktury. Při zobrazení tohoto formuláře je nejdříve vygenerovaný identifikátor nové faktury ve formátu *Rok-PrvníVolnéCeléNezáporné Číslo*, poté je vyplněn seznam výpůjček pro výběr výpůjčky, pro kterou chce uživatel fakturu zaevidovat. V případě, že je tento formulář otevřen po vytvoření výpůjčky nebo po kliknutí na tlačítko pro vytvoření v sekci výpůjčky, je seznam vyplněn pouze nově vytvořenou nebo vybranou fakturou z tabulky. Následně po vyplnění data vystavení je kontrolováno, zda datum vystavení je nejdříve v den ukončení výpůjčky. Po platném vyplnění tohoto pole je

automaticky vyplněno pole pro datum splatnosti, jehož hodnota je závislá na konfiguraci aplikace (k datu vystavení je přičten počet dní uvedený v konfiguraci). Toto datum může být uživatelem změněno, ale musí být nejdříve v den data vystavení. Po vyplnění data vystavení je také automaticky vyplněno pole pro datum uskutečnění zdanitelného plnění, které je nastaveno na datum ukončení výpůjčky a nemůže být uživatelem změněno. Dalším vstupním polem je zaškrtávací tlačítko pro volbu, zda se jedná o mimořádnou fakturu či nikoliv. Mimořádnou fakturou se rozumí faktura na různé sankce atd. V případě, že faktura není mimořádná, je nastavena pevně daná celková cena do faktury vypočtená z délky výpůjčky ceny za den vypůjčení plošiny, která nemůže být změněna. Pokud je faktura mimořádná, je nutné cenu faktury vyplnit manuálně. Dále je vyplněno pole pro sazbu DPH, která je uvedena v konfiguraci aplikace (pro mimořádnou fakturu je v konfiguraci sazba 0 %) a je vypočtena cena s DPH. Jsou vyplněny pole s údaji bankovního spojení získány taktéž z konfigurace. Následně je možné vyplnit nepovinné pole pro dodatečný popis faktury a potvrdit formulář. Po potvrzení formuláře je v hlavním *ViewModelu* volána metoda pro uložení záznamu do databáze a v případě úspěchu operace je uživateli zobrazena hláška.

Poslední operací je filtrování záznamů podle identifikátoru zákazníka či sériového čísla plošiny. Toto je implementováno tak, že k tabulce sekce Výpůjčky je svázán objekt *ICollectionView*, který umožňuje filtrování kolekce. Po změně textového pole v boxu operací je automaticky filtrován seznam záznamů.

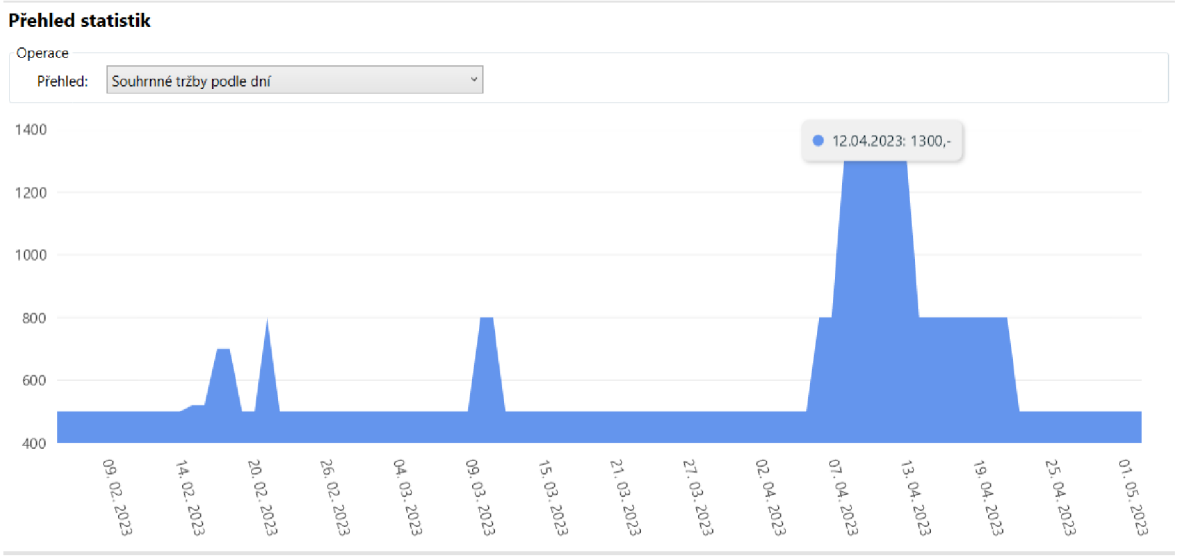
4.4.4 Sekce Statistiky

Zobrazení souhrnných grafů je implementováno v sekci Statistiky. V rámci požadavků na aplikaci byl formulován seznam grafů, které mají být implementovány. Vytvoření těchto grafů je implementováno v projektu *LiftApp.Statictics*, kdy pro požadované grafy jsou implementovány služby se stejnou signaturou a je možné je získat prostřednictvím služby *BorrowalReportServiceFactory*, která metodou *Create* vrátí instanci požadované služby podle výčtového parametru *BorrowalReportType*.

Pro vykreslení grafů byla využita knihovna **LiveChartsCore.SkiaSharpView.WPF** [48] verze 2.0.0-beta.701. Každá služba pro vytvoření grafu implementuje metodu *GetSeriesAndAxes*, která přijímá kolekci záznamů výpůjček a nepovinný argument s rozpětím let, který je využíván pouze u služby pro vygenerování grafu srovnání tržeb v měsících vybraných let. V této metodě je naplněna kolekce *ISeries[]* hodnoty v grafu a kolekce *Axis[]* popisující osu X. Tato dvojce kolekcí je poté touto metodou vrácena.

Pro všechny grafy jsou vytvořeny samostatné dvojice *View* a *ViewModel* a po kliknutí na tlačítko Statistiky v navigační liště je každý *ViewModel* inicializován a naplněn získanou dvojicí *ISeries[]* a *Axis[]*, které jsou svázány na element *CartesianChart*. Všechny implementované grafy lze přibližovat či oddalovat po ose X.

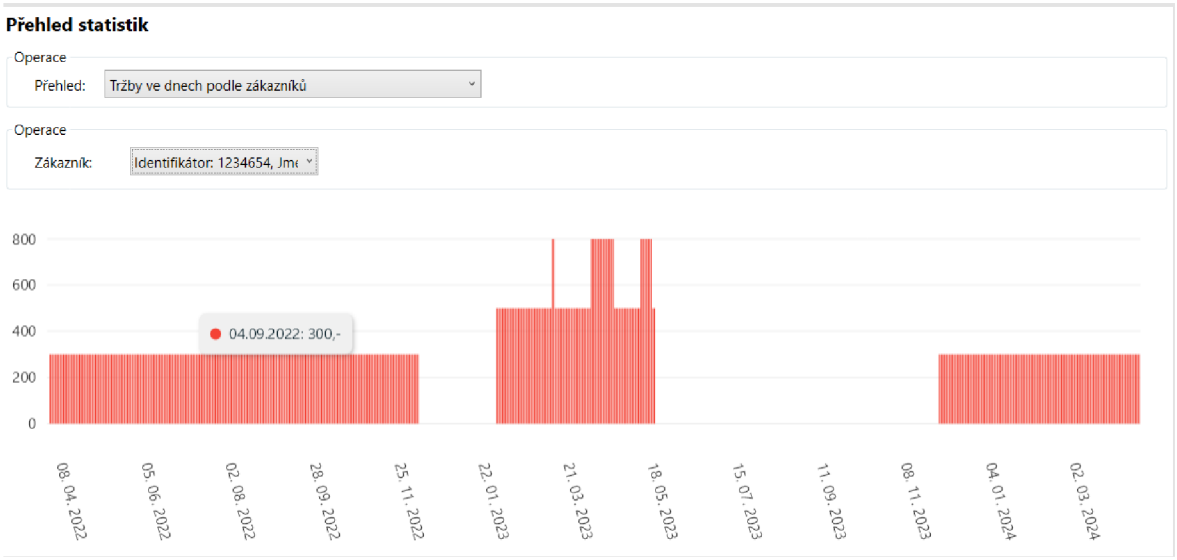
Prvním přehledem je graf souhrnných tržeb podle dní (viz. Obrázek 20). Tento přehled zobrazuje součet tržeb v každém dni vypočítávaném na základě ceny za den



Obrázek 20 - Graf souhrnných tržeb podle dní. Zdroj: vlastní

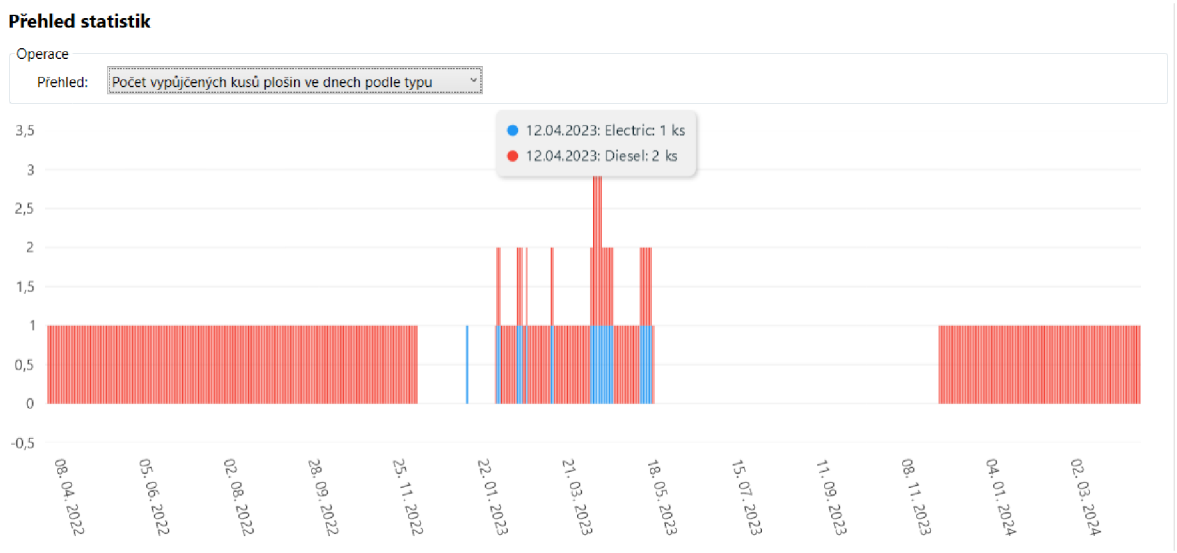
vypůjčení právě vypůjčených plošin. K tomuto přehledu není možná žádná operace a je vykreslován službou *OverallIncomeByDaysReportService*.

Druhým přehledem je graf tržeb ve dnech podle zákazníků. Tento graf je vykreslován službou *IncomeByCustomerReportService* a téměř totožný s předchozím až na skutečnost, že je filtrován podle vybraného zákazníka ze seznamu zákazníků (viz. Obrázek 21). Filtrování podle zákazníka je vykonáváno na samotné kolekci ve *ViewModelu* a graf není službou opakovaně vykreslován.



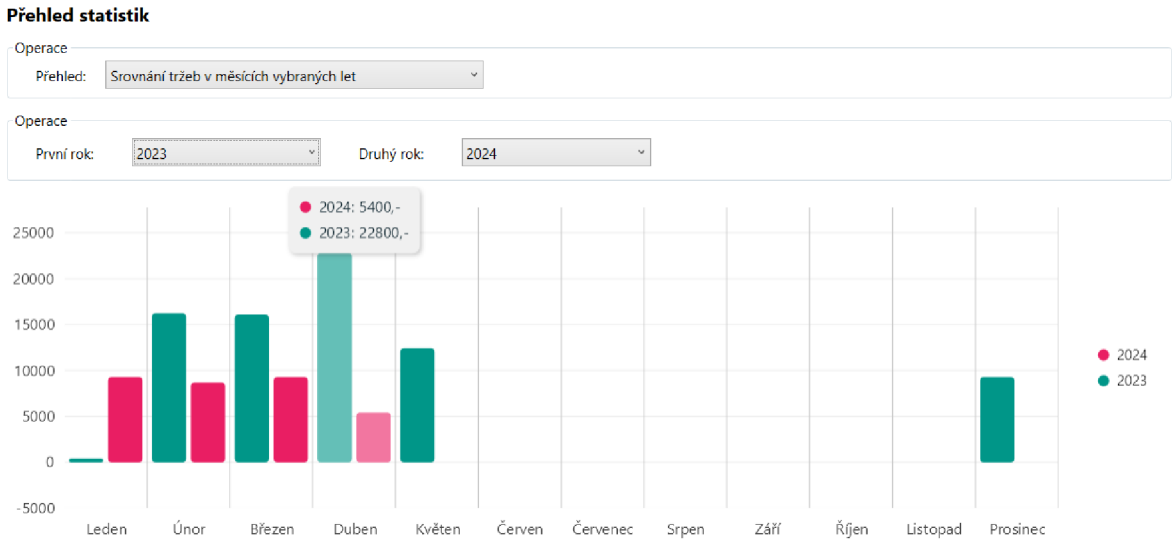
Obrázek 21 - Graf tržeb ve dnech podle zákazníků. Zdroj: vlastní

Třetím přehledem je graf počtu vypůjčených kusů plošin ve dnech podle typu (viz. Obrázek 22). Vypůjčené plošiny jsou rozlišovány podle typu pohonu. Tento graf je vykreslován službou *BorrowedLiftTypeByDaysReportService*.



Obrázek 22 - Graf počtu vypůjčených kusů plošin ve dnech podle typu. Zdroj: vlastní

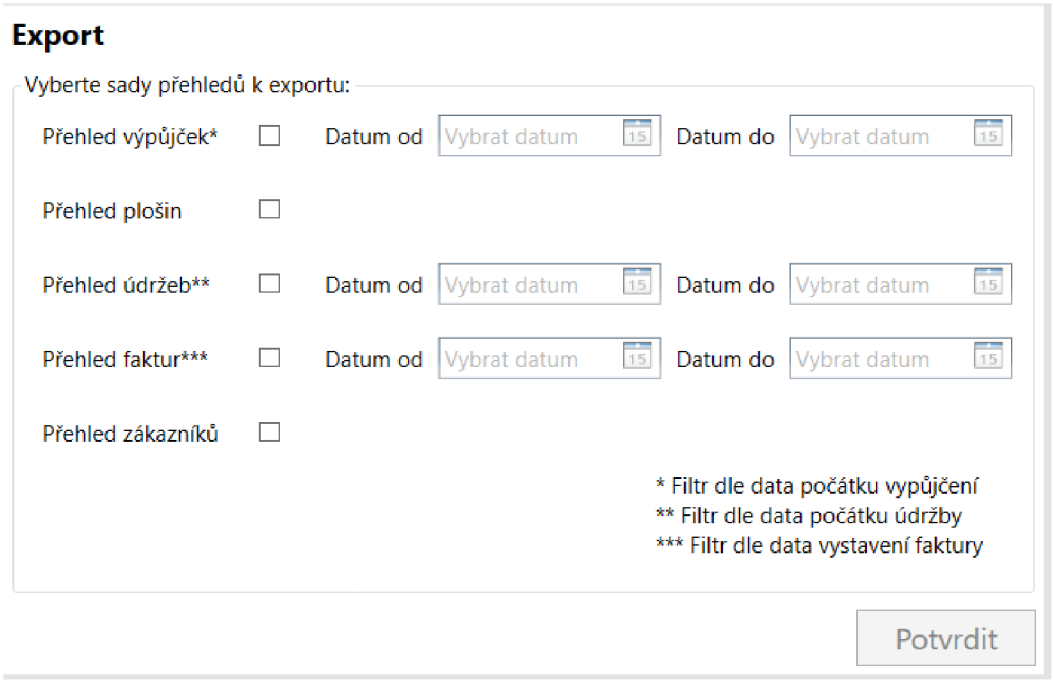
Posledním přehledem ve výběru je graf srovnání tržeb v měsících vybraných let (viz. Obrázek 23). O vykreslení tohoto grafu se zodpovědná služba *IncomeYearComparisonReportService* a tento graf zobrazuje ke srovnání tržeb v jednotlivých měsících let vybraných z rozevíracích seznamů nad tímto grafem. Při změně výběru let pro srovnání jsou roky validovány pro případ, že by uživatel v prvním poli vybral rok, který je dříve než druhý rok nebo je totožný s vybraným druhým rokem. Po výběru validní dvojice let je zavolána metoda služby pro znovuvykreslení grafu.



Obrázek 23 - Graf srovnání tržeb v měsících vybraných let. Zdroj: vlastní

4.4.5 Sekce Export

V sekci Export je možné vyexportovat přehledy dat do souboru tabulkového procesoru Microsoft Excel. Tato funkce je implementována v projektu *LiftApp.Export*. Byl implementován pohled, který nabízí volbu přehledů, které mohou být exportovány (viz. Obrázek 24). Pohled obsahuje pro každý přehled zaškrtávací pole a u vybraných také vstupní pole pro výběr časového rozpětí pro omezení počtu záznamů. Po kliknutí tlačítka Potvrdit je zobrazen systémový dialog pro výběr umístění a poté je do zvoleného umístění uložen vygenerovaný soubor, ve kterém jsou jednotlivé zvolené přehledy v samostatných listech.



Obrázek 24 - Implementovaný pohled sekce Export. Zdroj: vlastní

Pro práci s aplikací Microsoft Excel je využita knihovna **Microsoft.Office.Interop.Excel** verze 1.9. Export probíhá skrytým otevřením aplikace Excel nainstalované na pracovní stanici a načtením záznamů do předpřipravených šablon. Načtení záznamů do souboru Microsoft Excel s přehledy je implementováno ve službě *OverviewExportService*.

Třída *OverviewExportService* obsahuje dvě metody. První z nich, *SaveAs*, slouží k uložení souboru s přehledy do umístění definovaném přes parametr *fileName*. Druhá, *ExportAsync*, obsahuje logiku k vypsání záznamů do jednotlivých listů šablony (viz. Kód 28). Tato metoda přijímá parametr *directoryPath* určující cestu k uložení souboru a parametr typu *IEnumerable<OverviewConfiguration>*, který představuje kolekci objektů s informacemi, jaké všechny přehledy mají být exportovány a jestli mají být časově omezeny. V této metodě pak probíhá cyklus, který projde všechny objekty v parametru *IEnumerable<OverviewConfiguration>*. V každé iteraci je poté vykreslen požadovaný přehled pomocí služby *IOverviewSheetRenderer*, která je implementována pro každý jednotlivý přehled záznamů (viz. Obrázek 25). Služba *IOverviewSheetRenderer* listu určeného podle názvu v konfiguraci aplikace vypíše atributy záznamů do sloupců, které jsou uchovávány taktéž v konfiguraci. Uchováním názvu listů a pozice sloupců pro atributy v konfiguraci je docílena možná pozdější modifikace šablon, aniž by bylo nutné upravovat implementaci samotných služeb *IOverviewSheetRenderer*. Následně jsou ze souboru

odstraněny nevyužité listy a soubor je uložen dříve zmíněnou metodou *SaveAs*. Nový soubor je pojmenován ve formátu *Export – DartumAČasExportu*.

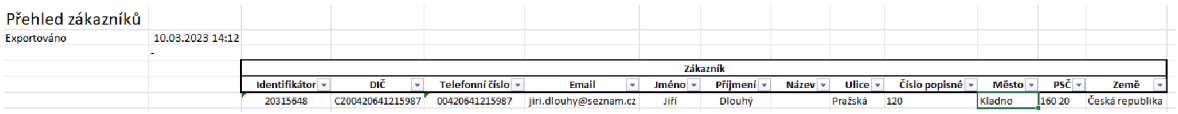
```
public async Task ExportAsync(string directoryPath, IEnumerable<OverviewConfiguration>
            overviewConfigurations)
{
    try
    {
        InitializeExcelApp();
        InitializeWorkbook(Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location)!,
            _overviewExportOptions.Value.ExcelTemplatePath));

        // Render contents
        foreach (var overviewConfiguration in overviewConfigurations)
        {
            var renderer = _serviceProvider.GetRequiredService<OverviewSheetRendererFactory>()
                .Create(overviewConfiguration.OverviewType);
            renderer.InitializeWorksheet(_workbook,
                        _overviewExportOptions.Value.SheetNames[overviewConfiguration.OverviewType]);
            await renderer.RenderContentAsync(overviewConfiguration.DateRange);
        }

        foreach(var sheetName in _overviewExportOptions.Value.SheetNames)
        {
            if (!overviewConfigurations.Select(oc => oc.OverviewType).Contains(sheetName.Key))
            {
                _workbook!.Worksheets[sheetName.Value].Delete();
            }
        }

        // Save
        var fileName = Path.Combine(directoryPath, $"Export - {DateTime.Now:dd-MM-yyyy HH-mm}.xlsx");
        SaveAs(fileName);
    }
    finally
    {
        CloseWorkbook();
        CloseExcelApp();
    }
}
```

Kód 28 - Metoda ExportAsync třídy OverviewExportService. Zdroj: vlastní



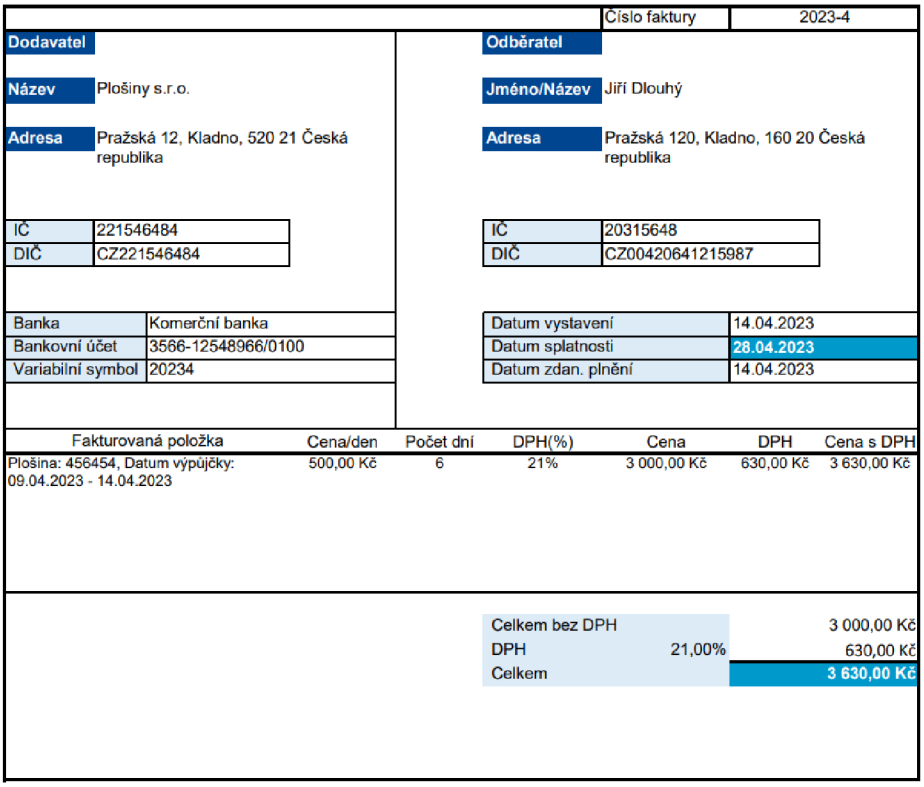
Přehled zákazníků												
Exportováno	10.03.2023 14:12											
		Zákazník										
	Identifikátor	DIČ	Telefonní číslo	Email	Jméno	Příjmení	Název	Ulice	Číslo popisné	Město	PSČ	Země
	20315648	CZ00420641215987	00420641215987	jiri.dlouhy@seznam.cz	Jiří	Dlouhý		Pražská	120	Kladno	160 20	Česká republika

Obrázek 25 - Exportovaný list s přehledem plošin. Zdroj: vlastní

4.4.6 Ostatní sekce

Zbylé sekce Plošiny, Zákazníci, Faktury a Údržby jsou určeny pro náhled a provádění operací se záznamy z databáze. Tyto sekce jsou vzhledem i funkčností podobné sekci Výpůjčky, která byla dříve podrobněji popsaná, tudíž není nutné tyto sekce znovu detailněji popisovat. Každá sekce obsahuje box s operacemi a tabulku se zobrazovanými záznamy. Sekce Plošiny umožnuje přidání nové plošiny do databáze, dále editaci stávajícího záznamu, označení plošiny jako vyřazené či označení zpět jako dostupné a filtrování záznamů podle sériového čísla. V sekci Zákazníci je možné pouze nahlížet mezi evidované zákazníky a filtrování zobrazovaných záznamů podle jejich identifikátoru, telefonního čísla či emailové

adresy. V sekci Faktury je možné nahlížet mezi evidované faktury, přidat novou fakturu, potvrdit její zaplacení, odstranit (pouze pokud není označena jako již zaplacená), filtrovat zobrazované záznamy podle identifikátoru faktury a export faktury do souboru PDF (viz. Obrázek 26), k čemuž je implementována služba *InvoiceExportService* v projektu *LiftApp.Export*. V sekci Údržby je pak možné přidat nový záznam o údržbě, editace existujícího záznamu o údržbě, odstranění záznamu a filtrování zobrazených záznamů podle sériového čísla plošiny.



				Číslo faktury	2023-4	
Dodavatel			**Odběratel**			
Název	Plošiny s.r.o.		**Jméno/Název**	Jiří Dlouhý		
Adresa	Pražská 12, Kladno, 520 21 Česká republika		**Adresa**	Pražská 120, Kladno, 160 20 Česká republika		
IČ	221546484		IČ	20315648		
DIČ	CZ221546484		DIČ	CZ00420641215987		
Banka	Komerční banka		Datum vystavení	14.04.2023		
Bankovní účet	3566-12548966/0100		Datum splatnosti	28.04.2023		
Variabilní symbol	20234		Datum zdan. plnění	14.04.2023		

Fakturovaná položka	Cena/den	Počet dní	DPH(%)	Cena	DPH	Cena s DPH
Plošina: 456454, Datum výpůjčky: 09.04.2023 - 14.04.2023	500,00 Kč	6	21%	3 000,00 Kč	630,00 Kč	3 630,00 Kč

Celkem bez DPH		3 000,00 Kč
DPH	21,00%	630,00 Kč
Celkem		3 630,00 Kč

Obrázek 26 - Faktura exportovaná do souboru PDF. Zdroj: vlastní

4.4.7 Konfigurace

Údaje jako připojovací řetězec k databázi, bankovní spojení společnosti používané při evidenci faktur, názvy listů šablony pro export přehledů atd. jsou umístěny mimo zdrojový kód aplikace v samostatném souboru *appsettings.json*. Výhodou této skutečnosti je možnost modifikovat tyto údaje bez nutnosti zásahu do zdrojového kódu aplikace. Během používání aplikace se může změnit např. číslo účtu společnosti a tento údaj je pak možné jednoduše změnit v konfiguračním souboru, který je umístěn v adresáři se samotnou aplikací. Konfigurace aplikace je inicializována při jejím spuštění a napříč aplikací je k ní přistupováno přes rozhraní *IOptions* zmíněném v kapitole 3.5.4.

4.4.8 Logování

Zaznamenávání uživatelových činností je důležité z hlediska pozdějšího ladění chyb, které jsou odhaleny až při jejím používání samotným uživatelem. Bez automatizovaného zaznamenávání činností ve standardizovaném formátu by následné řešení chyb bylo velmi obtížné, jelikož vývojář by mohl získat od uživatele pouze omezené informace o jeho činnosti, načež jaké další jeho činnosti chybě předcházely.

Z tohoto důvodu jsou prostřednictvím služby *ILogger* (popsané v kapitole 3.5.3) zaznamenávány prováděné činnosti napříč celou aplikací. V případě vyhození výjimky je se zprávou výjimky zaznamenán také zásobník volání, díky kterému je možné snadněji ladit vyskytlé chyby.

Pro uchování logů v souborech mimo aplikaci je využita knihovna **NLog.Web.AspNetCore** [49] verze 5.2.2. Formát názvu souboru a formát logů, které mají být uloženy, jsou nakonfigurovány v souboru *nlog.config*. Při spuštění aplikace je poté pomocí tohoto souboru inicializován poskytovatel logování *NLog* metodou *AddNLog*. Při běhu aplikace jsou poté logy automaticky zapisovány do textového souboru umístěného v adresáři aplikace (viz. Kód 29). Během používání jsou tvořeny nové soubory v každém dni logování a každý soubor má název označující den zaznamenání změn.

```
2023-03-06 14:32:30.2948||INFO|LiftApp.Export.Services.OverviewExportService|Opened Excel workbook
2023-03-06 14:32:30.7099||INFO|LiftApp.Export.Services.OverviewExportService|Saved file as:
C:\Users\ladam\Desktop\Export - 06-03-2023 14-32.xlsx
```

Kód 29 - Náhled logovaných zpráv do textového souboru. Zdroj: vlastní

4.5 Testování komponent

Při vývoji komponent aplikace je vhodné implementovat také testovací metody, které ověřují jejich funkčnost. Pokrytí zdrojového kódu testy umožňuje ověření správné funkčnost všech komponent, což snižuje možnost výskytu chyby po úpravě komponent aplikace.

Jedním z přístupů k vývoji software je vývoj řízený chováním (BDD) zmíněný v kapitole 3.10. Pro implementaci testů byla využita knihovna **LightBDD.XUnit2** [50] verze 3.6.1. Z názvu této komponenty lze vyčíst, že jde o část knihovny, která je integrována s nativní knihovnou **XUnit**.

Přístupem BDD byla v aplikaci vyvinuta komponenta *AppDbContext*, která je součástí vrstvy pro přístup k datům. Byly implementovány dvě testovací metody, které ověřují funkčnost komunikace s databází. Tyto metody implementují volání metod, které představují kroky scénáře (viz. Kód 30).

```
[Scenario]
[ClassData(typeof(TestBorrowalClassData))]
public async Task Test_Insert_Borrowal(Office office, Borrowal borrowal)
{
    await Runner.WithContext<DbContextContext>()
        .AddAsyncSteps(
        _ => _.Given_Reset_Db(),
        _ => _.And_Add_Office_With_Address_And_Single_Lift(office),
        _ => _.When_Borrowal_Is_Added(borrowal)
        ).AddSteps(
        _ => _.Then_Office_Has_Lift_That_Has_Borrowal(borrowal)
        )
        .RunAsync();
}

[Scenario]
[ClassData(typeof(TestInvalidEntitiesClassData))]
public async Task Test_That_It_Is_Unable_To_Add_Some_Entities_Without_Association(Maintenance maintenance,
Address address, Invoice invoice)
{
    await Runner.WithContext<DbContextContext>()
        .AddAsyncSteps(
        _ => _.Given_Reset_Db(),
        _ => _.Then_Adding_Maintenance_Without_Association_Throws_Exception(maintenance),
        _ => _.Then_Adding_Address_Without_Association_Throws_Exception(address),
        _ => _.Then_Adding_Invoice_Without_Association_Throws_Exception(invoice)
        )
        .RunAsync();
}
```

Kód 30 - Testovací metody pro komponentu AppDbContext. Zdroj: vlastní

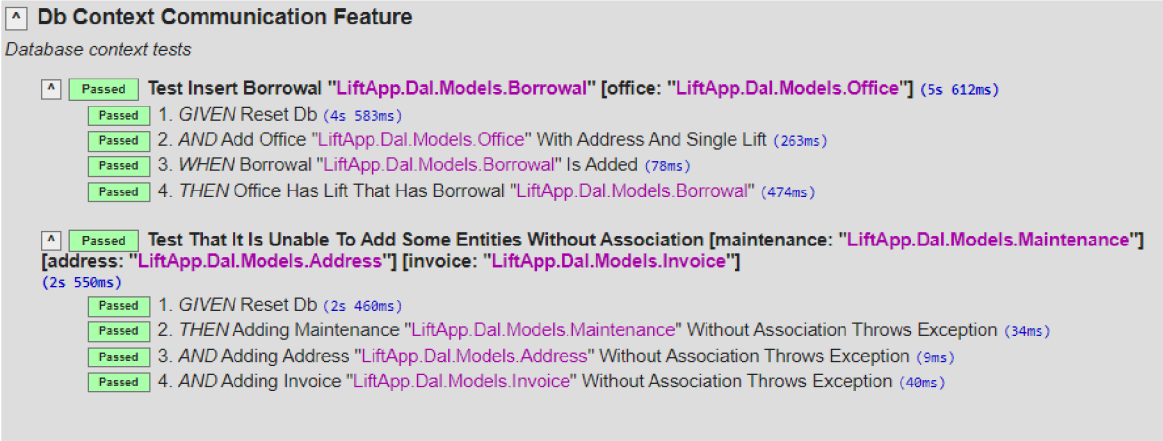
První metoda testuje scénář přidání nového záznamu o výpůjčce:

1. Je resetována databáze
2. A je přidána pobočka s adresou a určitou plošinou
3. Když je přidán záznam o výpůjčce pro dříve přidanou plošinu
4. Tak přidaná pobočka má plošinu a tato plošina má výpůjčku.

Druhá metoda testuje, zda vložení záznamů entit, u kterých je vyžadován vztah s jinou entitou, vyhodí výjimku. Metoda testuje vyhození výjimky při vložení entit Údržba, Adresa a Faktura.

Kroky obou metod jsou implementovány v jedné třídě typu testovací kontext (pojmenovaná *DbContextContext*). V konstruktoru této třídy je inicializován DI kontejner a je do něj zaregistrována pouze služba *AppDbContext*. Je tak zajištěna izolovanost prostředí pro testování. Při registraci služby *AppDbContext* je vložen připojovací řetězec pro testovací databázi, takže testování není ovlivněno záznamy uloženými v hlavní databázi.

Knihovna LightBDD umožňuje generování reportů o testování při každém spuštění testů. V adresáři aplikace generuje složku *Reports*, kam umisťuje soubor HTML s přehledem průběhu a úspěšnosti testů (viz. Obrázek 27).



Obrázek 27 - Vygenerovaný report o průběhu testů. Zdroj: vlastní

5 Výsledky a diskuse

5.1 Zhodnocení softwarového řešení

Implementovaná aplikace splňuje všechny formulované funkční i nefunkční požadavky. Softwarové řešení umožňuje spravovat evidenci záznamů o výpůjčkách plošin, jejich údržbách a evidenci faktur vystavovaných zákazníkům. Záznamy mohou být přidávány pomocí přehledných formulářů v konzistentní formě docílené vhodnou validací a vhodně navrženým datovým modelem. V aplikaci je možné provádět pouze operace, které konzistenci dat neohrozí. Jsou implementovány vizuální přehledy dat v čase, které mohou být využity vedením společnosti jako podpora při rozhodování. Zároveň je implementována možnost exportu datových sad do tabulkového procesoru, aby bylo možné provádět se záznamy operace, které v aplikaci implementovány nejsou.

Data jsou uložena mimo klientskou aplikaci, což snižuje riziko jejich ztráty a zaručuje možnost přístupu k nim z více pracovních stanic. Díky použití technologie Entity Framework Core je zaručena nezávislost aplikace na zvoleném databázovém systému – v případě požadavku může být nasazen zprostředkovatel komunikace pro jiný databázový systém, aniž by bylo potřeba zásadně měnit zdrojový kód aplikace.

Uživatelské rozhraní aplikace odpovídá navrženému rozložení pomocí drátových modelů. Byla použita architektura MVVM, která odděluje business logiku od prezentační vrstvy a je možné tyto dvě části aplikace vyvíjet nezávisle na sobě. Zároveň je takto business logika přenositelná do jiných frameworků pro vývoj aplikací, a to nejen desktopových.

5.2 Uživatelské testování

Po implementaci aplikace byla aplikace nasazena do zkušebního provozu a bylo provedeno uživatelské testování. Testování se účastnily 3 osoby, kterým byly předány testovací scénáře pro všechny případy užití. Při jejich vykonávání byla sledována interakce osob s aplikací a byly sbírány možné připomínky.

Uživatelské testování proběhlo vesměs bez problémů. Nebyly identifikovány žádné větší výhrady k funkčnosti aplikace, což lze přičíst dostatečně podrobné analýze případů užití, během které byly případy užití konzultovány s budoucími uživateli. Testovací subjekty kladně hodnotily přehledné a konzistentní rozložení uživatelského rozhraní napříč sekcemi, přehlednost formulářů a kontrolu správnosti vyplněných vstupů ve formulářích doplněnou informací, jaká hodnota je pro vstup korektní. Jediná výhrada se týkala přístupnosti

uživatelského rozhraní, a to malá velikost všech textů v aplikaci, která je ve WPF v základu nastavena na velikost 12. Řešením bylo nastavení velikosti všech textů na velikost 15 a následná úprava ovládacích prvků pro dosažení jejich správného zobrazení ve spojení s větší velikostí textu. Tato úprava rozhraní aplikace byla testovacím subjektem, který tento problém identifikoval, ohodnocena jako dostatečná.

5.3 Problémy při vývoji

Při implementaci aplikace bylo nutné řešit několik problémů. Prvním z nich byl problém s použitím knihovny CommunityToolkit.Mvvm ve vývojovém prostředí Visual Studio. Během sestavování aplikace byly opakovaně oznamované chyby o chybějících vlastnostech ViewModelů generované pomocí atributu *[ObservableProperty]*. Po restartu prostředí Visual Studio a znovusestavení projektu tyto chyby vždy zmizely. Během vývoje aplikace byl tento problém ignorován a před dalším potenciálním vývojem by bylo vhodné prozkoumat možnosti jeho řešení.

Druhým problémem bylo použití knihovny Microsoft.Office.Interop.Excel pro exportování přehledů dat do tabulkového procesoru Microsoft Excel. Problém spočíval samotném použití této knihovny, neboť tato knihovna omezuje kompatibilitu aplikace pouze pro OS Windows a kompilátor .NET je určen pro multiplatformní vývoj. Při sestavování aplikace tak kompilátor hlásil chybu s touto knihovnou a nebylo možné pomocí příkazu *dotnet build* tento projekt sestavit. Při sestavování pomocí vývojového prostředí Visual Studio se žádné problémy neobjevily. Tato skutečnost ale omezuje potenciální přenositelnost aplikační logiky na jiné platformy. Řešením by mohlo být např. odstranění závislosti aplikace na této knihovně a exportovat přehledy dat ve formátu CSV. Aby bylo možné v budoucnu upravovat datový model a migrovat změny do relačního databázového systému, byla aplikační logika pro migraci datového modelu izolována od zbytku logiky. Byl vytvořen druhý projekt, kam byla zkopírována část aplikace pro migrování datového modelu. Po provedení změn v tomto projektu je ale nutné tyto změny přenést také do hlavního projektu, aby byla zachována správná funkčnost aplikace.

5.4 Další možnosti vývoje

Vytvořená aplikace splňuje všechny požadavky formulované v rámci analýzy, díky navržené architektuře je ale možné ji jednoduše v budoucnu rozvíjet. Jedním ze směrů rozvoje by mohl být přenos aplikační logiky do jednoho z multiplatformních frameworků

(např. .NET MAUI, Avalonia atd…), kdy by bylo nutné znovu implementovat prezentační vrstvu pro každou platformu a uživatelé by pak mohli používat aplikaci nejen ze stolního počítače, ale také z mobilního telefonu či webu.

Dalším směrem by mohlo být rozšíření funkcionality aplikace pro podporu dalších podnikových procesů, jako např. detailnější správa údržeb plošin či správa zaměstnanců společnosti.

V případě volby cloudového řešení pro uchování dat (např. Microsoft Azure) by mohla být provedena integrace s aplikací Microsoft PowerBI, ve které by mohly být vytvořeny multidimenzionální pohledy na data prostřednictvím technologie OLAP, což by mohlo ještě více zefektivnit podporu v rozhodování.

Stávající aplikace obsahuje jednotkové testy pokrývající pouze některé operace s databázovým kontextem zprostředkovávajícím přístup k datům uloženým v relačním databázovém systému. Pro udržení funkčnosti aplikace při budoucím vývoji by bylo vhodné pokrýt testy všechny implementované komponenty, aby bylo možné při každém rozšíření snadno ověřit jejich správnou funkčnost.

V rámci optimalizace výkonu by bylo vhodné revidovat výkonnost dotazů na databázový systém. V rámci této aplikace je pro všechny dotazy využívána metoda Eager Loading, která nemusí být vždy nejvýkonnější. Bylo by vhodné tedy porovnat rychlost jednotlivých dotazů i s použitím jiných metod.

6 Závěr

Cílem této práce bylo vytvořit za pomoci metod softwarového inženýrství a technologií pro vývoj aplikací softwarové řešení pro společnost vypůjčující pracovní plošiny, které nahradí stávající systém nezávislých tabulek v tabulkovém procesoru.

První část cíle bylo analyzovat stávající řešení pomocí metod softwarového inženýrství. Byly formulovány požadavky na aplikaci a případy užití. Druhou částí cíle bylo navrhnout uživatelsky přívětivé uživatelské rozhraní ve kterém budou mít uživatelé možnost spravovat evidenci záznamů o výpůjčkách plošin. Třetí částí cíle bylo samotnou aplikaci implementovat ve zvoleném frameworku určeném pro vývoj desktopových aplikací. Cíl práce dále zahrnoval nasazení aplikace do testovacího provozu a provedení uživatelského testování, při kterém byl odhalen problém s uživatelským rozhraním, který byl následně opraven. Poslední částí cíle bylo navržení možných směrů v rozšíření aplikace.

Stanovené cíle byly splněny. Výsledkem práce je plně funkční informační systém, který je připraven k nasazení do ostrého provozu ve společnosti. Informační systém umožňuje provádění operací, které nevedou ke ztrátě dat, dále optimalizuje vybrané podnikové procesy a snižuje zatížení zaměstnanců zapojených v procesu, což může vést ke snížení provozních nákladů a zvýšení zisku společnosti. Dále poskytuje přehledy dat, které mohou být používány jako podpora při rozhodování vedení společnosti.

7 Seznam použitých zdrojů

[1] The Forgotten History of OOP. In: *Medium* [online]. [cit. 2023-01-09]. Dostupné z: https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f

[2] *Beginning SOLID Principles and Design Patterns for ASP.NET Developers: Patterns of Enterprise Application Architecture: Repository, Unit of Work, Lazy Load, and Service Layer*. Berkeley, CA.: Apress, 2016. ISBN 978-1-4842-1848-8.

[3] The SOLID Principles of Object-Oriented Programming Explained in Plain English. In: *FreeCodeCamp* [online]. [cit. 2023-01-09]. Dostupné z: https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

[4] VIRIUS, Miroslav. *Programování v C#: od základů k profesionálnímu použití*. Praha: Grada Publishing, 2021. Knihovna programátora (Grada). ISBN 978-80-271-1216-6.

[5] PRICE, Mark J. *C# 10 and .NET 6 - modern cross-platform development: build apps, websites, and services with ASP.NET Core 6, Blazor, and EF Core 6 using Visual Studio 2022 and Visual Studio Code*. Sixth edition. Birmingham: Packt, 2021. ISBN 978-1-80107-736-1.

[6] JOHN, Townsend. The History of .NET. In: *Omnitech* [online]. [cit. 2023-01-09]. Dostupné z: https://www.omnitech-inc.com/blog/the-history-of-net/

[7] Overview of .NET Framework. In: *Microsoft Learn* [online]. [cit. 2023-01-09]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview

[8] Desktop Guide (Windows Forms .NET). In: *Microsoft Learn* [online]. [cit. 2023-01-10]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-6.0

[9] What's a Universal Windows Platform (UWP) app?. In: *Microsoft Learn* [online]. [cit. 2023-01-10]. Dostupné z: https://learn.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide

[10] What is .NET MAUI?. In: *Microsoft Learn* [online]. [cit. 2023-01-10]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-7.0

[11] About the Uno Platform. In: *Uno Platform* [online]. [cit. 2023-01-11]. Dostupné z: https://platform.uno/docs/articles/what-is-uno.html

[12] Supported platforms. In: *Uno Platform* [online]. [cit. 2023-01-11]. Dostupné z: https://platform.uno/docs/articles/getting-started/requirements.html

[13] How Uno Platform works. In: *Uno Platform* [online]. [cit. 2023-01-11]. Dostupné z: https://platform.uno/docs/articles/how-uno-works.html

[14] BISSON, Simon. Getting started with Avalonia UI. In: *InfoWorld* [online]. [cit. 2023-01-11]. Dostupné z: https://www.infoworld.com/article/3650477/getting-started-with-avalonia-ui.html

[15] JAMES, Mike. Avalonia platform support - why it's simple. In: *Dev* [online]. [cit. 2023-01-11]. Dostupné z: https://dev.to/avalonia/avalonia-platform-support-why-its-simple-cjd

[16] Desktop Guide (WPF .NET). In: *Microsoft Learn* [online]. [cit. 2023-01-11]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-6.0#see-also

[17] XAML overview (WPF .NET). In: *Microsoft Learn* [online]. [cit. 2023-01-13]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/?view=netdesktop-6.0

[18] Application Management Overview. In: *Microsoft Learn* [online]. [cit. 2023-01-13]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/app-development/application-management-overview?view=netframeworkdesktop-4.8

[19] SERASINGHE, Sahan. Understanding .NET Generic Host Model. In: *Sahansera* [online]. [cit. 2023-01-13]. Dostupné z: https://sahansera.dev/dotnet-core-generic-host/

[20] .NET Generic Host in ASP.NET Core. In: *Microsoft Learn* [online]. [cit. 2023-01-13]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-7.0&viewFallbackFrom=aspnetcore-3.0

[21] KEMPÉ, Laurent. *WPF and .NET Generic Host with .NET Core 3.0* [online]. In: . [cit. 2023-01-13]. Dostupné z: https://laurentkempe.com/2019/09/03/WPF-and-dotnet-Generic-Host-with-dotnet-Core-3-0/

[22] Architectural principles. In: *Microsoft Learn* [online]. [cit. 2023-01-16]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles

[23] Dependency Injection Design Pattern in C#. In: *Dotnettutorials* [online]. [cit. 2023-01-16]. Dostupné z: https://dotnettutorials.net/lesson/dependency-injection-design-pattern-csharp/

[24] Dependency injection in .NET. In: *Microsoft Learn* [online]. [cit. 2023-01-16]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

[25] Logging in .NET. In: *Microsoft Learn* [online]. [cit. 2023-01-16]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/extensions/logging

[26] .NET Logging Basics. In: *Loggly* [online]. [cit. 2023-01-16]. Dostupné z: https://www.loggly.com/ultimate-guide/net-logging-basics/

[27] Configuration in .NET. In: *Microsoft Learn* [online]. [cit. 2023-01-16]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/extensions/configuration

[28] Options pattern in ASP.NET Core. In: *Microsoft Learn* [online]. [cit. 2023-01-16]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-7.0

[29] ABBA, Ihechikara Vincent. What is an ORM – The Meaning of Object Relational Mapping Database Tools. In: *FreeCodeCamp* [online]. [cit. 2023-01-17]. Dostupné z: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/

[30] SMITH, John P. *Entity Framework Core In Action*. Second edition. Shelter Island, NY: Manning, 2021. ISBN 978-1-61729-836-3.

[31] Database Providers. In: *Microsoft Learn* [online]. [cit. 2023-01-17]. Dostupné z: https://learn.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli

[32] Creating and Configuring a Model. In: *Microsoft Learn* [online]. [cit. 2023-01-17]. Dostupné z: https://learn.microsoft.com/en-us/ef/core/modeling/

[33] Entity Framework Core. In: *Microsoft Learn* [online]. [cit. 2023-01-17]. Dostupné z: https://learn.microsoft.com/en-us/ef/core/

[34] Fluent API - Configuring and Mapping Properties and Types. In: *Microsoft Learn* [online]. [cit. 2023-01-20]. Dostupné z: https://learn.microsoft.com/en-us/ef/ef6/modeling/code-first/fluent/types-and-properties

[35] Code First Data Annotations. In: *Microsoft Learn* [online]. [cit. 2023-01-20]. Dostupné z: https://learn.microsoft.com/cs-cz/ef/ef6/modeling/code-first/data-annotations

[36] SMITH, Josh. Patterns - WPF Apps With The Model-View-ViewModel Design Pattern. In: *Microsoft Learn* [online]. [cit. 2023-01-20]. Dostupné z: https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern

[37] LITVINAVICIUS, Taurius. *Exploring Windows Presentation Foundation: With Practical Applications in .NET 5*. 1. vydání. Berkeley, CA: Apress, 2020. ISBN 978-1-4842-6637-3.

[38] BOST, Kevin. Master the Basics of MVVM for Building WPF Applications. In: *IntelliTect* [online]. [cit. 2023-01-23]. Dostupné z: https://intellitect.com/blog/getting-started-model-view-viewmodel-mvvm-pattern-using-windows-presentation-framework-wpf/

[39] KANISOVÁ, Hana a Miroslav MÜLLER. *UML srozumitelně*. Brno: Computer Press, 2004. ISBN 80-251-0231-9.

[40] Types of UML Diagrams. In: *Lucidchart* [online]. [cit. 2023-01-23]. Dostupné z: https://www.lucidchart.com/blog/types-of-UML-diagrams

[41] CSER, Tamas. Behavior-driven development. In: *Functionize* [online]. [cit. 2023-03-13]. Dostupné z: https://www.functionize.com/automated-testing/behavior-driven-development

[42] Draw.io. In: *Diagrams.net* [online]. [cit. 2023-03-13]. Dostupné z: https://app.diagrams.net

[43] PostgreSQL: The World's Most Advanced Open Source Relational Database. In: *PostgreSQL* [online]. [cit. 2023-03-03]. Dostupné z: https://www.postgresql.org

[44] Npgsql - .NET Access to PostgreSQL. In: *Npgsql* [online]. [cit. 2023-03-10]. Dostupné z: https://www.npgsql.org

[45] Figma - the collaborative interface design tool. In: *Figma* [online]. [cit. 2023-03-10]. Dostupné z: https://www.figma.com

[46] Community Toolkit. In: *Github* [online]. [cit. 2023-03-10]. Dostupné z: https://github.com/CommunityToolkit

[47] Inheritance. In: *Microsoft Learn* [online]. [cit. 2023-03-05]. Dostupné z: https://learn.microsoft.com/cs-cz/ef/core/modeling/inheritance

[48] LiveCharts2: Beautiful, animated, automatically updated, cross-platform, object oriented and easy to use. In: *LiveCharts* [online]. [cit. 2023-03-10]. Dostupné z: https://lvcharts.com

[49] Nlog.Web: NLog integration for ASP.NET & ASP.NET Core 2-6. In: *Github* [online]. [cit. 2023-03-10]. Dostupné z: https://github.com/NLog/NLog.Web

[50] LightBDD: BDD framework allowing to create easy to read and maintain tests. In: *Github* [online]. [cit. 2023-03-10]. Dostupné z: https://github.com/LightBDD/LightBDD

[51] How is C# Compiled?. In: *Manning Free Content Center* [online]. [cit. 2023-01-09]. Dostupné z: https://freecontent.manning.com/how-is-c-compiled/

8 Přílohy

Přílohou této práce je nosič CD, který obsahuje dva repositáře s názvy *LifApp* a *LiftApp-migrations*. První z nich obsahuje samotný projekt aplikace. Druhý obsahuje projekt s aplikační logikou pro přenos datového modelu do databázového systému PostgreSQL. V obou repositářích jsou přiložené soubory README.md s popisem projektů.