VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

SERVEROVÁ APLIKACE PRO DISTRIBUOVANÉ VÝPOČTY

SERVER APPLICATION FOR DISTRIBUTED COMPUTING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. Marek Svozílek
AUTHOR

VEDOUCÍ PRÁCE — Ing. Jiří Kovář, Ph.D.
SUPERVISOR

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Ústav automatizace a informatiky
Student:	**Bc. Marek Svozílek**
Studijní program:	Strojní inženýrství
Studijní obor:	Aplikovaná informatika a řízení
Vedoucí práce:	**Ing. Jiří Kovář, Ph.D.**
Akademický rok:	2016/17

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Serverová aplikace pro distribuované výpočty

Stručná charakteristika problematiky úkolu:

Výpočet průtoku a tlaku v rozsáhlejší tlakové potrubní síti si mnohdy vyžaduje velkou výpočetní kapacitu počítače. Dlouhá doba řešení takové simulace znesnadňuje nebo přímo omezuje její operativní použití. Řešením může být použití výpočetní kapacity více počítačů na základě principů distribuovaných výpočtů. Úkolem této práce je návrh a realizace serverové aplikace pro realizaci distribuovaných výpočtů.

Cíle diplomové práce:

1) Vytvořte řešerši stávajících metod realizace distribuovaných výpočtů
2) Na základě rešerše vytvořte návrh serverové aplikace pro distribuované výpočty
3) Navrženou aplikaci realizujte (s využitím platformy .NET v jazyce C#)
4) Vytvořenou aplikaci testujte

Seznam doporučené literatury:

KSHEMKALYANI, Ajay D. a Mukesh SINGHAL. Distributed computing: principles, algorithms, and systems. Cambridge: Cambridge University Press, 2011. ISBN 978-0-521-18984-2.

ALBAHARI, Ben., Peter. DRAYTON a Brad. MERRILL. C# essentials: principles, algorithms, and systems. 2nd ed. Sebastopol, CA: O'Reilly, c2002. ISBN 05-960-0315-3.

CHENG, Steven., Peter. DRAYTON a Brad. MERRILL. Microsoft Windows Communication Foundation 4.0 cookbook for developing SOA applications: over 85 easy recipes for managing communication between applications. 2nd ed. Birmingham, U.K.: Packt Publishing, 2010. ISBN 9781849680769.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2016/17

V Brně, dne

L. S.

doc. Ing. Radomil Matoušek, Ph.D. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D děkan fakulty

ABSTRAKT

Tato diplomová práce se zabývá realizací serverové aplikace pro distribuované výpočty programované na platformě .NET a v jazyce C# ve vývojovém prostředí Visual Studio 2015. Diplomová práce je rozdělena na část teoretickou a praktickou. V teoretické částí se práce zabývá technologiemi, které jsou potřeba pro realizaci aplikace. Zde je popsáno, jak se každá technologie používá a nastavuje, aby posloužila účelu diplomové práce. V teoretické části je funkčnost ukázána na příkladech. Praktická část popisuje rozvržení aplikace a funkčnost jednotlivé části aplikace. Architektura každé části aplikace je prezentovaná v UML class diagramu.

ABSTRACT

This diploma thesis deals with the realization of a server application for distributed calculations programmed on .NET platform and in C # language in development environment Visual Studio 2015. The diploma thesis is divided into the theoretical and practical part. In the theoretical part, the thesis deals with technologies, which are needed for implementation of the application. Here is how each technology is used and set up to serve the purpose of the master's thesis. In the theoretical part, functionality is shown in the examples. The practical part describes the layout of the application and the functionality of the individual part of the application. The architecture of each part of the application is presented in the UML class diagram.

KLÍČOVÁ SLOVA

Server, klient, Microsoft, .NET, databáze, Visual Studio, Entity framework, NuGet balíček

KEYWORDS

Server, client, Microsoft, .NET, database, Visual Studio, Entity framework, Nuget packet

BIBLIOGRAFICKÁ CITACE

SVOZÍLEK, M. *Serverová aplikace pro distribuované výpočty.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2017. 81 s. Vedoucí diplomové práce Ing. Jiří Kovář, Ph.D.

PODĚKOVÁNÍ

Rád bych poděkoval svému vedoucímu práce Ing. Jiřímu Kovářovi, Ph.D. za odborné vedení, jeho ochotu, čas a cenné rady, které mi dával během konzultací. V neposlední řadě děkuji rodině a svým blízkým a kamarádům za podporu při vypracovávání této diplomové práce.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato práce je mým původním dílem, zpracoval jsem ji samostatně pod vedením Ing. Jiřího Kováře, Ph.D. a s použitím literatury uvedené v seznamu.

V Brně dne 26.5.2017 ……

Bc. Marek Svozílek

OBSAH

1 ÚVOD

V současné době je vysoký výpočetní výkon jednoho počítače velice nákladnou záležitostí. Proto se začalo využívat distribuovaných výpočtů, které využívají přerozdělení výpočetních výkonů mezi méně výkonné počítače a svým společným výkonem dosahují většího výkonu než jeden výkonnější počítač.

Práce je zaměřena na problematiku distribuovaných výpočtů a návrhem aplikace, která by dokázala daný problém vyřešit. Hlavním úkolem aplikace je posílat klientům soubory, které se zpracují na klientovi a následně se začne provádět výpočet, který má pevně stanovený počet cyklů. Po dokončení celého výpočtu se odešlou výsledky na server, kde se uloží do databáze a zašle se klientovi nový soubor a cyklus se opakuje opět od začátku. Hlavním úkolem diplomové práce je vytvořit automatickou distribuci těchto souborů, aby uživatel mohl ukládat výpočetní soubory do databáze, sledovat výsledky a nemusel ručně zadávat každý výpočetní úkon zvlášť.

Tento úkol by měl být řešen za pomocí technologie .NET a programovacího jazyku C#. Technologie .NET nabízí dvě možná řešení daného úkolu. Technologii WCF, která je robustní a disponuje značným množstvím způsobů komunikace. Oproti její konkurenci ASP.NET Web API, která je podporována pouze na HTTP protokolu. Jelikož by výsledný program měl být implementovaný na IIS (uživatel se nemusí starat o jeho spuštěný a následný běh) se u technologie WCF zužuje možnost komunikace, která je pouze pomocí HTTP protokolu.

První část práce je zaměřena na teoretické základy, které jsou potřeba k naprogramování celé aplikace. Teoretické základy jsou použity na názorných příkladech, které pomáhají lépe pochopit celou funkčnost technologie. Do teorie jsou zahrnuty novinky na platformě .NET, SQL databáze, programové způsoby, které lze zvolit pro komunikaci s databází. V této části jsou rozebrány dvě technologie, které se mohou použít pro vytvoření reálné aplikace pro distribuované výpočty. Tato část obsahuje i přehled tvoření celé architektury pomocí UML class diagramu. V závěru teoretické části je zmínka o využití automatizovaného testování.

Praktická část je zaměřena na prezentaci architektury a popsání jednotlivých částí, na které se dá celá aplikace rozdělit.

2 DISTRIBUOVANÝ SYSTÉM

Jedná se o systém autonomních počítačů, které jsou spolu vzájemně propojeny tak, aby se dosáhlo určeného cíle. Počítače v distribuovaném systému jsou nezávislé a nemají sdílení fyzické paměti a výpočetního výkonu. Komunikují mezi sebou pouze pomocí zpráv posílaných přes počítačovou síť nebo internet. Pomocí zpráv se mohou přenášet různé informace, ale nejčastější slouží k přenášení balíků dat, které obsahují informace pro výpočet na daném počítači.

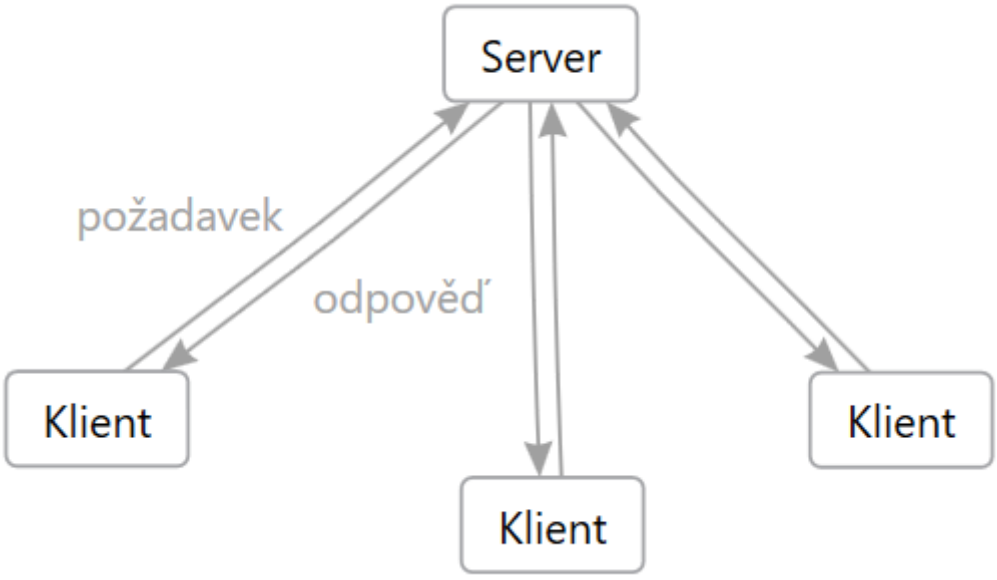
Počítače v počítačové síti mohou mít různé role. Role jednotlivého počítače je dána jeho konfigurací (disponující výpočetní síla) a programovým vybavením. Často je v systému používán jeden počítač (server) jako hlavní, který pak slouží ke komunikaci s ostatními počítači a k přerozdělování úkolů. [12]

2.1 Distribuované výpočty

V distribuovaných výpočtech se jedná o službu s centrálním zdrojem. Centrální zdroj je zastoupen serverem, který slouží k rozesílání odpovědí na požadavky od klientů. Každý klient dostane odpověď na svůj požadavek, kterou zpracuje a po zpracování ho odešle opět na server. Server data přijme a uloží si je a celý cyklus se opakuje znovu. Schéma znázornění můžeme vidět na Obr. 1.

Model klient/server lze znázornit na moderním Word Wide Web. Při načítání webové stránky od klienta se posílá požadavek na server, který zpracovává jeho informace a po zpracování je klientovi od serveru zaslána celá stránka, která se mu načítá v prohlížeči.

Pojmy klient a server jsou silně funkční abstrakce. Server je jednoduchá jednotka, která poskytuje službu jednomu nebo více klientům současně. Klient je jednotka, která spotřebovává danou službu a nemusí znát podrobnosti o tom, jak je služba poskytována a jak se nakládá se zpracovanými daty, které klient odeslal na server.



Obr. 1 Komunikace klient/Server [12]

Nikde není dané, že klient a server nemůže být na jednom zařízení a využívat stejnou architekturu. Například signály ze vstupního zařízení počítače jsou obecně dostupné programům běžících na počítači. Programy jsou klienti a vstupními zařízeními mohou být myš a klávesnice. Ovladače operačního systému jsou servery, které berou fyzický signál a mění ho na vstupní signál.

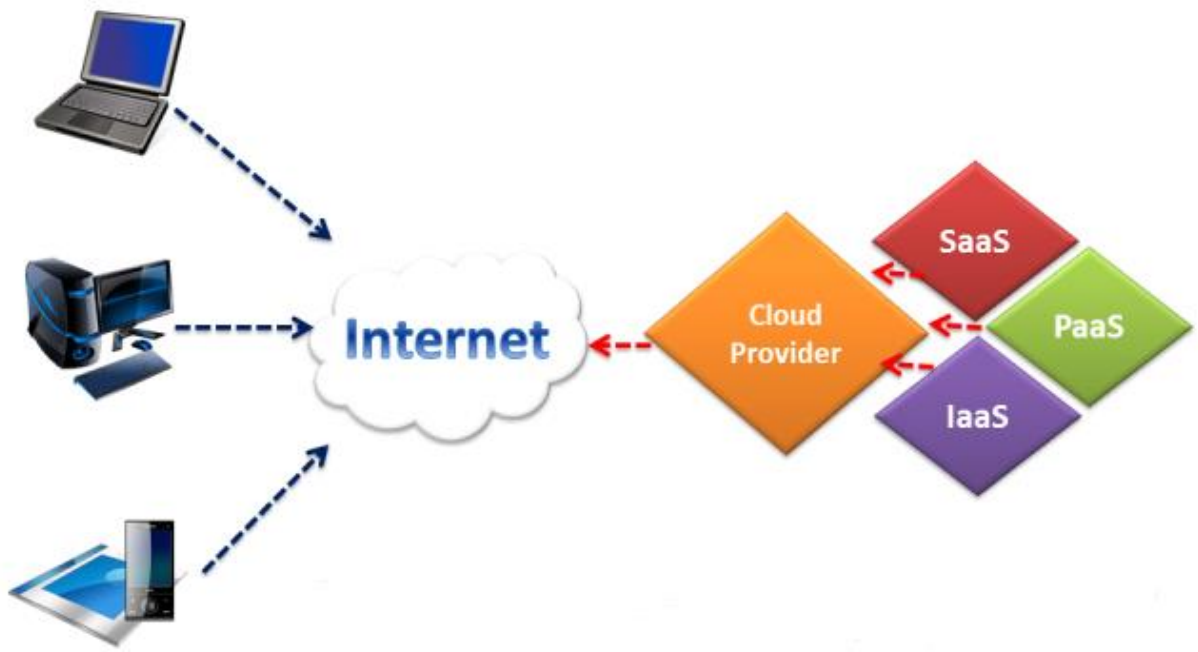
Nevýhodou systému klient/server je, že server je jediný bod selhání. Je to jediná složka se schopností rozesílat a sbírat data klientů. U klientů je jedno, zda fungují nebo ne, protože se dají lehce nahradit dle potřeby. Když bude vyřazen z provozu server, potom dochází k pádu celého systému.

Další nevýhoda nastává, když má server mnoho klientů, které musí obsluhovat, protože jsou kladeny velké nároky na fyzické vlastnosti serveru. Nelze tedy připojovat další klienty, aniž by to nemělo vliv na celkový výkon serveru. [12]

2.2 Cloudové výpočty

Pro cloudové výpočty je důležité, aby uživateli aplikace nezamrzala, nepadala nebo nebyla příliš zpomalena. Proto je kladen důraz na odezvu, která musí být skoro rovna nule. Uživatelé žijící v různých částech světa musí mít stejně fungující aplikaci, která nehlásí žádné problémy 24/7/365 dnů v roce.

Sálové počítače mnohdy nestačí na splnění požadavků od firem na zpracování velkých strukturovaných a nestrukturovaných datových souborů. Zde vzniká prostor pro cloudovou distribuovanou výpočetní techniku, která umožnuje zpracovávat funkce na velkých souborech dat. Facebook je společnost, která nejvíce využívá distribuované cloudové výpočty. Obyčejné distribuované výpočty nemohou poskytnout tak velkou dostupnost, odolnost proti selhání a škálovatelnost.



Obr. 2 Schéma cloudového výpočtu [8]

Cloudové výpočty jsou typ výpočtu, který je masivně škálovatelný a flexibilní v IT službách dodávaných uživateli, který využívá internetové technologie. Služby mohou zahrnovat infrastrukturu, platformy, aplikace a úložný prostor, který je uživateli plně k dispozici. U úložného prostoru nemusí uživatel provádět žádné další zálohování uložených dat na jiné místo, protože data už jsou na cloudu zálohována a je malá pravděpodobnost, že by došlo k jejich ztrátě. Za všechny zmíněné služby si uživatel musí platit, ale platí vždy jen za to, co ve skutečnosti využívá. Např.: potřebuje jeden den provádět spoustu výpočtů, zaplatí si výpočetní výkon na určitou dobu.

Cloudové výpočty se odkazují na poskytování služeb přes internet viz. Obr 2. Tyto služby mohou obsahovat cokoliv od podnikatelského softwaru, který je přístupný přes internet až ke skladování a výpočetním prostředkům. Zatímco distribuovaný výpočet je rozdělen na menší úkoly, které postupně zpracovává skupina počítačů, na cloudu se problém řeší hromadně ve stejnou dobu.

Mezi výhody cloudového řešení patří možnost práce na daném problému z jakéhokoliv místa na světě se zásadní podmínkou, která je, připojení k internetu. Díky této technologii se např. může firma stěhovat po světě a nemusí řešit technologické zázemí. Neplatí to jenom o stěhování firmy, ale i o kolektivní práci lidí z celého světa.

Další výhodou pro firmy je sdílení záznamů pro zaměstnance na jednom místě, nemusí komunikovat pomocí emailu nebo jiných technologií. Firma si může uložit své know-how na cloud a z tohoto zdroje mohou čerpat zaměstnanci. Když někdo provede aktualizaci záznamu, potom i ostatní pracují s nejnovějšími daty. Nevznikají tím zbytečné kopie souborů. [8]

3 .NET

Microsoft koncem 90. let minulého století začal s vývojem svého prvního frameworku, který uvedl na trh začátkem nového tisíciletí. V minulosti atraktivní a zajímavá platforma, která obsahovala knihovnu tříd. Pomocí této knihovny mohli programátoři volně psát aplikace v kterémkoli podporovaném jazyce (VB, C#, C++ atd.). Programy psané v .NET jsou vykonávány v softwaru nazývaném *Common Language runtime* (CLR). Je to aplikační virtuální stroj, který poskytuje zabezpečení, správu paměti a zpracování výjimek.

.NET získal širokou podporu ze strany programátorů a vývojářů, kterým se líbil konzistentní programovací model s přímou podporu zabezpečení, menší komplexní vývoj a ladění. Tím se aplikace stávala lépe udržovatelnou a nabízela lehčí možnost instalace na počítač. Díky tomuto přístupu se .NET uchytil v programátorském světě.

IT průmysl prochází neustálými změnami. Nevyhnuly se ani Microsoftu a jejímu .NET Framework. Největší změna nastala s příchodem nového frameworku .NET Core. [5].

3.1 .Net Core

Přibližně po 14 letech se Microsoft rozhodl uskutečnit další krok, který vedl k tomu, že vyvinul .NET Core. Rozhodl se udělat technologii, která nebude podporována pouze na platformě Microsoft, ale bude možné s její pomocí vyvíjet i na různých platformách (Mac OS, Linux). Vznikla tak multiplatformní technologie .NET Core, která je open source. Všechny zdrojové kódy, dokumentace a další věci, včetně diskuze s vývojáři, kteří se zabývají vývojem nové platformy, jsou k nalezení na GitHubu (https://github.com/dotnet/core).

V současné době je na .NET Core zaměřena pozornost vývojářů od Microsoftu, kteří se snaží o optimalizaci platformy, aby si byli jistí tím, že se jedná o dosud nejrychlejší vyvinutou platformu. Pokud programátor bude potřebovat technologii, která bude zvládat velké množství požadavků, při rychlé reakci na každý požadavek, tak nemůže najít jinou technologie než .NET Core vytvořený od Microsoftu.

Jelikož se jedná o čerstvou technologii, o které zatím nejsou vedeny debaty o problémech a jejich řešeních na diskuzních fórech a jiných stránkách, může se to jevit, jako nevýhoda. Při pokusech o optimalizaci se zatím nedosáhlo toho, aby nová technologie měla všechny funkce z předešlých technologií. Z toho důvodu odborníci radí programátorům, kteří chtějí psát nové aplikace pro klienty, aby psali aplikace v nové technologii .NET, ale přitom stále využívali podporu plného starého frameworku. A za několik let provedli úplnou migraci na novou technologii pomocí NuGet balíčků.

Během několika dalších let se daná technologie stane více robustní, vznikne větší množství programátorů, kteří budou tuto technologii využívat. Dále budou doprogramovány zbývající funkcionality starých frameworků do nové technologie a odstraněny některé chyby, které zatím nejsou objeveny.

3.2 NuGet balíček

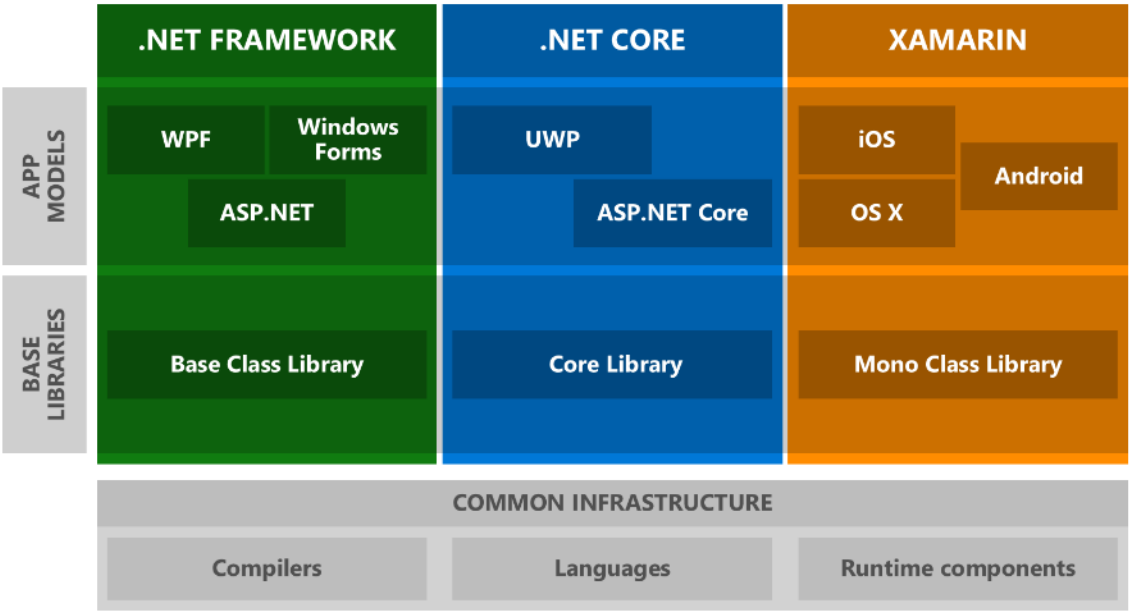
NuGet balíček je univerzální balíčkovací systém určený nejen pro web aplikace, ale i pro ostatní aplikace. Jedná o knihovnu, které má některé funkce, které jsou právě potřeba. Kdokoliv z programátorů může být autorem NuGet balíčku. Autor balíčku se rozhodne, zda chce, aby jeho balíček byl k dispozici pro širokou veřejnost nebo si ho ponechá jen pro firemní/soukromé účely.

Na web stránce www.nuget.or je NuGet repository, který slouží k prohledávání všech dostupných balíčků, které jsou na stránce k dispozici. Může se zde najít řešení problému, se kterým si programátor neví rady nebo vytvořenou knihovnu, kterou potřebuje a provede v ní potřebné úpravy.

Výhodou NuGet balíčku je, že se programátor nemusí starat o aktualizování daného balíčku. Tím je myšleno, že se nemusí hlídat, zda se na daném balíčku nevyskytla novější verze, kterou je potřeba stáhnout a zakomponovat znovu do aplikace. Místo toho jen zkontroluje možnost novějších verzí a pokud se rozhodne pro aktualizaci, tak vybere možnost aktualizace a stávající verze se nahradí novější.

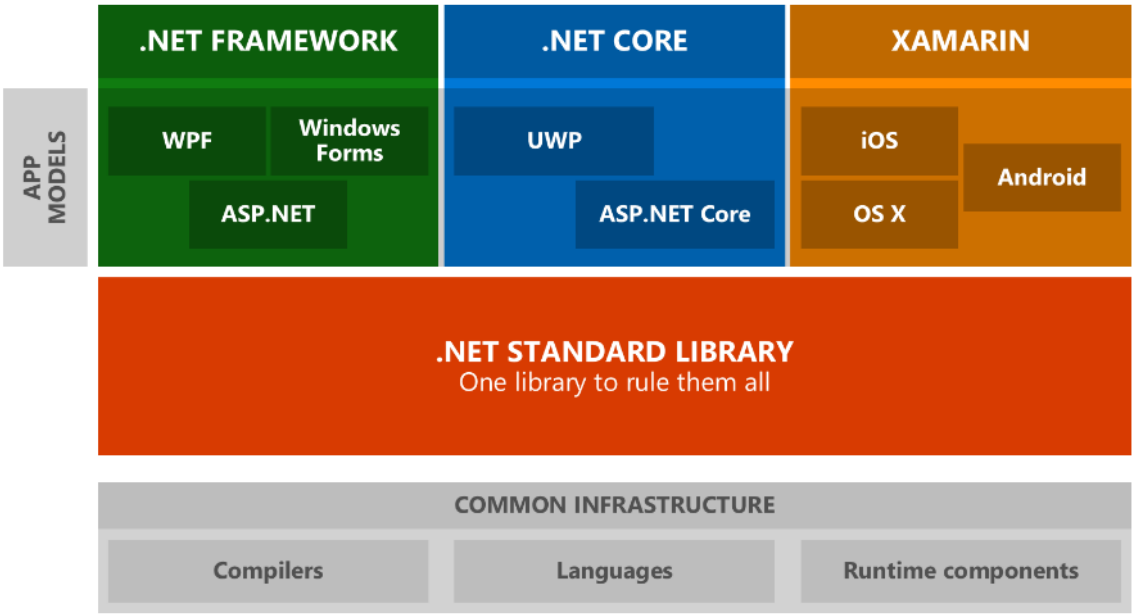
3.3 .NET Standard

Před zavedením .NET Standard existovaly tři hlavní odnože .NET. Což znamenalo, že bylo potřeba ovládat tři různé knihovny, pokud jsme chtěli psát program, který funguje na všech třech platformách viz Obr. 3. Vzhledem k tomu, že IT průmysl je rozmanitější, než byl v počátku vzniku .NET, bylo zapotřebí, aby přišel Microsoft se změnami, které by vedli k většímu pohodlí programátorů. Nebylo by potřeba znát tři základní knihovny, jejich funkce a myslet pokaždé na vkládání všech tří knihoven.



Obr. 3 NET základní knihovny [6]

To vše vedlo k tomu, že vznikla nová knihovna, která je společná pro všechny platformy a nese název .NET Standard Library. .NET Standard Library je formální specifikace .NET API, která nabízí jednotnou knihovnu se všemi API funkcemi pro všechny platformy a programátoři už nemusí hledat API funkce, které jim nabízela každá knihovna zvlášť. Teď mají jistotu, že když budou potřebovat funkci z knihovny, vědí, že je stejná pro všechny platformy, na kterých bude běžet jejich aplikace viz Obr. 4. [6]



Obr. 4 NET Standard [6]

4 DATABÁZE SQL

SQL (Structured Query Language) je Strukturovaný dotazovací jazyk, který vypadá jako strojová angličtina a slouží pro ukládání, manipulaci a načítání dat uložených v relační databázi. Má však přesně definovaná syntaktická a lexikální pravidla. [1]

SQL je standartní jazyk pro relační databázový systém. Všechny relační databáze používají podobný dotazovací jazyk SQL. Mezi základní dotazy, které slouží k potřebné manipulaci s daty z databáze jsou jednoduchá anglická slovíčka: [1]

- CREATE – vytvoří novou tabulku v databázi
- SELECT – vybere požadované sloupce v tabulce
- INSERT – vytvoří nový záznam v tabulce
- UPDATE – aktualizuje záznam v tabulce specifikovaný příkazem WHERE
- DELETE – smaže záznam z tabulky
- WHERE – slouží k přesnějšímu určení hodnot určených k práci

4.1 Bezpečnost

Mezi důležitou část zabezpečení, na kterou musí správce databáze klást důraz, jsou uživatelské účty a jejich role, které určují přístup a práva do konkrétních databází konkrétnímu uživateli. Názorné ukázky jsou prezentovány v MS SQL Server 2014 Management Studio. Management studio bylo vybráno z důvodu, že je volně dostupné na internetu.

4.1.1 Vytvoření uživatelského účtu

Při vytváření nového uživatelského účtu, jeho přístupových práv a rolí, je důležité nezapomenout, aby novému uživateli byla přidělena práva a role, které mu budou dostatečně stačit k tomu, aby pokryly všechny jeho potřeby a zabránily mu vytvoření škody v databázi.

Databáze je názornou ukázkou víceuživatelského prostředí. Přístup do databáze má více uživatelů různými způsoby. Část může data v databázi aktualizovat mazáním, vkládáním nových dat nebo změnou aktuálních dat. Další si data mohou jenom prohlížet a pracovat s nimi bez možnosti jejich změny. Proto je důležité vědět, co daný uživatel potřebuje dělat s daty a podle toho mu přesně nastavit jeho účet s právy.

MS SQL je úzce spojen s operačním systémem Windows. Pro přístup do databáze může zvolit administrátor dva způsoby:

- Integrované zabezpečení Windows
- Autentizace SQL Serveru

Zabezpečení Windows využívá prvky operačního systému, který má přístup k uživatelskému účtu a tím může sdílet citlivé informace o uživateli s MS SQL. Uživateli je pak vytvořen stejný účet, jako má při přihlášení ke svému účtu na počítači a tím mu odpadá starost, aby si musel pamatovat přístupové heslo. [3]

Pokud na jednom počítači pracuje více lidí pod stejným účtem, potom se zvolí autentizace MS SQL. Ta spočívá v tom, že se vytvoří nový účet pro přístup do databáze potřebnému uživateli.

4.1.2 Vytvoření uživatele v databázi

Vytvoření nového uživatele pro příslušnou databázi je možné SQL Server Management Studiu. V databázi, kde se má vytvořit nový uživatel, se ve složce *Security* vybere položka *User* a pomocí pravého tlačítka se vybere *New User*. Vyplní se požadovaná políčka a v levé části okna je zobrazena možnost *Membership*. Tato část je důležitá pro nastavení práv daného uživatele. [3]

4.1.3 Role

Z vytváření uživatelů pro databázi je patrné, že jejich přístupová práva jsou skoro totožná. Přesněji řečeno se vytvářejí skupiny uživatelů se stejnými přístupovými právy, proto je důležité přidělovat uživatelům jejich role, které omezí jejich možnosti v dané databázi.

Každý uživatel může mít přiřazeno několik rolí a to samé platí i naopak, jedná se o vztah M:N. Role v databázi jsou rozděleny do dvou částí, na serverové a databázové role. Serverové role se týkají operací, které lze provádět na úrovni serveru. Osoba umístěná v této roli bude moci provádět vše, co role povoluje. Serverové role nelze žádným způsobem modifikovat, jejich práva jsou pevně nastavená. Seznam serverových rolí můžeme vidět ve složce *Security* a podsložce *Server Roles.* [3]

Databázové role určují práva každému uživateli pro jednotlivou databázi, do které má přístup. V některé může mít práva na editaci dat, zatím co do jiné databáze nemusí mít vůbec žádný přístup nebo velice omezený. Omezený může být tak moc, že se může do dané databáze jenom dívat na data, ale nemůže tam už provádět jejich editaci. Role pro každou databázi se nastavuje v sekci *Membership,* kde se uživateli přidávají práva do dané databáze. [3]

4.2 Zálohování

Do databáze ukládáme data s rozdílnou hodnotou informací. Kdyby se s databází něco stalo a data budou nenávratně ztracena nebo poškozena, vznikne nenahraditelná škoda. V některých situacích může jít o nemyslitelnou situaci, která se nikdy nesmí stát. Pro případy, že by se stala nečekaná věc, která by poškodila data, se provádí zálohování dat.

Zálohování dat je proces, při kterém dochází k vytvoření kopií požadovaných dat na záložní servery nebo uložiště, kde jsou data archivována a nepracuje se s nimi. Vytvoření záložních kopií je nezbytné pro nápravu vzniklých škod. Jde o prevenci pro případ výpadku systému, která umožní uvést databázi zpátky do původního nepoškozeného stavu. Pro zálohování dat z databáze se využívají 3 modely: [3],[1]

- Kompletní zálohování
- Diferenciální zálohování
- Záloha translačního protokolu

Kompletní zálohování zálohuje všechny údaje, které jsou uloženy v databázi. Obnova údajů je jednoduchá, protože se provádí obnovení celé databáze ze zálohy. Nevýhodou je, že se provádí záloha celé databáze i dat, které nebyly od poslední zálohy pozměněny.

Diferenciální zálohování je rozdílné oproti kompletní záloze, že provádí zálohu jen těch dat, které byly pozměněny od poslední kompletní zálohy. Při obnově se nejdříve obnoví data z kompletní zálohy a až po nich se začnou obnovovat data z diferenciální zálohy. Velkou výhodou je, že se zálohuje menší množství dat. U transakčního protokolu se zálohují pouze transakce, které byly provedeny od poslední kompletní zálohy databáze. [1]

4.3 Transact SQL

Hlavní nevýhodou jazyka SQL je, že se při psaní dotazu nedá využívat žádných procedur, cyklů nebo podmínek. Proto má většinou každá moderní databázová platforma implementované určité rozšíření jazyka SQL. Díky tomuto rozšíření se jazyk SQL stává velice schopným jazykem, který umožnuje naprogramovat i ty nejsložitější algoritmy pro práci s daty. Rozšíření jazyka SQL na platformě Microsoft SQL Serveru má název Transact SQL, zkráceně T-SQL.

Některé bloky příkazů se vykonávají v cyklech, některé jen za určitých podmínek. Blok příkazů je v jazyku T-SQL ohraničený pomocí klíčových slov BEGIN a END: [3]

```
BEGIN
    (blok příkazů)
END
```

4.3.1 Deklarace proměnných

Nabídka podporovaných datových typů je u jazyka T-SQL velmi bohatá. K dispozici je vždy několik datových typů různého rozsahu pro textové proměnné, číselné proměnné, datum a čas, jednotky peněžní měny, logické proměnné a binární proměnné. Proměnné se deklarují pomocí příkazu DECLARE: [1]

```
DECLARE @jméno_proměnné datový_typ
```

4.3.2 Podmínka

Základní konstrukcí pro tvorbu podmínek je IF..ELSE. Jednotlivé bloky jsou ohraničené pomocí BEGIN..END: [3]

```
IF (podmínka)
BEGIN
    (blok příkazů)
END
ELSE IF (podmínka)
BEGIN
    (blok příkazů)
END
ELSE
BEGIN
    (blok příkazů)
END
```

4.3.3 Cyklus WHILE

Pro vytvoření cyklu se používá příkaz WHILE: [3]

```
WHILE podmínka
BEGIN
    (blok příkazů)
END
```

Cyklus se opakuje tak dlouho, dokud je splněna podmínka za příkazem WHILE. Z cyklu je také možné vyskočit dříve, pomocí příkazu BREAK.

4.4 Uložené procedury

Jak už vyplývá z názvu kapitoly, jedná se o uložené procedury přímo v databázi spolu s ostatními daty. Z praktického hlediska je možné do databáze uložit i skoro celou aplikační logiku pro zpracování těchto údajů. To umožnuje nejen jednodušší logiku aplikace komunikující s databází, ale zvyšuje to i spolehlivost celé aplikace, protože uložené procedury jsou zálohované společně s daty.

Tento způsob se v minulosti doporučoval, ale z praktického hlediska se doporučuje efektivní návrh tabulek na základě ukládaných a dotazovaných dat a správně rozvrhnout poměr logiky v databázi a v aplikaci, která využívá danou databázi. Uloženou proceduru je možné vytvořit pomocí příkazu CREATE PROCEDURE. Ukázka základní syntaxe pro uložené procedury vypadá následovně: [1]

```
CREATE PROCEDURE název_procedury [(seznam_parametrů)]
AS
        (blok příkazů)
GO;
```

Kód může obsahovat jakoukoliv kombinaci cyklů a podmínek z kapitoly T-SQL, ale také může obsahovat další uloženou proceduru, na kterou se odkazuje. Pro spuštění jiné procedury, stačí použít příkaz EXECUTE.

```
CREATE PROCEDURE procedura_1
AS
…
GO;
CREATE PROCEDURE procedura_2
AS
…
EXECUTE procedura_1
…
GO;
```

4.5 Funkce

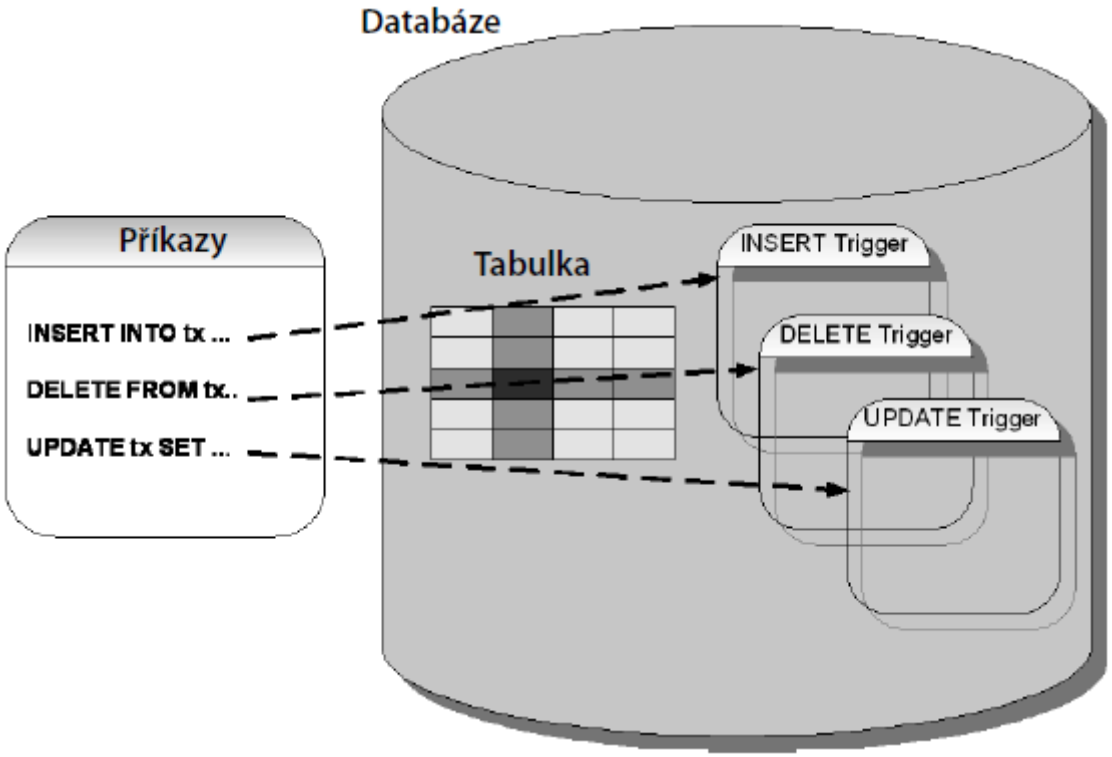
Funkce je speciálním případem uložené procedury, kdy dochází k návratu proměnné nebo rovnou celé tabulky hodnot. Tyto hodnoty se získávají po provedení těla funkce. Pro vytvoření funkce se používá příkaz CREATE FUNCTION: [3]

```
CREATE FUNCTION název_funkce [(seznam_parametrů)]
RETURNS (skalární_typ | TABULKA)
AS
BEGIN
        (blok příkazů)
RETURN (výraz stejného typu, jako je deklarovaný)
– v případě skalárního typu
– nebo
RETURN (příkaz SELECT)
– v případě výsledku typu TABULKA
```

```
END;
```

4.6 Trigger (Spouštěč)

Trigger je uložená procedura, která se volá automaticky, po vykonání předem definovaných událostí, které nastávají při manipulaci s daty, např. při vkládání, aktualizaci nebo mazání dat z tabulek. Trigger se nikdy nespouští pomocí zavolání uložené procedury, je totiž navázán na příkazy pracující s daty (INSERT, UPDATE, DELETE). Cílem triggeru není zálohování upravených dat s možností jejich obnovení, ale slouží ke kontrole zadávaných údajů, zjištění datové integrity a podobně. Na Obr. 5 je ukázáno, na jaké příkazy se může trigger navázat. [3]



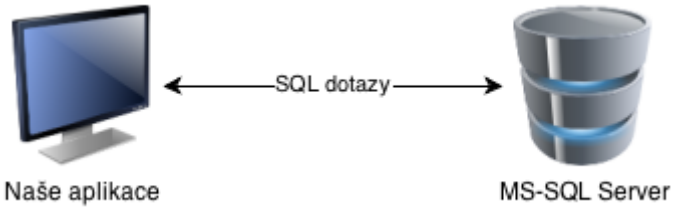
Obr. 5 Princip triggeru [3]

5 PŘIPOJENÍ K DATABÁZI

Aplikace, které pracují s různými daty a informacemi, si je musí někde ukládat, případně načítat. Řešením tohoto problému je databáze. Základní informace o databázi jsou uvedené v předešlé kapitole. Tato kapitola je zaměřena na možné technologie, které slouží k přenosu dat mezi aplikací a databází.

5.1 Připojená aplikace

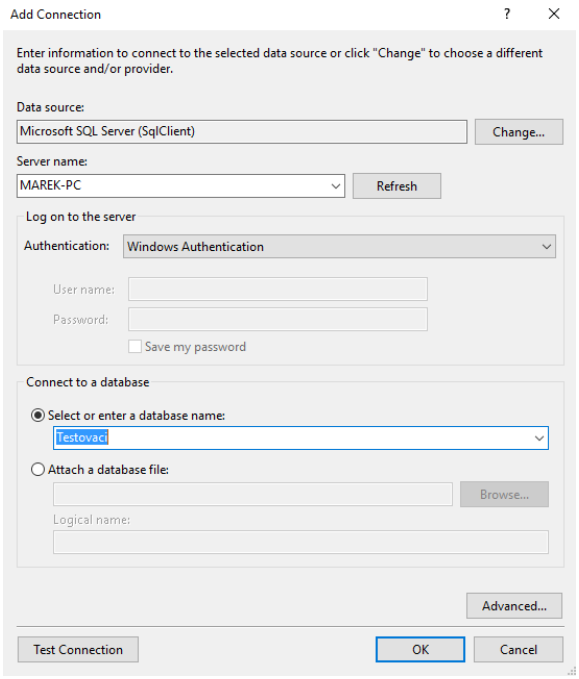
Přístup připojené aplikace se používá ve chvíli, kdy je potřeba intenzivně komunikovat s databází, číst nebo upravovat uložená data. Pomocí tříd *DataReader*, *Command* a *Connection* posíláme databázi přímo příkazy v jazyce SQL a dostáváme výsledky. Spojení mezi databází a aplikací je navázáno pokaždé, když se posílá SQL požadavek. Po jeho vyřízení se spojení opět ukončí. Princip fungování je znázorněn na Obr. 6: [18]



Obr. 6 Princip připojené aplikace [18]

5.1.1 Připojení

Pro připojení aplikace k databázi je zapotřebí tzv. *ConnectionString*. Jak je patrné z názvu, jedná se o řetězec, který obsahuje potřebné údaje k připojení k databázi. Většinou to bývá název databáze a heslo pro přístup viz. Obr 7.



Obr. 7 Visual studio – Napojení databáze

Existují dvě možnosti, jak získat *Connectionstring*. První možnost je postupovat dle návodu Visual studia, který ho vygeneruje nebo použijeme *SQLConnectionStringBuilder*, která se vytvoří přímo v programu. ,

V první variantě je potřeba ve Visual Studiu zobrazit *Server Explorer* (*View → Other Windows → Server Explorer*) a kliknout na přidat nové připojení. Zobrazí se okno, kde se vyplní potřebné údaje. V prvním okně *DataSource* se nastaví *Microsoft SQL Server (SqlClient)*, u *Server name* vybere SQL server, který se bude využívat. Nakonec se vybere databáze, na kterou se chceme připojit. V případě nejistot a ověření správného vyplnění, se klikne na tlačítka *Test Connection*, které je vidět na Obr. 10 vlevo dole. Po dokončení spojení se nám připojí databáze a po zobrazení *Properties* daného spojení, si můžeme přečíst *ConnectionString*. Ukázka vygenerovaného *ConnectionString* vypadá následovně: [19]

```
connectionString="Data Source=MAREK-PC;Initial
Catalog=Testovaci;Integrated Security=True"
```

Druhá možnost psaná v programu vypadá následovně:

```
SqlConnectionStringBuilder csb = new SqlConnectionStringBuilder();
csb.DataSource = @"MAREK-PC";
csb.InitialCatalog = "Testovaci";
csb.IntegratedSecurity = true;
string pripojovaciRetezec = csb.ConnectionString;
```

Důležité je, aby se nezapomněl připsat jmenný prostor (*using System.Data.SqlClient*).

Pro samotné připojení v programu je potřeba použít následující kód, abychom mohli začít pracovat s daty v databázi. Tento kód nám otevře spojení mezi databází a aplikací: [19]

```
SqlConnection pripojeni = new SqlConnection(connectionString)
pripojeni.Open();
pripojeni.Close();
```

5.1.2 Práce s SQL příkazy

V minulé podkapitole je ukázáno, jak se lze připojit do databáze. V této kapitole je ukázáno, jak se používají funkce, které komunikují s databází. Pro práci s databázovými příkazy slouží třída *SqlCommand*, která umožnuje spustit na SQL databází jakýkoliv dotaz. Ve třídě *Command* je potřeba naplnit hodnotu *Connection* řetězcem *Connectionstringem*. Do *CommandText* se píše SQL příkaz, který se má provést. Ke spuštění příkazu slouží metody *ExecuteReader()*, *ExecuteScalar()* a *ExecuteNonQuery()*. První metoda se využívá, pokud se vrací více dat (typický je *SELECT* s více daty). Druhá metoda slouží k návratu jedné hodnoty (při použití tzv. agregačních funkcí: počet hodnot v tabulce, součet hodnot atd.). Poslední metoda se používá pro změnu hodnot v databázi (*INSERT, UPDATE, DELETE*). Návratová hodnota této metody říká, kolik bylo pozměněno řádků. [19]

```
SqlCommand prikaz = new SqlCommand();
prikaz.Connection = pripojeni;
prikaz.CommandText = "SELECT COUNT(*) FROM Word";
int pocetSlovicek = (int) prikaz.ExecuteScalar();
pripojeni.Close();
```

Další důležitá metoda, která je neodmyslitelnou součástí přenosu dat je *AddWithValue*. Tato metoda se používá k přidávání parametrů. Existuje ještě jeden způsob, který je programově

přijatelnější, ale z bezpečnostního hlediska je nepřipustitelný. Nepodporovaný princip je založen na tom, že se do řetězce, kde se píše SQL příkaz vloží i parametry ve formátu *string*. Tento způsob se rychleji naprogramuje, ale není zabezpečen před tím, že by se v parametru mohl skrývat další SQL příkaz, který by nám celou databázi mohl smazat. Tento způsob napadnutí databáze je znám pod názvem SQL injekce. Abychom tomu zabránili, parametry se vkládají zvlášť a místo hodnoty v SQL dotazu se píše jenom odkaz na hodnotu se znakem @:[28]

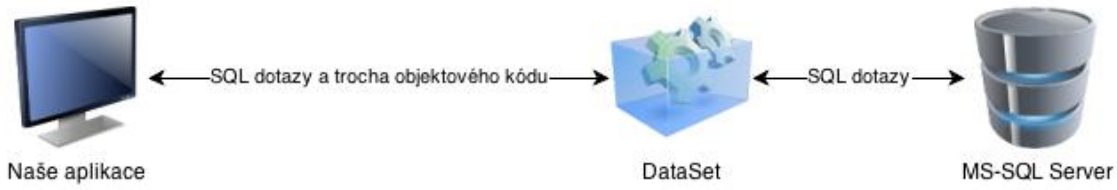
```
SELECT [Czech] FROM [Word] WHERE [English]=@slovo
prikaz.Parameters.AddWithValue("@slovo", slovo);
```

Jak je ukázáno na příkladu výše, jsou zde použité hranaté závorky. Tyto závorky jsou použité proto, aby byla ukázána možnost označení sloupečků. Dělá se to tak proto, že název sloupečku je stejný jako nějaký příkaz SQL. Tímto způsobem se dosáhne rozlišení sloupečku od příkazu SQL. [17]

5.2 Odpojená aplikace

Přístup odpojené aplikace se používá ve chvíli, kdy není vyžadováno načítání nových dat z databáze a postačí jedno načtení a budeme s daty pracovat delší dobu. Data se z databáze stáhnou do operační paměti, kde se uloží do tzv. *Dataset*. Aplikace pracuje s daty uloženými v *DataSetu* a jednou za čas provede aktualizaci dat v databázi a nahraje do ní případné změny v *DataSetu*. Použitím této technologie se nepracuje s nejaktuálnějšími daty, ale aplikace má rychlejší odezvu, protože nemusíme komunikovat s databází a čekat na její odpověď.

DataSet obsahuje tabulky, které jsou objekty. Do tabulky se mohou přidávat řádky a upravovat je bez psaní SQL příkazů. Pro spuštění dotazu na databázi se používá *DataAdapter*, pomocí kterého se naplní *DataSet* daty. Příkazy jsou už psány SQL jazyce. Princip fungování je znázorněn na Obr. 8: [18]



Obr. 8 Odpojená aplikace-princip [18]

5.2.1 Připojení

Způsob připojení databáze a získání *Connectionstringu* je popsán v předešlé podkapitole Připojená aplikace, proto zde není popsána.

5.2.2 DataSet

Jak už vypovídá z názvu, jedná se o množinu dat, které nám reprezentují data z databáze. Naplnění se provádí pomocí objektu *SqlDataAdapter*. Tomuto objektu se předává dotaz a spojení. Výsledný objekt se zapisuje do třídy *DataSet* pomocí metody *Fill*. Názorná ukázka kódu: [9]

```
SqlDataAdapter adapter = new SqlDataAdapter(dotaz, spojeni);
DataSet vysledky = new DataSet();
adapter.Fill(vysledky);
```

Na ukázce kódu níže je znázorněný výpis dat získaných z databáze:

```
vysledky.Tables[0].Rows[0]["Czech"]
```

Výsledek je třída *DataSet*, která obsahuje objekt *Tables*. V tomto objektu se nachází tolik tabulek, kolik jsme si vybrali pomocí příkazu *SELECT*. Tabulky jsou značené od 0 až do počtu vybraných tabulek (pokud chceme načítat více tabulek, je důležité do *DataSetu* vkládat tabulky postupně, aby se mohly dobře editovat). Dále má každá tabulka určitý počet řádků, které jsou značené od nuly a každý řádek má určitý počet sloupců, u kterých máme možnost výběru. První možnost je volba sloupce podle čísla nebo podle jejich názvu z databáze. Na příkladu je znázorněna ukázka z databáze.

Princip předávání parametru z *DataSetu* funguje stejně jako u připojené aplikace. Jediný rozdíl je v použitých metodách, které se musí použít, aby se předaly parametry do SQL dotazu, viz. níže:

```
adapter.SelectCommand.Parameters.AddWithValue("@slovo", slovo);
```

5.2.3 Úprava dat v DataSetu

Aby mohl být vložen nový záznam do tabulky, je potřeba postupovat následovně. Nejdříve se vytvoří nový řádek tabulky *DataRow*, pomocí metod *NewRow*. V dalším kroku se do nového řádku napíší hodnoty, které musí být vepsány pro každý sloupeček zvlášť. Nakonec se provedené změny v řádku přidají do tabulky v *DataSetu* a přidání nového řádku je hotové. [10]

```
DataRow noveSlovo = datovaSada.Tables["Slovicka"].NewRow();
noveSlovo ["Czech"] = "míč";
datovaSada.Tables["Slovicka"].Rows.Add(noveSlovo);
```

Je-li potřeba nějaký řádek v tabulce upravit, je zapotřebí ho nejdříve najít a vybrat do samotného řádku v třídě *DataRow*. Po nalezení požadovaného řádku se spustí úprava pomocí *BeginEdit*, která povolí provádět úpravy v daném řádku. Po provedení všech úprav se ukončí úpravy metodou *EndEdit*. [10]

```
DataRow[]Radky = datovaSada.Tables["Slovicka"].Select("Czech='míč'");
DataRow radek = Radky[0];
radek.BeginEdit();
radek["Czech"] = "Balón";
radek.EndEdit();
```

V případě potřeby mazání řádku se postupuje skoro stejným způsobem jako u upravování řádku. Najde se požadovaný řádek, který má být smazán a na daný řádek se použije metoda *Delete*, která smaže celý řádek.

```
radek.Delete();
```

5.2.4 Přenos z DataSetu do Databáze

Jelikož se jedná o způsob přenosu dat, který si všechna data vždy stáhne z databáze a pracuje se s nimi v programu, tak je potřeba, aby se všechny provedené změny přenesly a uložili do databáze. Za normálních podmínek by bylo potřeba napsat příslušný SQL příkaz k tomu, aby se dané změny provedly. Protože psaní těchto příkazů je velice zdlouhavé a šablonovité, proto existuje třída *SqlCommandBuilder*, která celou práci zjednodušuje a vše se provádí automaticky. Provedení příkazu je ukázáno na kousku kódu: [10]

```
SqlCommandBuilder cbSlovicka = new SqlCommandBuilder(adapter);
adapter.Update(datovaSada.Tables["Slovicka "]);
```

5.3 Entity framework

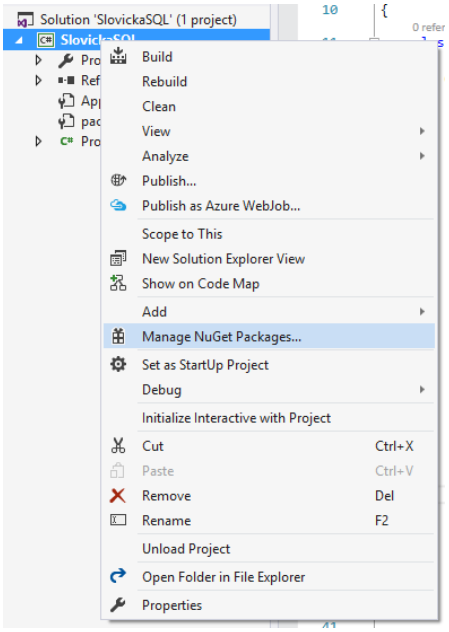
Pomocí Entity frameworku se vytváří datové třídy (objekty) odpovídající tabulkám v databázi. S těmito objekty můžeme pracovat jako s normálními daty v programu a nemusíme se starat o psaní SQL kódu pro aktualizaci dat v databázi. O tuto práci se stará Entity framework.

První programátoři mohli tento framework začít používat v roce 2008. Za několik let si prošel velkými změnami. Začínal jako normální framework, ale postupně ho Microsoft předělal do NuGet balíčku. Poslední verze Entity framework 6 je už dostupná pouze v NuGet balíčku.[4]

5.3.1 Objektově relační mapování

Entity framework patří do skupiny objektově relačně mapovaných (ORM) produktů Microsoftu. ORM je programovací technika, která mapuje relační databáze entity k objektům v určitém programovacím jazyce. Každá databáze je reprezentována objektem ORM v konkrétním programovacím jazyku. Databázové tabulky jsou reprezentovány třídami se společnými vztahy těchto tříd. ORM je zodpovědný za mapování a spojení mezi třídami databáze tak, aby v aplikaci byla databáze plně zastoupena třídami. Aplikace se nemusí starat o to, jak se připojí k databázi, jaké použije SQL příkazy atd., o všechno se postará ORM. [16]

5.3.2 Instalace Entity frameworku



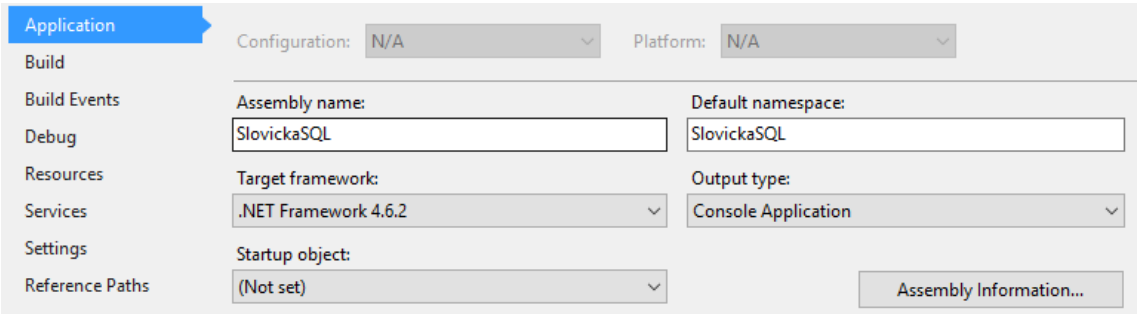
Obr. 9 Entity framework-NuGet balíček

Pro využívání Entity frameworku v aplikaci, je potřeba si ho stáhnout a nainstalovat pomocí NuGet balíčku. To můžeme udělat v jakémkoliv projektu, stačí jenom na projektu vybrat Manager NuGet Packages. Po výběru se zobrazí okno s vyhledáváním, kde se zadá Entity framework a vybereme požadovaný balíček, který se má stáhnout a nainstalovat. Ukázka výběru Manageru pro NuGet Balíčky je na Obr. 9.

5.3.3 Vytvoření Entity datového modelu

V minulé podkapitole je ukázáno, jak si stáhnout a nainstalovat Entity framework. Tato podkapitola je zaměřena na vytvoření entity modelu k existující databázi a pochopení základů. Nejdříve je potřeba si otevřít Visual Studio s projektem, kde je stažený a nainstalovaný Entity framework. Potom je potřeba na daném projektu otevřít *Properties* a nastavit *Target framework* na .NET Framework 4.6.2. Ukázku výběru Entity frameworku je ukázána na Obr. 10. Jedná se o nejnovější plnohodnotný framework, který Microsoft vydal. Je možnost si stáhnout ještě novější framework, ale ten je již vyvinutý na nové architektuře .NET Core, která ještě neobsahuje všechny funkce, které jsou na starém frameworku. Proto je řešením, prozatím použít plný framework a čekat, až bude více funkcí v .NET Core a potom provést migraci.

Po nastavení .NET frameworku už zbývá jen přidat *ADO.NET Entity Model*. Začátek bude stejný jako u NuGet balíčku, ale místo NuGet Packages zvolíme možnost *Add* a vybereme možnost *New Item*. Po kliknutí se zobrazí nabídka *New Item* a vybereme požadovaný model *ADO.NET Entity Data Model*.



Obr. 10 Entity framework-výběr frameworku

Po kliknutí na model se nám otevře *Entity Data Model Wizard*, kde je na výběr ze čtyř možností:

- *EF Designer from database* – slouží pro vytváření objektu na základě existující databáze
- *Empty EF Designer model* – vytvoří prázdný model v aplikaci, který se bude aktualizovat v závislosti na databázi
- *Code First from database* – vytvoří databázi v závislosti na datovém modelu aplikace
- *Empty Code First mode l* – vytvoří prázdný databázový model, který se bude aktualizovat v závislosti na modelu aplikace

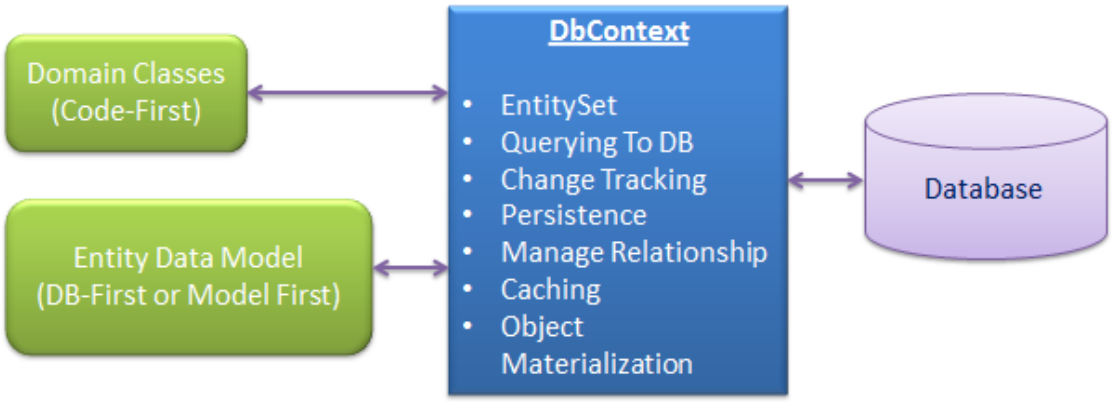
Dále se bude pracovat pouze s *EF Designer from database*. Po zvolení první možnosti se vybere databáze, ze které se bude vytvářet model a pokud neexistuje žádné spojení, musí se vytvořit nové. Při pokračování se zobrazí nabídka s výběrem tabulek, pohledů a uložených procedur. Po výběru obsahu modelu z nabídky je možnost zaškrtnutí všeho, co se má importovat společně s modelem a tím nastavit jeho vlastnosti. Objasnění možností pro zaškrtnutí nastavení modelu:

- *Pluralize or singularize generated object names* – u vazeb 1:1 nechává název v jednotném čísle a jestli se jedná o jiný druh vazby, potom se název změní na množné číslo
- *Include foreign key columns in the model* – existují-li vazby mezi tabulkami 1:N, tak u každé vazby N bude dopsaný i identifikátor hodnoty v databázi
- *Import selected stored procedures and functions into the entity model* – importuje vybrané uložené procedury a funkce. Před .NET Frameworkem 5.0 se tento import musel provádět ručně. [Entity Model]

Po dokončení výběru se stiskne tlačítko *Finish* a tím se ukončí celé tvoření modelu a proběhne generování objektů z databáze do aplikace. Visual Studio k tomu vytvoří i *Class diagram* s vazbami k nahlédnutí. [14]

5.3.4 DbContext

DbContext je důležitou součástí Entity Frameworku. Zajišťuje spojení mezi databází a objekty v aplikaci a práci s nimi viz Obr. 11.



Obr. 11 DbContext-schéma [14]

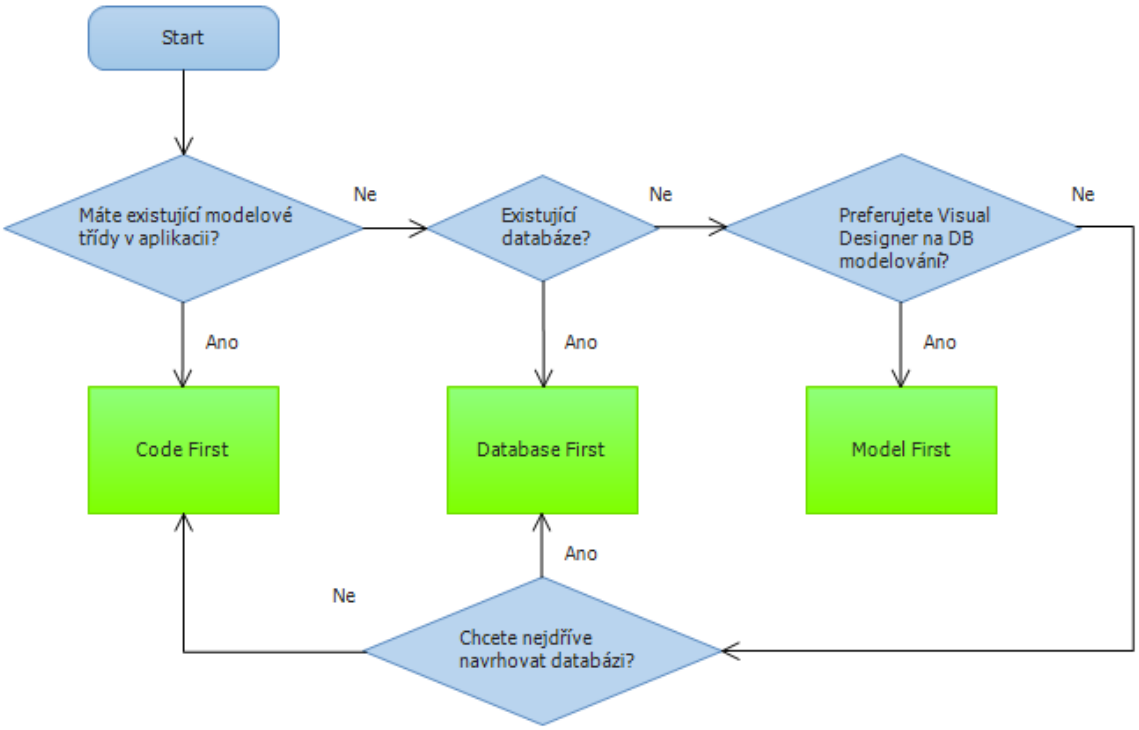
Dále je odpovědný za následující činnosti: [11]

- *EntitySet* – obsahuje sadu entit, pro všechny subjekty, které jsou mapovány na tabulky databáze
- *Querying to DB* – převádí LINQ to Entities dotazy do SQL dotazů a odesílá je do databáze
- *Change tracking* – udržuje informace o změnách, ke kterým došlo v aplikaci a po stažení z databáze
- *Persistence* – provádí vložení, aktualizaci, a odstranění na základě stavů entit
- *Manage Relationship* – řídí vztah za použití CSDL, MSL a SSDL v DB-First nebo Model-First přístupu nebo použitím API v Code-First přístupu
- *Caching* – provádí ukládání dat do mezipaměti v aplikaci, které získal během doby životnosti třídy kontextu
- *Object Materialization* – převádí řádky tabulek dat do entit objektů

Entity framework využívá tři rozdílné vývojové přístupy k používání entity frameworku v aplikaci:

- *Code First* – Používáme tehdy, máme-li už naprogramované třídy v programu, a chceme celé objekty přenášet do databáze, ale nemáme udělaný žádný model.
- *Model First* – Podstata model First je ta, že se nejdříve vytvoří model všech tříd s vazbami ve Visual Studiu, který poslouží k vygenerovat celá databáze a tříd v aplikaci.
- *Database First* – Tento princip využijeme tehdy, máme-li k dispozici už existující databázi. Aplikaci připojíme k databázi a potom nám Visual Studio vygeneruje celý model a třídy v aplikaci.

V případě nerozhodnosti při tvoření modelu se může použít diagram se zobrazeným možným řešením problému na Obr. 12:



Obr. 12 Entity framework-volba přístupu [11]

5.3.5 Vztahy mezi objekty

Entity framework podporuje celkově 3 druhy vztahů 1:1, 1:N, M:N. Důležité je vědět, jaký vztah mají třídy mezi sebou. Jestli z dané třídy může být použit jen jeden prvek z druhé třídy, potom se používá následující způsob odkazování:

```
public virtual název třídy název proměnné {get; set;}
```

Jedná-li se o druhou možnost, že se může odkazovat na více prvků, vypadá odkaz následovně:

```
public virtual ICollection<název třídy> název proměnné {get; set;}
```

Většinou název třídy a proměnné bývá totožný, lišit se může při generování v jednotném čísle názvu třídy a množném čísle názvu proměnné. [15]

5.3.6 Práce s daty

Aby se mohlo pracovat s daty z databáze v aplikaci, je nejdříve potřeba navázat spojení mezi aplikací a databází. Musí se vytvořit v aplikací proměnná, která se stará o spojení mezi databází a aplikací a přenosem dat mezi nimi. Pro vytvoření spojení slouží následující příkaz: [13]

```
Název_databázeEntities název_proměnné = new
Název_databázeEntities();
```

V aplikaci to vypadá následovně:

```
TestovaciEntities db = new TestovaciEntities();
```

Kde Testovací je název vytvořené databáze v MS SQL serveru.

Načítání dat z databáze:

Pro načítání dat z databáze slouží následující příkaz:

```
var WordsOnDatabase = (from d in db.Words select d);
```

Příkaz načte celou tabulku *Words* a uloží nám to do proměnné *WordsOnDatabase*, se kterou se dále pracuje.

Vytvoření nového záznam:

Pro vytvoření nového záznamu v proměnné *words* se používá následující příkaz:

```
db.words.Add(NewWord);
db.SaveChanges();
```

Kde první příkaz slouží k přidání nového záznamu typu *Word* k dalším záznamům v ní uložených. *NewWord* musí být stejného datového typu jako proměnná *words*. Další příkaz nám slouží k nahrání upravených dat v aplikaci do databáze.

Aktualizace stávajících dat:

Pro změnu dat stačí změnit data, která chceme změnit v proměnné *words* a použít výše zmíněný příkaz pro aktualizaci dat z aplikace do databáze:

```
db.SaveChanges();
```

Smazání záznamu:

Pro mazání dat slouží následující příkaz:

```
db.Words.Remove(WordDelete);
```

Kde *WordDelete* je datového typu *Word*. Pro nalezení požadovaného řádku můžeme použít následující příkaz, který je skoro stejný jako pro načítání dat z databáze:

```
var WordDelete = (from d in db.Words where d.ID == ID select d).Single();
```

Hledaný řádek se může hledat podle všech parametrů dané třídy. Příkaz S*ingle()* zajistí, že se vybere pouze jeden záznam, který je potřeba k identifikaci. Po nalezení a smazání požadovaného řádku se opět provede aktualizace dat s databází příkazem: [13]

```
db.SaveChanges();
```

5.3.7 Automapper

Automapper je NuGet balíček, který zajišťuje mapování jednoho objektu na druhý. Hlavní použití je mapování Entit do DTO (data transfer object) a zpět, protože se jedná skoro o totožné třídy. Proto existuje taková knihovna, která usnadní práci s kopírováním hodnot.

Mapování se používá proto, aby se neměnili data získaná z databáze, ale pracovalo se s jejich kopií. Další možností, proč používat mapování je nezveřejňování citlivých dat z databáze, které by se neměli dostat k uživateli. Pro nastavení mapování je důležité znát příkazy pro vytvoření mapování ze zdroje a uložení hodnot v cíli.

```
Mapper.Initialize(cfg =>
{
    cfg. CreateMap<zdroj, cíl>();
});
```

Příkaz uvedený výše vytvoří mapovaní mezi třídou, ze které se budou data kopírovat a třídou, kam se budou data kopírovat. Je doporučeno při používání více mapování na různé třídy používat každou inicializaci třídy zvlášť, protože při rozsáhlých a komplikovanějších programech je vytváření mapování komplikovanější a obsáhlejší a vnikal by v tom chaos. Při inicializaci každého nového mapování se předchozí mapování smaže a musíme ho v případě využití inicializovat znovu. Pokud nechceme nějaké hodnoty kopírovat, rozšíříme o specifikaci, která vypadá následovně:

```
cfg.CreateMap< zdroj, cíl >()
.ForMember(dest => dest.Nazev_promene, opt => opt.Ignore());
```

V případě, že se hodnota nebude kopírovat, ale v cílové třídě se bude nacházet, její hodnota bude `null` nebo v případě `integer` 0. Pokud se bude provádět přepsání stávající proměnné s daty novými, stará data se smažou. Pro kopírování mezi proměnnými se používá následující příkaz:

```
Trida_cile nazev_cile = Mapper.Map<trida_cile>(zdrojovy_objekt);
```

Automapper se snaží namapovat co nejvíce hodnot. Nejsnáze se mapují proměnné, které mají stejný nebo velice podobný název. V případě, že chceme mapovat do jiných proměnných je potřeba provést specifikaci daného mapování při jeho inicializaci. [27]

6 TECHNOLOGIE PRO DISTRIBUOVANÉ VÝPOČTY

Jedná se o technologie, pomocí kterých se vytvářejí aplikace pro distribuční systémy s modelem klient/server. Na platformě .NET patří mezi nejznámější technologie Windows Communication Foundation (WCF), která byla vyvinuta s .NET Frameworkem 3. Další technologií, která je k dispozici od .NET je ASP.NET Web API. Tato technologie je jenom jednodušší varianta, která slouží pro méně složité aplikace.

6.1 Windows Communication Foundation

WCF je název API navržené speciálně pro proces budování distribučních systémů. Na rozdíl od jiných technologií, které jsou k dispozici, WCF poskytuje jediný, jednotný a rozšiřitelný objektový model programování, který se může použít k interakci s celou řadou dříve používaných distribučních technologií. Když se vytváří aplikace pro distribuované výpočty za použití WCF, musí se vytvořit tři provázané sestavy: [2]

- *WCF Service assembly* – toto `*.dll` obsahuje třídy a rozhraní celé funkčnosti, které potřebujeme pro externí volání
- *WCF Service host* – tento aplikační modul je entita hostující aplikace *WCF Service assembly*
- *WCF client* – aplikace, která přistupuje k funkčnosti služby pomocí proxy adresy

6.1.1 Endpoint

Endpoint patří k jedné z nejdůležitějších částí služby. Pomocí něho je zprostředkována celá komunikace mezi hostující službou a klientem. V některých případech bývá prezentována jako ABC (Address, Binding, Contract) dohoda, která patří ke stavebním blokům WCF aplikace:

- *Address* – Popisuje polohu služby v síti. V kódu je reprezentována se `System.Uri` typu. Jinak je typické uložení hodnoty v `*.config` souboru.
- *Binding* – Říká způsob komunikace WCF služby. Jaký bude zvolen síťový protokol, mechanismy kódování a transakcí.
- *Contract* – Poskytuje popis jednotlivých metod atd., je nezávislý na volbě adresy a bindingu.

ABC neříká jakým postupem se má programovat. V mnoha případech začíná vyvíjení WCF popisem jednotlivých služeb a používaných metod a následně vytvořením adres a vazeb. [2]

Address

Adresa poskytuje dva důležité prvky. První je umístění endpointu v síti, tím je myšleno, na jaké adrese ho nalezneme a druhý prvek říká, jaký protokol budeme využívat pro komunikaci se službou.

Adresa podporuje následující přenosové protokoly: [2]

- HTTP – http|https://název počítače|domény[:port]/cesta
- TCP – net.tcp:// název počítače|domény[:port]/cesta
- MSMQ – net.msmq://hostname/[private]/queueName
- IPC – net.pipe:// název počítače|domény[:port]/cesta

Binding

Obsahuje sadu několika prvků, které definují způsob, jakým služba komunikuje s klientem. Transportní prvek a kódování zpráv jsou dvě nejdůležitější součásti bindingu. Níže jsou uvedeny nejčastěji používané typy: [26]

- *Basic Binding* – Základní binding ve WCF web service využívající protokol HTTP využívající třídu BasicHttpBinding.
- *Web Service Binding* – Vychází z *Basic binding* a navíc má několik WS-* specifikací. Využívá třídu WSHttpBinding.
- *IPC Binding* – Jedná se o nejrychlejší a nejbezpečnější binding ze všech. Využívá netNamedPipeBinding třídu.
- *TCP Binding* – Je považována za nejspolehlivější a pro komunikaci využívá TCP protokol ve stejné síti. Kódování zpráv se provádí v binárním formátu. Využívá NetTCPBinding třídu.
- *WS Dual Binding* – Stejný jako *WS binding* s rozdílem, že podporuje obousměrnou komunikaci. Využívá WSDualHttpBinding třídu.
- Web Binding – Je navržen tak, aby reprezentoval službu v podobě HTTP požadavků (HTTP GET, HTTP POST atd.). Využívá WebHttpBinding třídu.
- *MSMQ Binding* – komunikace pomocí MSMQ (message queue – velmi spolehlivé doručování zpráv, často používané). Využívá NetMsmqBinding třídu.
- *Federated WS Binding* – Specifická forma *WS binding* a nabízí podporu pro federované zabezpečení. Využívá WSFederationHttpBinding třídu.
- *MSMQ integration Binding* – komunikace mezi WCF aplikací a již existující MSMQ aplikací Využívá WSFederationHttpBinding třídu.

Contract

Contract vrstva se nachází na úrovni aplikační vrstvy, kde se provádí zpracování, řízení a přenášení dat. Contract je v podstatě rozdělený na čtyři druhy: [24]

- *Service contract* – Poskytuje informace o klientovi stejně jako o endpointech i protokolů, které mají být použity v komunikačním procesu.
- *Data contract* – Je zde definována výměna dat. Vyměněná data mezi klientem a službou musí být v souladu s data contract.
- *Message contract* – Slouží k řízení data contract. Nejdůležitějším úkolem je formátování typu parametrů pro SOAP zprávy. WCF používá formát SOAP za účelem komunikace. SOAP je zkratka pro Simple Object Access Protocol.
- *Policy and Binding* – Jsou zde definovány určité předpoklady pro komunikaci s obsluhou. Klient musí dodržovat tento contract.

6.1.2 Vytváření Endpointu

Endpointy je možné vytvářet dvěma způsoby.

Administrativním způsobem je zápis proveden do *.config souboru: [2]

```
<configuration>
<system.serviceModel>
<client>
<endpoint address = "net.tcp://localhost:8090/MagicEightBallService"
binding = "netTcpBinding"
contract = "ServiceReference1.IEightBall"
name = "netTcpBinding_IEightBall" />
</client>
</system.serviceModel>-
</configuration>
```

Další možností je vytvářit endpoint přímo ve zdrojovém kódu aplikace: [2]

```
// Create the host.
myHost = new ServiceHost(typeof(MathService));
// The ABCs in code!
Uri address = new Uri("http://localhost:8080/MathServiceLibrary");
WSHttpBinding binding = new WSHttpBinding();
Type contract = typeof(IBasicMath);
// Add this endpoint.
myHost.AddServiceEndpoint(contract, binding, address);
// Open the host.
myHost.Open();
```

6.1.3 Hostování služby

Po vytvoření WCF služby je dalším krokem nastavení způsobu hostování aplikace. To je známé jako WCF service hosting. WCF služba může být hostována některým ze čtyř následujících způsobů: [25]

- *IIS hosting* – Internet Information Services (IIS) je podobný modelu od ASP.NET, jenom hostuje WCF službu. Mezi vlastností IIS hostingu patří automatické zpracování aktivace služby. Rovněž nabízí sledování procesu nečinnosti vypnutí, recyklace a mnoho dalších funkcí s cílem usnadnit WCF service hosting.
- *Self-Hosting*– WCF služba je hostované v aplikaci, která ji spravuje. To vyžaduje potřebný kód pro ServiceHost inicializaci v aplikaci. WCF služba může být umístěna v různých aplikacích jako jsou konzolové aplikace, formuláře systému Windows atd.
- *WAS hosting* – Windows Activation Service (WAS), je nejvýhodnější hostování, protože stejné sledovací funkce jako IIS, ale je podporován na více protokolech.
- *Windows Service Hosting* – Jedná se o způsob hostování služby WCF service pro místní klienty systému. Všechny verze Windows podporují tento typ hostování a zde může Service Control Manager řídit proces životního cyklu služby WCF.

6.2 ASP.NET Web API

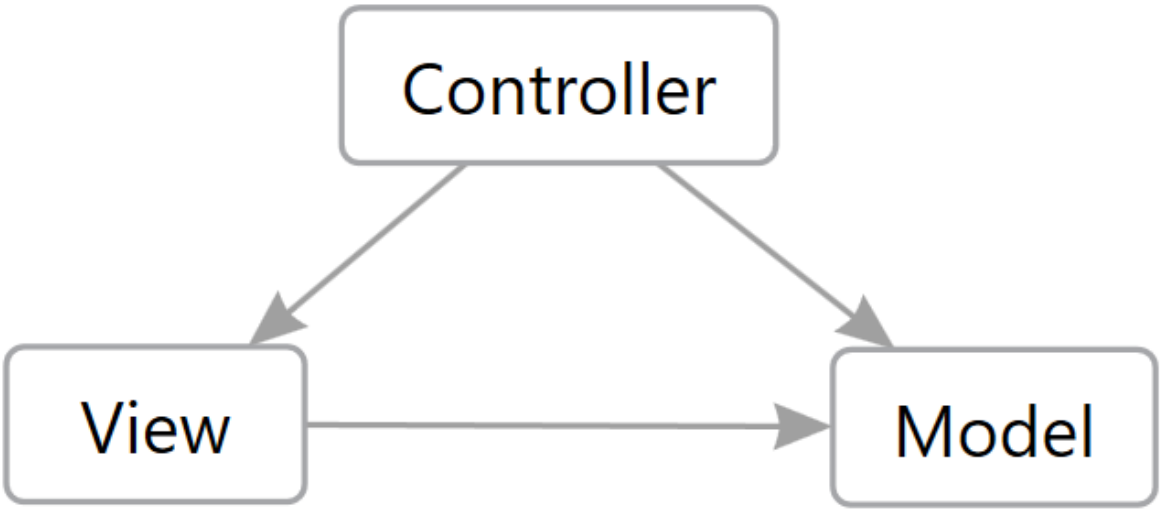
WCF aplikace mohou komunikovat s velkou škálou klientů pomocí různých protokolů. WCF je robustní, ale jestli vše, co je potřeba, jsou jen jednoduché služby založené na protokolu HTTP, vytvářet to pomocí WCF může být zbytečně komplikované. K tomuto účelu byla vytvořena technologie .NET Web API, která vychází z WCF.

Základem pro ASP.NET Web API slouží technologie MVC (Model-View-Controller). Jedná se o doplňkovou technologii, která vychází z MVC technologie a využívá řadu stejných pojmů jako jsou např. models, controllers a routing. Web API byla poprvé vydána s Visual Studiem 2012 jako součást MVC 4. [2]

6.2.1 ASP.NET MVC

Jedná se o framework od Microsoftu, který slouží k vyvíjení webových aplikací kombinující funkce MVC architektury, up-to-date nápady z agilního vývoje a části z existující platformy ASP.NET. MVC architektura odděluje uživatelské rozhraní (UI) od aplikační do tří částí, schéma je zobrazeno na Obr. 13: [7]

- **Model** – Množina tříd, která popisuje data, se kterými se pracuje.
- **View** – Definuje, jak bude zobrazeno uživatelské rozhraní aplikace. Je to čistě HTML kód.
- **Controller** – Soubor tříd, který obstarává komunikaci od uživatele, celý tok aplikací a aplikační logiku.



Obr. 13 MVC architektura [7]

6.2.2 Vytváření Endpointu a práce s ním

Na rozdíl od WCF se endpointy u Web API vytváří jenom ve zdrojovém kódu aplikace. Endpoint je asynchronní metoda, která se nachází ve vytvořené třídě pro *controller* ve složce *Controllers*. Aby třída mohla sloužit jako endpoint, musí ještě dědit od třídy *ApiController*, je-li kód psán jenom ve Web API nebo ze třídy *Controller*, která je součástí MVC. Každý endpoint může sloužit jenom k dělání jednoho ze čtyř účelů, kterými lze pokrýt plnou funkčnost aplikace:

- GET – slouží k načítání stávajících dat.
- SET – slouží k aktualizaci stávajících dat

- POST – slouží k vytváření nových dat
- DELETE – slouží k mazání stávajících dat

Každá metoda musí mít předem definovaný účel. Který se buď píše do názvu dané metody *GetNázevMetody* nebo následujícím způsobem `[HttpGet]` nebo alternativou, která se nachází před každou metodou. Dále se tam musí nacházet i trasování k dané metodě. Při použití čisté Web API používáme příkaz: [2]

```
[Route("Názevcontroller/nazevmetody/{parametr1,{parametr2}")]
```

Tento příkaz slouží k úplnému nastavení HTTP cesty, pod kterou se nachází daná metoda na webu. Při použití MVC a Web Api tento způsob nefunguje a používá se způsob z MVC ve kterém se píše pouze název metody a zbytek z úplné cesty je nastaven ve složce *App_Start* ve třídě *RouteConfig*:

```
[ActionName("nazevmetody")]
```

V případě, že je potřeba poslat parametry máme dva způsoby. První způsob je připojení parametru za URL adresu, kde stačí pouze za adresu dopsat parametry, které se mají poslat. Je jedno, zda se posílá řetězec nebo číselná hodnota, protože datový typ se převede automaticky v závislosti na parametru v metodě. Ukázka přidání parametrů do URL požadavku:

```
/ nazevmetody?parametr1=hodnota&parametr2=hodnota
```

Pokud se tímto způsobem bude posílat třeba heslo, tak zde hrozí nebezpečí, že bude prozrazeno, protože není obtížné odchytit URL požadavek a přečíst si posílané parametry. Pro tyto případy existuje druhý způsob, který je bezpečnější. Nehrozí tím únik citlivých informací, které jsou uloženy v těle zprávy a nejsou přístupné zvenčí. Na následujícím kódu je znázorněno, jak probíhá načítání dat z URL kódu a z těla požadavku:

```
nazevmetody ([FromBody()] JObject parametr, string parametr2)
```

Nejčastějším ze způsobů, jakým jsou posílána data v těle je datový typ Json nebo XML. Do těchto typů se zakódují požadovaná data a jsou posílána s požadavkem. V těle metody jsou potom data rozbalena a pracuje se s nimi obvyklým způsobem. [4]

7 UML

Jeden obrázek řekne více než tisíc slov, toto rčení by se dalo přirovnat k UML diagramu. Objektově orientované koncepty byly zavedeny mnohem dříve, než vzniklo UML. V té době nebyly žádné standartní metody pro upevnění objektově orientovaného vývoje. V tu chvíli byl vyvinut UML.

Je to zkratka pro Unified Modeling Language (UML), který je standartním jazykem pro specifikaci, vizualizaci, konstrukci a dokumentování softwarového systému. Jazyk byl vytvořen Object Management Group (OMG) a první návrh UML 1.0 byl v lednu roku 1997.

UML je odlišný jazyk oproti běžným programovacím jazykům jako je C++, Java, a další. Není to programovací jazyk, ale jeho nástroje mohou být použity pro generování kódu v různých jazycích pomocí UML diagramů. UML má přímou souvislost s objektově orientovaným návrhem a analýzou.

UML diagramy nejsou vyvinuty jen pro vývojáře, aby si nechávali generovat kódy a usnadnili si tím práci, ale také pro firemní uživatele a obyčejné lidi, kteří chtějí pochopit systém. Jedná se o jednoduchý a snadno pochopitelný mechanizmus na modelování všech druhů praktických systémů. [21]

UML zahrnuje následujících devět schémat, kde *Interaction diagram* zastupuje *Sequence a Collaboration diagram*: [22]

- *Class diagram*
- *Object diagram*
- *Use case diagram*
- *Interaction diagram*
- *Activity diagram*
- *Statechar diagram*
- *Deployment diagram*
- *Component diagram*

V další podkapitole je rozebrán jen *Class diagram*, protože tento diagram je jeden z nejčastěji používaných diagramů pro návrh architektury aplikace a v této práci je pomocí něho popsána celá architektura aplikace.

7.1 Class diagram

Je statický diagram, který představuje statický pohled na aplikaci. Používá se nejen pro vizualizace, popisování a dokumentaci systému, ale také pro konstrukci spustitelného kódu aplikace. Popisuje parametry a metody v dané třídě. Využívá se při modelování objektově orientovaných systémů, protože je to jediný UML diagram, ve kterém se mohou přímo mapovat objektově orientované jazyky.

Patří mezi nejpopulárnější UML diagram používaný pro navrhování aplikace. Proto je důležité umět s ním pracovat. Je to v podstatě grafické znázornění statického vzhledu systému, kde jsou představeny vlastnosti daných tříd, které v celku dávají dohromady celý systém.

Při návrhu je důležité se řídit následujícími body: [20]

- Názvy tříd by měli popisovat smysl dané třídy.
- Každý prvek a vztahy by měly být předem určeny.
- Prvky a metody by měly mít jasný název.
- Používat minimální počet prvků ve třídě (eliminování nepotřebných).
- Používat poznámky, aby byl jasně pochopený záměr.
- Před provedením konečné verze je potřeba se ujistit, že taková podoba je už konečná a správná.

Po dokončení úprav v diagramu následuje vygenerování příslušného kódu aplikace.

8 AUTOMATIZOVANÉ TESTOVÁNÍ APLIKACÍ

Při psaní kódu aplikace by se mělo testovat, zda je daná část kódu naprogramovaná správně a zda je splněna její funkčnost. Po každém napsání části kódu je potřeba zkontrolovat, zda se náhodou neudělalo něco, čím by se dosavadní funkčnost některých částí programu narušila

Aby se nemuselo provádět ruční testovaní každé části, naprogramují se automatizované UNIT testy, které provedou kontrolu aplikace pokrytou UNIT testy a zajistí funkčnost daného kódu. V případě chybného provedení testu se ve výsledku zobrazí, které testy neproběhly správně.

Dalším možným typem testování pro aplikace pracující s http požadavky jsou Web testy. Jejich programování je jednoduché, stačí napsat patřičný HTTP požadavek, který slouží k získávání nebo ukládání dat na webové službě. Výsledkem je status odpovědi 200 a pokud se vrací nějaká data v HTTP požadavku, potom jsou přiložena také.

8.1 UNIT testy

UNIT testy slouží k testování tříd a jejich metod. Většinou jsou dlouhé jenom několik řádků a doba trvání jednoho testu je v řádech sekund. Tím je zajištěna rychlá kontrola funkčnosti aplikace. S jistotou lze říct, že se test provede přesně tak, jak je naprogramován a provedou se všechny testy na rozdíl od ručního testování, kde se může zapomenout otestovat některou část kódu.

Když se píší UNIT testy, je potřeba vědět, že neslouží k testování ukládání dat do databáze. Na toto testování jsou jiné testy, které poznají, zda jsou data uložena v databázi. UNIT testy především slouží k testování funkčnosti knihoven, popřípadě jejich metod, které plní nějaký konkrétní programový účel a vrací výsledky. K otestování správnosti funkce slouží třída *Assert* a její metoda *AreaEqual*, která ohodnotí výsledek z metody a porovná ho s požadovaným výsledkem. Metoda může mít až tři parametry. Poslední parametr je nepovinný, který určuje přesnost výsledku, na který se má porovnávat: [23]

```
Assert.AreaEqual(výsledek,metoda(),přesnost);
```

8.2 Web testy

Web testy slouží k testování správné funkčnosti endpointů. Na rozdíl od UNIT testu se nezabývají pouze jednou konkrétní metodou, ale zkontrolují celý endpoint, zda proběhl správně se statusem 200 nebo při vykonávání testu nastala nespecifikovaná chyba se statusem 500, který informuje o neočekávané chybě za běhu endpointu.

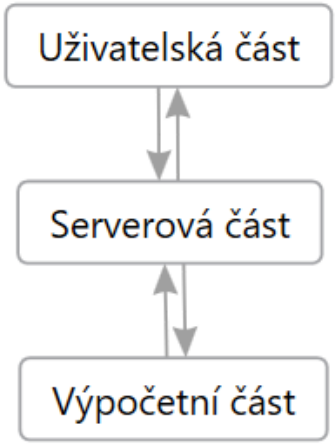
Web testy se vytvářejí velice jednoduše. Do existujícího testovacího projektu, kde jsou uložené UNIT testy přidáme *Web Performance Test*, do kterého se potom vepisují HTTP požadavky, kterými se testuje správnost endpointů. Existují zde dvě možnosti požadavků, první je metoda GET a druhá POST. Do každé z těchto metod se mohou vkládat parametry za URL adresu nebo do těla požadavku.

Po napsání příslušných testů, kterými se provede testování endpointů, se spustí testy, které proběhnou systematicky od shora dolů. Nato je potřeba myslet, pokud existuje test, který

pracuje s daty, které jsou nahrány v jiném prováděném testu. Kdyby se neupravilo pořadí testů, mohl by test skončit neúspěchem v závislosti na tom, jak je ošetřen celý program. [4]

9 PRAKTICKÉ ŘEŠENÍ APLIKACE

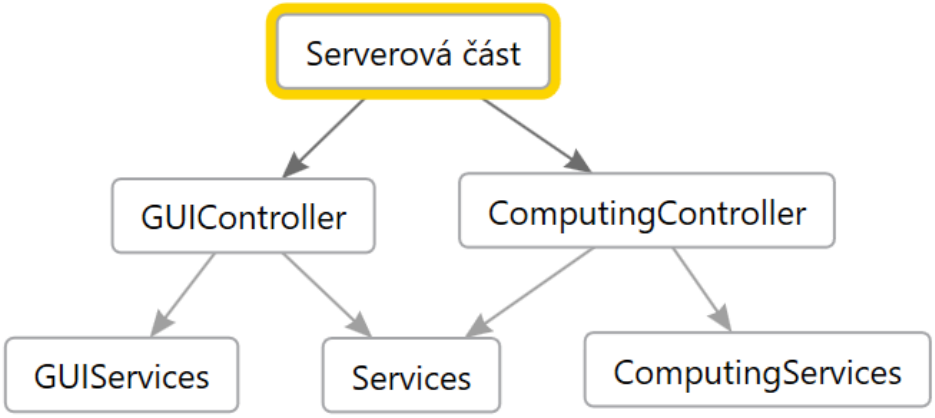
Celá serverová aplikace pro distribuované výpočty je rozdělena na tři části, které využívají společné datové modely a komunikují mezi sebou pouze přes HTTP požadavky. Každá z těchto částí je rozebrána v následujících podkapitolách, kde je vysvětleno, jakým způsobem spolu komunikují a jaké funkce každá část plní.



Obr. 14 Rozdělení aplikace

9.1 Serverová část

Serverová část aplikace je nejdůležitější částí celé aplikace, protože komunikuje se zbývajícími částmi aplikace, jak můžeme vidět na Obr. 14. Serverová část obsahuje dva controllery, *GUIController* a *ComputingController*. V těchto controllerech jsou zabudované endpointy, na které se dotazují zbývající části aplikace. Endpointy zpracují požadavek a v závislosti na požadavku volí jednu z příslušných *services*, kterou mají k dispozici. *Services* slouží k ukládání a načítání dat z databáze. Celé schéma zobrazení serverové části je znázorněné na Obr. 15.



Obr. 15 Aplikace - Serverová část

9.1.1 Controllery

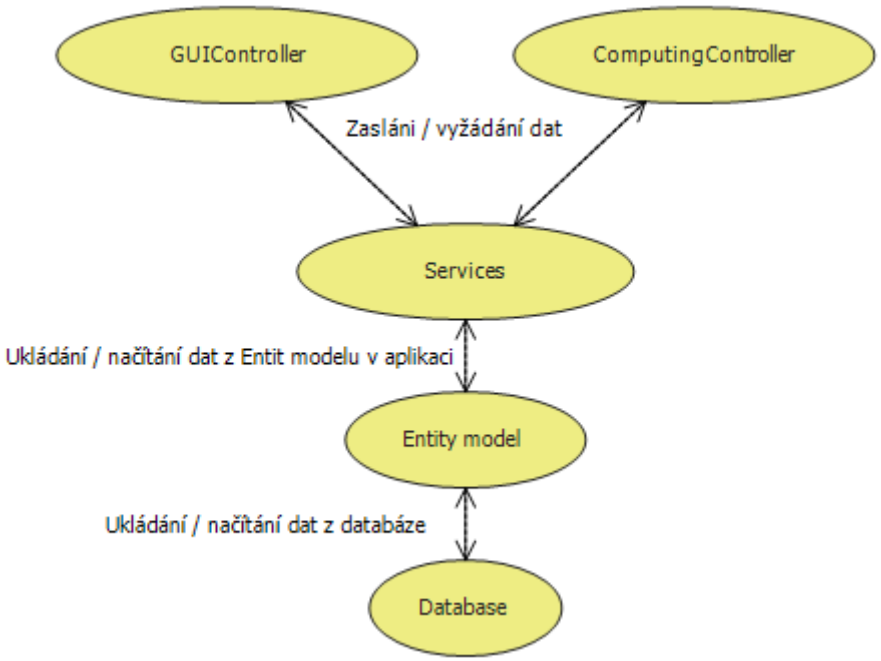
Každý controller obsahuje metody, které jsou využívány příslušnou částí aplikace. *GUIController*, jak už vyplývá z názvu slouží pro komunikaci s klientskou částí. Je znázorněn v class diagramu, který je v příloze A.

Každá metoda má v hranatých závorkách definované, jakým HTTP požadavkem s ní lze komunikovat a pod jakou adresou je k nalezení. Všechny metody obsahují v názvu *async*, který znamená, že daná metoda bude asynchronní. Návratová hodnota funkcí je *IHttpActionResult* zabalená v třídě *Task<>*. *IHttpActionResult* umožní při bezchybném dokončení metody vracet data ve formátu *Json*, kde jsou data zabalena do řetězce a poslána. V případě špatného dokončení metody se zavolá výjimka, která se odešle ve zprávě, že došlo k chybě za běhu programu. Ukázka volání metody:

```
public async Task<IHttpActionResult> NewUserLoginTest(string login)
```

Jak už bylo výše zmíněno, tak jsou metody psány asynchronně, proto i funkce psané v *services* musí být volány s použitím příkazu *await*, který zaručí, že program počká na dokončení dané metody a až dostane výsledek, potom teprve bude pokračovat v běhu. V době, kdy čeká na výsledek se nevyužívané vlákno uvolní pro jiné úkony a po dokončení metody, na kterou se čeká, se vlákno obnoví a program pokračuje dál.

V případě, že vznikne nějaká výjimka, program vygeneruje chybovou hlášku, která se uloží do databáze a metoda vrátí chybovou hlášku, že nastal neočekávaný problém. Pro případ, že by vypadla celá databáze je to zabezpečeno ukládáním chybových hlášek do souboru. Pro každou chybovou hlášku se generuje samostatný *.txt soubor, který se jmenuje podle data, kdy byla chyba vygenerována. Kdyby se náhodou stalo, že by bylo vygenerováno více chyb v jeden okamžik, tak program přiřadí pořadové číslo souboru s informací, kolikátá chyba se v daný okamžik vygenerovala.



Obr. 16 Tok dat v aplikaci

Druhý controller, který komunikuje s výpočetní části programu a slouží k posílání výpočetních souborů a ukládání výsledků, je naprogramován stejným způsobem jako *GUIController*. Jeho class diagram je znázorněn v příloze A. Na Obr. 16 je ukázka toku dat v aplikaci.

9.1.2 Services

Na Obr. 15 je možné vidět, s jakými services komunikuje každý controller. Každý controller má zvlášť svoji services, která využívá pouze své specifické privátní funkce, u kterých je zbytečné, aby byly k dispozici u jiného controlleru, který je stejně nevyužívá. Potom oba controllery mají jednu společnou services, která právě obsahuje funkce, které využívají společné privátní funkce. Class diagramy těchto tříd jsou znázorněny v příloze A.

V každé services jsou metody, které slouží k práci s daty v databázi. Když se pracuje s daty, které jsou načítány z databáze a potom vráceny v návratové hodnotě, tak se vždy vrací kopie dat z databáze. Kopie se provádí pomocí třídy *Mapper*, která vrací kopii dat načtených z databáze. Dělá se to jednak z důvodů bezpečnostního, aby se nepracovalo přímo s daty z databáze a také Web API nedovolí posílat data stažená přímo z databáze. Musí se nejdříve provést jejich nahrání do jiné proměnné a potom se mohou teprve posílat.

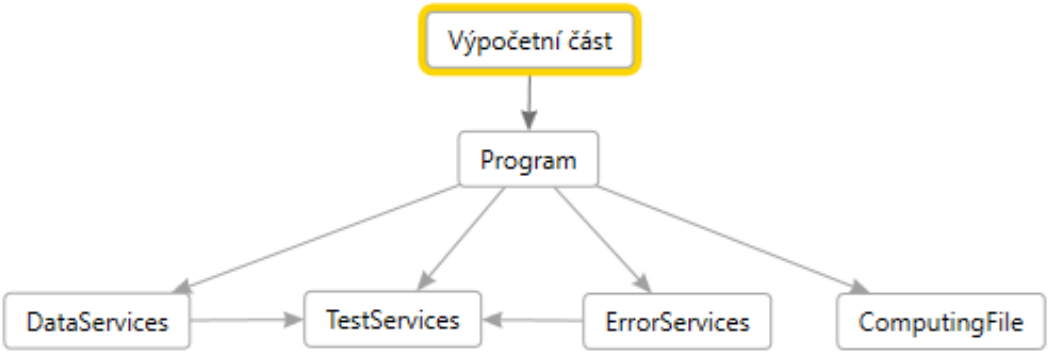
Z class diagramu v příloze A kde jsou znázorněny metody všech services a je patrné z názvu metod, jaký účel by měly plnit, až na privátní metodu v *ComputingServices*, která se jmenuje *PingServer()*. Tato metoda se volá z veřejné metody *CheckClient()*, která slouží ke kontrole klientů, kteří provádí výpočet a jsou pořád aktivní a není žádný problém ve spojení s nimi. Při prvním připojení se každý výpočetní klient uloží do databáze pod jeho IP adresou. Po uplynutí předem stanovené doby se zavolá funkce, která provede kontrolu všech klientů, zda jsou pořád připojeni do sítě a není žádný problém v komunikaci. Zjištění se provádí pomocí ping na IP adresu od klienta, kterou provádí zmiňovaná funkce *PingServer()*. Pokud při pokusu o ping vypršel timeout, tak se daný klient z databáze smaže a tím se udržuje aktualizovaný stav všech klientů.

9.2 Výpočetní část

Výpočetní část aplikace je obyčejná konzolová aplikace, která běží na každém klientském počítači, který má sloužit k provádění výpočtu. Když se program spustí, tak se provádí základní konfigurace celé aplikace. Důležité je nastavení správné IP adresy serveru, kam se budou směřovat veškeré dotazy. Pokud se aplikaci nepodaří spojit se serverem, vyzve k opětovnému zadání IP adresy. Spojení se nejdříve testuje posláním ping na uvedenou adresu. Pokud proběhl ping správně pošle se testovací spojení, které zaregistruje klienta do databáze na serveru. Klient je zaregistrován pod svojí IP adresou, která je načtena ze systému, aby nemohlo docházet ke špatnému zadání IP adresy a mohlo se testovat spojení ze strany serveru.

Po úspěšném složení úvodních testů, aplikace pošle požadavek na server, že je připravena k zahájení výpočtu a požaduje zaslání nějakého souboru, který může začít počítat. Před každým odesláním požadavku na server se provádí test spojení, zdali server není náhodou vypnutý. Když test spojení proběhne správně odešle se požadavek na server, aby byl poslán soubor, který se má počítat. Po stažení souboru se začne s výpočtem. Dokončení každého cyklu se odešle na server, aby se mohlo aktualizovat procento výpočtu u souboru uloženého v databázi. Po dokončení výpočtu jsou nashromážděné výsledky odeslány na server, kde proběhne jejich uložení a celý cyklus se opakuje znovu.

V případě, že by se stala nějaká neočekávaná chyba během výpočtu nebo práci se souborem, klient zahlásí chybu, která se odešle a uloží do databáze. Klient potom vypíše chybovou hlášku a ukončí činnost. Schéma výpočetní části je znázorněné na Obr. 17 a class diagram celé části je znázorněn v příloze B.



Obr. 17 Aplikace - Výpočetní část

Posílání dat mezi serverem a jeho částmi se provádí pomocí třídy *HttpClient*. Tato třída obsahuje metody, které slouží k posílání HTTP požadavků. Při programování byly využity tyto metody:

- `PostAsync(URLadresa, data)` – Slouží k posílání zabezpečených dat, ke kterým není přístup během posílání. Data se připojují ve formátu třídy *StringContent*.
- `PutAsync(URLadresa, data)` – Slouží k aktualizaci stávajících dat v databázi. Data se posílají zabalená stejně jako u metody *PostAsync()*.
- `GetStringAsync(URLadresa)` – Slouží k přenosu dat od serveru ke klientovi. Stáhne si řetězec, kde jsou zapsána všechna data a následně ho převede na objekt, který byl do řetězce převeden.
- `DeleteAsync(URLadresa)` – V URL adrese se pošle hodnota, která má být odstraněna z databáze a jako návratová hodnota je vrácen status zprávy 200, pokud vše dopadlo dobře.

Při posílání dat se metodou *PutAsync(),PostAsync()* a *DeleteAsync()* provádí kontrola úspěšného dokončení následovně. Nejdříve se zkontroluje, zda se nevyskytuje v návratové hodnotě žádná výjimka. Pokud se nevyskytuje, tak se otestuje z návratové hodnoty hodnota *response.Result.ReasonPhrase*, která se porovná, zda obsahuje řetězec ‚OK'. Kdyby se v návratové hodnotě vyskytla nějaká výjimka, tak by hodnota *ReasonPhrase* byla *null*.

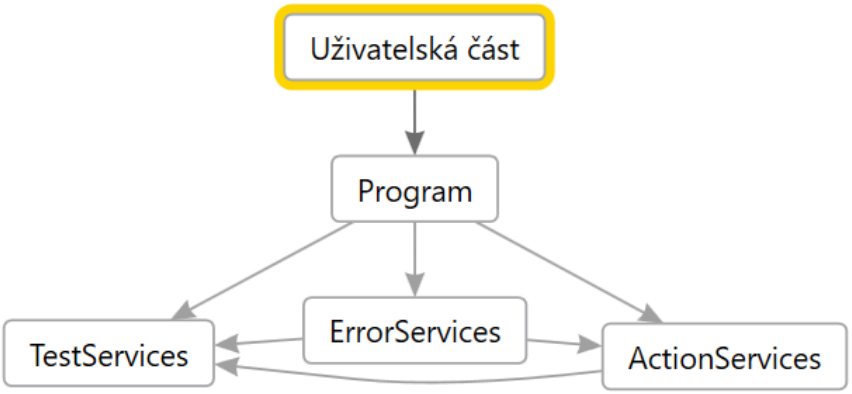
Při kontrolování přijatých dat z metody *GetStringAsync()* se kontroluje zda řetězec, který se vrací není roven *null*. Pokud řetězec obsahuje nějaká data, provede se jejich transformace na příslušný objekt.

Když odešleme jakýkoliv požadavek, tak je vrácen i status zprávy. Při správném průběhu, kdy nedošlo k žádné výjimce se vrátí status roven 200. Když se vyvolá nějaká výjimka za běhu a není ošetřena, dostaneme status roven 500. Pokud je špatně napsaný dotaz a URL adresa daného endpointu neexistuje, dostaneme zpátky status 405. Statusů je velké množství a

každý znamená něco jiného. Zde jsou uvedeny jen ty nejdůležitější, se kterými se programátor nejčastěji potká.

9.3 Uživatelská část

Uživatelská část je psána stejným způsobem jako klientská část. Před prvním šířením této aplikace je důležité nastavit IP adresu serveru. Předpokládá se, že se tato adresa už měnit nebude, proto je zvoleno programové nastavení této adresy.



Obr. 18 Aplikace - Uživatelská část

Schéma klientské části je znázorněné na Obr. 18 a detailnější class diagram je v příloze C. V třídě program běží hlavní část aplikace, kde se provádí načítání seznamu dle práv přihlášeného uživatele. Přihlášení probíhá vyžádáním loginu a hesla od uživatele. Pokud je vše zadáno správně, potom uživatel má na výběr z možností, které jsou zobrazeny na Obr.21.

Při přihlašování se vyplní login uživatele a potom je požadováno heslo. Při jeho zadávání se psané znaky do příkazového řádku schovávají pod znak „*“. Tím se zaručí nezveřejnění hesla. Funkce, která se používá pro načítání hesla je ukázána na Obr. 20.Po zadání hesla se vygeneruje příslušný *HashCode*, který zaručí jedinečnost zadaného hesla a velice komplikuje rozšifrování a tím zjištění hesla.

Po načtení celého hesla od uživatele se provede generování *HashCode* ze zadaného hesla v serverové části. Nejdříve se převede heslo na posloupnost bytů, ze kterých se vytvoří požadovaný *HashCode* za použití knihovny *Cryptography* a v ní zvoleného způsobu hashování *SHA256Managed.* Vytvořený *HashCode* je posloupnost bytů, která se v konečné fázi převede na řetězec, se kterým se už potom pracuje a posílá se k ověření.

```
var bytes = new UTF8Encoding().GetBytes(password);
byte[] hashBytes;
using (var algorithm = new System.Security.Cryptography.SHA256Managed())
{
    hashBytes = algorithm.ComputeHash(bytes);
}
password= Convert.ToBase64String(hashBytes);
```

Obr. 19 Generování HashCode

Existuje několik způsobů hashování, které nabízí daná knihovna. Programátor si může zvolit typ, kterým chce hashovat heslo a také na jakou délku ho chce hashovat. V programu je zvolena délka algoritmu 256 bitů. Tato velikost je dostatečně bezpečná, ale pro větší bezpečnost se může použít délka algoritmu 512 bitů. Názorná ukázka funkce, která načítá heslo a následně ho převádí na *HasCode* je prezentována na Obr. 19.

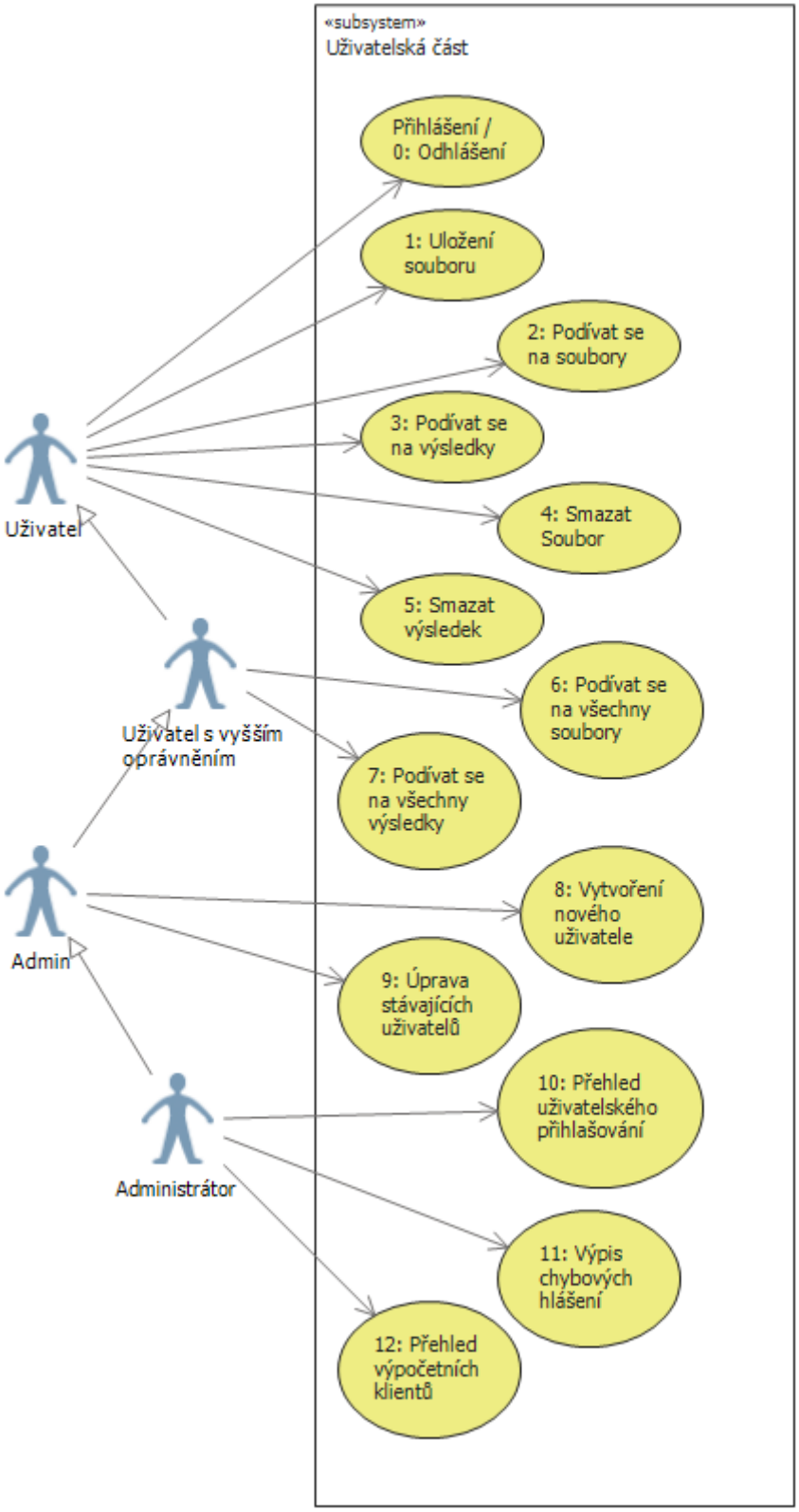
Při zadání čísla z výběru se provede příslušná akce. Pokud bude zadáno jiné číslo, než je možné z výběru, potom se akce bude opakovat, dokud se nezadá správné číslo. Podtržené řádky označují konec práv každé role. Každá nová role obsahuje schopnosti předešlé a některé nové.

Při načítání souboru je ošetřeno, zda daný soubor vůbec existuje na dané adrese, která je požadována po uživateli. Úkony číslo 5 a 6 jsou sice zahrnuty v základní roli, ale opět zde hraje roli právo uživatele. Pokud má uživatel větší roli než 2, potom se mu zobrazí všechny nahrané soubory a výsledky z databáze a může s nimi pracovat dle svého uvážení.

Při vytváření nového uživatele se hned při zadání loginu provádí kontrola, zda se daný login nevyskytuje v databázi. Pokud se už vyskytuje v databázi, vypíše uživateli chybovou hlášku a bude požadovat zadání nového loginu. Stejná kontrola je provedena, pokud je požadována změna u některého uživatele.

```
public string ReadPassword()
{
    string password = "";
    ConsoleKeyInfo info = Console.ReadKey(true);
    while (info.Key != ConsoleKey.Enter)
    {
        if (info.Key != ConsoleKey.Backspace)
        {
            Console.Write("*");
            password += info.KeyChar;
        }
        else if (info.Key == ConsoleKey.Backspace)
        {
            if (!string.IsNullOrEmpty(password))
            {
                password = password.Substring(0, password.Length - 1);
                int pos = Console.CursorLeft;
                Console.SetCursorPosition(pos - 1, Console.CursorTop);
                Console.Write(" ");
                Console.SetCursorPosition(pos - 1, Console.CursorTop);
            }
        }
        info = Console.ReadKey(true);
    }
    Console.WriteLine();
    return password;
}
```

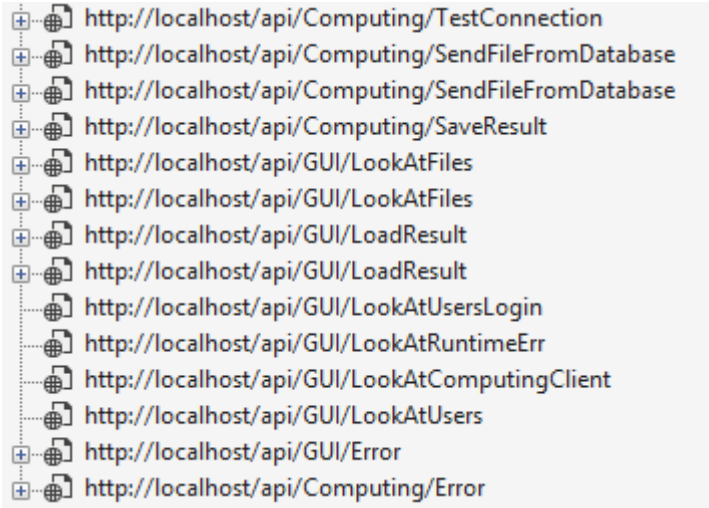
Obr. 20 Načítání hesla od uživatele



Obr. 21 Možnosti uživatele v závislosti na roli

9.4 Testování aplikace

Pro automatizované testování aplikace jsou dva způsoby, kterými lze aplikaci testovat. První jsou UNIT testy, které se provádí na testování správné funkčnosti metod, které mají vrátit hodnotu, která se poté porovnává. Tento způsob není ideální pro testování metod, které komunikují s databází. Proto existují web testy, které testují celé metody k příslušným endpointům a byly aplikovány na testování serverové části. Na Obr. 22 jsou znázorněny možné web testy.



Obr. 22 Ukázka web testů

Web testy psané ve Visual Studiu neumožnují posílat DELETE a PUT požadavku. Tím je znemožněno testování všech těchto metod. Po spuštění testů se provedou všechny testy a zobrazí se nám výsledky, které jsou vidět na Obr. 23. Je na něm zobrazen stav výsledku, zda proběhl daný test správně nebo s jakou chybou, čas, který byl potřeba pro požadavek a velikost odpovědí. Celá testovaná aplikace je provozována na IIS. V případě, že se spouští testování po delší době nebo po aktualizaci programu, tak první požadavek trvá déle, protože čeká, než rozběhne celá aplikace na IIS.

Request	Status	Total Time	Request Time	Request Bytes	Response Bytes
http://localhost/api/Computing/TestConnection	200 OK	0,020 sec	0,020 sec	0	4
http://localhost/api/Computing/SendFileFromDatabase	200 OK	0,015 sec	0,015 sec	0	86 278
http://localhost/api/Computing/SendFileFromDatabase	200 OK	0,509 sec	0,509 sec	0	2 690
http://localhost/api/Computing/SaveResult	200 OK	0,054 sec	0,054 sec	298	0
http://localhost/api/GUI/LookAtFiles	200 OK	0,515 sec	0,515 sec	0	3 394
http://localhost/api/GUI/LookAtFiles	200 OK	0,514 sec	0,514 sec	0	2 827
http://localhost/api/GUI/LoadResult	200 OK	0,017 sec	0,017 sec	0	19 174
http://localhost/api/GUI/LoadResult	200 OK	0,016 sec	0,016 sec	0	2
http://localhost/api/GUI/LookAtUsersLogin	200 OK	0,013 sec	0,013 sec	0	6 696
http://localhost/api/GUI/LookAtRuntimeErr	200 OK	0,017 sec	0,017 sec	0	13 418
http://localhost/api/GUI/LookAtComputingClient	200 OK	0,011 sec	0,011 sec	0	186
http://localhost/api/GUI/LookAtUsers	200 OK	0,013 sec	0,013 sec	0	314
http://localhost/api/GUI/Error	200 OK	0,022 sec	0,022 sec	263	0
http://localhost/api/Computing/Error	200 OK	0,024 sec	0,024 sec	262	0

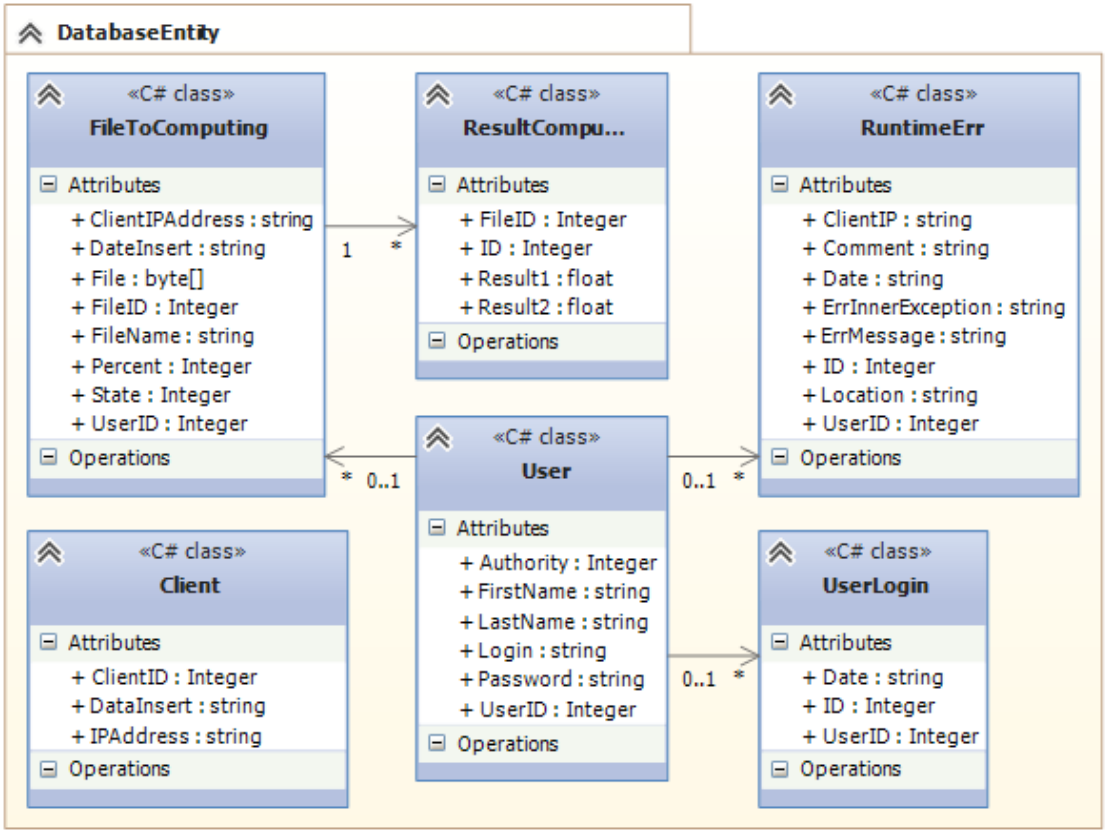
Obr. 23 Výsledky web testů

9.5 Databázový model

Celá datová architektura je vytvořena v MS SQL Server s pomocí Management Studia. V tomto programu byly vytvořeny všechny tabulky, se kterými komunikuje aplikace. Tabulky a jejich jednotlivé prvky s datovými typy jsou psané v jazyce C# na Obr. 24. Pro datový typ *string* je v databázi volen datový typ *nvarchar(délka)*. *Délka* je číslo, kolik prvků má maximálně obsahovat daný řetězec. Pro každý parametr je délka rozdílná, například pro heslo se používá délka 50 a pro *IPAddress* je délka 20. Jako datový typ *byte[]* je použit *varbinary(MAX)*.

Mezi tabulkami, které obsahují prvky, které na sebe odkazují, je vytvořen vztah, který hlídá, zda vůbec existuje odkaz na prvek, který se má vložit do tabulky. Pro porozumění je vytvořen uživatel, který má *UserID* rovno 1. Do tabulky *FileToComputing* je přidán soubor, který bude obsahovat *UserID* rovno 1. Databáze provede kontrolu, zda se v tabulce User nachází uživatel s *UserID* rovno 1. Pokud se daný uživatel nachází v tabulce, nový soubor se přidá do tabulky. V případě, že by se uživatel v tabulce nenacházel, tak by ani nebyl přidán soubor. Toto zabezpečení slouží k ochraně záznamů, aby se do tabulek nedostaly záznamy, které by tam vůbec neměly být.

Jediná tabulka, která je bez vazby na jinou tabulku, je tabulka *Client*. Do této tabulky se ukládá záznam připojení výpočetního klienta. Ukládá se do ní datum, kdy výpočetní klient navázal spojení se serverem a jeho komunikující IP adresa. Tato adresa je velice důležitá, protože v programu se provádí kontrola, zda pořád existuje spojení mezi serverem a klientem. V případě, že by došlo k chybě ve spojení a spojení by nebylo, tak bude daný klient s této tabulky odebrán, aby byl pořád udržován aktuální stav výpočetních klientů.

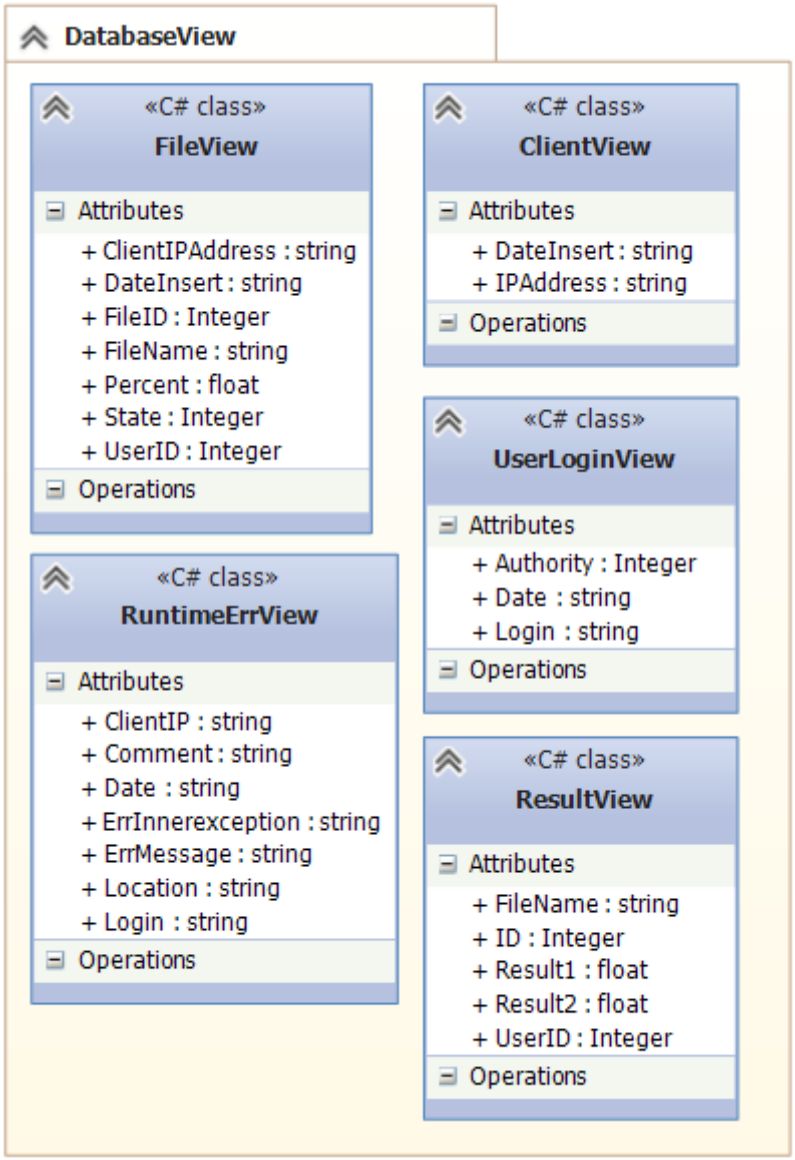


Obr. 24 Databázové tabulky

Do tabulky RuntimeErr se ukládají výjimky, které byly vyvolány při běhu programu. Ukládají se zde záznamy ze všech tří části aplikace. Tato tabulka obsahuje prvky, jako datum vzniku chyby, text chybové hlášky, která byla vyvolána, místo, kde chyba byla vyvolána. Místem je myšleno, v jaké části aplikace byla chyba vyvolána (serverová, klientská, uživatelská).

Tabulka ResultComputing slouží k ukládání vypočtených výsledků, které jsou zaslány od výpočetního klienta. V této tabulce jsou důležité pouze dva parametry *FileID* a *ID*. Zbývající parametry slouží pouze k testovacím účelům. V případě aplikace na konkrétní výpočetní případ se musí daná tabulka upravit, aby odpovídala vypočteným výsledkům.

Dále v databázi byly ještě vytvořeny pohledy, které jsou složené z více tabulek. Pohledy slouží k zobrazování informací v ucelenějším celku, který se využívá v zobrazovaní záznamů v uživatelské části. Na Obr. 25 jsou znázorněné názvy pohledů se všemi sloupečky, které se v nich nachází.



Obr. 25 Databázové pohledy

9.6 Komunikace mezi databází a serverem

Komunikace mezi serverovou částí a databází je postavena na technologii Entity Framework. Verze Entity frameworku, která je v aplikaci nainstalovaná a patří mezi nejaktuálnější verze (verze 6.1.3) v době, kdy byla programována aplikační část. Protože celý model databáze byl vytvořen dříve, než bylo započato s programováním, byla využita možnost vygenerování datového modelu v aplikaci za použití existující databáze. Tento model je znázorněn v příloze D.

Vytvořený datový model je přímá prezentace dat z databáze. Pro další práci s těmito daty je doporučeno pracovat s jejich kopiemi, které se vytváří pomocí technologie Automapper. Verze Automapperu, která je v aplikaci nainstalovaná patří mezi nejaktuálnější verze (verze 6.0.2) v době, kdy byla programována aplikační část.

Pro načítání celého seznamu dat s databáze pomocí entity frameworku se používaly příkazy z knihovny *Linq* a výsledná data byla zapsána do proměnné. V případě, že se jednalo o celou databázi, potom se data zapisovala za pomoci *ToList()* a při výběru jednoho záznamu se používá *Single()*. Pro výběr souboru na počítání se bral soubor se stavem 0 a těch se mohlo v tabulce nacházet více. Při použití příkazu *Single()* by nastala chyba, protože by hledaná hodnota nebyla pouze jedna, ale bylo by více souborů se stavem 0. Při tomto způsobu se používá *FirstOrDefault()*, který vezme první z možných hodnot, v případě, že by nebyla nalezena žádná možnost, potom vrátí *null*.

10 ZÁVĚR

Cílem této práce bylo vytvoření serverové aplikace pro distribuované výpočty s využitím nových technologií. V teoretické části je seznámení s problematikou, možnými technologiemi, které se mohou využít při programovaní této aplikace. V práci jsou vysvětleny základy technologií s ukázkami na příkladech. Tím by mohlo být pochopení problematiky jednodušší než v knihách, které jsou zaměřeny obecně a nejsou znázorněny konkrétní možnosti, které jsou v některých případech potřeba.

Začátek této práce je zaměřen na distribuované výpočty, kde je nastíněna současná problematika a možnosti řešení problému, který spočívá ve vytvoření vlastní distribuované sítě nebo využití cloudového řešení, které může být v jistých případech i levnějším řešením.

Další část práce je zaměřena na popsání nejnovější technologie od Microsoftu .Net Core, která je vyvíjena v posledních letech a je řazena k jedné z hlavních a razantních změn provedených na platformě .NET, které by měly usnadnit práci programátorům využívající danou technologii. Jednou z největších změn byl přechod na multiplatformní verzi a vyvinutí jedné knihovny, která je společná pro všechny platformy.

Výpočty musí mít možnost si ukládat na nějaké místo vypočítané výsledky a odněkud čerpat zadání výpočtu. K tomuto účelu je využíváno MS SQL, o kterém je další kapitola v práci. Je v ní prezentováno, jak si nastavit SQL server pro více uživatelů, nastavit jim správné přístupy a práva. Pro větší bezpečnost jsou zmíněny i způsoby zálohování dat uložených v databázi. Data se z databáze musí různými způsoby stahovat, aby se s nimi mohlo pracovat. Na toto téma je zaměřena další kapitola, kde jsou vysvětleny možné způsoby komunikace mezi aplikací a databází. Je zde zmíněn i model vícevrstvých aplikací a prezentována programová pomůcka, která usnadňuje práci s kopírováním dat stažených z databáze do vyšších vrstev aplikace.

V další kapitole jsou rozebrány technologie, kterými lze programovat distribuované výpočty. Jedná se o WCF a ASP.NET Web API. WCF je starší technologie, která slouží k velkému množství možností. Dá se s ní naprogramovat velice robustní aplikace, na velkém množství protokolů. Na rozdíl od WCF je ASP.NET WEB API menšího zaměření, pracuje pouze s HTTP požadavky, a slouží k programování velice jednoduchých aplikací, které nejsou náročné na požadavky.

Celá aplikace je navrhována pomocí UML class diagramu, pomocí kterého se lépe chápe celá architektura aplikace a v neposlední řadě se dají vygenerovat třídy, které se mohou použít v aplikaci.

V práci je zmíněné i automatizované testování, které usnadňuje kontrolu naprogramované části aplikace bez opětovného ručního testování všech částí. Jedná se o užitečnou věc, která bývá v hodně případech opomíjena z důvodu časové tísně.

Poslední kapitola je zaměřena na popis kompletní aplikace, která je rozdělena do tří hlavních částí, které mezi sebou komunikují pomoci HTTP požadavků. Serverová část zpracovává požadavky od klienta a uživatele a ukládá data do databáze. Výpočetní část zpracovává výpočty a posílá data serveru. Poslední částí je uživatelské rozhraní, které dává přehled o kompletním dění celé aplikace. Uživatelé zde mají přístup v závislosti na přihlašovacích údajích, které mají přiřazená práva, která jim omezují možnosti.

SEZNAM POUŽITÝCH ZDROJŮ

[1] LACKO, Ľuboslav. *1001 tipů a triků pro SQL*. Brno: Computer Press, 2011. ISBN 978-80-251-3010-0.

[2] TROELSEN, Andrew a Philip JAPIKSE. *C# 6.0 and the .NET 4.6 Framework*. Seventh Edition. New York: Springer Science+Business Media, 2015. ISBN 978-148-4213-339.

[3] LACKO, Ľuboslav. Mistrovství v SQL Server 2012: [kompletní průvodce databázového experta]. Brno: Computer Press, 2013. ISBN 978-80-251-3773-4.

[4] NAGEL, Christian. *Professional C# 6 and .Net Core 1.0*. 1. Indianopolis: John Wiley & Sons, 2016. ISBN 11-190-9660-X.

[5] .NET Core. *What is .NET Core?* [online]. Cranbury: infragistics, 2017 [cit. 2017-01-19]. Dostupné z: http://www.infragistics.com/community/blogs/marketing/archive/2015/01/13/what-is-net-core-and-why-does-it-matter.aspx

[6] .Net Standart. *.NET Blog* [online]. Remond: Microsoft, 2016 [cit. 2017-01-20]. Dostupné z: https://blogs.msdn.microsoft.com/dotnet/2016/09/26/introducing-net-standard/

[7] ASP.NET MVC - Pattern. *Tutorials Point* [online]. Hyderabad: Tutorials Point, ©2017 [cit. 2017-02-3]. Dostupné z: https://www.tutorialspoint.com/asp.net_mvc/asp.net_mvc_pattern.htm

[8] Cloud Computing. *Cloud Computing vs. Distributed Computing* [online]. Redwood City: DeZyre, 2015 [cit. 2017-03-03]. Dostupné z: https://www.dezyre.com/article/cloud-computing-vs-distributed-computing/94

[9] DataSet část 1. *IT network* [online]. Praha: IT network, 2014 [cit. 2017-02-25]. Dostupné z: http://www.itnetwork.cz/csharp/databaze/c-sharp-net-tutorial-databaze-ado-net-dataset-sqldataadapter

[10] DataSet část 2. *IT network* [online]. Praha: IT network, 2014 [cit. 2017-01-25]. Dostupné z: http://www.itnetwork.cz/csharp/databaze/c-sharp-net-tutorial-odpojena-databazova-aplikace-dataset-podruhe

[11] DbContext. *Entity Framework Tutorial* [online]. -: Entity Framework Tutorial, 2016 [cit. 2017-01-27]. Dostupné z: http://www.entityframeworktutorial.net/EntityFramework4.3/dbcontext-vs-objectcontext.aspx

[12] Distributed and Parallel Computing. *Distributed and Parallel Computing* [online]. California: EECS Instructional Support, 2014 [cit. 2017-01-30]. Dostupné z: http://www-inst.eecs.berkeley.edu/~cs61a/sp12/book/communication.html

[13] Entity Framework - Database Operations. *Tutorials Point* [online]. Hyderabad: Tutorials Point, ©2017 [cit. 2017-02-13]. Dostupné z: https://www.tutorialspoint.com/entity_framework/index.htm

[14] Entity Model. *Entity Framework Tutorial* [online]. -: Entity Framework Tutorial, 2016 [cit. 2017-01-21]. Dostupné z: http://www.entityframeworktutorial.net/EntityFramework5/create-dbcontext-in-entity-framework5.aspx

[15] Entity Relationships. *Entity Framework Tutorial* [online]. -: Entity Framework Tutorial, 2016 [cit. 2017-01-28]. Dostupné z: http://www.entityframeworktutorial.net/entity-relationships.aspx

[16] Objektově relační mapování. *LINQ to SQL: Basic Concepts and Features* [online]. Toronto: CodeProject, 2011 [cit. 2017-01-27]. Dostupné z: https://www.codeproject.com/Articles/215712/LINQ-to-SQL-Basic-Concepts-and-Features

[17] Práce s SQL příkazy. *IT network* [online]. Praha: IT network, 2014 [cit. 2017-01-25]. Dostupné z: http://www.itnetwork.cz/csharp/databaze/c-sharp-tutorial-databaze-ado-net-insert-update-delete-count

[18] Přehled přístupu do databáze. *IT network* [online]. Praha: IT network, 2014 [cit. 2017-01-25]. Dostupné z: http://www.itnetwork.cz/csharp/databaze/c-sharp-tutorial-pristupy-pro-praci-s-relacni-databazi

[19] Připojená aplikace. *IT network* [online]. Praha: IT network, 2014 [cit. 2017-02-22]. Dostupné z: http://www.itnetwork.cz/csharp/databaze/c-sharp-tutorial-pripojena-databazova-aplikace

[20] UML - Class Diagram. *Tutorials Point* [online]. Hyderabad: Tutorials Point, ©2017 [cit. 2017-01-31]. Dostupné z: https://www.tutorialspoint.com/uml/uml_class_diagram.htm

[21] UML - Overview. *Tutorials Point* [online]. Hyderabad: Tutorials Point, ©2017 [cit. 2017-01-31]. Dostupné z: https://www.tutorialspoint.com/uml/uml_overview.htm

[22] UML Building Blocks. *Tutorials Point* [online]. Hyderabad: Tutorials Point, ©2017 [cit. 2017-02-24]. Dostupné z: https://www.tutorialspoint.com/uml/uml_building_blocks.htm

[23] Unit testy. *IT network* [online]. Praha: IT network, 2014 [cit. 2017-05-10]. Dostupné z: https://www.itnetwork.cz/csharp/testovani/testovani-v-csharp-net-uvod-do-unit-testu-a-priprava-projektu

[24] WCF - Architecture. *Tutorials Point* [online]. Hyderabad: Tutorials Point, ©2017 [cit. 2017-03-04]. Dostupné z: https://www.tutorialspoint.com/wcf/wcf_architecture.htm

[25] WCF - Hosting WCF Service. *Tutorials Point* [online]. Hyderaba: Tutorials Point, ©2017 [cit. 2017-03-04]. Dostupné z: https://www.tutorialspoint.com/wcf/wcf_hosting_service.htm

[26] WCF - Service Binding. *Tutorials Point* [online]. Hyderabad: Tutorials Point, ©2017 [cit. 2017-03-04]. Dostupné z: https://www.tutorialspoint.com/wcf/wcf_service_binding.htm

[27] What is AutoMapper? *Kunal-chowdhury* [online]. Kolkata: kunal-chowdhury, 2013 [cit. 2017-02-13]. Dostupné z: http://www.kunal-chowdhury.com/2013/01/what-is-automapper-and-how-to-map-objects.html#sY0FC8W6EYTJHzSI.97

[28] Zabezpečení přenosu dat. *IT Network* [online]. Praha: IT network, 2014 [cit. 2017-01-25]. Dostupné z: http://www.itnetwork.cz/csharp/databaze/c-sharp-tutorial-pripojena-databazova-aplikace-zpracovani

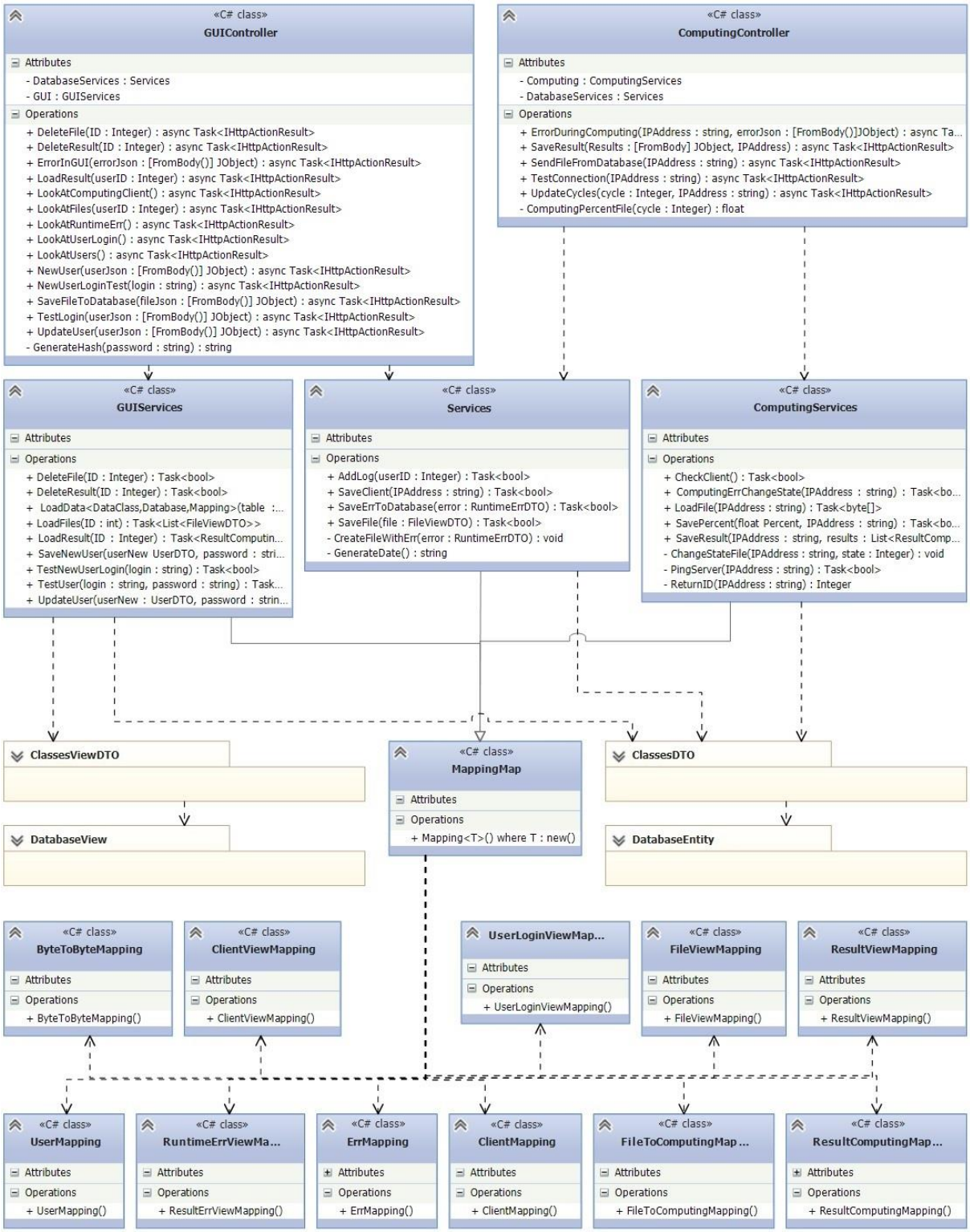
SEZNAM ZKRATEK

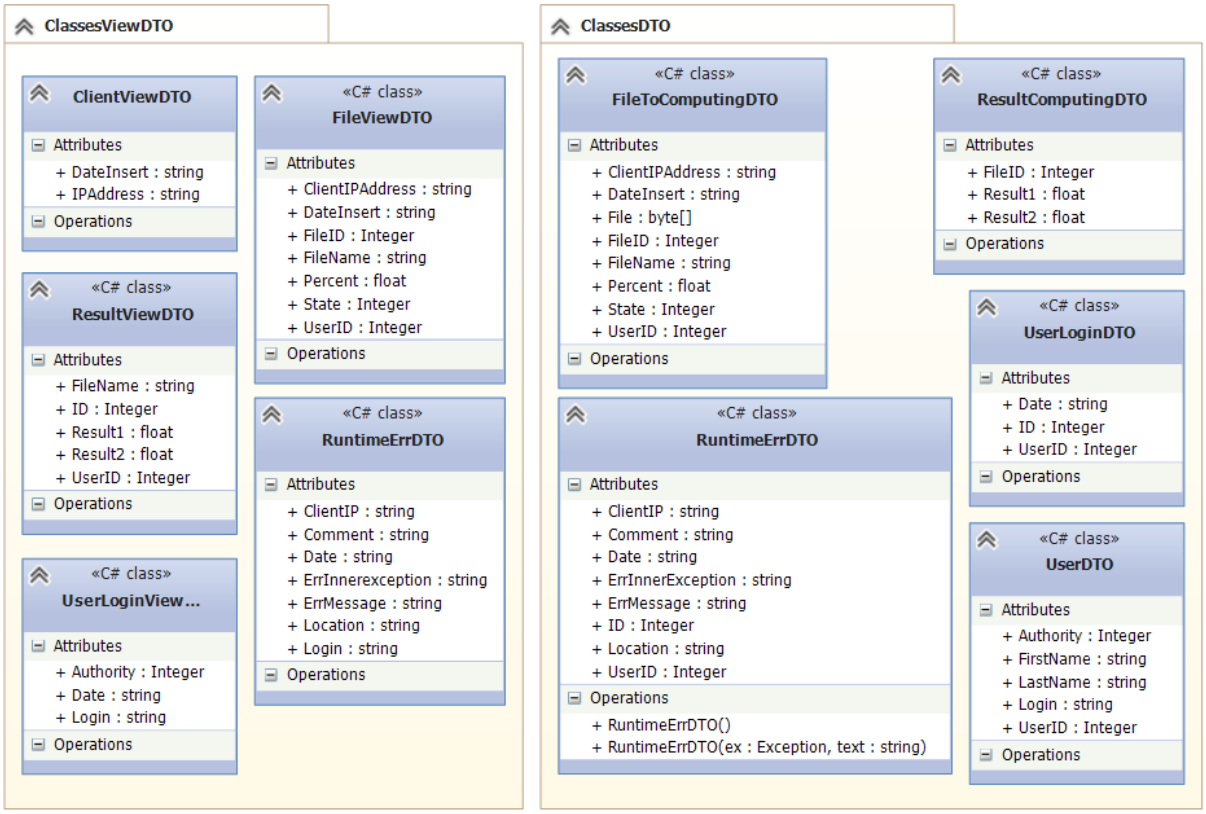
VB - Visual Basic
CLR – Common Language Runtime
SQL – Structured Query Language
ORM – Object-Relational Mapping
DTO – Data Transfer Object
WCF – Windows Communication Foundation
MVC – Model View Controller
UML – Unified Modeling language
OMG – Object Management Group
ABC – Address Binding Contract
SOAP – Simple Object Access Protocol.
IIS – Internet Information Services
UI – User Interface
WAS – Windows Activation Service
IT – Information Technology
TCP – Transmission Control Protocol
IPC – Inter-process Communication
HTTP – Hypertext Transfer Protocol
WS – Web Service
MSMQ – Microsoft Message Queuing

SEZNAM OBRÁZKŮ

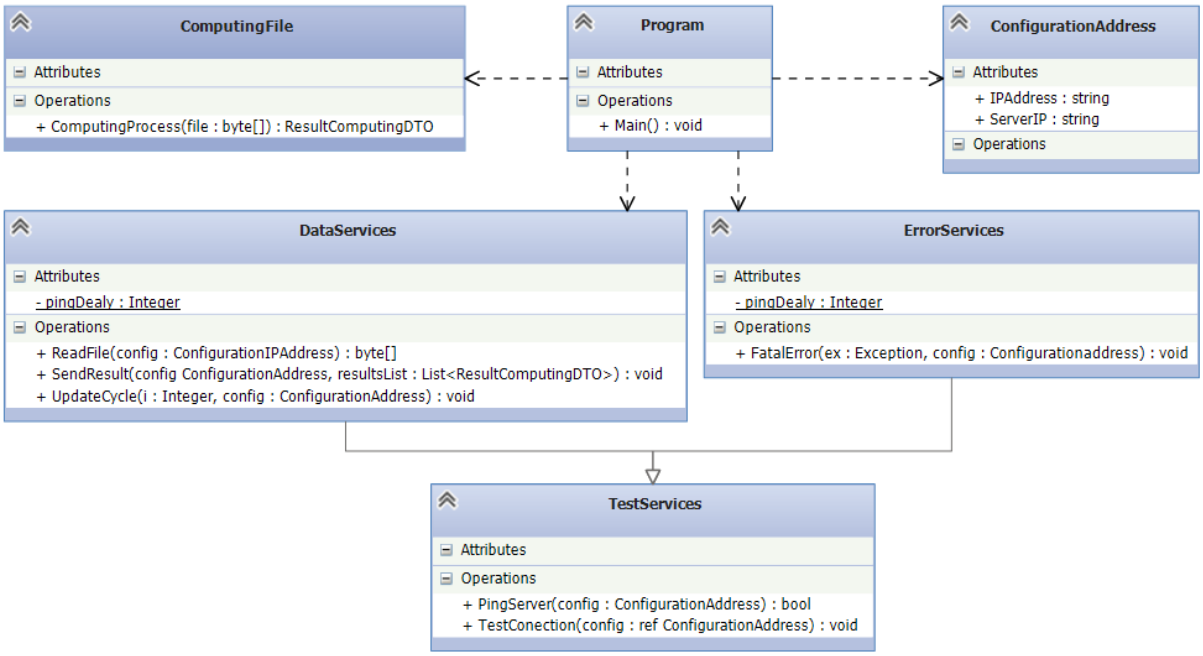
SEZNAM PŘÍLOH

PŘÍLOHA A – CLASS DIAGRAM SERVEROVÉ ČÁSTI

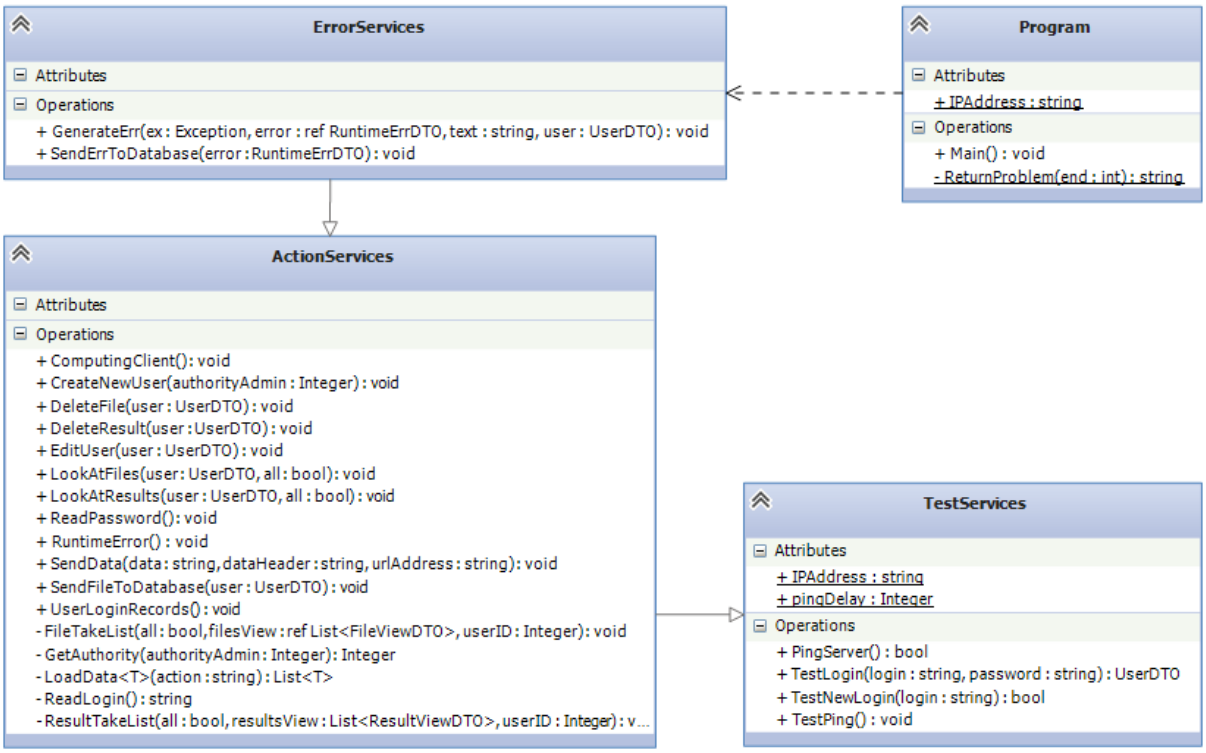



ClassesViewDTO
ClientViewDTO
Attributes
+ DateInsert : string
+ IPAddress : string
Operations
«C# class»
FileViewDTO
Attributes
+ ClientIPAddress : string
+ DateInsert : string
+ FileID : Integer
+ FileName : string
+ Percent : float
+ State : Integer
+ UserID : Integer
Operations
«C# class»
ResultViewDTO
Attributes
+ FileName : string
+ ID : Integer
+ Result1 : float
+ Result2 : float
+ UserID : Integer
Operations
«C# class»
RuntimeErrDTO
Attributes
+ ClientIP : string
+ Comment : string
+ Date : string
+ ErrInnerexception : string
+ ErrMessage : string
+ Location : string
+ Login : string
Operations
«C# class»
UserLoginView...
Attributes
+ Authority : Integer
+ Date : string
+ Login : string
Operations
ClassesDTO
«C# class»
FileToComputingDTO
Attributes
+ ClientIPAddress : string
+ DateInsert : string
+ File : byte[]
+ FileID : Integer
+ FileName : string
+ Percent : float
+ State : Integer
+ UserID : Integer
Operations
«C# class»
ResultComputingDTO
Attributes
+ FileID : Integer
+ Result1 : float
+ Result2 : float
Operations
«C# class»
UserLoginDTO
Attributes
+ Date : string
+ ID : Integer
+ UserID : Integer
Operations
«C# class»
RuntimeErrDTO
Attributes
+ ClientIP : string
+ Comment : string
+ Date : string
+ ErrInnerException : string
+ ErrMessage : string
+ ID : Integer
+ Location : string
+ UserID : Integer
Operations
+ RuntimeErrDTO()
+ RuntimeErrDTO(ex : Exception, text : string)
«C# class»
UserDTO
Attributes
+ Authority : Integer
+ FirstName : string
+ LastName : string
+ Login : string
+ UserID : Integer
Operations
DatabaseView
«C# class»
FileView
Attributes
+ ClientIPAddress : string
+ DateInsert : string
+ FileID : Integer
+ FileName : string
+ Percent : float
+ State : Integer
+ UserID : Integer
Operations
«C# class»
ClientView
Attributes
+ DateInsert : string
+ IPAddress : string
Operations
«C# class»
UserLoginView
Attributes
+ Authority : Integer
+ Date : string
+ Login : string
Operations
«C# class»
RuntimeErrView
Attributes
+ ClientIP : string
+ Comment : string
+ Date : string
+ ErrInnerexception : string
+ ErrMessage : string
+ Location : string
+ Login : string
Operations
«C# class»
ResultView
Attributes
+ FileName : string
+ ID : Integer
+ Result1 : float
+ Result2 : float
+ UserID : Integer
Operations
DatabaseEntity
«C# class»
FileToComputing
Attributes
+ ClientIPAddress : string
+ DateInsert : string
+ File : byte[]
+ FileID : Integer
+ FileName : string
+ Percent : Integer
+ State : Integer
+ UserID : Integer
Operations
1
*
«C# class»
ResultCompu...
Attributes
+ FileID : Integer
+ ID : Integer
+ Result1 : float
+ Result2 : float
Operations
«C# class»
RuntimeErr
Attributes
+ ClientIP : string
+ Comment : string
+ Date : string
+ ErrInnerException : string
+ ErrMessage : string
+ ID : Integer
+ Location : string
+ UserID : Integer
Operations
* 0..1
«C# class»
User
Attributes
+ Authority : Integer
+ FirstName : string
+ LastName : string
+ Login : string
+ Password : string
+ UserID : Integer
Operations
0..1 *
«C# class»
Client
Attributes
+ ClientID : Integer
+ DataInsert : string
+ IPAddress : string
Operations
0..1 *
«C# class»
UserLogin
Attributes
+ Date : string
+ ID : Integer
+ UserID : Integer
Operations

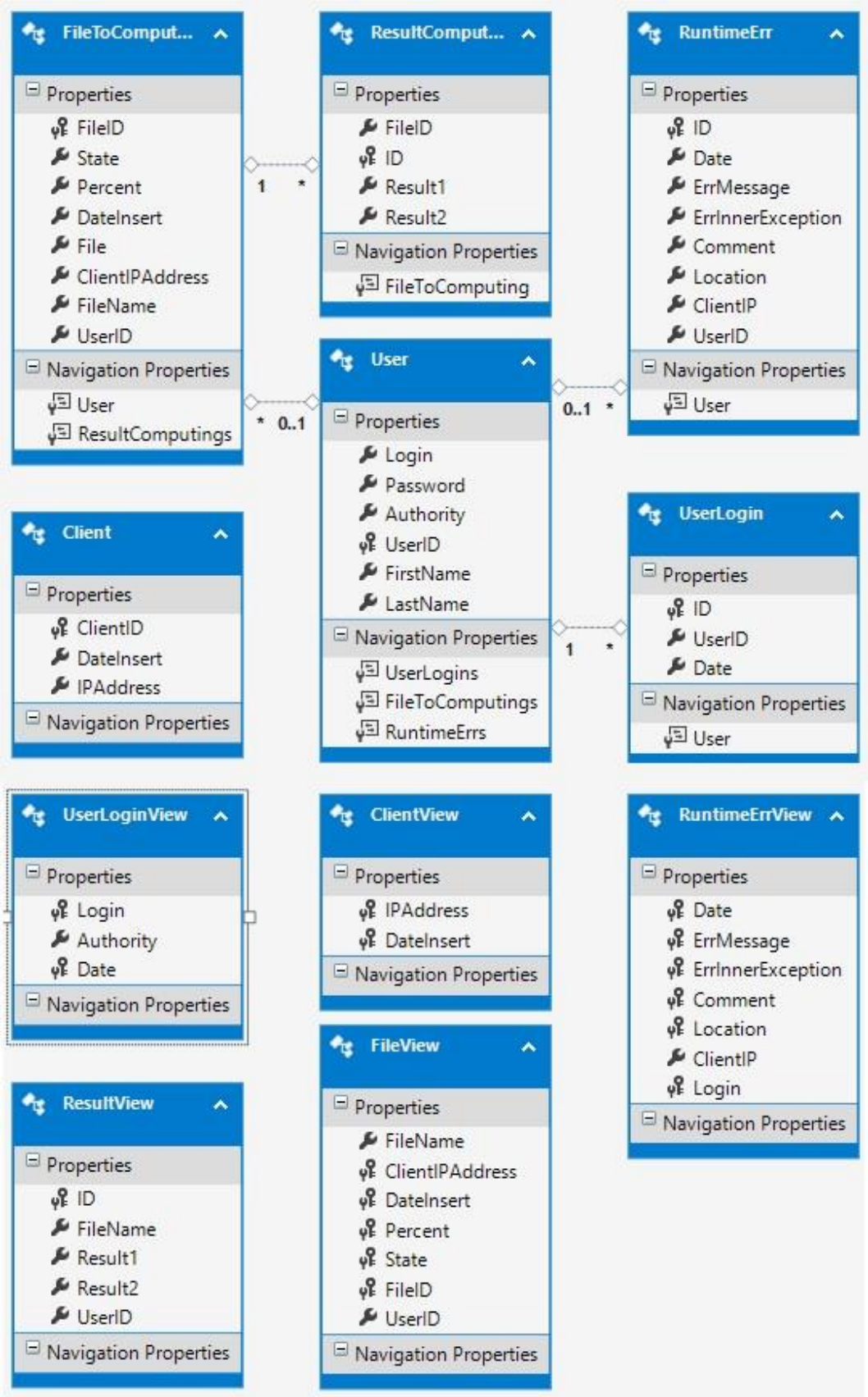
PŘÍLOHA B – CLASS DIAGRAM VÝPOČETNÍ ČÁSTI



PŘÍLOHA C – CLASS DIAGRAM UŽIVATELSKÉ ČÁSTI



PŘÍLOHA D – ENTITY FRAMEWORK – DATOVÝ MODEL



PŘÍLOHA E – DATOVÝ NOSIČ

Elektronická verze práce.

Program serverová aplikace pro distribuované výpočty