MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Tomáš Smolka

Informační systém osobní přepravy

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce:	Mgr. Pavel Ježek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Softwarové a datové inženýrství

Praha 2019

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Chtěl bych poděkovat vedoucímu práce Mgr. Pavlu Ježkovi, Ph.D. za trpělivost a pomoc při vypracování této práce, firmě Axis Transport s.r.o. za spolupráci a nasazení mé práce v praxi, Danielu Doskočilovy za přečtení práce a poskytnutí zpětné vazby, a mámě za podporu během bakalářského studia.

Název práce: Informační systém osobní přepravy

Autor: Tomáš Smolka

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., katedra

Abstrakt: Cílem této práce bylo vytvořit informační systém pro malé firmy podnikající v oboru smluvní osobní přepravy, které nemají přístup k vhodnému softwarovému řešení pro svoji činnost.

Vzniklý informační systém poskytuje řešení pro celý životní cyklus objednávky na přepravu, od založení objednávky od cestovní agentury, přes naplánování realizace objednávky řidičem, až po vytvoření faktury pro cestovní agenturu.

Z důvodu téměř neexistujících standardů v oboru osobní přepravy na formáty souborů (objednávky, faktury atd.) jsou všechny vstupy a výstupy systému řešeny pomocí pluginů.

V textové části práce je podrobně představeno fungování firmy poskytující smluvní osobní přepravu. Dále je rozebrán návrh a samotný vývoj informačního systému. Následně je popsána struktura implementovaného softwarového řešení.

V teoretické části práce byla provedena hloubková analýza interních procesů firmy poskytující smluvní osobní přepravu, na jejímž základě byl proveden návrh a následná implementace systému.

Klíčová slova: smluvní osobní přeprava informační systém plugin systém wpf mvvm wcf mef

Title: Personal Transport Information System

Author: Tomáš Smolka

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., department

Abstract: The aim of this bachelor thesis is to create an information system for small companies which are running a business in the field of the contractual passenger transport and which do not have an access to a suitable software solution for their business.

This information system, which has arisen, provides the solution for a complete cycle of a transport order which starts with a formation of an order from a travel agency. In the next step, there is a realisation of the planned order which is processed by a driver and last step is a creation of an invoice for the travel agency.

On the grounds of the fact that there are not almost any existent standards for the formats of the requested files in the field of the contractual passenger transport (for example: the orders, the invoices, etc.), all the inputs and the outputs of the system are solved by means of plug-ins.

In the text part of the thesis, there is presented the operation of the contractual passenger transport company in details. In addition, there is an analysis of the the system draft and there is also examined the development of the system itself. This analysis is followed by a description of the implemented solution structure.

In the theoretical part, there is executed an in-depth analysis of the internal processes which are performed by the company of the contractual passenger transport. On the basis of this analysis, the thesis proposes the draft of the system and the subsequent implementation of the system.

Keywords: information system mef mvvm plug-in system contractual passenger transport wcf wpf

Obsah

1. Úvod

V minulosti autor této práce pracoval na pozici řidiče ve společnosti Axis Transport s.r.o. poskytující smluvní osobní přepravu. Při tom si všiml špatně fungující administrativy, a to zejména v důsledku nevhodného softwarového vybavení pro tuto činnost. Tyto administrativní problémy často vedou k nemalým finančním ztrátám či negativním hodnocením od klientů.

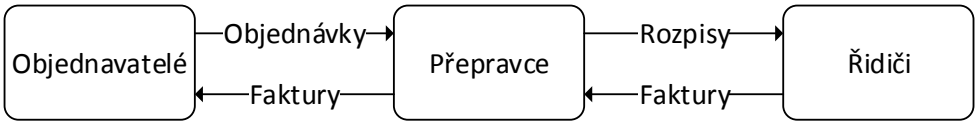
Diskusí s řidiči z jiných společností jsme zjistili, že i v dalších společnostech je situace podobná. Pouze největší společnosti v oboru si mohou dovolit zaplatit vývoj specializovaného informačního systému přesně pro jejich potřeby.

V této práci se budeme snažit najít vhodné administrativní řešení nejen pro výše zmíněnou firmu Axis Transport s.r.o. (dále označovaná jako *partnerská firma*), ale i další malé či středně velké firmy podnikající v oboru smluvní osobní přepravy.

1.1 Smluvní osobní přeprava

Na úvod si musíme vysvětlit co to vlastně smluvní osobní přeprava je a jak fungují firmy tuto službu poskytující. Jak už název napovídá, jedná se o přepravu osob na základě uzavřené smlouvy mezi firmou, která si přepravu objednává, dále *objednavatel*, a firmou, která přepravu poskytuje, dále *přepravce*.

Přepravce funguje jako prostředník mezi objednavateli přepravy a jednotlivými řidiči, kteří přepravu poskytují. Objednavatelé zasílají přepravci objednávky na přepravu, ten je ukládá a později rozděluje mezi řidiče, kteří přepravu realizují. Poté co je přeprava poskytnuta, přepravce zasílá objednavateli fakturu za poskytnutou přepravu a současně řidiči zasílají faktury za poskytnutou přepravu přepravci. Popsané schéma fungování ilustruje Obrázek 1.1.



Obrázek 1.1: Schéma poskytování smluvní osobní přepravy

Nyní si projdeme detailní popis jednotlivých účastníků a jejich činnosti. Začneme u objednavatelů, v drtivé většině se jedná o *cestovní agentury*, proto dále v textu budeme pojem objednavatele a cestovní agentury ztotožňovat. Cestovní agentury využívají smluvní osobní přepravu v případech, kdy nabízejí takzvané „package holiday“, tedy dovolenou jako kompletní balíček zahrnující letenky, transfer z letiště do hotelu a zpět, ubytování v hotelu a případně další služby.

Cestovní agentury zasílají přepravci objednávky na přepravu osob, v oboru se obvykle používá termín objednávka na transfer. Pro formát objednávek neexistuje žádný jednotný standard a jednotlivé cestovní agentury používají vlastní formáty, obvykle dané jejich vlastním informačním systémem. Objednávky jsou typicky

zasílány emailem, a to buď jako HTML tělo zprávy či jako příloha ve formátu `.pdf` nebo `.xml`.

Podobně jako pro formát objednávek, tak ani pro jejich obsah neexistuje žádný jednotný standard, přesto ale mají ideově podobný obsah a lze v nich vždy najít minimálně tyto informace:

- Referenční číslo objednávky – identifikátor, pomocí kterého cestovní agentura, přepravovaný klient a přepravní společnost identifikují konkrétní objednávku při vzájemné komunikaci.
- Jméno klienta – jméno alespoň jedné z přepravovaných osob. Často se používá cedulka s tímto jménem k nalezení klienta na místech s vysokou koncentrací osob, například na letišti či vlakovém nádraží.
- Počet osob – celkový počet přepravovaných osob. Může také obsahovat informaci o počtu a věku přepravovaných dětí.
- Popis objednané služby – informace o počtu a typu vozů (např. osobní vůz, minivan atd.), kterými má být přeprava realizována. Dále také informace, zda je transfer privátní nebo sdílený (v jednom voze jsou přepravovány osoby z více nezávislých objednávek, tzv. *shuttle*).
- Datum a čas vyzvednutí
- Instrukce k vyzvednutí – obvykle adresa počáteční destinace, ale často obsahuje také další informace k vyzvednutí. Například pokud je klient vyzvednut na letišti, tak obsahuje číslo letu, kterým klient přilétá.
- Instrukce k vyložení – obvykle adresa cílové destinace, ale může obsahovat další informace k vyložení klienta.

Přepravní firma potřebuje nějaké administrativní řešení, tak aby objednávku, která často dorazí i několik měsíců před plánovaným provedením, mohla po přijetí založit, tak aby ji bylo možné několik dní před provedením naplánovat, ale také kdykoliv upravit či stornovat. Právě na toto řešení se v této práci zaměřujeme a v kapitole 1.2 si detailně popíšeme požadavky na toto řešení a v kapitole 1.3 si pak popíšeme současně řešení v partnerské firmě. Ještě předtím si však musíme dovysvětlit další aspekty činnosti přepravní firmy, protože zmiňované administrativní řešení v mnoha z nich hraje významnou roli.

Při zakládání objednávky na transfer je nutné určit cenu, za kterou bude později vyfakturovaná. V některých případech je tato cena součástí objednávky, v jiných případech je nutné tuto cenu určit podle ceníku, v ojedinělých případech je cena určena jinak (například telefonicky dohodou pro speciální objednávky).

Pro provozování osobní přepravy musí mít firma k dispozici dostatek řidičů a vozů. V oboru je pravidlem, že řidiči poskytují přepravu vlastními vozy, a to zejména z důvodu nižších počátečních investicí při vzniku firmy, a také kvůli tradičně špatnému zacházení řidičů s firemními vozy. Dalším pravidlem je, že řidiči pracují na živnostenský list namísto pracovní smlouvy, a tedy za vykonanou práci firmě zasílají faktury, jejichž správnost firma musí být schopná ověřit dle vlastních záznamů.

Obvykle 3 až 7 dní před plánovaným vyzvednutím zákazníka je nutné objednávky naplánovat. Plány se obvykle vytvářejí pro jednotlivé dny a nazývají se *rozpisy*. V rozpisech jsou k objednávkám přiřazeni řidiči. Obvykle je každému řidiči přiřazena série jednotlivých objednávek, ale nastávají i složitější situace, kdy je k jedné objednávce přiřazeno více řidičů, například v situaci, kdy není k dispozici vůz s odpovídající kapacitou a je nutné poslat více menších vozů, nebo je řidiči přiřazeno více objednávek, které vykoná současně jako jeden sdílený transfer.

Sestavený rozpis je pak nutné předat řidičům a také uložit, aby jej bylo možné v případě nutnosti editovat, či případně dohledat v situacích, kdy je nutné určit, kterému řidiči byla daná objednávka přiřazena, například v případě stížnosti nebo negativního hodnocení od klienta.

Vedle řidičů firma také zaměstnává operátory, kteří jednak zodpovídají za telefonickou a emailovou komunikaci s cestovními agenturami a koncovými klienty a vyřizování jejich požadavků, a tak řeší neočekávané situace jako například zpoždění příletů letadel. K tomu, aby operátor mohl pracovat, musí mít k dispozici objednávky na transfery a rozpisy řidičů.

Na konci životního cyklu objednávky transferu je její fakturace cestovní agentuře. Zde opět neexistuje jednotný formát, ve kterém cestovní agentury faktury přijímají, typicky je to soubor ve formátu `.pdf` nebo `.xlsx`, ale můžeme se setkat i s proprietárními formáty pro import do informačního systému dané cestovní agentury.

1.2 Požadavky na administrativní řešení

Administrativní řešení, které umožňuje objednávky na transfery zakládat, vyhledávat a plánovat, je pro fungování firmy poskytující smluvní osobní přepravu naprosto klíčové. Toto řešení často může i rozhodnout o úspěchu či neúspěchu firmy na trhu, protože firma funguje pouze jako prostředník mezi cestovními agenturami a řidiči a tyto činnosti představují zásadní většinu práce přepravní firmy.

Nyní na základě znalostí z předchozí kapitoly a konzultací s partnerskou firmou sestavíme seznam požadavků, které jsou na toto administrativní řešení kladeny.

P1 Dostupnost

Operátoři obvykle pracují z domova (tzv. Home-office), a proto je nutné, aby bylo možné ke všem informacím přistupovat vzdáleně přes internet. Současně je nutné, aby přístup do systému byl zabezpečený, jelikož se v něm budou nacházet citlivé informace o firmě a jejích klientech.

P2 Import objednávek

Objednávky přicházejí z různých zdrojů a v různých formátech a je nutné se těmto podmínkám flexibilně přizpůsobovat, jelikož je velmi pravděpodobné, že každá nová cestovní agentura bude mít vlastní formát objednávek.

P3 Tvorba a distribuce rozpisů

Vytváření rozpisů je důležitá část administrativy firmy a je proto nutné,

aby tato činnost byla prováděna efektivně a spolehlivě. Je tedy nutné minimalizovat riziko selhání, jako například ztráta objednávky. Rozpisy je dále nutné denně distribuovat řidičům opět efektivním a spolehlivým způsobem.

P4 Tvorba a export faktur
Pro firmu je nezbytné pravidelně vytvářet faktury za realizované objednávky a tyto faktury exportovat ve formátech požadovaných cestovními agenturami. U každé objednávky musí být možné ověřit, že byla vyfakturovaná a za jakou částku.

P5 Kontrola faktur řidičů
Řidiči pro firmu pracují na živnostenský list a zasílají faktury za vykonanou přepravu. Pro firmu je pak zásadní mít možnost tyto faktury porovnat s vlastními záznamy a odhalit případnou nepoctivost řidičů, která by vedla k finančním ztrátám.

P6 Historie
U objednávek je nutné udržovat historii, aby bylo možné v případě problému při provedení či stížností od zákazníka odhalit zodpovědnou osobu či chybu v zadání objednávky.

P7 Náklady
Firmy v tomto oboru v dnešní době pracují s poměrně malými maržemi a proto náklady musí být brány v úvahu pří návrhu nového administrativního řešení. To znamená nejen co nejnižší pořizovací náklady na samotný systém, ale také možnost využít existující infrastruktury, v případě partnerské firmy se jedná o kancelářské počítače s operačním systémem Windows 7, 8 nebo 10 a server s operačním systémem Windows Server 2012 R2.

1.3 Existující řešení

V současné době firma využívá k práci s objednávkami Excel tabulku, kde jsou jednotlivé transfery reprezentovány jako řádky s údaji v několika sloupcích. Toto řešení zhodnotíme z pohledu seznamu potřeb společnosti, který jsme sestavili v kapitole 1.2.

Požadavek dostupnosti (P1) se částečně naplňuje posíláním celé tabulky emailem mezi operátory, což vede k častým chybám, kdy operátoři nepracují s aktuální verzí nebo úplnou verzí. Dalším omezením je, že aktuální verzi má vždy právě jedna osoba, ostatní si ji musí v případě potřeby vyžádat.

Import objednávek (P2) je realizován ručním přepisováním do tabulky, což je časově náročné a vede k zbytečným chybám. Zároveň to také omezuje růst firmy, protože takto lze zpracovat pouze omezený počet objednávek.

Tvorba rozpisů (P3) je realizována sloupcem tabulky kam se vepisují jména řidičů. Stejně jako v předchozím bodě je tento postup neefektivní a vede ke zbytečným chybám. I zde tento postup omezuje růst firmy, protože takto lze vytvářet rozpisy pouze omezené velikosti. Takto vytvořené rozpisy jsou potom řidičům distribuovány pomocí emailu. Tento způsob distribuce funguje dobře, protože umožňuje řidičům k vytvořeným rozpisům přistupovat kdykoliv z mobilních telefonů.

Faktury (P4) jsou vytvářeny vyfiltrováním požadovaných transferů a jejich nakopírováním do nové tabulky pro danou fakturu a poté ruční editaci do vhodného formátu. Tento postup funguje, ale opět je časově náročný.

Kontrolu faktur řidičů (P5) nelze podle dat v tabulce realizovat, protože chybí informace o výši odměny pro řidiče. Tento údaj se nevyplňuje, protože u každého řidiče je cena individuální a vytváření rozpisů je už tak časově velmi náročné. Tento fakt v některých případech vede k nezjištěnému zneužívání řidiči, kdy si fakturují vyšší částky, než na které mají právo, což vede k finančním ztrátám pro firmu.

Historii (P6) lze v tabulce obvykle dohledat, ale jak bylo zmíněno u bodu (P1), občas dochází k její ztrátě kvůli špatnému naplnění toho bodu. V tabulce také chybí historie jednotlivých změn, a tak není možné odhalit případnou chybu zaměstnance při provádění změn.

Existující řešení bylo zvoleno zejména kvůli prakticky nulovým pořizovacím a provozním nákladům (P7) a z tohoto pohledu je vyhovující.

Z předchozí diskuse je nám zcela zřejmé, že aktuální situace v administrativě firmy je nevyhovující a vede nejen k větší časové zátěži zaměstnanců, ale i k finančním ztrátám. Aby firma mohla růst a být konkurenceschopná je nutné hledat lepší administrativní řešení.

1.4 Možná řešení

Nyní prozkoumáme navrhovaná řešení a podíváme se na jejich výhody a nevýhody.

Zlepšit stávající systém
Po doporučení autora přešla firma od aplikace Excel k používání Google Dokumenty, tato webová aplikace umožňuje přístup více uživatelů ke sdílené tabulce v úložišti Google Drive. Touto změnou bylo výrazně usnadněno sdílení dat mezi uživateli (P1), jedná se však pouze o malé zlepšení a stávající řešení je stále nevyhovující.

Generický objednávkový systém
Na trhu existuje mnoho generických (univerzálních) objednávkových systémů, ale žádný z těch, které se podařilo dohledat, nepodporuje funkce specifické pro smluvní osobní přepravu (např. vytváření rozpisů jízd), a proto je nelze v tomto oboru plnohodnotně využít.

Informační systém na zakázku
Toto řešení volí největší firmy na trhu a zcela jistě se jedná o optimální řešení, bohužel je toto řešení zcela mimo finanční možnosti malých a středně velkých firem v tomto oboru a v našem případě jej tedy nelze využít.

Vytvořit vlastní systém
Vzhledem k tomu že žádné z výše navrhovaných řešení nesplňuje naše požadavky, tak jsme se rozhodli vyvinout vlastní řešení, které bude zaměřené na naplnění potřeb partnerské firmy a podobně velkých firem v oboru smluvní osobní přepravy.

Tímto řešením se má stát právě tato bakalářská práce v níž se pokusíme navrhnout a implementovat softwarový informační systém, který bude možné použít jako dostupné administrativní řešení ve firmách poskytujících smluvní osobní přepravu.

1.5 Shrnutí cílů

Cílem této práce je navrhnout a implementovat softwarový informační systém pro malé a středně velké firmy poskytující smluvní osobní přepravu, který bude naplňovat požadavky P1 až P7 sestavené v kapitole 1.2.

2. Analýza zadání

Na základě diskuse v předchozí kapitole jsme se rozhodli pro vývoj vlastního informačního systému pro správu informací o osobní přepravě (dále pouze označovaný jako *informační systém*) a v této kapitole se budeme zabývat analýzou a následným návrhem systému. Jednotlivé podkapitoly se zabývají různými problémy návrhu systému, jejich možnými řešeními a odůvodněním vybraného řešení.

Během návrhu našeho systému jsme se několikrát setkali se situací, že jsme hledali vhodný již existující framework nebo knihovnu pro řešení daného problému. V takové situaci jsme vždy brali v potaz také cenu, protože se snažíme vytvořit dle požadavku P7 cenově dostupný systém, a tedy i v případě kdy by například byla k dispozici akademická licence, která by dovolovala bezplatné použití v této práci, ale placené použití při komerčním nasazení, které je cílem této práce, tak jsme se snažit hledat jiné alternativy nebo aspoň uvážit náklady při komerčním použití.

2.1 Volba jazyka a vývojového prostředí

Na základě cílů práce očekáváme, že náš systém bude potřebovat komunikovat přes internet, pracovat s databází a poskytovat bohaté grafické rozhraní, chceme proto zvolit vysokoúrovňový programovací jazyk, který nabízí vhodné frameworky pro tyto problémy.

Vhodnými kandidáty splňující zmíněné požadavky jsou Java a `C#`. Náš systém by bylo možné implementovat v kterémkoliv z těchto jazyků. Na základě předchozí autorovy zkušenosti nejen s jazykem `C#`, ale i mnoha frameworky v tomto jazyce, volíme tento jazyk a platformu .NET, která je s ním spojená.

Jako vývojové prostředí použijeme Visual Studio 2017 vyvíjené společností Microsoft, která též vyvíjí jazyk `C#` a toto prostředí tedy poskytuje výbornou integraci nejenom s jazykem `C#`, ale i jeho frameworky.

Pro zálohování a správu kódu budeme využívat Visual Studio Team Services. Tato služba je dobře integrovaná s naším vývojovým prostředím a pro jednotlivé vývojáře je zdarma.

2.2 Architektura a komponenty systému

Při návrhu architektury našeho systému budeme vycházet z požadavků P1 – P7, které na náš systém klademe.

Vzdálený přístup

První požadavek je možnost přístupu uživatelů do systému odkudkoliv přes internet (viz P1), to znamená že budeme potřebovat centrální server, kde budou uložena sdílená data. S tímto serverem následně budou komunikovat klientské aplikace, které budou umožňovat uživatelům práci se systémem pomocí uživatelského rozhraní, proto volíme architekturu server-klient.

Klientská aplikace
Musíme analyzovat kolik klientů budeme potřebovat a jaké funkce budou muset poskytovat. Víme, že se systémem budou pracovat jednak operátoři, kteří potřebují přístup k tvorbě a editaci transferů a rozpisů. Dále vedení firmy, které kromě funkcí pro operátory také potřebuje přístup k vytváření faktur pro cestovní agentury a řidiče a správě ceníku. Protože vedení firmy potřebuje přístup k nadmnožině funkcí pro operátory, budeme tyto funkce implementovat v rámci jednoho klienta (dále označovaný jako *firemní klient*) a přístup k různým funkcím omezíme pomocí uživatelských rolí. V tomto klientu budeme implementovat funkce dle požadavků P2 až P5.

Distribuce rozpisů
Dle požadavku P3 bude nutné zajistit distribuci rozpisů vytvořených ve firemním klientu k řidičům, pro tento účel jsme uvažovali o vytvoření klienta pro řidiče, do kterého se řidiči budou moci přihlásit a prohlížet jim přiřazené jízdy. Tento klient musí být vhodný pro mobilní platformy, protože řidiči k němu musí mít přístup při práci.

Monitorování řidičů
Také by bylo vhodné, aby řidiči mohli prostřednictvím tohoto klienta potvrzovat přijetím rozpisů a dále také zaznamenávat informace o průběhu přepravy, jako například že řidič čeká na klienta, je na cestě s klientem nebo dokončil přepravu. Tyto informace by bylo možné zaznamenávat spolu s GPS pozicí řidiče a zobrazovat je ve firemním klientu, aby je mohli operátoři využít v situacích, kdy buď potřebují ověřit, zda je řidič opravdu na místě, anebo v situaci kdy potřebují najít nejbližšího volného řidiče. Při analýze (viz kapitola 2.2.1) se ukázalo, že realizovat tuto funkci by bylo zcela mimo možnosti této práce.

Zrušení klienta pro řidiče
Poté co jsme analyzovali náročnost a přínosnost implementace tohoto klienta (viz kapitola 2.2.5) a s přihlédnutím k upuštění od realizace sledování pozic řidičů, tak jsme se rozhodli klienta pro řidiče nevytvářet a jeho funkci pro distribuci rozpisů nahradit pomocí pluginu, který umožní snadno vytvářet emailové zprávy s pomocí aplikace Outlook, kterou už firma používá.

Po rozdělení systému na základní komponenty se nyní budeme zabývat jejich analýzou a návrhem jejich architektury.

2.2.1 Server

Centrální server rozdělíme do dvou komponent, datového úložiště a aplikačního serveru. K tomuto rozdělení přistupujeme, protože ukládání dat a vzdálený přístup k nim přes internet jsou dvě rozdílné úlohy, a proto je vhodné je i v návrhu řešit samostatně.

Datové úložiště
Jako datové úložiště použijeme SQL databázi, protože budeme potřebovat ukládat velké množství záznamů, jejichž položky jsou pevně dané, a databáze nám k těmto datům umožní rychlý přístup a zároveň bude zajišťovat jejich základní integritu.

Aplikační server
Tuto komponentu budeme realizovat jako službu, kterou budeme hostovat na serveru s operačním systémem Windows Server (viz požadavek P7). Tato služba bude implementovat veškerou business logiku serveru, bude tedy zajišťovat klientům nejen zajišťovat přístup k datům, ale také bude provádět autentizaci a autorizaci uživatelů a kontrolovat správnost jejich akcí.

2.2.2 Sledování pozic řidičů

Při návrhu systému jsme zvažovali možnost zaznamenávat pozice řidičů pomocí klienta pro řidiče a poté pozice řidičů zobrazovat v uživatelském rozhraní firemního klienta.

Abychom mohli tato data zobrazovat budeme potřebovat komponentu uživatelského rozhraní, které dokáže na pozadí mapových podkladů zobrazovat pozice objektů, takovou komponentu budeme dále označovat jako *mapová komponenta*. Takovou komponentu můžeme buď na-implementovat vlastní nebo najít vhodný framework pro uživatelské rozhraní, pro který je taková komponenta k dispozici.

Bing Maps
Protože implementace vlastní mapové komponenty by byla zcela mimo rozsah této práce, budeme hledat již existující implementaci. Jelikož jsme se rozhodli používat platformu .NET, začali jsme naše hledání u Bing Maps [1], protože se také jedná o produkt firmy Microsoft a očekáváme snadné propojení s .NET platformou. Bing Maps nabízí mapovou komponentu jak pro webové aplikace, tak pro desktopové aplikace využívající framework WPF nebo UWP [2].

Funkce těchto komponent nám plně dostačují ve všech případech, ale narazili jsme na problém s omezením použití samotných Bing Maps pro interní komerční aplikace. Pro takové použití je vyžadován takzvaný *„Bing Maps Enterprise Key“* [3], který je zpoplatněn. Cena tohoto klíče není nikde na webu přímo uvedena, a tak jsme zaslali žádost o cenu, ke které jsme připojili popis použití v našem systému.

Byly jsme kontaktování firmou *WIGeoGIS*, která zastupuje Bing Maps v řadě evropských zemí včetně České Republiky. Ve zprávě jsme obdrželi informace, že na náš případ se vztahuje licence pro *„Mobile Asset Management (MAM)“*, přičemž využití této licence znamená minimální roční platby ve výši 2 700 Euro. Tato částka je zcela v rozporu s požadavkem na nízké náklady na provoz systému (viz P7) a proto jsme možnost použití Bing Maps zamítli.

GMap.Net a OpenStreetMap
Dále jsme zkoumali mapovou komponentu GMap.Net [4]. Jedná se o open source implementaci mapové komponenty pro Windows Forms a WPF, která podporuje řadu poskytovatelů mapových podkladů, například Google Maps, Bing Maps, OpenStreetMap nebo Mapy.cz, a také nabízí možnost implementace vlastního poskytovatele.

Bohužel žádný z poskytovatelů, kterého se nám podařilo dohledat nenabízí možnost použití jeho mapových podkladů zdarma v komerční interní aplikaci, použití je buď zpoplatněno nebo není povoleno.

Zajímavou alternativu nabízí *OpenStreetMap*, kde je možné si mapové podklady stáhnout a vytvořit vlastní mapový server. Touto cestou by bylo možné

získat mapové podklady i pro naše účely zdarma a použít je v mapové komponentě GMap.Net[4]. Zároveň je však tato cesta velmi náročná a zahrnuje řadu kroků, zejména instalace databázového serveru podporujícího mapová data, importování stažených mapových podkladů do databáze, instalace a konfigurace serveru, který bude generovat mapové dlaždice. Instalace celé této infrastruktury by překročila výkonové i prostorové kapacity existujícího serveru, a proto jsme tuto cestu zamítli.

Zhodnocení
Ač by funkce sledování řidičů představovala užitečnou součást systému, její implementace by byla nejen časově ale i finančně velmi náročná. Dále také nepředstavuje kritickou součást k naplnění požadavků na náš systém, a proto jsme se ji po konzultaci s partnerskou firmou rozhodli neimplementovat.

2.2.3 Vstupní a výstupní formáty

Důležitým aspektem našeho systému je flexibilita, co se týče vstupních a výstupních souborů, protože jak jsme diskutovali v kapitole 1.1, v oboru smluvní osobní přepravy neexistují žádné standardní formáty souborů. Ke vstupům či výstupům systému se vztahují požadavky P2 až P5. Zejména u importování objednávek do systému (viz P2) je možnost pracovat s různými vstupními formáty kritická, protože každá cestovní agentura zasílá objednávky v jiném formátu a je velmi pravděpodobné, že každá nová cestovní agentura sebou přinese další nový formát.

Tyto informace musíme vzít v úvahu a navrhnout architekturu našeho systému tak, aby byl v tomto ohledu snad rozšiřitelný. Začneme tím, že si rozmyslíme, na kterých místech budeme s externími soubory pracovat:

- Import objednávek,
- Export faktur pro cestovní agentury,
- Export přehledů pro řidiče,
- Export výsledků hledání transferů (volitelné).

Dle principů objektového programování pro každou z těchto funkcí připravíme rozhraní, které jednotlivé implementace příslušného rozšíření budou naplňovat. Spolu s tím jsme také řešili, jakým způsobem se budou tyto implementace do systému přidávat. Rozhodovali jsme se mezi těmito dvěma řešeními.

V kódu klienta
V tomto řešení v kódu klienta definujeme místo, kde se budou implementace jednotlivých rozhraní registrovat, a to buď jako typ nebo se předá rovnou instance. Výhoda tohoto řešení spočívá ve velmi jednoduché implementaci. Nevýhodou je, že pokaždé když budeme chtít provést změnu v některém z rozšíření, tak budeme muset znovu přeložit celého klienta.

Plugin systém

Jednotlivá rozšíření budou implementována jako samostatné knihovny (pluginy), které budou distribuovány samostatně a klient si je při startu načte z předem stanoveného místa. Toto řešení umožní snadné aktualizace jednotlivých rozšíření. Další velkou výhodou tohoto přístupu je, že pokud bude rozšíření implementovat třetí strana, tak nebude potřebovat přistup ke zdrojovému kódu klienta, ale pouze k dokumentaci příslušného rozhraní. Tato vlastnost se ukázala být velice důležitá pro spolupracující firmu, protože panovaly obavy, že pokud v budoucnu nebude autor k dispozici, bude velmi náročné zajistit implementaci nových rozšíření pro případné nové cestovní agentury.

Nevýhodu náročnější implementace plugin systému můžeme odstranit použitím frameworku MEF (*Managed Extensibility Framework*), který umožnuje snadnou tvorbu plugin systému a navíc je standardní součástí platformy .NET. Framework umí ze zvolené složky obsahující knihovny v podobě `.dll` souborů načíst všechny implementace zvoleného rozhraní a vytvořit jejich instance, které umístí do zvolené kolekce pro toto rozhraní.

Protože jsme našli způsob, jak plugin systém snadno přidat do systému a jeho vlastnosti jsou pro spolupracující firmu důležité, tak jsme se rozhodli pro toto řešení s použitím frameworku MEF.

2.2.4 Firemní klient

Aby bylo možné pracovat s daty na serveru, budeme potřebovat klientskou aplikaci. Ta by měla mít maximálně jednoduché a intuitivní grafické uživatelské rozhraní, protože předpokládáme, že se systémem budou pracovat i technicky méně zkušení uživatelé.

Pro implementaci grafického uživatelského rozhraní existují na platformě .NET dva typy aplikací, mezi kterými jsme se rozhodovali.

Webová aplikace

Webová aplikace je v dnešní době modernější řešení než desktopová aplikace, jehož velkou výhodou je, že klientskou aplikaci bude možné používat na jakémkoliv zařízení s webovým prohlížečem a navíc bez instalace. Další výhodou je, že bude snadné distribuovat aktualizace a pluginy.

Nevýhodou je náročnější implementace logiky aplikace, kterou je nutné implementovat v jazyce javascript, který je netypový a navíc různé webové prohlížeče podporují různé verze. Implementace grafického rozhraní by také byla náročně, jelikož by bylo nutné vyřešit korektní zobrazení v různých prohlížečích. Svět webových aplikací se navíc dnes rychle vyvíjí a aplikaci by tedy bylo nutné aktivně udržovat aby fungovala v moderních prohlížečích.

Plugin systém

Výhodou je existence frameworků pro snadnou tvorbu grafického uživatelského rozhraní a lepší výkon při práci s větším množstvím dat. Nevýhodou je závislost na platformě .NET framework, a tedy operačních systémech Windows, ale pro nás je to naopak výhoda, protože je to naše cílová platforma (viz požadavek P7), a naopak tedy budeme lépe schopni využít všech možností této platformy.

Častým důvodem pro volbu desktopové aplikace před webovou aplikací je možnost off-line použití. To však v našem případě neplatí, protože i pro desktopovou aplikaci budeme potřebovat stále připojení k centrálnímu serveru kde budou uložena všechna data. Volba desktopové aplikace by nám však v budoucnu umožnila rozšíření systému o lokální cache, které by zajistila alespoň částečné fungování systému v případě výpadku centrálního serveru.

Rozhodli jsme se tedy implementovat klienta jako desktopovou aplikaci, protože desktopová aplikace bez nutnosti údržby může fungovat mnoho let, což je důležité z pohledu nákladů na provoz aplikace (viz požadavek P7). Dále také očekáváme, že klientská aplikace bude poměrně rozsáhlá a náročná na implementaci a použití webových technologií by ji ještě dále zkomplikovalo.

Desktopová aplikace nám také umožní potenciální integraci s jinými programy, například při exportu faktury do souboru aplikace Excel můžeme výsledný soubor rovnou v této aplikaci otevřít pro dodatečnou editaci. Dále také potenciální rozšíření systému o lokální cache pro zajištění off-line provozu je pro nás významné, protože i krátký výpadek systému může firmě způsobit velké problémy, protože po dobu výpadku zcela ztratí přehled o aktuálních objednávkách.

2.2.5 Distribuce balíčku pluginů

Protože jsme se rozhodli implementovat firemního klienta jako desktopovou aplikaci, budeme muset řešit distribuci pluginů k jednotlivým klientům. Při řešení tohoto problému jsme zvažovali několik možných řešení, jak pluginy k uživatelů distribuovat.

A - Ručně

Náš systém nebude distribuci řešit a uživatelé si sami zajistí nakopírování souborů pluginů do příslušného adresáře jejich klienta. Toto řešení nevyžaduje žádnou práci z pohledu implementace, ale je nejméně uživatelsky přívětivé, což je značný problém, protože očekáváme, že mnoho uživatelů našeho systému budou méně zkušení uživatelé PC.

B – Součást systému

Distribuce pluginů klientům bude řešena přímo v systému. Pro takové řešení jsme se rozhodovali mezi dvěma způsoby.

B1 - Jednotlivě

Jako součást firemního klienta naimplementujeme systém, který umožní uživateli instalovat a aktualizovat jednotlivé pluginy, které budou staženy ze serveru. Tento systém je flexibilní a umožní uživatelům plnou kontrolu nad tím, které pluginy a v jaké verzi používají.

B2 - Balíček

V tomto řešení jsou všechny pluginy součástí jediného balíčku, který se automaticky distribuuje všem klientům. Pokud dojde k přidání, odebrání či aktualizaci

pluginu, tak se aktualizuje celý balíček na novou verzi a tato verze se distribuuje všem klientům. Toto řešení není tak flexibilní jako předchozí řešení, ale jeho výhodou je, že nevyžaduje žádné akce od uživatele.

Závěr

V první verzi jsme naimplementovali a nasadili řešení, které pracuje s pluginy jednotlivě, protože je nejflexibilnější. Uživatelé však s tímto řešení nebyly spokojeni, protože po nich vyžadovalo mnoho akcí, aby byly všechny jejich pluginy aktuální. Myslíme, že velkou roli v neoblibě tohoto systému hrálo to, že jsme pluginy museli často aktualizovat, jelikož stále probíhal vývoj a pluginy jsme často měnili. Uživatelům se také často stávalo, že zapomněli plugin aktualizovat a přišli na to až ho použili, čemuž často předcházela nějaká příprava (např. vyhledání transferů k exportu), kterou museli opakovat po dodatečné aktualizaci, která vyžaduje restart klienta.

Na základě zpětné vazby od uživatelů jsme změnili distribuci pluginů na řešení používající jediný balíček, jehož aktualizace nevyžaduje žádný vstup od uživatele. Uživatelé tuto změnu hodnotili velmi pozitivně, a i my zpětně hodnotíme toto řešení jako lepší, protože naši uživatelé nevyžadují flexibilitu, kterou nabízelo předchozí řešení, ale naopak ocení jednoduchost použití.

2.2.6 Distribuce rozpisů

Důležitou funkcí našeho systému je vytváření rozpisů (P3). Tyto rozpisy je pak nutné dále distribuovat řidičům, proto jsme se při návrhu našeho systému zvažovali vedle firemního klienta vytvořit i klienta pro řidiče.

Tento klient musí být vhodný pro mobilní telefony, aby ho řidiči mohli používat během práce. Dále také musí fungovat na všech běžných mobilních platformách, tedy zejména *iOS* a *Android*, aby stejným způsobem bylo možné distribuovat rozpisy všem řidičům.

Xamarin Forms

Zkoumali jsme možnost použití frameworku Xamarin Forms, který umožňuje implementovat *„cross-platform“* mobilní aplikace v jazyce `C#` na platformě .NET. Tyto aplikace pak mohou fungovat na mobilních platformách iOS, Android a Windows Phone.

Použití tohoto frameworku by nám značně usnadnilo naplnění požadavku, aby byl klient pro řidiče dostupný na všech mobilních platformách. Při prvních experimentech s tímto frameworkem jsme však narazili na řadu problémů.

Prvním zásadním problémem je, že k přeložení aplikace pro iOS je nutné mít počítač s Mac OS 10.11 nebo novější s příslušnou verzí Visual Studia. Z důvodu politiky firmy Apple je možné jakékoliv aplikace pro iOS přeložit pouze na počítačích od této firmy, protože pro jiné platformy neexistují příslušná SDK.

Dalším problémem bylo, že se jedná o nový framework, který je stále v relativně rané fázi vývoje. V našem případě se to projevilo například neexistujícím UI editorem pro XAML, který definuje podobu uživatelského rozhraní. Jediným způsobem, jak se podívat, jak naše aplikace bude vypadat, bylo ji přeložit a spus-

tit. Vývoj uživatelského rozhraní tímto způsobem by byl velice zdlouhavý, zvláště s přihlédnutím k tomu, že s tímto frameworkem nemáme předchozí zkušenosti.

Vzhledem k těmto problémům jsme se rozhodli framework Xamarin Forms nepoužít, a protože implementace dvojice samostatných klientů pro řidiče pro platformy iOS a Android by byla zcela mimo rozsah této práce, tak jsme se rozhodli mobilního klienta pro řidiče neimplementovat a hledat jiné řešení distribuce rozpisů k řidičům.

Export rozpisů pomocí pluginu

Protože existující systém distribuce rozpisů pomocí emailu funguje v praxi dobře, rozhodli jsme se tuto možnost přidat i do našeho systému. Abychom zachovali možnost budoucí změny způsobu distribuce rozpisů, tak i pro tuto funkci vytvoříme rozhraní pro plugin a pak spolu se systémem dodáme plugin, který umožní rozpis vytvořený v našem systému exportovat do aplikace Outlook, kterou firma aktuálně úspěšně používá.

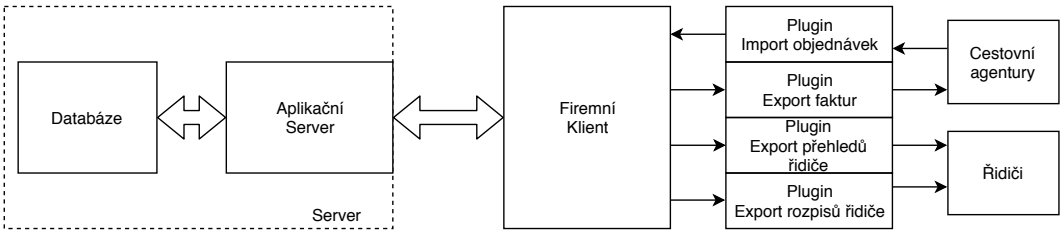
Ohlédnutí

Zpětně hodnotíme jako chybu, že jsme v této fázi návrhu nezvážili možnost implementace mobilního klienta pro řidiče formou webové aplikace. Pravděpodobně by i takto navržený klient pro řidiče byl mimo rozsah této práce, ale bylo by vhodné tuto možnost zvážit v případě budoucího rozšíření této práce.

2.2.7 Finální návrh systému

Poté co jsme udělali první návrh našeho systému, tak jsme provedli analýzu jeho jednotlivých komponent, kdy jsme se jednak zaměřovali na možnosti, jak tyto komponenty realizovat, tak jsme se snažili zhodnotit časovou náročnost a potenciální přínosy pro systém jako celek.

Výsledný návrh systému, který můžeme vidět na obrázku 2.1, je založen na architektuře server-klient, kde server bude dále rozdělen na databázi a aplikační server. Klient bude jediný v podobě firemního klienta, kterého budou používat jak operátoři, tak vedení firmy. Tento klient bude dále rozšiřitelný pomocí pluginů, které budou zajišťovat práci s externími vstupními a výstupními soubory, tedy import a export transferů, export faktur a export rozpisů pro řidiče.



Obrázek 2.1: Finální architektura systému

2.3 Databáze

Po navržení základní architekturu celého systému, se můžeme pustit do podrobné analýzy jednotlivých jeho částí. Začneme ze zdola, tedy od databázové vrstvy, a propracujeme se až k firemnímu klientu, se kterým budou pracovat samotní uživatelé.

2.3.1 Návrh schématu databáze

Dobrý návrh databáze bude pro nás klíčový, protože na ní budou stát všechny další vrstvy naší aplikace, a proto je nutné dobře prozkoumat reálnou doménu se kterou pracuje. Základy fungování přepravní společnosti jsme si popsali v kapitole 1.1, a nyní na základě těchto informací musíme sestavit schéma databáze, tedy vytvořit jednotlivé entity, jejich položky a vzájemné vztahy, tak abychom byli schopni do databáze uložit všechny potřebné informace.

Z diskuse o fungování společnosti nám vyplynuly čtyři základní entity, se kterými budeme pracovat, tedy transfer, cestovní agentura, řidič a rozpis. Podle těchto entit jsme rozdělili následující analýzu do jednotlivých sekcí. V každé sekci se zabýváme jednou z těchto entit a entitami, které s ní souvisí.

Transfer a související entity

Základní entitou naší databáze je transfer, protože představuje základ činnosti firmy a vše další je na ni navázáno. Protože transfer představuje objednávku na přepravu osob, tak jsme požádali spolupracující firmu o několik ukázek objednávek, abychom zjistili, jaké informace jsou běžně k transferu poskytovány. Příklad jedné takové objednávky je na obrázku 2.2. Dalším zdrojem informací o tom, jaké položky je nutné u transferu uchovávat je pro nás Excel tabulka z kapitoly 1.3, kterou firma aktuálně používá ke správě transferů. Na základě těchto zdrojů jsme stanovili seznam položek, které budeme muset ke každému transferu uchovávat.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<booking>
    <reference>Tran_9475</reference>
    <lead_passenger>
        <name>Petr</name>
        <surname>Novak</surname>
        <email>Petr@novak.com</email>
        <mobile>+420 123 456 789</mobile>
    </lead_passenger>
    <transfers>
    <transfer>
        <transfer_document>Tran_9475/1</transfer_document>
        <vehicle>
        <code>ch36</code>
        <title>Private Coach</title>
        <max_passengers>36</max_passengers>
        <type>coach</type>
        </vehicle>
        <passengers>
        <total_passengers>12</total_passengers>
        <number_children>11</number_children>
        <number_babies>0</number_babies>
        </passengers>
        <origin type="airport">
        <id>12143</id>
        <name>Prague airport</name>
        <pickup_time>2018-06-30 10:55:00</pickup_time>
        <flight>
            <airline>China Airlines</airline>
            <flight_number>CI 9069</flight_number>
            <date>2018-06-30 10:55:00</date>
            <origin_airport>AMS</origin_airport>
        </flight>
        </origin>
        <destination type="city">
        <accommodation>
            <name></name>
            <address>
                U Stadionu 12, 288 02 Nymburk, Czech Republic
            </address>
        </accommodation>
        <id>22706</id>
        <name>Nymburk</name>
        </destination>
        <extras>
        <extra>
            <name>
                Skis / Snowboard (max. 15 kg./unit)
            </name>
            <quantity>12</quantity>
        </extra>
        </extras>
    </transfer>
    </transfers>
</booking>
```


Obrázek 2.2: Příklad objednávky transfer

První položkou je referenční číslo, které slouží k identifikaci transferu při komunikaci jak s cestovní agenturou, tak případně s klientem. Každá cestovní agentura nějaký takový identifikátor používá, ale jejich formáty se liší, ale vždy se jedná o řetězec písmen a čísel. V některých případech mohou být tyto identifikátory dva, a to zejména v případech, kdy je klientovi přidělen samostatný identifikátor.

Dále je nutné znát jméno klienta, tato informace je důležitá hned z několika důvodů. Jednak ji využívají řidiči k nalezení klienta na místech s vysokou koncentrací osob, jako jsou například letiště či recepce hotelů. Řidič napíše jméno na cedulku, a to umožní klientovi řidiče snadno rozpoznat. Jméno klienta je také často důležité pro operátora v situaci, kdy volá klient, který nezná číslo své rezervace, například pokud neví, kde ho na rezervaci najít nebo ji ztratil.

V souvislosti s klientem mohou být také uvedeny kontaktní informace, nejčastěji telefonní číslo nebo email, ale nebývá to pravidlem, protože cestovní agentury často vyžadují, aby se s klienty komunikovalo pouze jejich prostřednictvím. V mnoha případech by přímá komunikace nebyla ani možná kvůli jazykové bariéře, protože balíčkové dovolené si často kupují lidé, kteří nemluví žádným světovým jazykem.

Dále bude nutné ukládat počet přepravovaných osob a také počet přepravovaných dětí, protože je pro ně nutné dle zákona při přepravě zajistit dětskou sedačku nebo podsedák.

Další částí informací o transferu je informace odkud a kam bude přeprava probíhat. Tuto informaci nebudeme reprezentovat jako položky transferu, ale jako samostatnou entitu cesta, která se bude skládat z počáteční a koncové destinace, což budou také samostatné entity. Toto řešení přirozeně vyplývá z reálné situace a omezí duplikaci informací v databázi. Dále nám to umožní snadno pro transfery vytvářet ceníky, protože se můžeme odkazovat na jednotlivé cesty, což by v případě pouze textové reprezentace počáteční a koncové destinace transferu nebylo možné.

Protože cesta nemusí vždy postihovat všechny informace o místě vyzvednutí a vyložení, doplníme transfer o položky s informacemi k vyzvednutí a vyložení, kde budou uloženy informace specifické pro daný transfer, například číslo letu při příletu na letiště.

Aby bylo možné vytvářet zmíněné ceníky dle zvyklostí v oboru, budeme ještě potřebovat informaci o voze, kterým má být transfer poskytnut, například osobní vůz versus minivan. Entitu, která tuto informaci bude reprezentovat budeme nazývat služba, abychom pokryli i situace kdy se nejedná přímo o vůz, jako například transfer typu „shuttle“, tedy sdílený vůz.

Takto navržená reprezentace transferu nám umožňuje uložit všechny základní informace k transferu, ale v praxi se často objevuje mnoho doplňujících informací, specifických pro daný transfer, a proto potřebujeme mít možnost ke každému transferu volitelně ukládat doplňující informace, které budeme reprezentovat entitou poznámka k transferu a ke každému transferu jich bude možné přiřadit libovolné množství. V každé poznámce bude kromě samotného textu také typ poznámky pro snazší případné další zpracování, dále viditelnost, která určí, zda má být poznámka distribuovaná i k řidičům, a informaci, zda je poznámka potvrzená, což je užitečné v případě, kdy poznámka vyžaduje další zpracování, například pokud byl transfer vytvořen na poslední chvíli a je nutné jej přidat do

existujícího rozpisu.

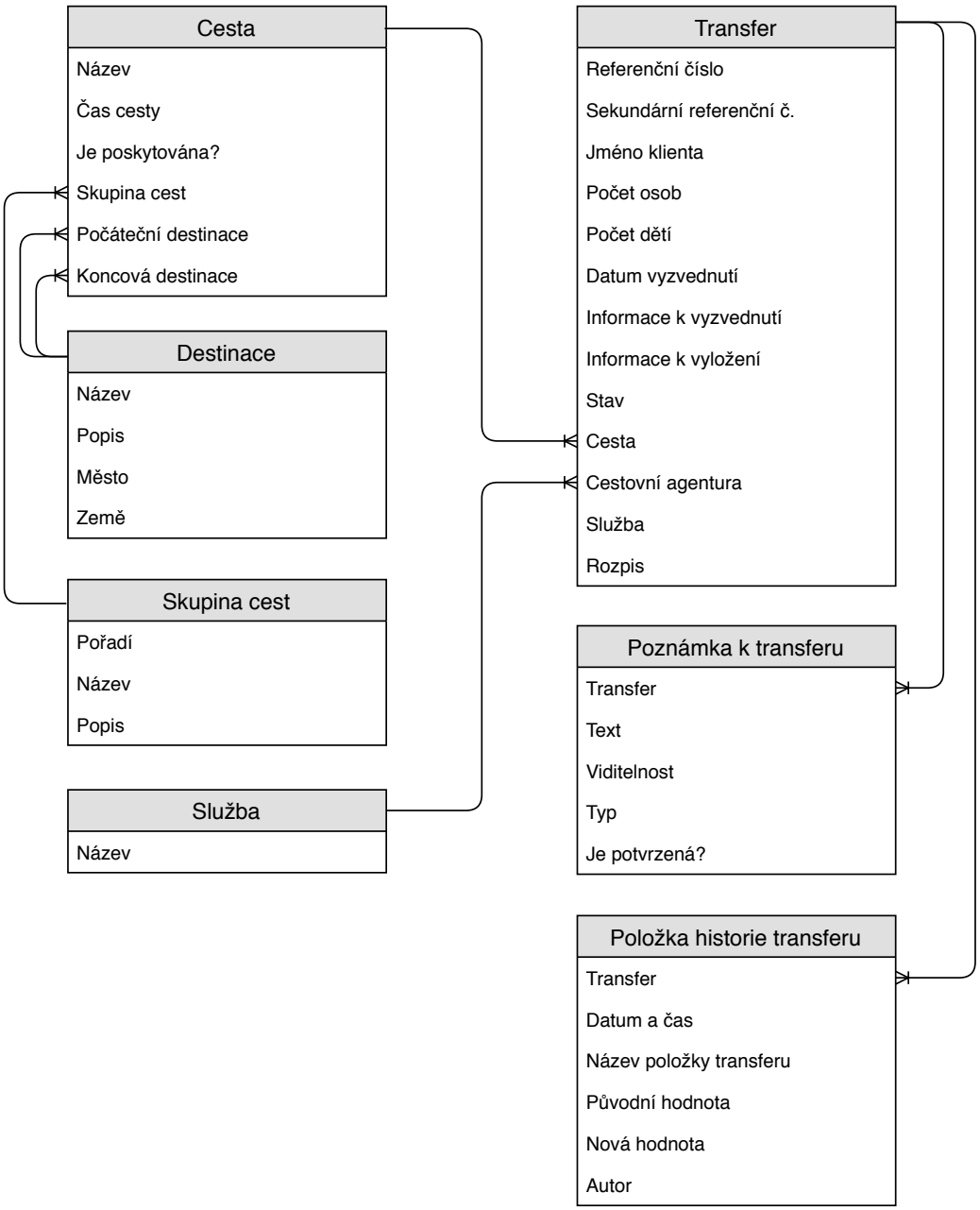
Když jsme takto vytvořené entity prezentovali partnerské firmě pomocí pohledů nad příslušnými tabulkami, tak se ukázalo, že bude potřeba zlepšit možnosti seskupování a řazení transferů. Navrhovali jsme transfery seskupovat a řadit podle cesty, ale u tohoto řešení jsme objevili dva problémy. Prvním je, že seskupování transferů podle cesty je v mnoha případech nedostačující, příkladem jsou přílety na letiště Praha s různou cílovou destinaci, protože firma preferovala mít všechny transfery, které míří do různých částí Prahy v jedné skupině, dále pak ostatní transfery z letiště Praha ve skupině druhé, protože v rámci těchto skupin jsou transfery z pohledu rozdělení mezi řidiče velmi podobné, tedy krátká jízda do centra Prahy versus dlouhá jízda mimo Prahu.

Druhý problém je nemožnost uživatelsky definovat pořadí skupin transferů, protože navrhované seřazení podle názvů cest je nevhodné. Příkladem je opět situace firmy se kterou spolupracujeme, drtivá většina (více než 90 procent) jejich transferů je z nebo na Letiště Praha, a proto je vhodné, aby transfery na méně běžných cestách byly na začátku, protože je pak daleko jednodušší si jich všimnout, než když se vyskytují až na konci.

Abychom tyto nedostatky našeho návrhu odstranili, tak jsme přidali novou entitu skupina cest, každá cesta pak bude patřit do právě jedné této skupiny a transfery se budou v uživatelském rozhraní seskupovat podle těchto skupin. Navíc tato entita má uživatelsky nastavitelný index pořadí, kterým se určí vzájemné pořadí těchto skupin při zobrazení.

Další navrhovanou změnou, kterou považujeme spíše za návrh na rozšíření funkcionality než za chybu návrhu, bylo přidání sledování změn, a to zejména u transferů. Za tímto účelem jsme přidali novou entitu položka historie transferu, která bude pro každou změnu položky transferu uchovávat následující údaje:

- jméno položky, u které došlo ke změně,
- původní hodnota,
- nová hodnota,
- datum a čas změny,
- Datum a čas vyzvednutí
- autor změny (uživatel systému).



Obrázek 2.3: Schéma pro transfer a související entity

Cestovní Agentura a související entity

Další klíčovou entitou je pro nás cestovní agentura. S touto entitou souvisí dvě důležité funkce našeho systému, první je možnost vytvářet ceníky, podle kterých se budou počítat ceny transferu. Druhou funkcí je pak vytváření faktur za provedené transfery.

U cestovní agentury bude nutné ukládat všechny informace, které mohou být potřebné pro tvorbu faktur. O vypracování seznamu těchto údajů jsme požádali spolupracující firmu. Tyto položky můžeme vidět na obrázku schématu pro cestovní agenturu.

Ceníky budeme reprezentovat entitou ceník cestovní agentury, která bude vázaná na konkrétní cestovní agenturu. U ceníku budeme ukládat jeho jméno a po-

pis, aby uživatelé mohli u ceníku uložit informace o jaký ceník se jedná a jaký je jeho účel. Dále bude u ceníku uložená informace o jeho platnosti ve formě data začátku a konce platnosti, přičemž tato data odpovídají datům realizace transferu, nikoliv objednání. Toto odpovídá běžným postupům v oboru, kdy se ceníky určují dopředu na jeden kalendářní rok a nastavují cenu transferů realizovaných v tomto období, i když jsou transfery objednány s předstihem.

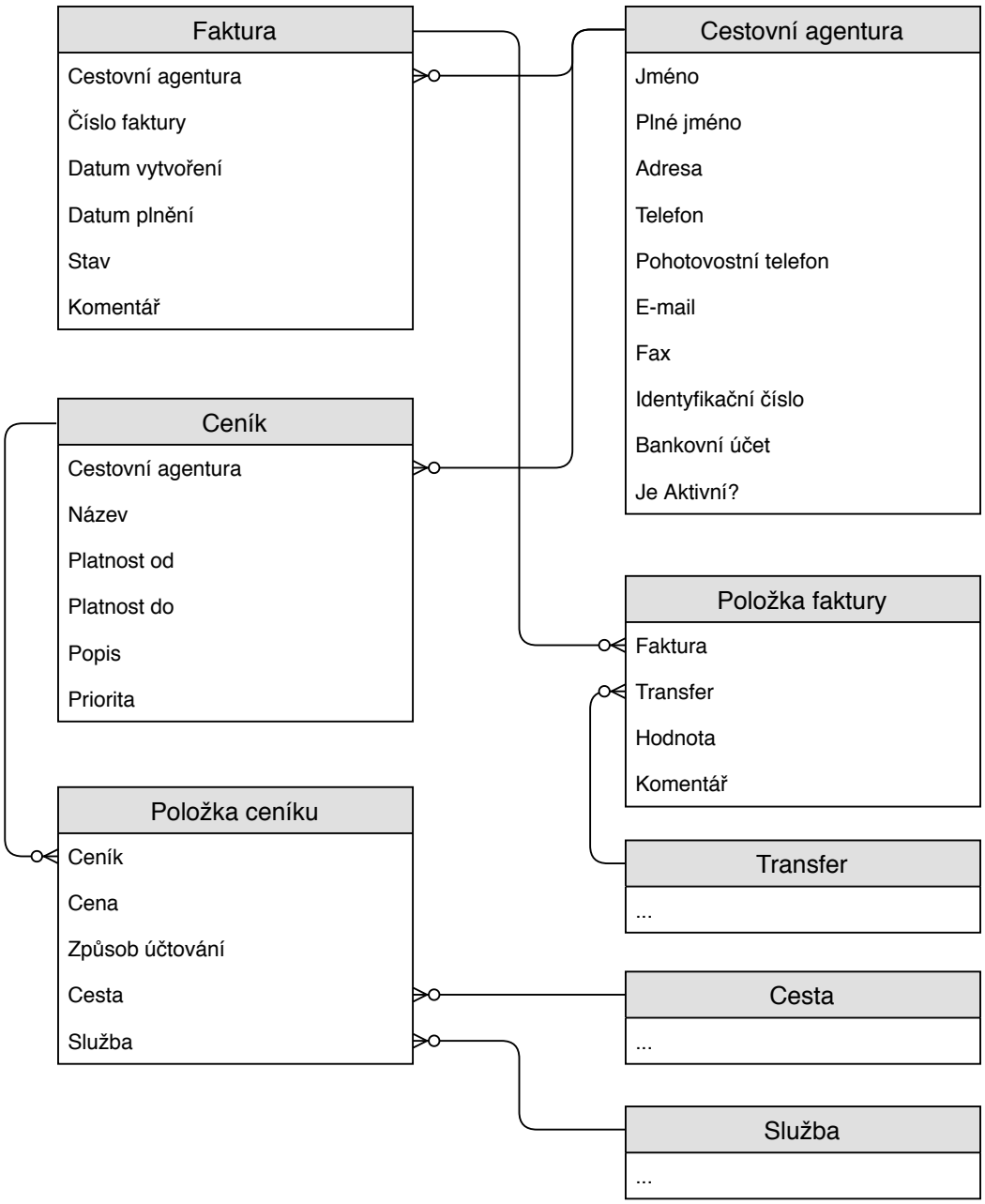
K ceníku pak budou patřit jednotlivé položky s cenami. Jak jsme již diskutovali v předchozí části této kapitoly věnované transferu a souvisejícím entitám, tak cena transferu je určena podle jeho cesty a služby. Každá položka ceníku tedy musí souviset s konkrétní cestou a službou, a navíc v rámci ceníku musí existovat nejvýše jedna položka s danou kombinací cesty a služby, aby bylo určení ceny jednoznačné.

S určením ceny pak souvisí další problém, tedy jakým způsobem cenu uložit. Triviální řešení by bylo, aby se cena přímo neukládala, ale vždy znovu dopočítala z ceníku. Toto řešení je však nevhodné, protože úprava ceníku změní ceny všech transferů jejichž cena se od daného ceníku odvíjí, navíc tyto transfery už můžou být realizované nebo dokonce i vyfakturované. Dále také uživatel nemá možnost nastavit transferu cenu mimo ceník, což je poměrně častý případ, například když jsou u transferu požadovány další extra služby nebo je naopak poskytnuta sleva, například z důvodu pozdního příjezdu řidiče.

Lepší možností by bylo ukládat cenu přímo u transferu, to by odstranilo oba výše zmíněné problémy, ale cena by stále byla reprezentována pouze jako jedno číslo, což může být například v případě zmíněných extra služeb nevhodné, protože není možné jednotlivé položky ceny oddělit. Proto jsme se pro cenu transferu rozhodli zavést samostatnou entitu položka ceny transferu, která bude kromě samostatné hodnoty obsahovat také popis a k transferu bude moci příslušet libovolný počet těchto položek. Podle ceníku pak bude vytvořena jedna tato entita s ceníkovou cenou transferu, kterou bude moci uživatel upravit, ale také přidat další položky s kladnou i zápornou hodnotou.

Tuto entitu navíc také využijeme k implementaci i další požadované funkce, a to vytváření faktur pro cestovní agentury. Fakturu budeme reprezentovat novou entitou Faktura pro cestovní agenturu, ta bude obsahovat standardní informace, které jsme opět získali od partnerské firmy. Jednotlivé položky faktury budou reprezentovány právě položkami ceny transferu. To umožní pro každý transfer vytvořit malý účet kde budou jednotlivé položky s cenou či slevou a pak obvykle jedna (ale bude podporováno i více) položka ceny pro fakturu, která nastaví zůstatek na účtu pro transfer na nulu.

Tuto část návrhu jsme také konzultovali s partnerskou firmou a objevili jeden nedostatek. Tím je problém krátkodobých akčních ceníků, kdy je obvykle pro malou skupinu cest či služeb upravena cena. V našem řešení by bylo nutné velký celoroční ceník rozdělit na dva, jeden s platností do začátku akce a druhý s platností od konce akce. Toto rozdělování by bylo velmi pracné, protože by vedlo k přepisování a upravování často velmi velkého ročního ceníku, a proto jsme se rozhodli k entitě ceník cestovní agentury přidat položku priorita, která umožňuje existující ceník překrýt novým ceníkem s vyšší prioritou, což je vhodné například pro zmiňované krátkodobé akční ceníky.



Obrázek 2.4: Schéma pro cestovní agentury a související entity

Řidič a související entity

Další velkou částí našeho schématu je entita řidič a s ní související entity, zároveň tato část úzce souvisí s entitou rozpis a s ní souvisejícími entitami, které budeme analyzovat v následující sekci.

Jak jsme již řekli, základní entitou této částí schématu je řidič. Tato entita reprezentuje řidiče pracujícího pro přepravní firmu, který realizuje přidělené transfery v rámci rozpisu. U řidiče bude kromě jména nutné ukládat kontaktní údaje, aby je každý operátor, který bude se systémem pracovat, měl k dispozici.

Ke každému řidiči je také nutné uložit informace o jeho voze spolu s ceníkem pro tento vůz. Rozhodli jsme se, že pro každého řidiče budeme podporovat libovolný počet vozů a s každým vozem pak bude svázaný jeho ceník. Toto ře-

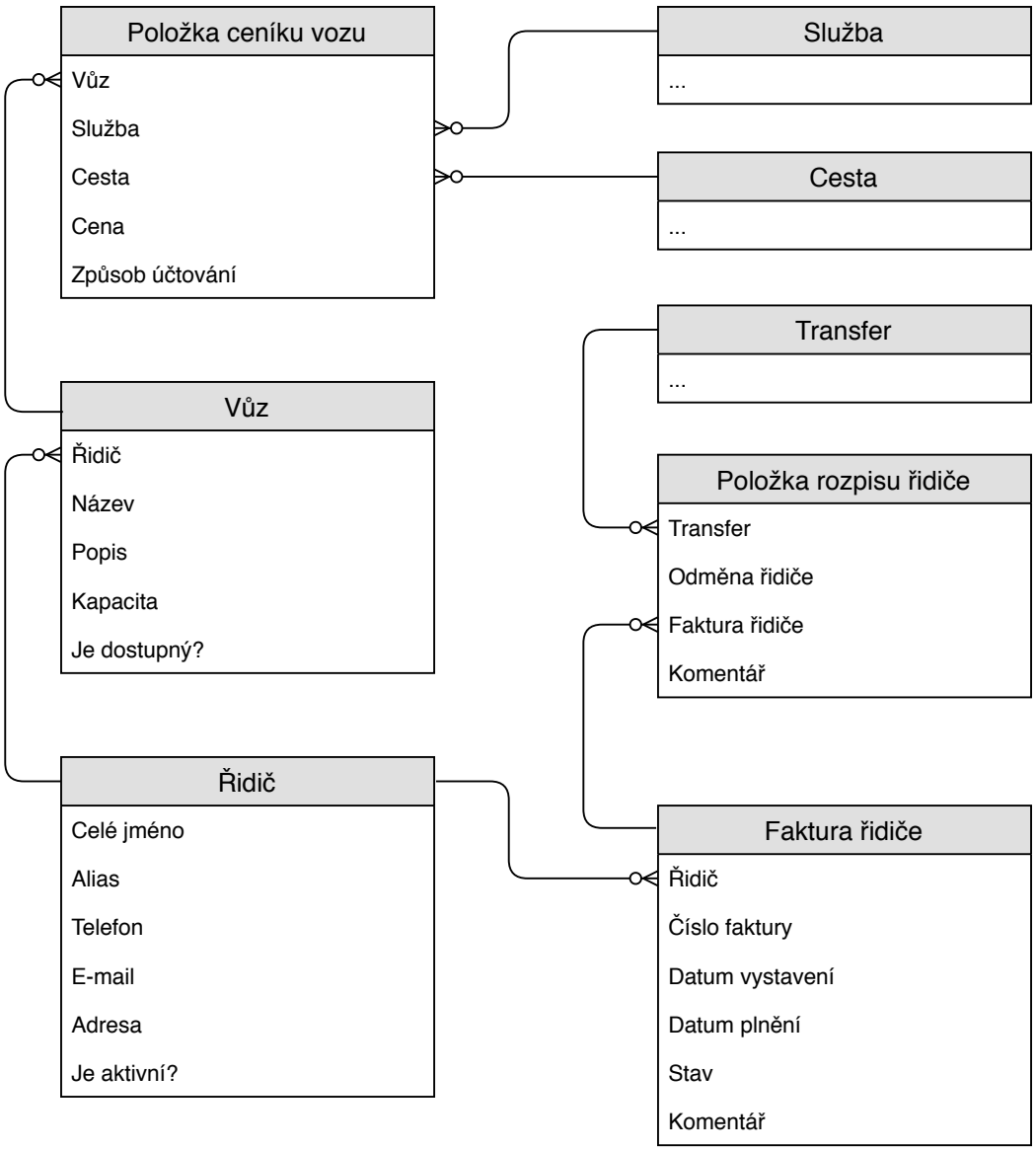
šení umožňuje snadno řešit situaci, kdy si řidič pořídí nový vůz a s tímto vozem má souviset nový ceník nebo už řidič dva vozy má a každý vůz má jiný ceník, například pro osobní vůz a minivan.

Jednotlivé vozy řidiče budeme reprezentovat entitou vůz, u které budeme ukládat popis vozu, jeho kapacitu (tj. maximální počet přepravovaných osob bez řidiče) a informaci, zda je vůz dostupný, aby bylo možné již nedostupné vozy vypnout bez jejich mazání.

Dále bude možné ke každému vozu přidávat položky jeho ceníku reprezentované entitou položka ceníku vozu. Tyto položky budou mít stejnou logiku jako položky ceníku pro cestovní agenturu, tedy určují odměnu za transfer s danou kombinací cesty a služby.

Další důležitou funkcí této části schématu je vytváření faktur pro řidiče. Na rozdíl od faktur pro cestovní agentury se v tomto případě nebude jednat o skutečné faktury ale spíše jen *kontrolní přehledy*. Protože jak jsme diskutovali v kapitole 1, tak řidiči pro přepravní firmu pracují na základě *„přepravní“* smlouvy, a proto je povinnost vystavovat faktury na nich, ale přepravní společnost potřebuje možnost tyto faktury kontrolovat. U přehledů budeme ukládat stejné položky jako u standardní faktury, aby bylo možné v budoucnu přehledy používat jako standardní faktury, pokud by tato potřeba vznikla.

Jednotlivé položky přehledů budou současně položky rozpisu řidiče, u kterých bude vyplněna cena stejně jako u položek faktur pro cestovní agentury. Více se těmto položkám budeme věnovat v následující části.



Obrázek 2.5: Schéma pro řidiče a související entity

Rozpis a související entity

V poslední části naší analýzy schématu databáze se budeme věnovat tvorbě rozpisů, tedy přiřazení transferů k řidičům. Toto je část, která je autorovi dobře známá, protože se při práci řidiče s rozpisy denně setkával.

Začněme tedy s požadavky na funkcionalitu ohledně rozpisů, protože situace není tak jednoduchá, aby nám postačovalo jednoduché přiřazení jednotlivých transferů k jednotlivým řidičům.

Za prvé musíme být schopni k jednomu transferu přiřadit více řidičů. Tato situace nastává poměrně často, a nejčastějším důvodem je, že transfer je objednán pro minivan či minibus, a to vozidlo není zrovna v dobu transferu k dispozici a transfer je nutné realizovat pomocí více osobních automobilů.

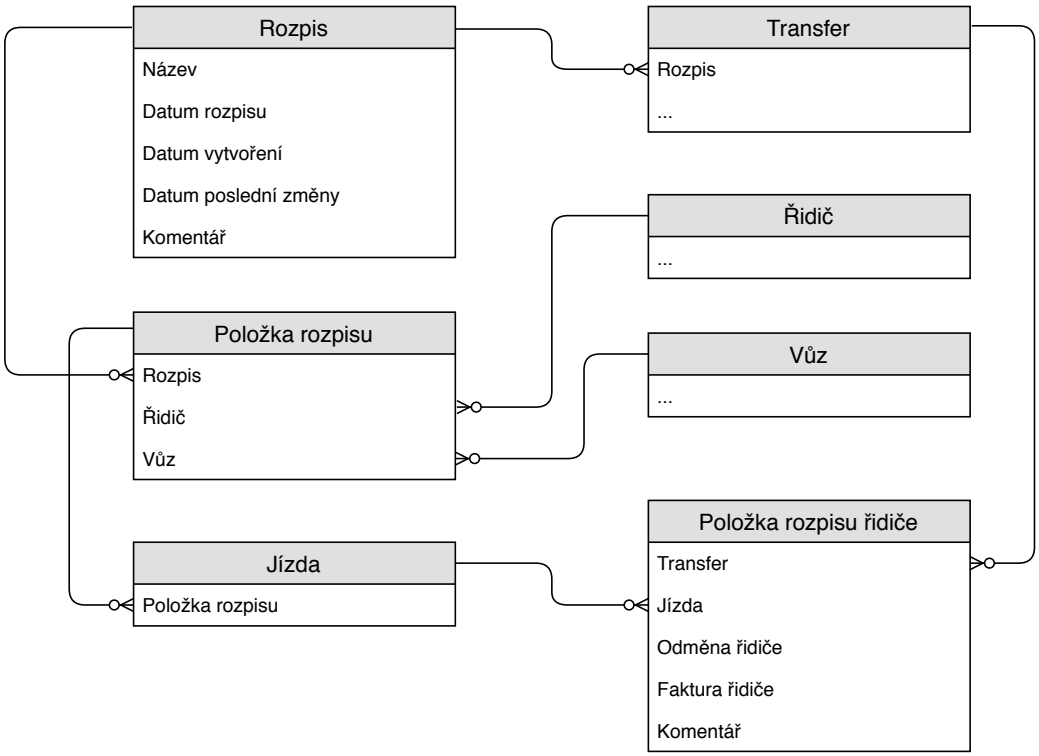
Za druhé musíme naopak být schopni k jednomu řidiči přiřadit více transferů v rámci jedné cesty a tato situace musí být jasně odlišena od situace, kdy má řidič přiřazeno za sebou několik transferů se stejnou cestou. Protože tato situace

je již o něco složitější, než jednoduché přiřazení řidiče k transferu uveďme příklad. Chceme odlišovat situace, kdy má řidič za sebou tři transfery z centra Prahy na letiště Praha od situace, kdy má řidič v rámci jedné cesty z centra na letiště vyzvednout klienty ze tří hotelů a odvést je na letiště. Mohlo by se zdát, že transfery bude možné odlišit podle různé služby, privátní osobní vůz vs. shuttle, ale to bude za prvé pro řidiče nepřehledné a může docházet k chybám, také je občas nutné z důvodu nedostatku vozů dokonce i privátní transfery sloučit do jednoho shuttle-u.

Dále také musí být možné rozpis uložit v rozpracovaném stavu, tedy v situaci, kdy jsou už do rozpisu přiřazené transfery, ale některé z těchto transferů ještě nejsou přiřazené k řidičům. K této situaci opět dochází poměrně často, protože na začátku vytváření rozpisu není hned zřejmé, kolik bude potřeba řidičů a případně jaké typy vozů.

Abychom mohli poskytnout všechny požadované funkce, vytvořili jsme následující sadu entit. Základní entitou je rozpis, ta reprezentuje jednu instanci rozpisu transferů pro řidiče, která má jméno a datum, případně nějaké komentáře. Očekávané použití je jeden rozpis na "den", kde den nemusí odpovídat kalendářnímu dni od půlnoci do další půlnoci, ale spíše dni z pohledu činnosti firmy od prvních ranních transferů po poslední transfery, které jsou často po půlnoci. V některých situacích může také dávat smysl vytvořit více rozpisů pro jeden den, například pokud firma poskytuje transfery v Praze a Brně, nebo také jeden rozpis na více dnů, například velká skupina objednávek se specifickými požadavky, které jsou schopni naplnit jen někteří řidiči.

Každý transfer pak bude patřit do nejvýše jednoho rozpisu. Rozpis se pak bude dále skládat z jednotlivých položek reprezentovaných entitou položka rozpisu. Každá tato entita bude představovat unikátní kombinaci řidiče a jednoho jeho vozu a bude se skládat z položek reprezentovaných entitou jízda, která bude představovat skupinu transferů, které mají být provedeny současně. Jednotlivé transfery v jízdě budou reprezentovány entitou položka rozpisu řidiče, která bude současně sloužit jako položka faktury řidiče a bude tedy obsahovat výši odměny pro řidiče za odpovídající transfer. Tato odměna se bude počítat podle ceníku odpovídajícího vozu a uživatel by měl mít možnost ji upravovat.

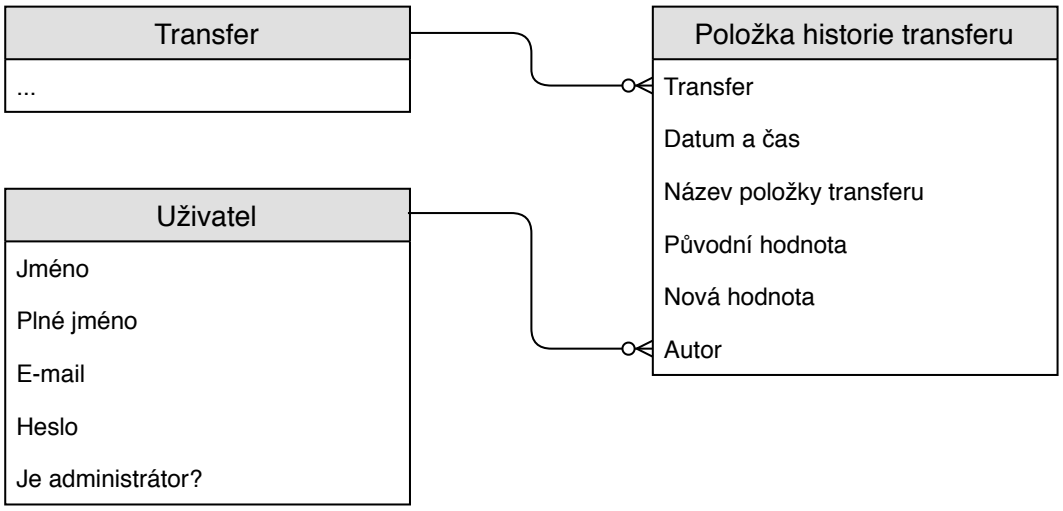


Obrázek 2.6: Schéma pro rozpis a související entity

Entita uživatel

Poslední významnou entitou je uživatel. Tato entita slouží k uložení informací o uživatelích našeho systému za účelem implementace autentifikace a autorizace. Autentifikační údaje jsou uživatelské jméno a hesla. Pro autorizaci rozlišujeme pouze dvě role, operátor a administrátor. Role administrátora má povoleno vše co operátor, takže nám stačí ukládat pouze informaci, zda je daný uživatel administrátor.

Na entitu uživatele se dále odkazuje entita položka historie transferu a tento odkaz reprezentuje autora dané změny.



Obrázek 2.7: Schéma pro uživatele a související entity

2.3.2 Volba DB serveru

Poté, co jsme navrhli schéma pro naši databázi musíme, zvolit databázový server, který budeme používat, abychom mohli naše schéma zapsat do SQL skriptu, protože každý databázový server používá vlastní „dialekt“ SQL jazyka.

Než začneme se samotným výběrem, je nutné identifikovat, jaké vlastnosti budou pro náš případ ty nejdůležitější. Protože databázi budeme používat pouze jako úložiště dat a veškerá logika bude implementována na vyšších vrstvách, a zároveň je naše schéma poměrně jednoduché, tak nám bude postačovat základní funkcionalita a u databázového serveru tedy nebudeme vyžadovat žádné pokročilé funkce.

Podle požadavku P7 budeme hledat databázový server, jehož licence je v ideálním případě bezplatná nebo aspoň cenově dostupná. Dále musí být kompatibilní s Windows Server 2008, dle požadavku P7. Dále samozřejmě požadujeme kompatibilitu s naší platformou, tedy .NET.

Na základě těchto požadavků jsme se rozhodovali mezi Microsoft SQL Server, Oracle Database a MySQL. V prvních dvou případech se jedná o komerční databáze, u nichž je nabízena i bezplatná varianta, která má omezení na výkon a velikost. V našem případě neočekáváme velkou frekvenci požadavků na databázi, takže výkonové omezení pro nás není podstatné. Velikostní omezení je v případě databáze od Microsoftu 10 GB [5] a 12 GB u databáze od Oracle [6]. V našem případě by v současné době tato omezení nehrála roli, ale v případě zvýšení požadavků na množství uložených dat (v důsledku růstu firmy) můžou způsobit nutnost přechodu na placenou verzi.

Třetí zmíněná databáze MySQL, je ve verzi Community dostupná zdarma pod licencí GPL [7]. U této verze nejsou žádná omezení na výkon či velikost (mimo těch technických). I v případě této databáze existují placené verze, ale ty se liší pouze přidanými funkcemi, které jsou v našem případě nepodstatné. Dále tato databáze po instalaci společně s grafickým nástrojem pro správu zabírá nejméně diskového prostoru, což hraje roli u serveru s omezenou diskovou kapacitou.

Na základě uvedených faktů jsme se rozhodli pro použití databáze MySQL, protože naplňuje všechny naše požadavky a zároveň nepředstavuje omezení do budoucnosti, které by vyžadovalo zakoupení placené verze.

2.4 Aplikační server

Aplikační server je vrstva naší aplikace, které bude poskytovat webové rozhraní (dále *API*) dostupné přes internet, skrze které bude klientská aplikace moci pracovat s daty v databázi. Toto API musí poskytovat základní funkce pro práci s daty, tedy načtení všech entit daného typu, případně s možností filtrování. Pro entity, u kterých to dává smysl, vytváření nových entit a upravování existujících entit.

2.4.1 Možnosti realizace

Platforma .NET nabízí širokou škálu možností, jak realizovat aplikační server s námi požadovanou funkcionalitou. Tyto možnosti se liší jak v šíři svých

možností, tak zejména v náročnosti na implementaci. Při našem výběru jsme uvažovali tyto možnosti.

Vlastní implementace

Platforma .NET umožňuje implementovat server kompletně od základu vlastními silami. Takové řešení by nám poskytlo plnou kontrolu nad funkcionalitou našeho serveru a umožnilo vytvořit implementaci přesně dle našich požadavků, avšak tento přístup by byl velice náročný a zcela mimo rozsah této práce.

SuperSocket

SuperSocket je *„light weight“* framework pro serverové aplikace umožnující obousměrnou komunikaci mezi klientem a serverem (viz dokumentace [8]). Framework je velmi dobře rozšiřitelný včetně možnosti implementace vlastního komunikačního protokolu. Na druhou stranu je tento framework pro naše účely příliš nízkoúrovňový, a protože nám postačuje pasivní server, tak bychom nedokázali plně využít jeho možnosti.

.NET Remoting

.NET Remoting je framework pro „zveřejňování objektů“, umožňuje pracovat s objekty napříč aplikačními doménami, procesy, ale i počítači. Poskytuje tedy vysokou úroveň abstrakce, protože volání metod na vzdálených objektech je stejné jako na „lokálních“ objektech. Toto je vhodné pro naše účely, kdy požadujeme pouze jednoduchý server pro přístup k datům. V dnešní době je však tento framework považován za zastaralý.

WCF

WCF (*Windows Communication Foundation*) je framework zajišťující komunikaci mezi aplikace a umožňující vytvářet servisně orientované aplikace. Plně nahrazuje zmíněný .NET Remoting a poskytuje stejnou úroveň abstrakce.

Framework funguje tak, že umožňuje zveřejňovat kolekci služeb, kde každá služba má jeden či více koncových bodů, které se skládají z následujících částí:

- Adresa – kde se koncový bod nachází (např. HTTP adresa).
- Binding – specifikace spojení, určuje zejména přenosový protokol (TCP, HTTP atd.), kódování (text, binární data) a požadavky na zabezpečení.
- Kontrakt – určuje co služba „nabízí“, definuje se v kódu jako rozhraní (ve smyslu jazyka `C#`) a tím určuje jaké metody je možné na službě volat, včetně jejich parametrů a návratových hodnot.

WCF také nabízí vestavěná řešení pro zabezpečení komunikace a pro autentizaci a autorizaci uživatelů.

Rozhodnutí

Na základě provedené analýzy jsme se rozhodli implementovat náš server pomocí frameworku WCF. Pro naše účely nabízí správnou úroveň abstrakce a také poskytuje vestavěná řešení pro řadu našich problémů. Dále je také pro nás výhodné, že je možné jej hostovat na webovém serveru IIS (*Internet Information Services*), který už je přítomný na serveru partnerské firmy.

2.4.2 Tvorba API ve WCF

Jak jsme zmínili v předchozí kapitole, tak API se ve WCF implementuje pomocí služeb. Nejdůležitější částí každé služby je kontrakt, jehož realizace se skládá z rozhraní, které definuje, jaké metody služba poskytuje, a implementace tohoto rozhraní, kde bude samotná funkcionalita služby.

Rozhraní, které službu definuje je standartní rozhraní jazyka `C#`, ale navíc je nutné přidat několik atributů. Prvním je atribut `ServiceContract`, kterým označíme celé rozhraní. Jednotlivé metody rozhraní pak musíme označit atributem `ServiceContract`. Dále je nutné, aby třídy, které používáme jako parametry nebo návratové hodnoty, byly serializovatelné, k tomu poskytuje `C#` několik možností, přičemž v případě WCF je doporučováno použít atributy `DataContract` a `DataMember`. Ukázku jednoduchého rozhraní pro službu ve WCF i se zmíněnými atributy můžeme vidět na ukázce kódu 2.1.

Když víme, jak realizovat kontrakt pro WCF službu, tak můžeme provést první návrh, jak budou vypadat naše služby. Na začátku jsme zvažovali tři možné přístupy. Za prvé pro každou entitu se, kterou budeme chtít pracovat, tak vytvoříme samostatnou službu, což by vedlo k poměrně vysokému počtu služeb a jejich koncových bodů, ale na druhou stranu budou implementace služeb menší. Druhou zvažovanou variantou bylo mít pro každou klíčovou entitu a její související entity (viz návrh schématu 2.3.1) jednou službu, čímž bychom dostali celkem pět služeb, což už považujeme za rozumný počet. Poslední, třetí možnost je, že budeme mít jednou velkou službu, což by bylo nejjednodušší na správu konfigurace, ale výsledná implementace bude velká a tedy nepřehledná.

Nakonec, protože se nám žádná z těchto možností nezdála optimální, tak jsme se rozhodli vrstvu aplikačního serveru rozdělit na dvě podvrstvy. Na nižší vrstvě bude pro každou významnou entitu jedna služba, které pro tuto entitu bude implementovat základní operace, a na vyšší vrstvě pak bude jen jedna služba, která bude poskytovat výsledné API pomocí služeb nižší vrstvy. Tím získáme možnost snadno využívat služby nižší vrstvy opakovaně na více místech, například přečtení entity cesta za účelem kontroly její existence můžeme využít jak při zakládání transferu, tak při zakládání položky ceníku pro cestovní agenturu. Vyšší vrstva bude dále zodpovědná za zachycení a zpracování chyb, to znamená jejich zaznamenávání a návrat chybové zprávy z API.

Tento dvouvrstvý návrh sice vede na větší počet tříd, ale výsledkem je přehlednější kód, kde se žádná část kódu zbytečně neopakuje.

Později jsme do toho návrhu přidali ještě druhou API službu, která slouží k distribuci pluginů. Zde jsme se rozhodli pro samostatnou službu, protože její funkce je zcela nezávislá na funkci služby první. Navíc tato služba nepoužívá zabezpečení a proto potřebuje vlastní konfiguraci.

```
namespace WcfServiceLibrary1
{
    [ServiceContract]
    public interface IService1
    {
        [OperationContract]
        string GetData(int value);

        [OperationContract]
        CompositeType GetDataUsingDataContract(
            CompositeType composite);

    }

    [DataContract]
    public class CompositeType
    {
        bool boolValue = true;
        string stringValue = "Hello ";

        [DataMember]
        public bool BoolValue
        {
            get { return boolValue; }
            set { boolValue = value; }
        }

        [DataMember]
        public string StringValue
        {
            get { return stringValue; }
            set { stringValue = value; }
        }
    }
}
```

Ukázka kódu 2.1: Rozhraní kontraktu pro WCF službu

2.4.3 Autentizace a autorizace ve WCF

V naší aplikaci budou uložena citlivá data, a proto je nutné ji zabezpečit. K tomu bude nutné naimplementovat mechanismus autentizace, tedy ověření identity volajícího služby. Dále, protože jsme se rozhodli rozlišovat dvě role uživatele, administrátor a operátor (viz kapitola 2.2), tak musíme také naimplementovat mechanismus pro autorizaci, tedy ověření, zda má volající povolení požadovanou akci provést.

Autentizace

WCF nabízí podporu pro řadu autentizačních mechanismů, přičemž ty nejpoužívanější jsou Windows autentizace, `X509` certifikát a uživatelské jméno a heslo. My budeme používat poslední jmenovaný, protože je nejvhodnější pro aplikace fungující přes internet.

Mechanismus pro přihlašování uživatelským jménem a heslem, můžeme využít

tak, že naimplementujeme potomka třídy `MembershipProvider`, přičemž je pro nás důležitá hlavně metoda `ValidateUser`, ve které můžeme příchozí uživatelské jméno a heslo zkontrolovat a povolit či zamítnout přístup.

Tyto údaje budeme kontrolovat proti údajům v databázi, přičemž heslo dle běžných doporučení budeme ukládat hashované. Při vytváření hashe použijeme takzvanou *„sůl“*, což na náhodný textový řetězec přidaný k heslu. Tím zajistíme, že i pokud si dva uživatelé zvolí stejné heslo, tak bude výsledná hash odlišná. K vytváření hashe použijeme kryptografickou funkci PBKDF2, jejíž implementace je dostupná ve třídě `Rfc2898DeriveBytes`. Byty hashe a *„soli“* budeme do databáze ukládat jako jeden textový řetězec vytvořený pomocí kódování base64.

Autorizace

Pro autorizaci poskytuje WCF vestavěný mechanismus, pro jehož použití musíme stejně jako v předchozím případě vytvořit potomka předdefinované třídy, v tomto případě třídy `RoleProvider`. Zde jsou pro nás důležité dvě metody, první se jmenuje `IsUserInRole` a zajišťuje ověření, že uživatel, identifikovaný pomocí uživatelského jména, je v dané roli. Druhá důležitá metoda se jmenuje `GetRolesForUser` a vrací všechny role, které má daný uživatel.

Protože máme pouze dvě uživatelské role, stačí nám v databázi pouze ukládat příznak, zda je daný uživatel administrátor. Toto řešení není sice do budoucna snadno rozšiřitelné, ale lépe vyhovuje naší situaci, kdy každý uživatel je operátor a jen někteří uživatelé jsou současně také administrátoři. Tedy jediná informace, která nás při autorizaci zajímá je, zda je volající administrátor.

Poté, co jsme nad touto datovou strukturou naimplementovali vlastní `RoleProvider`, tak je nasazení autorizace snadné. V implementaci kontraktu použijeme atribut PrincipalPermission a nastavíme SecurityAction na demand, což znamená, že požadujeme, aby volající uživatel byl v nastavené roli. A dále pak nastavíme, kterou roli požadujeme. Příklad použití v našem kódu můžeme vidět na ukázce kódu 2.2.

```
[PrincipalPermission(
    SecurityAction.Demand,
    Role = IsopRoleProvider.RoleAdmin)]
public IsopUserResult AddUser(IsopUser user, string password)
{ ... }
```

Ukázka kódu 2.2: Použití autorizace

2.4.4 Konfigurace WCF

Framework WCF řeší řadu problémů za nás, a to zejména v oblasti přenosu dat přes síť a zabezpečení, takže jejich řešení nemusíme implementovat sami, ale současně s tím přichází nutnost nastavit správnou konfiguraci, aby server fungoval dle našich požadavků. Při práci s WCF jsme museli provést následující konfigurace.

Binding

Pro naše účely, zabezpečený přenos dat přes veřejný internet, jsou nejvhodnější bindingy *NetTcpBinding* a *WSHttpBinding*. První z nich je určen pouze pro WCF-WCF komunikaci, ale je lépe optimalizován, zejména protože se pracuje s binárními daty namísto textových dat. Druhý implementuje webové standarty WS-* (WS-Addressing, WS-Security a WS-ReliableMessaging), což zaručuje interoperabilitu s technologiemi mimo WCF. Protože se jedná pouze o konfiguraci, a ne o implementaci, tak jsme zvolili optimálnější NetTcpBinding a pokud někdy budeme potřebovat kompatibilitu kterou nabízí druhá možnost, tak můžeme konfiguraci změnit nebo používat oba bindingy současně.

Při nasazení v praxi jsme narazili na problém, že při použití bindingu NetTcpBinding se z některých míst nebyli schopni připojit. Tyto problémy vyřešilo použití bindingu WSHttpBinding. Domníváme se, že příčinou těchto problémů bylo blokování některých TCP portů, které neovlivnilo WSHttpBinding binding, protože používá standardní HTTP port 80.

Zabezpečený binding a vývoj

Pro náš případ zřejmě potřebujeme zabezpečený způsob přenosu dat, ale protože implementace autentizace uživatelů byla v plánu až jako jedna z posledních položek vývoje, tak jsme během vývoje používali nezabezpečený binding BasicHttpBinding. S tímto binding-em je také výrazně jednodušší používat WCF Test client, testovacího klienta pro služby WCF, který umožňuje volat jednotlivé metody kontraktu a číst vrácené hodnoty. To umožňilo nezávislé testování funkčnosti služby před propojením s naší klientskou aplikací.

Problém s velikostí dat

Během vývoje jsme narazili na problém, že občas při přenosu většího seznamu transferů dojde k výjimce. Studiem výjimky jsme zjistili, že se pokoušíme přenést větší objem dat, než nastavení našeho binding-u dovoluje. Proto jsme konfiguraci binding-u doplnili o nastavení *readerQuotas*, které určuje mimo jiné maximální velikost přenášených dat.

2.4.5 Přístup k databázi

Při návrhu aplikačního serveru jsme řešili jak z kódu přistupovat k databázi. Na platformě .NET pro tento účel existuje řada ORM (Object-Relation Mapping) frameworků, které umožňují pracovat s relačními daty v objektovém jazyce jakým je `C#`. My jsme při výběru brali v úvahu tři nejpoužívanější:

Entity framework – (dále EF) je rozsáhlý ORM framework vyvíjený společností Microsoft jako součást platformy .NET. Umožňuje pracovat s databází bez psaní SQL skriptů, a to včetně vytvoření databázové schématu podle entit definovaných jako třídy v `C#`, což umožňuje odstranit závislost na konkrétním databázovém serveru.

NHibernate – je port ORM frameworku Hibernate pro platformu .NET. Tento framework je populární v prostředí programovacího jazyka Java a poskytuje v zásadě stejnou funkcionalitu jako EF.

Dapper.NET – je „micro“ OMR framework, který na rozdíl od předchozích dvou, poskytuje pouze omezenou funkcionalitu, zejména se jedná o provádění SQL dotazů a mapování výsledků na objekty. Výhodou je, že poskytuje v mnoha situacích výrazně vyšší výkon, protože programátor si píše SQL dotazy sám a nejsou generovány strojově jako v předchozích případech.

Dapper.NET nabízí zajímavou alternativu k velkým ORM frameworků, ale pro naši situaci, kdy neočekáváme velkou zátěž databáze, protože dotazy vznikají pouze na základě akcí uživatelů, a navíc i počet uživatelů bude omezený, tak nepředstavuje vhodnou volbu. Jeho výhoda v rychlosti by nebyla využita, a navíc by znamenal nutnost ručního psaní SQL dotazů.

Nakonec jsme se rozhodli použít EF, protože z pohledu našich potřeb je ekvivalentní s NHibernate, a autor s ním má předchozí dobré zkušenosti.

Při používání EF jsme narazili na to, že entity, které jsme používali nebylo možné přenášet pomocí WCF služby. Problém byl v tom, že u EF používáme takzvané generování proxy tříd. To znamená, že EF pro námi definované entity vygeneruje jejich potomky, kteří obsahují logiku, která umožňuje sledovat změny hodnot jednotlivých vlastností. To nám umožňuje načíst jednu nebo více entit, provést požadované změny a pak jen metodou SaveChanges, všechny změny uložit a EF se už postará o vše potřebné.

Existence proxy tříd však vede k tomu, že naše entity není možné serializovat pro přenos pomocí WCF služby. Protože jsme chtěli nadále používat proxy třídy, tak jsme se rozhodli k entitám vytvořit DTO (Data Transfer Object) třídy, které jsou v často shodné s databázovými entitami a slouží pro přenos dat. Později jsme je dále upravili pro použítí jako modely v návrhovém vzoru MVVM v klientské aplikaci (viz kapitola 2.5.2)

Tento problém nás současně přivedl ke správnému návrhu vrstev, kde databázová vrstva (reprezentovaná databázovými entitami a EF frameworkem) je oddělena od vrstvy WCF služeb (reprezentovaná DTO třídami). Toto oddělení nám dává nezávislost WCF služby, s jejíž objekty pracujeme v klientské aplikaci, na struktuře databáze.

Nevýhodou tohoto řešení je, že muselo vzniknout velké množství kódu na mapování mezi entitami a DTO třídami, kde se jen kopírují hodnoty proměnných. Tato nevýhoda by mohla být odstraněna použitím frameworku *AutoMapper*, který je přesně pro tuto situaci určen, ale jeho existence nám v době implementace nebyla známa.

2.5 Klient

Klientská aplikace představuje hlavní část našeho systému, protože v ní bude implementována většina logiky, a navíc řada jejích funkcí bude rozšiřitelná pomocí pluginů.

Primárním úkolem naší aplikace bude umožnit uživateli pracovat s daty v databázi, tak abychom dosáhli naplnění požadavků P2 až P5 na náš informační systém.

Pří návrhu aplikace se musíme soustředit na to, že naši uživatelé budou často technicky méně zkušení uživatelé a celé grafické rozhraní naší aplikace by tedy mělo být maximálně jednoduché a intuitivní. Důležitá bude také konzistence návrhu, tedy například pokud se destinace editují jedním způsobem, měli by se

podobným způsobem editovat také cesty a další související entity. To umožní uživatelům se rychle naučit pracovat s naším systémem, což je pro firmu využívající náš systém důležité i z ekonomického hlediska.

2.5.1 Framework pro grafické rozhraní

Na platformě .NET jsou pro tvorbu grafických desktopových aplikací dostupné tři běžně používané frameworky, od nejstaršího k nejnovějšímu to jsou WF (*Windows Forms*), WPF (*Windows Presentation Foundation*) a UWP (*Universal Windows Platform*).

Jako první jsme zkoumali UWP, protože se jedná o nejnovější ze tří zmíněných frameworků. Protože však UWP podporuje v době našeho rozhodování pouze operační systém Windows 10 a my dle cíle práce P7 chceme využít existující počítačové vybavení, které používá starší verze operačního systému Windows, tak jsme tento framework vyřadili z výběru jako první.

Protože zbývající dva frameworky nám z pohledu podporovaných operačních systému vyhovují, tak jsme se snažili analyzovat jaké komponenty budeme při vytváření naší aplikace potřebovat abychom se mezi nimi mohli rozhodnout. V naší aplikaci bude klíčová práce s transfery, určitě budeme chtít zobrazovat jejich seznamy, které budou reprezentovat denní přehledy nebo výsledky hledání. Navíc budeme chtít tyto transfery seskupovat (viz návrh schéma pro transfery) a seřazovat. Celkově budeme často zobrazovat tabulková data a k tomu by bylo vhodné mít k dispozici komponentu, která nám to snadno umožní. Windows Forms poskytují komponentu DataGridView a WPF nabízí komponentu DataGrid, po zběžném prostudování dokumentace k oběma zmíněným komponentám věříme že obě by bylo možné použít v našem případě.

Při studiu zmíněné dokumentace jsme narazili na to, že WPF poskytuje dva zcela odlišné přístupy k propojení uživatelského rozhraní definovaného v jazyce XAML a kódu v jazyce `C#`. První z těchto přístupů se označuje jako událostmi řízený (*„Event Driven“*), a jedná se o „klasický“ přístup, který používá také WF. Každá komponenta zveřejňuje sadu událostí (*„Events“*) a tyto události můžou být obsluhovány ve třídě, které reprezentuje dané okno nebo složenou komponentu. Tato třída je umístěna v *„code behind“*, což je soubor se `C#` kódem, který je vázán na příslušný XAML soubor.

Druhý, modernější, přístup používá návrhový vzor MVVM (*Model-View-ViewModel*). Základní myšlenka MVVM je vytvořit vrstvu, která oddělí datovou vrstvu od vrstvy uživatelského rozhraní. Toto řešení přináší snadnější údržbu, rozšiřitelnost a testovatelnost.

Protože se v naší aplikaci soustředíme na kvalitu uživatelského rozhraní (viz 2.5), tak očekáváme, že ho při vývoji budeme často měnit. Pro tyto situace je vhodnější druhý z přístupů, díky oddělení logiky od grafické reprezentace. Volíme tedy framework WPF, zejména kvůli možnosti použít MVVM.

Ohlédnutí

Dle očekávání jsme během vývoje několikrát zásadně přepracovávali naše uživatelské rozhraní a použití návrhového vzoru MVVM se ukázalo být velice důležité. Pokud bychom používali klasický událostmi řízený přístup, tak by změny

v uživatelském rozhraní byly daleko náročnější a pravděpodobně bychom od nich museli upustit a ponechat uživatelské rozhraní v nevyhovujícím stavu.

2.5.2 Návrhový vzor MVVM ve WPF

Použití MVVM hrálo v naší aplikaci důležitou roli a považujeme za škodu, že je tento přístup často opomíjen ve prospěch klasického událostmi řízeného přístupu. Dokonce jsme narazili na řadu bakalářských prací, v nichž je nepoužití MVVM ve WPF označovaná za pochybení, a proto bychom se chtěli použití MVVM více věnovat.

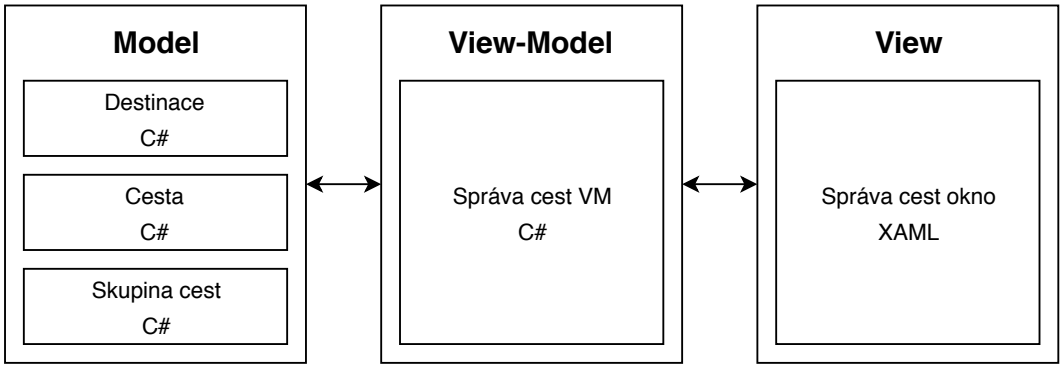
Vrstvy MVVM

Model – reprezentuje datovou vrstvu, v našem případě se jedná zejména o data získaná prostřednictvím klienta pro komunikaci s aplikačním serverem a s ním spojené entity (transfer, řidič atd.). Model neví nic o existenci či stavu uživatelského rozhraní.

View – definuje uživatelské rozhraní a reaguje na akce uživatele (kliknutí myší, stisk klávesy atd.). Ve WPF se zapisuje v deklarativním jazyce XAML a související „code behind“ by měl být prázdný nebo minimální. Reference na ViewModel se uchovává ve vlastnosti DataContext.

ViewModel – spojuje Model a View a je zde realizována business logika aplikace. Nemá přímou referenci na View, ale na místo toho vystavuje sadu vlastností a příkazů, které jsou s View propojeny pomocí bindingu. Ve WPF je binding realizován pomocí Binderu, který je součástí jazyka XAML.

Vrstvy MVVM a jejich propojení ilustruje obrázek 2.8.



Obrázek 2.8: MVVM – příklad pro okno pro správu cest

Binding

Binding realizuje propojení vrstev View a ViewModel tak, že propojuje vlastnosti těchto vrstev. Toto propojení může být obousměrné nebo jednosměrné. Pokud chceme, aby propojení fungovalo ve směru z ViewModelu do View, tak je nutné, aby ViewModel implementoval rozhraní `INotifyPropertyChanged`, pomocí kterého informuje binder, že došlo ke změně hodnoty a ten může tuto změnu propagovat do View. Dále pokud je daná vlastnost kolekce, tak tato kolekce musí

implementovat rozhraní InotifyCollectionChanged [9]. Pro tento účel je k dispozici generická kolekce `ObservableCollection`, které se chová stejně jako často používaná kolekce `List`.

V souvislosti s bindingem jsme řešili, jak zobrazovat kolekce seřazené, například seřadit transfery podle skupiny cest a poté podle času vyzvednutí. Zjistili jsme, že pokud kolekci seřadíme dle našich požadavků, tak binding pořadí zachová. Problém však nastal při vložení nové položky, aby byla vložena na správné místo museli jsme vyhledat index pozice kam patří a vložit ji na tuto pozici, to vedlo k oddělení samotné kolekce od logiky na vložení nové položky. To je způsobeno tím, že ObservableCollection je určena pro neseřazená data. Přirozeně jsme hledali variantu pro práci se seřazenými daty, podobně jako pro `List` existuje `SortedList`, taková varianta ale bohužel není k dispozici.

Protože potřebujeme seřazená data zobrazovat hned na několika místech, tak jsme se rozhodli pokusit o vlastní implementaci seřazené kolekce implementující rozhraní `INotifyCollectionChanged`. Nahlédnutím do implementace ObservableCollection jsme zjistili, že se jedná jen o obálku nad zmiňovanou kolekcí List. Pokusili jsme se o podobné řešení nad kolekcemi SortedList a SortedSet, ale ani v jednom z případů jsme neuspěli. Problém je, že návrh rozhraní nepočítá s použitím pro seřazená data a pracuje s nimi pomocí indexů a při vložení nové položky může docházet ke změně až všech těchto indexů, proto tuto událost musíme hlásit jako *„reset“*, což vede k tomu že, se všechny položky bindují znova a vytvářejí se pro ně nové vizuální elementy, které je reprezentují. Domníváme se, že toto omezení rozhraní `INotifyCollectionChanged` je důvodem proč hledaná *„SortedObservableCollection“* neexistuje.

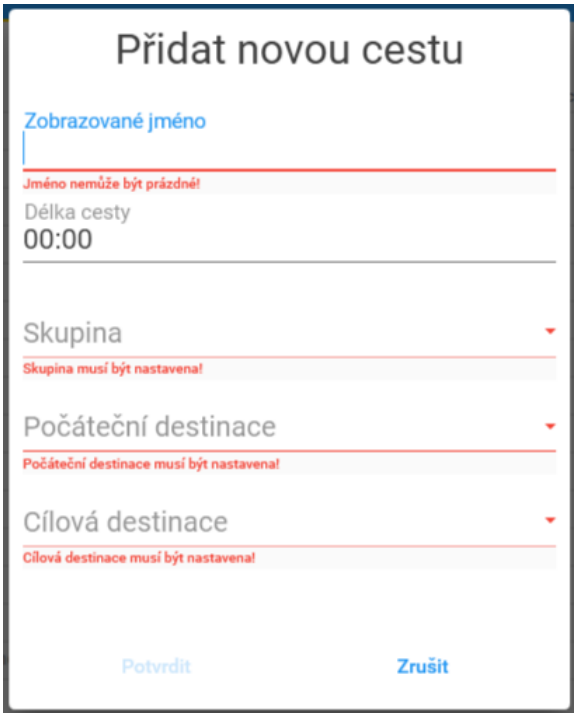
Později se nám podařilo nalézt řešení, které používá třídu `CollectionView`, což je právě třída, která má za úkol řešit omezenou podporu práce s kolekcemi při bindingu. Umožňuje snadno implementovat filtrování, seřazování a seskupování jako další vrstvu nad existující kolekcí.

Modely

Jako modely používáme DTO třídy z aplikačního serveru. Původně jsme zvažovali vytvoření další sady tříd, na které budeme DTO třídy mapovat a používat je jako modely, podobně jako je to u databázových entit a DTO, ale nakonec se ukázalo jedno oddělení jako dostatečné.

Tyto DTO třídy jsme museli pro použití v návrhovém vzoru MVVM doplnit o implementaci rozhraní INotifyPropertyChanged, a navíc jsme také přidali implementaci rozhraní `INotifyDataErrorInfo`. Toto rozhraní slouží k propojení se systémem pro zobrazování chybových hlášek ve WPF, jako například že je dané pole povinné. To velice usnadňuje implementaci zobrazování chybových hlášek v uživatelském rozhraní.

Rozhraní `INotifyDataErrorInfo` je založeno na tom, že u každé proměnné se ukládá seznam chyb v podobě textových řetězců a tento seznam se aktualizuje při každé změně hodnoty odpovídající proměnné. Tento seznam chyb, pokud není prázdný, se pak zobrazuje u odpovídající komponenty uživatelského rozhraní, na kterou je proměnná navázána pomocí bindingu. Výsledek použití tohoto konceptu můžeme vidět na obrázku 2.9, přičemž pro dosažení tohoto výsledku jsme pouze museli implementovat zmíněné rozhraní.



Obrázek 2.9: MVVM – chybová hláška

2.5.3 Material design

Náš systém má komerční ambice, a protože kromě samotných funkcí zákazníci také často berou v potaz vzhled, tak jsme hledali způsob, jak vylepšit už poněkud *„okoukaný“* standardní vzhled WPF aplikace.

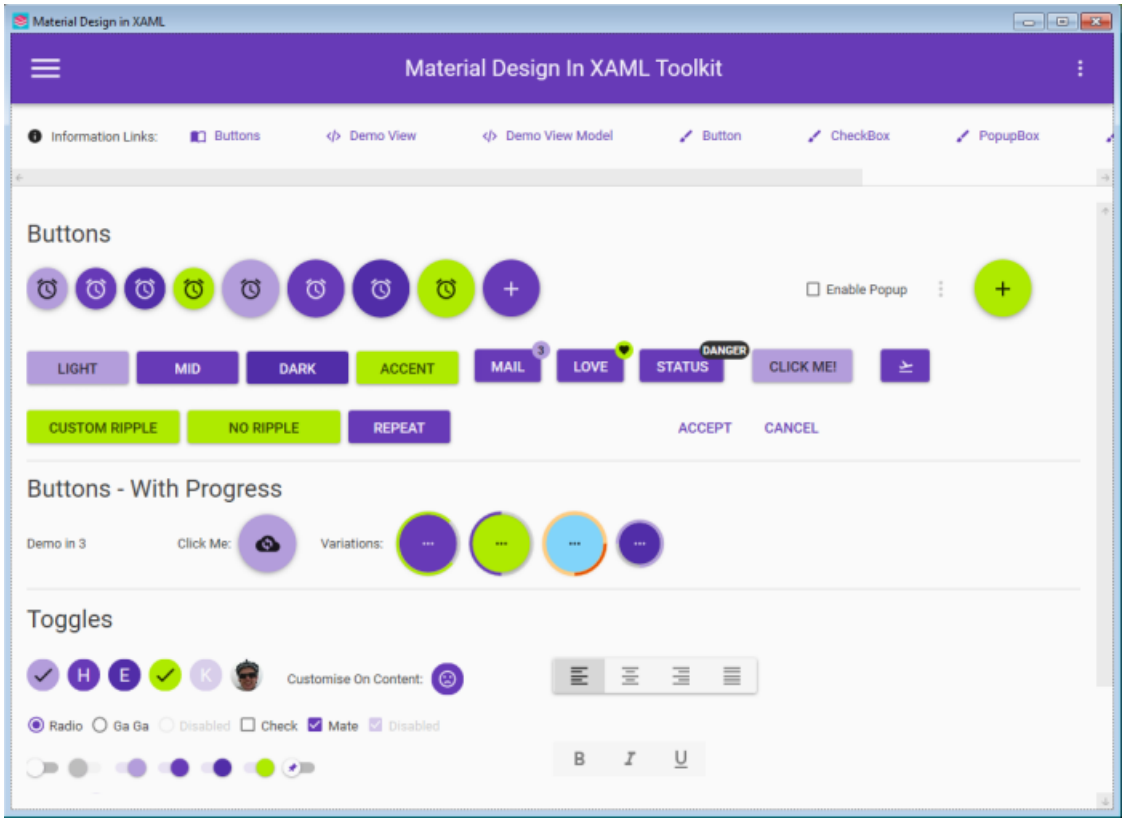
Autorovy se osobně líbí styl Material Design, jehož autorem je společnost Google. Při hledání fráze *„WPF Material Design“* jsme narazili na povedený framework Material Desing in XAML. Tento framework poskytuje *„material design style“* pro většinu komponent ve WPF a výrazně tak mění vzhled WPF aplikace (viz Obrázek 2.9 a 2.10). Obrázek však nedokáže ilustrovat animace a další podobné součásti stylu, a proto v případě zájmu doporučujeme navštívit webové stránky projektu [10], kde lze kromě samotného frameworku najít také aplikaci, která demonstruje všechny vytvořené styly, a z níž pochází ukázka na obrázku 2.10.

Použití tohoto frameworku hodnotíme velice pozitivně, protože dal našemu systému výrazně lepší vzhled, který zaujal řadu lidí a vyvolal zájem o naší aplikaci. Pozitivně vzhled aplikace hodnotilo vedení partnerské firmy s tím, že aplikace vypadá moderně a profesionálně, což zvýšilo jejich důvěru v náš produkt.

Kromě stylování existujících WPF komponent framework také nabízí několik vlastních komponent, například tlačítko s více akcemi, karty, dialogy anebo hodiny pro výběr času. Dále také poskytuje celou sadu standardních *„Material Design“* ikon, které lze snadno použít přímo v XAML kódu.

2.5.4 Kontext

Při implementaci klientské aplikace jsme brzy narazili na potřebu sdílet zdroje, například nastavení nebo informace o přihlášeném uživateli, mezi jednotlivými



Obrázek 2.10: Ukázka aplikace frameworku Material Desing in XAML

okny aplikace, a proto jsme hledali způsob jak toho dosáhnout.

Na začátku jsme k řešení tohoto problému použili návrhový vzor Singleton a požadované zdroje jsme zpřístupnili jako globální statické proměnné, tedy tvz. *Singletony*. Pro každý sdílený zdroj jsme navíc vytvořili rozhraní aby nevznikla pevná vazba na konkrétní třídu, která tento zdroj poskytuje. Toto řešení však přineslo několik problémů. Prvním bylo že vytvořilo uvnitř aplikace skryté vazby a stav aplikace nebyl najednou určen nejen ViewModely jednotlivých oken. Dále také toto řešení velice omezilo možnosti případného testování, protože aplikace by mohla být testována pouze jako celek.

Dalším problémem také je, že pokud bychom chtěli tyto zdroje poskytnout některému z pluginů, tak by tento plugin musel referencovat klientskou aplikaci namísto malé sdílené knihovny a dále by také museli být tyto zdroje přístupné veřejně nejen pro čtení, ale také pro zápis.

Museli jsme také řešit kam jednotlivé singletony umístit, jednou možností bylo, aby byly vždy součástí třídy, která nastavuje jejich hodnotu, to by umožnilo mít právo zápisu omezené jen na danou třídu pomocí privátního *„Setteru“*. To by však znamenalo, že jednotlivé singletony by byly v různých třídách a tyto třídy by nebylo možné měnit.

Z těchto důvodů jsme se rozhodli všechny zdroje reprezentované singletony zpřístupnit z jednoho místa kterému říkáme kontext. Toto řešení nás později dovedlo k návrhovému vzoru Dependency Injection, kde se obrací způsob, jakým jednotlivé ViewModely získávají sdílené zdroje. Namísto toho, aby se ViewModely staraly o získání těchto zdrojů, tak je vyžadují ve svém konstruktoru a ten kdo

daný ViewModel vytváří, tak musí zajistit jejich dodání.

Tento přístup nám dal kontrolu nad přístupem ke sdílenému kontextu, jednotlivá okna ho předávají do oken, která vytváří. Kontext vytváříme při startu aplikace a postupně do něj můžeme přidávat další zdroje. Tento přístup nám také umožňuje vytvářet pluginy, kterým při inicializaci předáme tento kontext a tím jim dáme přístup ke všem zdrojům klientské aplikace.

Tento přístup také výrazně zlepšuje možnosti testování, protože nám umožňuje testovat jednotlivé ViewModely individuálně tak, že jim předáme testovací kontext.

3. Rozšiřitelnost pomocí Pluginů

Klientská aplikace našeho informačního systému je v mnoha směrech rozšiřitelná pomocí externích pluginů. Protože tato funkce může být zajímavá i pro čtenáře, které jinak problém osobní přepravy nezajímá, tak se problematice pluginů budeme věnovat v této samostatné kapitole.

Motivace

V oboru osobní přepravy (viz kapitola 1.1) prakticky neexistují standardy, a proto potřebujeme, aby náš systém mohl snadno reagovat na změny vstupních či výstupních formátů souborů. Toho jsme docílili tím, že všechny vstupy a výstupy jsou řešeny pomocí pluginů. Tato rozšiřitelnost umožňuje snadno upravovat systém, a to jak podle potřeb jednoho uživatele (např. zahájení spolupráce s novou cestovní agenturou s jinými požadavky), tak pro různé uživatele (např. různé interní požadavky). Použití pluginů také umožňuje snadno implementovat stejnou funkci různými způsoby (např. export faktury do různých formátů) podle aktuální potřeby.

3.1 Úvod do MEF

Na základě diskuse v kapitole 2.2.3 jsme se rozhodli pro implementaci plugin systému použít framework MEF (Managed Extensibility Framework)[11], který je standardní součástí platformy .NET.

Plugin systém se skládá ze dvou hlavních součástí, první z nich je aplikace, která jednotlivé pluginy importuje a používá, dále jen aplikace. Druhou součástí jsou pak samotné pluginy.

Implementace plugin systému pomocí frameworku MEF je poměrně jednoduchá a sestává se z následujících kroků.

1 - Rozhraní pluginu

Prvním a nejdůležitějším krokem je návrh rozhraní, skrz které bude plugin používán. Jedná se o standardní rozhraní ve smyslu jazyka `C#`. Toto rozhraní tedy sdílí aplikace a plugin samotný a umožňuje jejich vzájemnou komunikaci. Sdílení tohoto rozhraní se obvykle realizuje pomocí samostatné knihovny, ale plugin také může přímo referencovat aplikaci. Na ukázce kódu 3.1 můžeme vidět rozhraní, které budeme dále v našem příkladu používat.

```
public interface ICalculator
{
  String Calculate(String input);
}
```

Ukázka kódu 3.1: Rozhraní pluginu ICalculator

2 - Implementace pluginu

Plugin samotný se realizuje jako samostatný projekt dynamicky linkované knihovny, výsledkem po kompilaci je tedy `DLL` soubor. V tomto projektu implementujeme rozhraní pluginu z předchozího kroku, při implementaci postupujeme standardním způsobem. Jediným rozdílem je, že musíme naši implementaci označit atributem `System.ComponentModel.Composition.ExportAttribute`, kde argumentem je typ rozhraní pluginu, které implementujeme. Na ukázce kódu 3.2 můžeme vidět použití tohoto atributu při implementaci rozhraní pluginu z prvního kroku.

```
[Export(typeof(ICalculator))] class MySimpleCalculator
: ICalculator { }
```

Ukázka kódu 3.2: Export implementace pluginu

3 - Umístění instance pluginu

Dále musíme rozhodnout, kam má framework MEF uložil instanci importovaného pluginu. Vybranou proměnnou nebo vlastnost označíme atributem `System.ComponentModel.Composition.ImportAttribute`, kde argumentem je opět typ rozhraní pluginu. Příklad můžeme vidět na ukázce kódu 3.3, kde je také příklad podpory kolekce pluginů pro dané rozhraní.

```
[Import(typeof(ICalculator))] public ICalculator calculator;
[Import(typeof(ICalculator))]
public IEnumerable<ICalculator> calculator2;
```

Ukázka kódu 3.3: Proměnná pro import pluginu

4 - Kompozice

Posledním krokem je zajištění toho, že MEF do proměnné z předchozího kroku uloží instanci odpovídajícího pluginu. K tomu potřebujeme instanci třídy `CompositionContainer` [12] (dále označována jako *kontejner*). Tato třída zodpovídá za samotnou realizaci kompozice. Do kontejneru je nutné předat jednu nebo více instancí třídy `ComposablePartCatalog`, která reprezentuje zdroj, kde lze hledat implementace pluginů. Jedná se o abstraktní třídu, která má několik potomků. Nejčastěji se používá třída DirectoryCatalog, které reprezentuje adresář s `DLL` soubory obsahujícími samotnou implementaci pluginu. Nakonce už stačí jen na kontejneru zavolat metodu `ComposeParts` a předat jí instanci třídy, do které se mají umístit instance naimportovaných pluginů, viz předchozí krok.

3.2 Implementace plugin systému

V našem systému řeší práci s pluginy třída PluginManager, která je implementací rozhraní IPluginManager, a je součástí kontextu aplikace (viz kapitola

4.3.1). Rozhraní publikuje kolekce jednotlivých typů pluginů.

Import pluginů do odpovídajících kolekcí je řešen v metodě LoadPlugins a relevantní části kódu můžeme vidět na ukázce 3.4. Implementace vychází z postupu uvedeného v předchozí kapitole.

```
class PluginManager : IpluginManager {
  public ResourceBuildResult LoadPlugins(IIsopSettingManager setting) {
    var catalog = new AggregateCatalog();
    catalog.Catalogs.Add(new DirectoryCatalog(
  setting.Configuration.AppConfiguration.PluginPackageDir));
    var container = new CompositionContainer(catalog);
    container.ComposeParts(this);
  }
}
```

Ukázka kódu 3.4: Implementace importu pluginů

Jak můžeme vidět na ukázce kódu, tak pluginy se načítají z adresáře, kde jsou umístěny jako jednotlivé `DLL` soubory. Cesta k tomuto adresáři je součástí nastavení aplikace. Pluginy jsou do tohoto adresáře staženy ze služby pro práci s pluginy, která je součástí serveru (viz kapitola4.2.2)

Načítání externích knihoven

Při implementaci plugin systému jsme řešili problém, jak umožnit pluginům používat externí knihovny, které nejsou standardní součástí platformy .NET. Problémem je, že MEF zajišťuje pouze načtení `DLL` s pluginy a pokud tyto pluginy referencují nějako externí knihovnu, tak při jejím použití se ji pokusí běhové prostředí jazyka načíst. Pokud knihovna není součástí platformy .NET a ani není přítomná v adresáři odkud byla aplikace spuštěna, tak dojde k výjimce.

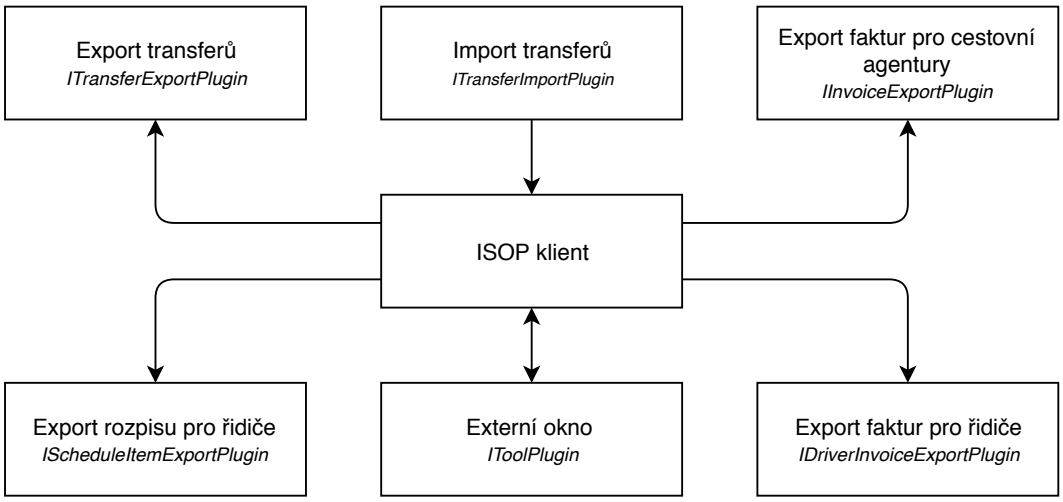
Tento problém jsme vyřešili implementací reakce na událost `AssemblyResolve` na instanci třídy `AppDomain` pro naši aplikaci, která je dostupná skrze vlastnost `AppDomain.CurrentDomain`. K vyvolání této události dochází ve chvíli kdy se nepodaří načíst požadovanou knihovnu. V metodě `LoadAssamblyFromPluginStore`, která na tuto událost reaguje se pokusíme požadovanou knihovnu nalézt v adresáři, kde se nachází soubory pluginů. Relevantní části naší implementace můžeme vidět na ukázce kódu 3.5. Toto řešení nám umožňuje potřebné externí knihovny snadno distribuovat společně s pluginy.

3.3 Plugin systém v klientské aplikaci

Plugin systém v naší klientské aplikaci řeší všechny vstupy a výstupy (viz kapitola 2.2.3) a jejich přehled spolu s odpovídajícími názvy rozhraní můžeme vidět na obrázku 3.1

```
public ResourceBuildResult LoadPlugins(IIsopSettingManager setting)
{
  ...
  var domain = AppDomain.CurrentDomain;
  domain.AssemblyResolve += LoadAssamblyFromPluginStore;
  ...
}
private Assembly LoadAssamblyFromPluginStore(
  object sender, ResolveEventArgs args)
{
  string assemblyPath = Path.Combine(
    _pluginPackageDir,
    AssemblyName(args.Name).Name + ".dll");
    if (!File.Exists(assemblyPath)) return null;
    var assembly = Assembly.LoadFrom(assemblyPath);
    return assembly;
}
```

Ukázka kódu 3.5: Načítání externích knihoven pro pluginy



Obrázek 3.1: Přehled pluginů a jejich rozhraní

Jak jsme diskutovali v předchozí kapitole, tak základem implementace plugin systému je návrh rozhraní, pomocí kterého aplikace komunikuje s pluginem. V našem systému máme několik různých typů pluginů a každý z nich je reprezentován vlastním rozhraním a všechna tato rozhraní mají společného předka `IPlugin`, který reprezentuje plugin jako takový. Strukturu pluginů a jejich jednotlivé vlastnosti si postupně projdeme v následujícím textu.

IPlugin

Rozhraní `IPlugin` umožňuje identifikaci pluginů a práci s nastavením pluginů, které podrobněji diskutujeme v další podkapitole. Rozhraní má následující položky:

- PluginId: int – unikátní identifikátor pluginu.

- Name: string – jméno pluginu, identifikace pluginu pro uživatele.
- Description: string – popis pluginu a jeho funkce.
- Version: int – verze pluginu.
- GetDefaultSetting(): Dictionary<string, string> - metoda, která vrací defaultní nastavení pluginu.
- SetSettingDictionary(Dictionary<string, string> setting): void – předání nastavení do pluginu.
- GetUiSettingControl(): IPluginUiSetting – metoda, která vrací instanci objektu uživatelského rozhraní, pomocí kterého může uživatel upravovat nastavení pluginu. Tato funkce je volitelná a pokud ji plugin nepodporuje, tak vrací null. Tento objekt je popsán rozhraním IPluginUiSetting, které má následující položky:
  - SetCurrentSetting(Dictionary<string, string> setting): void – nastaví nastavení pluginu do uživatelského rozhraní.
  - GetCurrentSetting(): Dictionary<string, string> - metoda, která vrací nastavení z uživatelského rozhraní.

Nastavení pluginů

Nastavení pluginu je reprezentováno jako slovník textových klíčů a hodnot. Ukládání nastavení zajišťuje Správce nastavení, který je součástí kontextu (viz kapitola 4.3.1).

Načtení nastavení probíhá tak, že napřed se z pluginu načte defaultní nastavení, poté je toto nastavení aktualizováno hodnotami ze Správce nastavení. Toto řešení umožňuje snadno v nových verzích pluginu přidávat nové položky nastavení, které jsou tak při načítání nastaveny na defaultní hodnoty. Díky tomu nenastane stav s chybějícími hodnotami, jak by tomu bylo v případě, kdy by se defaultní nastavení nenačítalo.

Import transferů - ITransferImportPlugin

Pluginy implementující rozhraní `ITransferImportPlugin` slouží k importu transferů do systému. Toto rozhraní má následující položky:

- MenuHeader: string – název, pod kterým se plugin zobrazuje v menu v hlavním okně klienta.
- Initialize(...): bool – inicializace pluginu, kterou je nutné zavolat před prvním použitím pluginu. Slouží k předání dat načítaných z databáze do pluginu, jedná se o seznam cestovních agentur, cest a služeb.
- Current: ImportTransferResult – aktuálně zpracovávaný transfer spolu se zdrojovým textem, ze kterého byl transfer načten.
- MoveNext(): bool – načte další transfer do položky Current a vrací zda se načtení povedlo.

- Reset(): void – volá se vždy před použitím pluginu k vynulování předchozího stavu.

Export transferů - ITransferExportPlugin

Pluginy implementující rozhraní `ITransferExportPlugin` slouží k exportu transferů ze systému, například do souboru. Toto rozhraní má následující položky:

- Export(...) - přijímá kolekci transferu a parametry hledání, které tyto transfery vrátilo.

Export faktur pro cestovní agentury - IInvoiceExportPlugin

Pluginy implementující rozhraní `IInvoiceExportPlugin` slouží k exportu faktur pro cestovní agentury ze systému, aby mohly být odeslány k proplacení. Toto rozhraní má následující položky:

- Export(...) - přijímá fakturu pro cestovní agenturu, která se exportuje, a dále informace o společnosti dle nastavení systému.

Export přehledu transferů pro řidiče - IDriverInvoiceExportPlugin

Pluginy implementující rozhraní `IDriverInvoiceExportPlugin` slouží k exportu přehledu transferů pro řidiče ze systému. Toto rozhraní má následující položky:

- Export(...) - přijímá přehled transferů řidiče k exportu.

Export rozpisu pro řidiče - IScheduleItemExportPlugin

Pluginy implementující rozhraní `IScheduleItemExportPlugin` slouží k exportu rozpisu pro řidiče ze systému. Toto rozhraní má následující položky:

- Export(...) - přijímá rozpis řidiče k exportu.

Plugin pro rozšíření systému o další okna - IToolPlugin

Pluginy implementující rozhraní `IToolPlugin` slouží k rozšíření systému o další okna, implementují rozšiřující funkce, například funkce specifické pro danou firmu, které by nebyly užitečné pro ostatní. Toto rozhraní má následující položky:

- GetToolWindow(...): object - přijímá kontext klientské aplikace (viz kapitola 4.3.1) a vrací okno k zobrazení uživateli.
- MenuHeader: string – jméno, pod kterým je plugin zobrazen v hlavním menu aplikace.

3.4 Sada demonstračních pluginů

Pro demonstraci toho jak jednotlivé pluginy implementovat, a také toho jak je v systému použít, jsme připravili sadu jednoduchých pluginů. Tyto pluginy jsou součástí našeho řešení, které se nachází v příloze A.1.

Náhodný transfer - RandomTransferImportPlugin

Jednoduchý plugin pro import transferů, které jsou náhodně generované, slouží k testování či předvedení funkcionality importovaní a zpracování transferů.

Outlook export - OutlookScheduleItemExportPlugin

Plugin pro export rozpisu řidiče do aplikace Microsoft Outlook, kde je vytvořen email, do jehož těla je vygenerován exportovaný rozpis. Dále je také předvyplněn příjemce emailu emailem řidiče podle rozpisu a předmět emailu jménem řidiče a datem prvního transferu.
Upozornění: Tento plugin vyžaduje aby na počítači byla nainstalována a nakonfigurována aplikace Microsoft Outlook (testováno pro verzi 2017).

Kontrola rezervací - CheckToolPlugin

Tento plugin přidává do systému nové okno, kde může uživatel otevřít XML soubor se seznamem rezervací a ten porovnat se stavem systému.

Excel Export - ExcelTransferExportPlugin

Plugin pro export transferů z hlavního okna aplikace do souboru aplikace Excel. Během vývoje systému sloužil k exportu transferů ze systému k ručnímu vytvoření rozpisu pro řidiče.
Upozornění: Tento plugin vyžaduje aby na počítači byla nainstalována a nakonfigurována aplikace Microsoft Excel (testováno pro verzi 2017).

Řidiči - Excel export - ExcelDriverInvoiceExportPlugin

Plugin pro export přehledu pro řidiče do souboru aplikace Excel.
Upozornění: Tento plugin vyžaduje aby na počítači byla nainstalována a nakonfigurována aplikace Microsoft Excel (testováno pro verzi 2017).

Export faktury do CSV - CsvInvoiceExportPlugin

Tento plugin slouží k exportu faktury pro cestovní agenturu do souboru `CSV`. Tento plugin vychází z pluginu používaného v praxi, kde po exportu je `CSV` soubor nahrán do systému cestovní agentury skrze webové rozhraní.

Import transferů z Gmail - GmailTransferImportPlugin

Komplexní plugin, ze kterého vychází všechny pluginy na import transferů používané v praxi. Tento plugin importuje transfery z těla nebo příloh emailů, které načítá z Gmail účtu.

Pro práci s Gmail účtem používáme sadu knihoven přímo od společnosti Google, jejichž dokumentaci můžeme najít na webových stránkách [13]. Na těchto stránkách také nalezneme postup jak Gmail API povolit a zprovoznit.

Přes toto API pak můžeme pracovat s emaily. V prvním kroku provádíme dotaz na maily filtrované na nastaveného odesílatele, kterým by v reálném případě byla cestovní agentura. Obdržíme seznam mailů, tento seznam obsahuje pouze identifikátory jednotlivých mailů, které je nutné po jednom stáhnout. Poté tyto maily zpracováváme po jednom od nejstaršího.

V tomto demo pluginu čteme objednávky z `XML` příloh. V reálném nasazení čteme objednávky také z příloh ve formátu `PDF` nebo z `HTML` těla mailu.

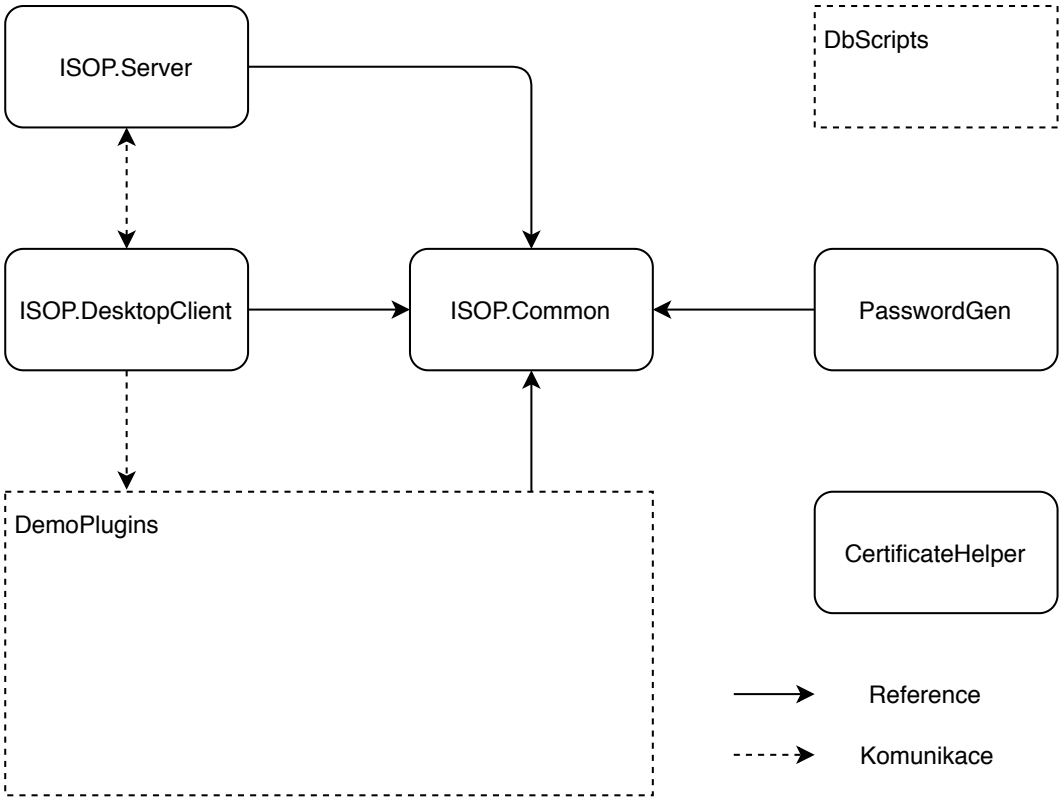
Teto plugin je nastaven na použití testovacího emailového účtu *"isop.demo@gmail.com"* s přístupovým heslem *"IsopSystem"*

4. Vývojová dokumentace

V této kapitole se budeme věnovat samotné implementaci našeho informačního systému. Zaměříme se jak na organizaci celého řešení, tak na důležité části naší implementace, zejména významné třídy a principy jejich fungování. Cílem však není zde poskytnout kompletní dokumentaci každého řádku kódu. K tomu slouží zdrojový kód samotný, který je napsán samo-vysvětlujícím stylem a v místech, kde jsme to považovali za potřebné, je doplněn o komentáře.

4.1 Organizace ISOP solution

Celá naše implementace se nachází v jednom Visual Studio solution, a je přiložena k této práci v příloze A.1. Solution se skládá z několika projektů a složek s dalšími soubory, které jsou společně s jejich vzájemnými vztahy znázorněny na obrázku 4.1 a jsou popsány v následujícím textu. Soubory, které nejsou součástí některého z projektů (například databázové skripty nebo poznámky k vývoji), jsou zde umístěny, aby byly součástí správy kódu, což nám umožňuje snadnou správu a zálohování celého projektu.



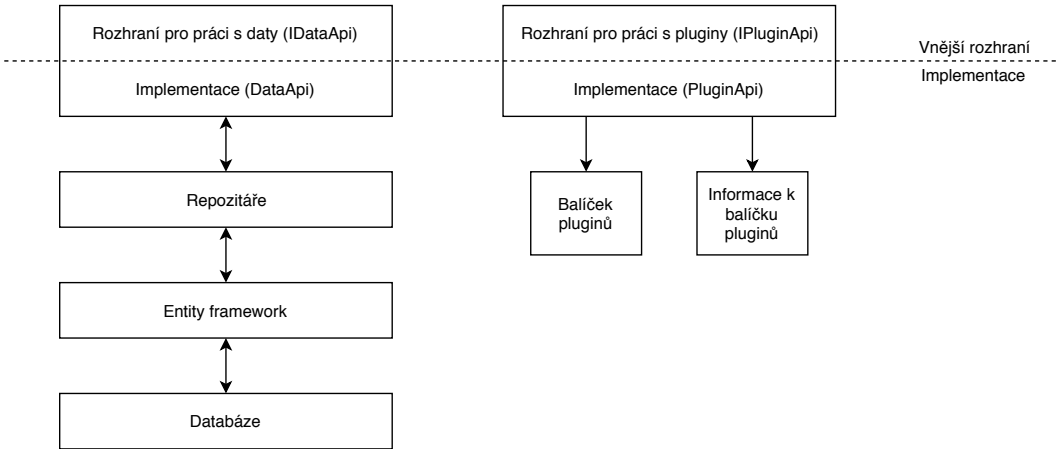
Obrázek 4.1: Organizace ISOP solution

- **ISOP.Common** – Projekt sdílené knihovny, kterou referencují všechny další komponenty systému, tedy aplikační server, klientská aplikace a všechny pluginy pro klientskou aplikaci. Zde jsou umístěny všechny třídy a rozhraní, které používá víc než jedna komponenta systému.

- **ISOP.Server** – Projekt aplikačního serveru, který je realizován pomocí frameworku WCF. Tomuto projektu se detailně věnujeme v kapitole 4.2.
- **ISOP.DesktopClient** – Projekt klientské desktopové aplikace realizované pomocí frameworku WPF.
- **DemoPlugins** – Složka obsahující projekty jednotlivých ukázkových pluginů, kterým se podrobně věnujeme v kapitole 3.
- **CertificateHelper** – Jednoduchá aplikace, která usnadňuje instalaci certifikátu na klientské počítače.
- **PasswordGen** – Konzolová aplikace, která umožňuje z uživatelského hesla v podobě textového řetězce vytvořit hash, který je možné vložit do databáze a tím nastavit uživatelské heslo bez použití klientské aplikace. Toho jsme využívali zejména během vývoje, protože správa uživatelů byla jednou z posledních implementovaných funkcí.

4.2 Server

Implementace aplikačního serveru, z návrhu architektury informačního systému v kapitole 2.2.7, se nachází v projektu ISOP.Server (dále jen *server*). Na obrázku 4.2 můžeme vidět základní strukturu celé implementace rozdělenou do jednotlivých částí a jejich vzájemné propojení.



Obrázek 4.2: Struktura implementace serveru

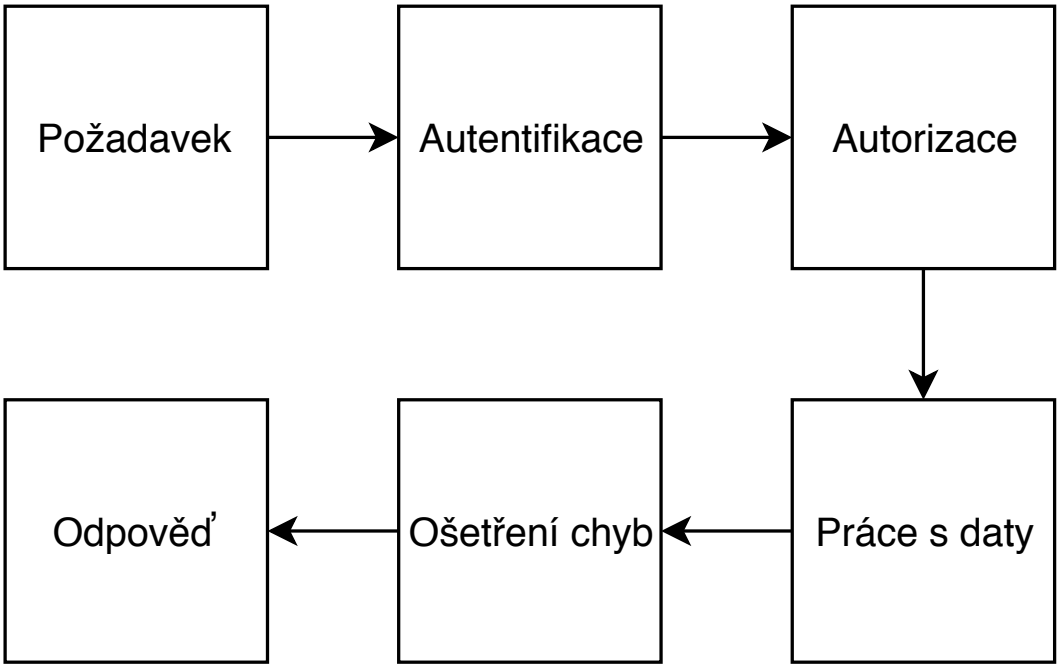
Naše implementace je založená na frameworku WCF[14], pro který jsme se rozhodli na základě analýzy v kapitole 2.4.1. Tento framework slouží k vytváření a publikování služeb, jak můžeme vidět na obrázku 4.2. V našem případě máme dvě služby, které mohou klienti serveru využívat. První služba slouží pro práci s daty v databázi a druhá pro práci s pluginy. Implementace našich služeb se nachází ve složce Api a rozhraní těchto služeb se pak nachází v podsložce Interfaces.

Služba pro práci s daty je definována rozhraním `IDataApi` a implementována ve třídě `DataApi`. Služba pro práci s pluginy je definována rozhraním `IPluginApi` a implementována ve třídě `PluginApi`.

4.2.1 Služba pro práci s daty

Tato služba publikuje sadu funkcí pro práci s daty v databázi. Tyto funkce jsou entitně-orientované, to znamená že pracují s jednotlivými entitami. Například pro entitu Route, publikuje funkce GetRoutes, AddRoute a UpdateRoute, tedy funkce načíst všechny cesty, přidat cestu a aktualizovat cestu.

Služba má kromě samotné práce s daty několik dalších zodpovědností, které musí řešit, a proto jsme jejích implementaci rozdělily do několika tříd. Rozdělení proběhlo na základě těchto zodpovědností, jak můžeme také vidět na obrázku 4.3, v pořadí jejich řešení při obsluze požadavku na službu.



Obrázek 4.3: Zodpovědnosti služby pro práci s daty

Autentizace

První zodpovědností služby je ověření identity klienta služby. Na základě analýzy v kapitole 2.4.3 pro tento účel používáme *„Membership provider“*, což je standardní mechanismus frameworku WCF pro autentizaci. Pro využití tohoto mechanismu musíme udělat několik kroků.

- Implementace potomka třídy `MembershipProvider`, v našem případě se implementace nachází ve třídě `IsopMembershipProvider` ve jmenném prostoru `ISOP.Server.Security`. Implementovali jsme pouze metodu `ValidateUser`, protože další funkce tohoto mechanismu nevyužíváme.
- Aby WCF framework mohl používat náš Membership provider, bylo nutné v konfiguračním souboru `App.config` v sekci `systém.web` > `membership` přidat naši implementaci membership provider, přesnou syntaxi znázorňuje ukázka kódu 4.1.

```
<system.web>
    <membership defaultProvider="IsopMembershipProvider">
        <providers>
        <add name="IsopMembershipProvider"
    type="ISOP.Server.Authorization.IsopMembershipProvider, ISOP.Server" />
        </providers>
    </membership>
</system.web>
```

Ukázka kódu 4.1: Přidání IsopMembershipProvider do konfigurace

- Vytvoření Behavior konfigurace, tedy konfigurace chování pro službu, na ukázce kódu 4.2 můžeme vidět konfiguraci kterou jsme použili. V položce `userNameAuthentication` jsme nastavili `userNamePasswordValidationMode` na `MembershipProvider` a `membershipProviderName` na jméno našeho membership provider-u z předchozího kroku.

```
<behavior name="SecureBehaviorConfiguration">
...
<serviceCredentials>
    <serviceCertificate
    findValue="CN=IsopServerCert"
    storeLocation="LocalMachine"
    storeName="My" />
    <userNameAuthentication
        userNamePasswordValidationMode="MembershipProvider"
        membershipProviderName="IsopMembershipProvider" />
</serviceCredentials
...
</behavior>
```

Ukázka kódu 4.2: Konfigurace chování služby

  V této konfiguraci také nastavujeme certifikát, kterým se bude server autentizovat klientovi.

- Nakonec naší službě pro práci s daty v konfiguraci nastavíme konfiguraci chování služby, kterou jsme vytvořili v předchozím kroku, přesný zápis můžeme vidět v ukázce kódu 4.3.

```
<service
    behaviorConfiguration="SecureBehaviorConfiguration"
    name="ISOP.Server.Api.DataApi">
    <endpoint
    ...
    </endpoint>
</service>
```

Ukázka kódu 4.3: Konfigurace služby pro práci s daty

Autorizace

V našem systému podporujeme dle analýzy v kapitole 2.2 dvě role, Operátor a Administrátor. K realizaci podpory rolí používáme opět standardní mechanismus framework WCF. Pro využití tohoto mechanismu musíme udělat několik kroků.

- Implementace potomka třídy `RoleProvider`, v našem případě se implementace nachází ve třídě `IsopRoleProvider` ve jmenném prostoru `ISOP.Server.Security`. Implementujeme pouze funkce `IsUserInRole` a `GetRolesForUser`, tedy ověření zda je daný uživatel v dané roli a načtení všech rolí pro daného uživatele, ostatní metody jsme ponechali bez implementace. Protože rozlišujeme pouze dvě role a navíc každý uživatel má roli operátora, tak v databázi pouze ukládáme informaci zda má uživatel také roli administrátora, a to v položce `IsAdmin`.
- Aby WCF framework mohl používat náš Role provider, bylo nutné v konfiguračním souboru `App.config` v sekci `systém.web` > `roleManager` přidat naši implementaci role provider, přesnou syntaxi můžeme vidět na ukázce kódu 4.4.

```
<system.web>
    ...
    <roleManager enabled="true" defaultProvider="IsopRoleProvider">
        <providers>
        <add name="IsopRoleProvider"
    type="ISOP.Server.Authorization.IsopRoleProvider, ISOP.Server" />
        </providers>
    </roleManager>
</system.web>
```

Ukázka kódu 4.4: Přidání IsopRoleProvider do konfigurace

- Do stejné konfigurace chování služby jako v případě IsopMembershipProvider (viz ukázka kódu 4.2) jsme pak také přidali položku `serviceAuthorization`, kde jsme nastavili `principalPermissionMode` na `UseAspNetRoles` a `roleProviderName` na jméno našeho RoleProvider-u z předchozího kroku.
- Nakonec jsme všechny metody služby `DataApi`, které mají být přístupné pouze uživatelům v roli administrátora, označili atributem `PrincipalPermission` (viz ukázka kódu 4.5).

```
    [PrincipalPermission(
        SecurityAction.Demand, Role = Constants.Roles.RoleAdmin)]
    public IsopUsersResult GetUsers()
    { ... }
```

Ukázka kódu 4.5: Použití role pro omezení přístupu pouze pro administrátory

Práce s daty

Implementace samotné práce s daty v databázi se nachází ve třídách ve jmenném prostoru `ISOP.Server.Repositories` a tyto třídy označujeme jako repositáře. Pro každou skupinu entit z návrhu databáze v kapitole 2.3.1 existuje jeden repositář, který pro tyto entity poskytuje sadu základních funkcí pro práci s nimi (například načíst všechny entity vyhovující nějaké podmínce, přidat novou entitu, aktualizovat existující entitu).

V repositářích používáme k přístupu k databázi Entity Framework 6 (dále *EF6*), který je do projektu přidán jako Nuget balíček. Spolu s tím balíčkem také používáme balíčky `MySql.Data` a `MySql.Data.EntityFramework`, které jsou nutné pro přístup EF6 k databázi MySql.

Objektový model databáze se nachází ve třídě `DbModel`, kde jsou jednotlivé tabulky mapovány na vlastnosti typu `DbSet<T>`. Generickým parametrem `T` je třída na kterou se mapují jednotlivé záznamy z odpovídající tabulky. Tyto třídy označujeme jako *Entity* a nacházejí se ve jmenném prostoru `ISOP.Server.DB.Entities`. Jednotlivé položky těchto tříd přímo odpovídají sloupcům v příslušné tabulce.

Ve funkcích repositářů také probíhá mapovaní dat z objektů Entit do objektů DTO (*Data Transfer Object*). Tyto objekty se používají ke komunikaci serveru a klienta, a proto jsou umístěny ve společném projektu `ISOP.Common`. Mapování je realizováno jako rozšiřující metoda (Extension method) `ConvertToDto`, která je pro všechny entity implementována ve třídě `EntityExtensions`.

Uvnitř funkcí repositářů se také provádí validace vstupních dat. V případě neplatného vstupu, kdy není možné pokračovat, tak dojde k vyvolání výjimky `IsopRepositoryException` s popisem chyby.

Ošetření chyb a logování

Jediná logika, která je přímo implementována ve třídě `DataApi` je ošetření výjimek. Tyto výjimky mohou vznikat jak v repositáři pokud nejsou data validní, tak při práci s databází, například kvůli porušení nějakého integritního omezení.

Volání repositářů je obalené `try-catch` blokem, který umožňuje zachycení případné výjimky. Tuto vyjímku zaznamenáme do logu a poté vrátíme negativní výsledek s obecnou zprávou že došlo k chybě.

Logování je řešeno pomocí frameworku log4net[15], který je do našeho projektu přidán jako Nuget balíček a jeho nastavení se nachází v souboru `App.config` v sekci `log4net`. Použití tohoto frameworku je obaleno třídou `Log`, které řeší přístup k instanci rozhraní `ILog`.

Odpověď

Objekty, které reprezentují odpověď na volání služby jsou instance potomků třídy `ServiceCallResultBase`, která umožňuje k odpovědi přidat informaci zda byla operace úspěšná a dále zprávy, které se zobrazí uživateli v případě neúspěchu.

V případě potomků třídy `ServiceCallResultBase`, by se mohlo zdát vhodné udělat třídu generickou namísto samostatné implementace pro každou entitu nebo její seznam, ale přístup není ve frameworku WCF doporučován, protože omezuje kompatibilitu s potenciálními klienty mimo platformu .NET, důvodem je že standard SOAP který WCF používá nepodporuje generické objekty.

4.2.2 Služba pro práci s pluginy

Služba `IPluginApi` slouží pro práci s balíčkem pluginů. Implementace této služby se nachází přimo ve třídě `PluginApi`. Služba pracuje s pluginy jako s jedním balíčkem, který má nějakou verzi a obsahuje sadu pluginů. Služba umožňuje jednak zjistit aktuální verzi balíčku, tak si tento balíček stáhnout.

4.3 Klient

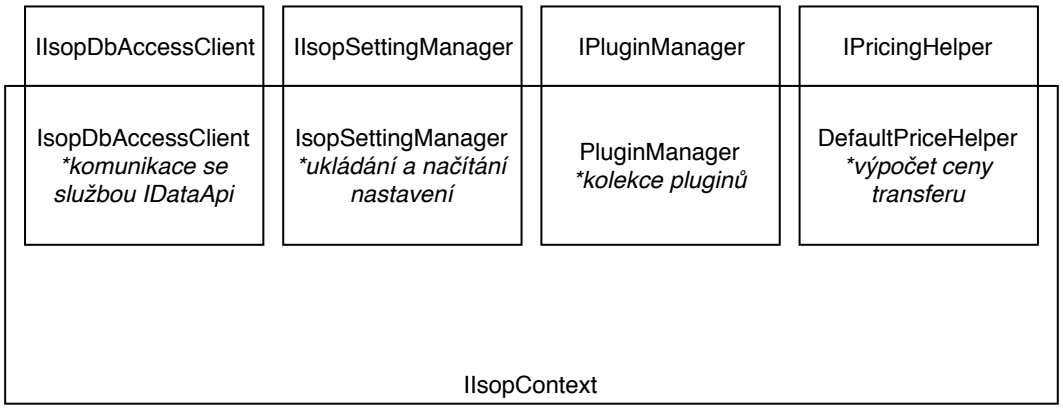
Klient pro náš systém je implementován v projektu `ISOP.Client`. Jedná se o okenní aplikaci založenou na frameworku WPF. Její návrh diskutujeme v kapitole 2.5, zde si představíme jak vypadá výsledná implementace a jak fungují základní mechanismy.

4.3.1 Kontext

Jedním z důležitých konceptů naší implementace je kontext, jedná se o kolekci zdrojů (ve zdrojovém kódu jako *Resource*), které využívají jednotlivá okna aplikace. Kontext je reprezentován rozhraním `IIsopContext`, které se nachází ve jmenném prostoru `ISOP.Common.ResourceInterfaces` spolu s rozhraními pro jednotlivé zdroje. Implementace kontextu se nachází ve třídě `IsopContext` ve jmenném prostoru `ISOP.DesktopClient.ResourceManagement` a je řešena jako slovník, kde klíčem je celý název rozhraní zdroje a hodnota je pak instance tohoto zdroje.

Rozhraní pro kontext a zdroje jsou umístěny ve sdílené knihovně, aby je bylo možné předat do pluginu typu `IToolPlugin`. To nám umožňuje pomocí plugin do systému přidávat nová okna, která mají plný přístup ke všem zdrojům aplikace.

Jednotlivé zdroje můžeme vidět na obrázku 4.4 spolu s jejich rozhraními a třídami, které je implementují.



Obrázek 4.4: Jednotlivé zdroje kontextu

Klient pro IDataApi

Zdroj definovaný rozhraním `IIsopDbAccessClient` je implementován ve třídě `IsopDbAccessClient` klienta, který umožnuje používat službu pro práci s daty

(viz kapitola 4.2.1) našeho serveru. Implementace je řešená jako obálka automaticky vygenerovaného klienta `DataApiClient`. Ten je vygenerován po přidání reference na službu pro práci s daty do projektu `ISOP.Client` na základě informací, které o naší službě publikuje framework WCF. Tento klient implementuje rozhraní naší služby `IDataApi` a řeší za nás volání této služby.

V naší implementaci přidáváme ošetření výjimek a opakování volání pokud dojde k problému. Pokud k výjimce dojde tak oznámení o chybě vracíme stejně jako v pokud by k chybě došlo na serveru, tedy také pomocí příslušných potomků třídu `ServiceCallResultBase` (viz kapitola 4.2.1).

Správce nastavení

O ukládání a načítání nastavení aplikace se stará správce nastavení, tento zdroj je definován rozhraním `IIsopSettingManager`, a je implementován ve třídě `IsopSettingManager`. Nastavení samotné je reprezentováno rozhraním `IIsopConfiguration`, které je implementováno třídou `IsopConfiguration`. Nastavené se ukládá do souboru `IsopConfiguration.bin`, v tomto souboru je uložena za-serializovaná instance nastavení. Serializace je realizována pomocí třídy `BinaryFormatter`.

Samotná aplikace má jen pár základních položek nastavení a hlavním účelem tohoto zdroje je podpora nastavení pro pluginy, kterou diskutujeme v kapitole 3.3.

Správce pluginů

Zdroj definovaný rozhraním `IPluginManager` je implementován ve třídě `PluginManager` a udržuje kolekce instance jednotlivých typů pluginů. Problematice pluginů se podrobně věnujeme v samostatné kapitole 3.

Pomocník pro výpočet ceny transferu

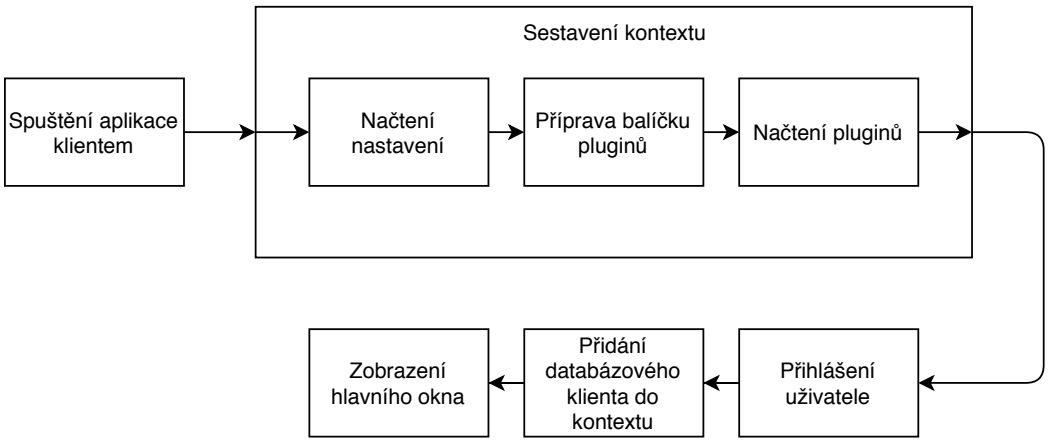
Rozhraní `IPricingHelper`, které je implementováno ve třídě `DefaultPriceHelper`, reprezentuje pomocníka pro výpočet ceny transferu, který se používá jak při ručním zadání transferu, tak při importu pomocí pluginu.

4.3.2 Spuštění aplikace

Při spuštění aplikace dochází k řadě důležitých kroků než se uživatel dostane do hlavního okna, kde může začít pracovat. Posloupnost těchto kroků můžeme vidět na obrázku 4.5, a v následujícím textu si je postupně projdeme v jejich chronologickém pořadí.

1. Spuštění aplikace uživatelem – start aplikace začíná ve třídě `App` a s ní spojeném souboru `App.xaml`, kde pomocí atributu `StartupUri` přesměrujeme další běh do sestavení kontextu aplikace.

2. Sestavení kontextu – vytváření jednotlivých součástí kontextu. Implementace se nachází ve třídě `IsopContextBuilder` a zavoláním metody Build dojde k následujícím krokům:



Obrázek 4.5: Kroky startu aplikace

(a) Načtení nastavení – načtení perzistentního nastavení aplikace ze souboru `IsopConfiguration.bin`. Implementace práce s nastavením se nachází ve třídě `IsopSettingManager`.

(b) Příprava balíčku pluginů - komunikací se službou pro práci s pluginy (viz kapitola 4.2.2) se stáhne a rozbalí balíček pluginů. Tato logika je implementována ve třídě `PluginHelper`.

(c) Načtení pluginů – práci s pluginy diskutujeme v kapitole 3.

3. Přihlášení uživatele – po sestavení kontextu pokračuje spouštění aplikace do okna pro přihlášení, kde uživatel zadá uživatelské jméno a heslo. Tyto údaje jsou pak odeslány na server k ověření.

4. Přidání klienta pro `IDataApi` do kontextu – po úspěšném přihlášení uživatele se do kontextu přidá klienta služby pro práci s daty. V tomto klientu jsou uloženy přihlašovací informace uživatele.

5. Zobrazení hlavního okna – aplikace má k dispozici kompletní kontext a uživatel může začít pracovat.

4.3.3 Implementace oken aplikace - návrhový vzor MVVM

Implementace jednotlivých oken naší aplikace jsou v mnoha ohledech podobné a používají řadu společných principů, které si projdeme v této podkapitole. Základem naší implementace oken je použití návrhového vzoru MVVM, který jsme diskutovali v kapitole 2.5.2.

Model

Roli modelů v našem případě plní DTO objekty (dále v tomto textu označovány jako modely), se kterými pracuje klient pro `IDataApi` službu. Jedná se tedy o stejné objekty, které se používají ke komunikaci mezi klientem a serverem. Implementace modelů se nachází ve jmenném prostoru `ISOP.Common.DTOs`.

Zde jsme uvažovali i o samostatné sadě objektů, které by sloužily jako modely, a DTO objekty by jsme na ně mapovali, ale tyto objekty by pravděpodobně byly v

současné situaci identické. Z toho důvodu jsme od implementace těchto objektů upustili, navíc případná dodatečná změn by nebyla náročná. Tohle řešení nám také umožňuje implementovat validaci přímo na těchto objektech a používat ji v klientské aplikaci i serveru.

Pro správnou funkci modelu MVVM ve WPF naše modely implementují dvě rozhraní, a to `InotifyPropertyChanged` a `INotifyDataErrorInfo` (viz kapitola 2.5.2). Implementace těchto rozhraní je ve třídě `DtoBase`, která je bázovou třídou pro všechny modely.

Implementace rozhraní `InotifyPropertyChanged` se skládá z definice události `PropertyChanged` a metody pro její vyvolání, kde je argumentem jméno vlastnosti, u níž došlo ke změně.

Implementace rozhraní `INotifyDataErrorInfo`, které můžeme vidět na ukázce kódu 4.6, používá slovník (třída Dictionary), kde klíčem je jméno proměnné a hodnotou je seznam textových řetězců, které reprezentují jednotlivé chyby. Pokud je seznam prázdný, tak se klíč odstraňuje, což umožňuje snadnou implementaci vlastnosti `HasError` pomocí počtu klíčů ve slovníku, tedy nulový počet klíčů značí stav bez chyb. Tento způsob je výrazně rychlejší než ověření, že seznamy pro všechny klíče jsou prázdné.

```
namespace System.ComponentModel
{
    public interface INotifyDataErrorInfo
    {
        bool HasErrors { get; }

        event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

        IEnumerable GetErrors(string propertyName);
    }
}
```

Ukázka kódu 4.6: Rozhraní INotifyDataErrorInfo

Ve spojitosti s implementací tohoto řešení jsme ještě narazili na problém, že instance slovníku klíčů je `NULL`, pokud instance vznikla jako návratová hodnota z volání služby aplikačního serveru. Instanci slovníku klíčů vytváříme v konstruktoru a už ji nikdy neměníme. Problém je v tom, že tato instance se nepřenáší, protože není označena atributem DataMember. Slovník můžeme tímto atributem označit, ale pro privátní proměnné to není doporučováno, jelikož toto řešení funguje pouze za určitých podmínek v závislosti na použité serializaci. Například funguje pro `BinaryFormatter`, ale už ne pro `XmlSerializer`.

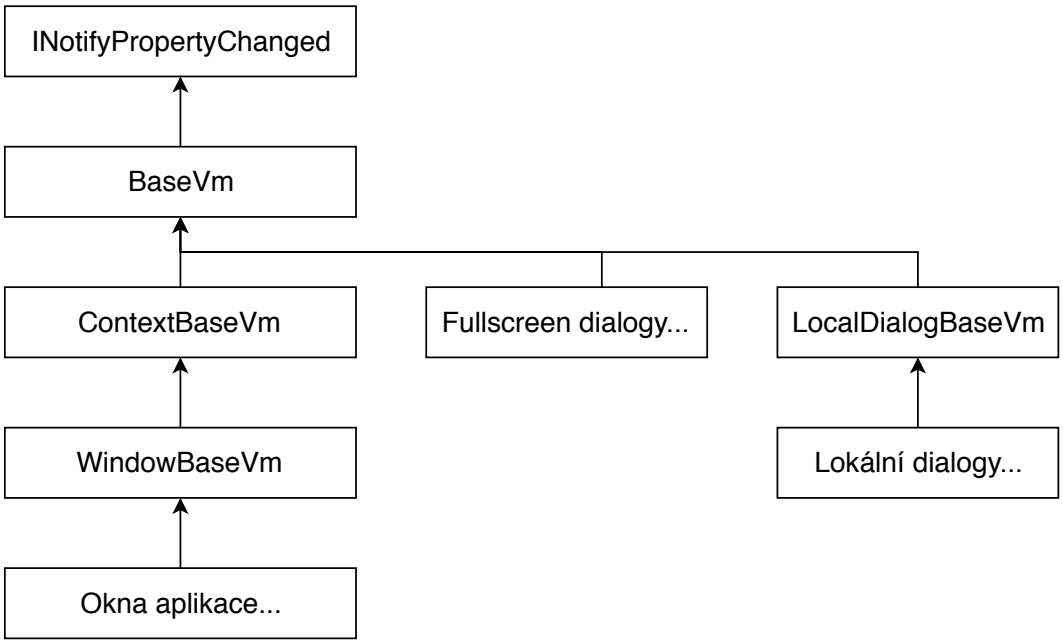
Navíc chyby nepotřebujeme přenášet, protože je lze dopočítat z hodnot proměnných, ale ani to by nemělo být potřeba, protože ze serveru přenášíme uložená data, která by měla být vždy validní a toto rozhraní slouží jen pro účely uživatelského rozhraní. Z těchto důvodů jsme zvolili řešení, že přidáme metodu `OnDeserializing`, kterou označíme stejnojmenným atributem, který zajišťuje, že se tato metoda bude volat po dokončení deserializace. V této metodě pak vytvoříme instanci slovníku, pokud neexistuje.

ViewModel

Bázové třídy pro implementaci ViewModel-ů se nachází ve jmenném prostoru ISOP.DesktopClient.GUI.ViewModels, samotné ViewModel-y se pak nacházejí v podprostorech Controls (VM pro komponenty), Dialogs (VM pro dialogy) a Windows (VM pro okna).

Hierarchii tříd pro implementaci VM můžeme vidět na obrázku 4.6, funkce jednotlivých tříd jsou následující:

- BaseVm – implementuje rozhraní InotifyPropertyChanged, které stejně jako v případě modelů slouží k notifikaci grafického rozhraní o změnách datových položek.
- ContextBaseVm – slouží k uložení kontextu aplikace (viz kapitola 4.3.1).
- WindowBaseVm – bázová třída pro VM pro jednotlivá okna.



Obrázek 4.6: Hierarchie tříd pro implementaci VM

5. Nasazení

Cílem této kapitoly je umožnit čtenáři zprovoznit náš systém s pomocí připravených souborů v příloze A.2. Podrobně si popíšeme jednotlivé kroky, které jsou nutné ke zprovoznění systému na lokálním počítači.

5.1 Předpoklady

Abychom mohli server systému ISOP úspěšně nainstalovat a připravit k použití, budeme potřebovat následující:

- Počítač s operačním systémem Windows 7 nebo novějším.
- Nainstalovaný IIS (Internet Information Services) ve verzi 7.0 nebo vyšší.
- Nainstalovaný .NET framework ve verzi 4.6.1 nebo vyšší.

5.2 Databáze

Aplikační server systému ISOP využívá k ukládání dat databázi MySQL. Tato databáze není součástí systému a je nutné si ji stáhnout samostatně ze stránek výrobce na odkazu [16].

Je nutné nainstalovat následující komponenty:

- *MySQL Community Server* – samotný databázový server
- *MySQL Workbench* – grafický klient pro správu databáze
- *Connector/Net* – ovladač pro platformu .NET

Při instalaci následujeme pokyny jednotlivých instalačních balíčků nebo můžeme využít manuál na stránkách výrobce na odkazu [17].

Během instalace pro lokální nasazení IP adresu nemusíme řešit a jako port použijeme defaultní hodnotu 3306.

Po instalaci vytvoříme účet pro ISOP systém s uživatelským jménem *„IsopAccess“* a heslem *„LongAndSecurePassword“*. Tomuto účtu pak dále musíme přidělit práva na čtení a zápis do databáze.

Dále na databázovém serveru musíme vytvořit samotnou databázi a k tomu slouží trojice SQL skriptů ve složce `/deploy-localhost/db-scripts`.

- `isop-db-create.sql` – vytvoří databázové schéma a všechny potřebné tabulky. Dále vytvoří prvního uživatele s uživatelským jménem a heslem *„IsopSystem“*.
- `isop-db-setup.sql` – Naplní databázi položkami vhodnými pro firmu nabízející své služby v České Republice, jedná se o destinace, skupiny cest, cesty a služby. Tento skript není nutný ke spuštění systému.
- `isop-db-demo.sql` – Naplní databázi ukázkovými daty, která jsou blíže popsána v kapitole 5.6. Tento skript je možné úspěšně provést, pokud byl úspěšně proveden předchozí skript. Tento skript není nutný ke spuštění systému a použijeme jej pouze v případě, že chceme předvést funkce systému.

5.3 Certifikáty

Aby správně fungovaly autentizační a šifrovací mechanismy aplikačního serveru je nutné nainstalovat bezpečnostní certifikáty, které jsou umístěné ve složce `/deploy-localhost/certificates`.

Certifikáty instalujeme do následujících umístění:

- `IsopRootCert.cer` – instalujeme do úložiště Trusted Root Certification Authorities pro Local Computer.
- `IsopServerCert.pfx` – instalujeme do úložiště Personal pro Local Computer. Tento certifikát obsahuje nejen veřejný klíč, ale také privátní klíč, jehož heslo je *„AxisTransportIsop“*. Poznámka: náš server (resp. uživatelský účet pod kterým běží) musí mít přístup k privátnímu klíči tohoto certifikátu – v případě vývoje ve VS je to uživatelský účet, v případě serveru je to účet pod kterým běží hostovací služba IIS. Podrobnější informace můžeme najít na stránce [18].

5.4 Nasazení serveru

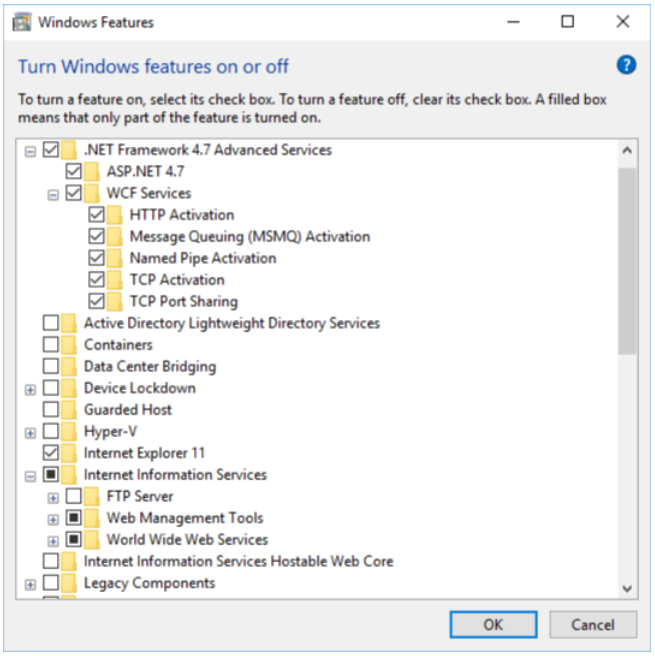
Server nasazujeme na hostovací server IIS (Internet Information Services), který musí být na počítači nainstalován, aby bylo možné server nasadit. IIS je volitelná součást operačního systému Windows, kterou lze nainstalovat přes Windows Features, nutné součásti jsou (viz obrázek 5.1):

- .NET Framework 4.(6 nebo 7) Advanced Services
  - ASP.NET 4.(6 nebo7)
  - WCF Services
    - HTTP Activation
- Internet Information Services
  - Web Management Tools
  - World Wide Web Services

Dále musíme provést samotné nasazení serveru na IIS, to uděláme přes IIS Manager. Na položce *„Default Web Site“* vytvoříme novou aplikaci, které dáme jméno a alias *„isop“* a nastavíme jí nějakou fyzickou složku na disku. Do této složky pak nakopírujeme soubory serveru, které najdeme v příloze ve složce `/deploy-localhost/deploy/server`.

Dále na stejné položce vytvoříme novou složku, kterou pojmenujeme *„isop-update“*. Do této složky můžeme nakopírovat soubory klientské aplikace a pokud je tato verze novější než aktuální klientova, při nejbližším spuštění dojde k automatické aktualizaci. Při prvotním nasazení použijeme soubory klienta z přílohy, které jsou umístěné ve složce `/deploy-localhost/deploy/client`, později by jsme zde umístili souboru nové verze.

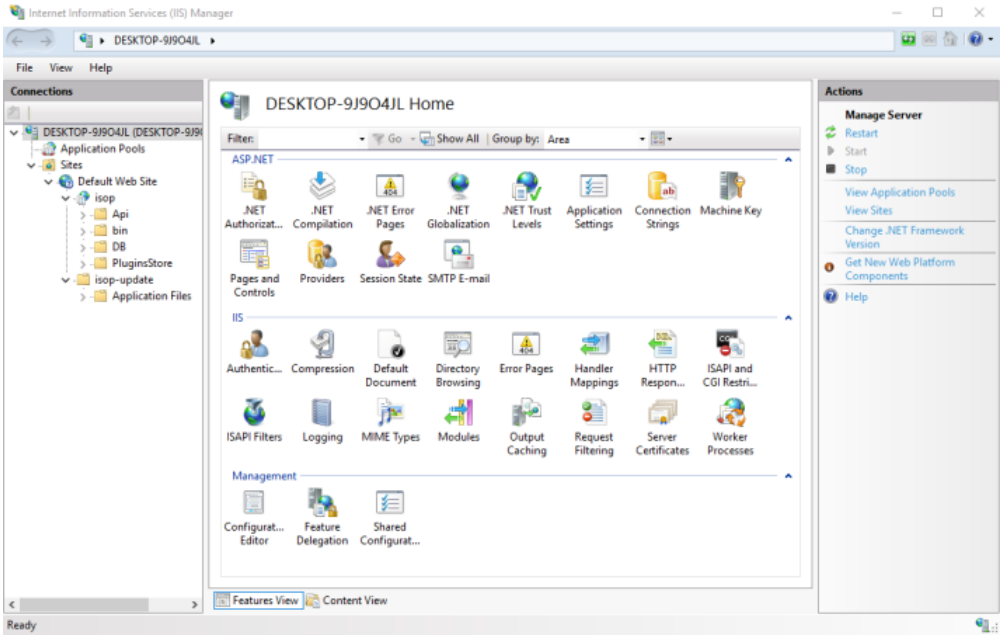
Po provedení těchto kroků by měly být služby serveru dostupné na adresách pod tímto odstavcem. To můžeme ověřit tím, že tyto adresy zadáme do prohlížeče



Obrázek 5.1: Windows features pro hostování WCF v IIS

a dostaneme se na stránku s návodem na připojení ke službě. Na obrázku 5.2 můžeme vidět finální stav IIS po nasazení našeho systému.

Služba pro práci s daty: http://localhost/isop/ISOP.Server.Api.DataApi.svc
Služba pro práci s pluginy: http://localhost/isop/ISOP.Server.Api.DataApi.svc



Obrázek 5.2: IIS – konečný stav po nasazení

5.5 Nasazení klient

Tuto část nasazení provádí samotní uživatelé systému, a proto jsme zde kladli důraz na maximální jednoduchost. Instalace se provádí pouze spuštěním souboru `Setup.exe`. Instalátor je umístěn v příloze ve složce `/deploy-localhost/deploy/client`.

Před zahánením samotné práce s klientskou aplikací doporučujeme si přečíst návody na obsluhu, které se nacházejí v příloze A.3.

5.6 Poznámky k testovací datům

Ukázková data se nacházejí ve skriptu `isop-db-demo.sql` (viz instalace databáze v podkapitole 5.2). Tento skript obsahuje následující data:

- Transfery pro období 20.12.2019 až 31.12.2019.
- Sadu řidičů i s vozy a ceníky pro rok 2019.
- Sadu cestovních agentur i s ceníky pro rok 2019.
- Kompletně zpracovaný rozpis pro 20.12.2019.
- Rozpracovaný rozpis pro 21.12.2019.
- Fakturu pro cestovní agenturu Sunny Transfers za období 20.12.2019 až 31.12.2019.

Závěr

V závěru této práce zhodnotíme, jak se nám podařilo naplnit požadavky, které vzešly z analýzy v kapitole 1.2. Tyto požadavky vycházely z reálných potřeb naší partnerské firmy, a proto budeme jejich naplnění hodnotit i z pohledu praktického nasazení v této firmě.

P1 Dostupnost
Náš systém se skládá z centrálního serveru a klientské aplikace, přičemž tyto části spolu můžou zabezpečeně komunikovat přes internet. Uživatelé můžou klientskou aplikaci snadno nainstalovat na vhodný počítač a pracovat odkudkoliv, kde mají dostupné připojení k internetu.

P2 Import objednávek
Objednávky je možné zadávat jak ručně, tak je importovat pomocí pluginů. Podařilo se nám také implementovat plugin, který objednávky importuje rovnou z emailového účtu a zpracování objednávek je tedy snadné.

P3 Tvorba a distribuce rozpisů
Systém umožňuje vytvářet, editovat a ukládat rozpisy. Vytvořené rozpisy je pak možné exportovat pomocí pluginu. V praxi partnerská firma používá export do aplikace Outlook, kde plugin připraví email k odeslání a uživatel již jen provede kontrolu, případně může rovnou doplnit další informace.

P4 Tvorba a export faktur
Systém umožňuje vytvářet a ukládat faktury pro cestovní agentury. Tyto faktury lze pak exportovat opět pomocí pluginu. V praxi se používá plugin na export do souboru aplikace Excel nebo do souboru .csv, který se importuje do systému cestovní agentury.

P5 Kontrola faktur řidičů
Systém umožňuje vytvářet a ukládat faktury/přehledy pro řidiče, ty je pak možné exportovat opět pomocí pluginu. V praxi možnost snadné kontroly faktur řidičů odhalila řadu pochybení a zabránila nemalým finančním ztrátám.

P6 Historie
U každého transferu je vedena historie všech změn včetně uživatele, který změnu provedl.

P7 Náklady
Systém je možné nasadit na běžně dostupnou infrastrukturu, v případě serveru se jedná o počítač s operačním systémem Windows Server a veřejně dostupnou IP adresou. Klientskou aplikaci je pak možné používat na běžném počítači s operačním systémem Windows 7 nebo novějším a s přístupem k internetu.

Všechny použité frameworky a knihovny jsou dostupné zdarma i pro komerční použití a nepředstavují další náklady. Také databáze MySql je dostupná zdarma i pro komerční použití.

Podařilo se nám vyvinout komplexní informační systém, který umožňuje přepravní firmě efektivně fungovat a řeší všechny její základní administrativní potřeby. Je nutné také podotknout, že náš systém je možné v mnoha ohledech vylepšit či rozšířit, ale i v současném stavu plní všechny důležité funkce.

Nasazení v praxi

Partnerská firma náš systém nasadila do praktického použití již během vývoje, přibližně v lednu roku 2017 a používala ho až do ukončení své činnosti na podzim roku 2018. Při ukončení provozu databáze obsahovala přibližně 100 tisíc objednávek (toto číslo zahrnuje i objednávky naimportované z předchozího řešení v podobě Excel tabulky).

Systém byl nasazen na virtuálním serveru, který byl pronajatý u společnosti Active 24 s.r.o., a používal operační systém Windows Server 2012 R2. Náklady na provoz dosahovali přibližně osmi tisíc korun ročně, čímž byl splněn požadavek P7 na nízké náklady.

Denně jsme prováděli zálohování databáze pomocí externího nástroje určeného pro MySQL databázi.

Možná pokračování

V budoucnu bychom rádi pokračovali ve vývoji tohoto systému a jeho rozšiřování o další funkce, níže uvádíme několik možných úprav či rozšíření:

- Zjednodušení procesu nasazení a aktualizace – zejména v případě serveru (viz kapitola 5).
- Off-line cache pro klientskou aplikaci – v současné době není možné klientskou aplikaci používat bez připojení k serveru. Dostatečným vylepšením by byla možnost prohlížení údajů po dobu výpadku připojení.
- Implementace klienta pro řidiče, který by jim umožnil prohlížet přiřazené transfery a zadávat jejich stav – čeká, naložil klienta, dokončil transfer. Tohoto klienta by bylo vhodné implementovat jako webovou aplikaci.
- Překlad klientské aplikace do anglického jazyka (případně dalších jazyků) – umožnilo by nabídnout systém i v zahraničí.
- Vytvoření sady standardních pluginů, případně také vytvoření pluginů na import transferů od největších cestovních agentur na trhu.

Seznam použité literatury

[1] Microsoft - Bing Maps. https://www.microsoft.com/en-us/maps. Zkontrolováno: 2019-06-30.

[2] Microsoft. Bing Maps - Choose your API. https://www.microsoft.com/en-us/maps/choose-your-bing-maps-api. Zkontrolováno: 2019-06-30.

[3] Microsoft. Bing Maps - Get Started Today. https://www.microsoft.com/en-us/maps/create-a-bing-maps-key. Zkontrolováno: 2019-06-30.

[4] GMap.NET - Github. https://github.com/radioman/greatmaps. Zkontrolováno: 2019-06-30.

[5] Sql Server 2017 Editions. https://www.microsoft.com/en-us/sql-server/sql-server-2017-editions. Zkontrolováno: 2019-06-30.

[6] Oracle Database 18c Express Edition (XE). https://www.oracle.com/database/technologies/appdev/xe.html. Zkontrolováno: 2019-06-30.

[7] MySQL Community Server. https://dev.mysql.com/downloads/mysql/. Zkontrolováno: 2019-06-30.

[8] SuperSocket 1.6 Documentation. http://docs.supersocket.net/v1-6/en-US. Zkontrolováno: 2019-06-30.

[9] INotifyCollectionChanged Interface Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.inotifycollectionchanged. Zkontrolováno: 2019-06-30.

[10] Material Design In XAML. http://materialdesigninxaml.net/. Zkontrolováno: 2019-06-30.

[11] Managed Extensibility Framework (MEF). https://docs.microsoft.com/en-us/dotnet/framework/mef/. Zkontrolováno: 2019-06-30.

[12] CompositionContainer Class. https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.composition.hosting.compositioncontainer. Zkontrolováno: 2019-06-30.

[13] GMail API .NET Quickstart. https://developers.google.com/gmail/api/quickstart/dotnet. Zkontrolováno: 2019-06-30.

[14] Juval Lowy. *Programming WCF Services*. 3rd Edition. O'Reilly Media, Inc., 2010.

[15] The Apache log4net project. https://logging.apache.org/log4net/. Zkontrolováno: 2019-06-30.

[16] MySQL Community Downloads. https://dev.mysql.com/downloads/. Zkontrolováno: 2019-06-30.

[17] Installing and Upgrading MySQL. https://dev.mysql.com/doc/refman/8.0/en/installing.html. Zkontrolováno: 2019-06-30.

[18] System.Security.Cryptography.CryptographicException: keyset does not exist. https://stackoverflow.com/questions/12106011/system-security-cryptography-cryptographicexception-keyset-does-not-exist. Zkontrolováno: 2019-06-30.

Seznam obrázků

Seznam ukázek kódu

A. Přílohy

A.1 Implementace

Složka */source* obsahuje Visual Studio solution se všemi projekty a zdrojovými kódy celého systému včetně demonstračních pluginů.

A.2 Testovací nasazení

Složka */deploy-localhost* obsahuje soubory nutné ke zprovoznění systému dle návodu, který je popsán v kapitole 5.

A.3 Návody

Složka */manual* obsahuje následující uživatelské návody na obsluhu systému po jeho nasazení:

- *isop-manual-operator.pdf* - manuál k funkcím systému ke kterým mají přístup uživatelé v roli operátora.
- *isop-manual-manager.pdf* - manuál k funkcím systému ke kterém má přístup pouze administrátor systému. Administrátor má také přístup k funkcím operátor, které jsou popsány v návodu výše.

A.4 Text práce

Složka */text* obsahuje soubor práce.pdf s textem této práce a soubory abstrakt.pdf a abstraktEN.pdf s abstraktem této práce v českém a anglickém jazyce.